# Retek® Merchandising System 10.0


## Operations Guide – Volume 2:

### Message Publication and Subscription Designs

Retek

The software described in this documentation is furnished under a license agreement and may be used only in accordance with the terms of the agreement.

No part of this documentation may be reproduced or transmitted in any form or by any means without the express written permission of Retek Inc., Retek on the Mall, 950 Nicollet Mall, Minneapolis, MN 55403.

Information in this documentation is subject to change without notice.

Retek provides product documentation in a read-only-format to ensure content integrity. Retek Customer Support cannot support documentation that has been changed without Retek authorization.

**Corporate Headquarters:**

Retek Inc.
Retek on the Mall
950 Nicollet Mall
Minneapolis, MN 55403

888.61.RETEK (toll free US)
+1 612 587 5000

**European Headquarters:**

Retek
110 Wigmore Street
London
W1U 3RW
United Kingdom

Switchboard:
+44 (0)20 7563 4600

Sales Enquiries:
+44 (0)20 7563 46 46
Fax: +44 (0)20 7563 46 10

Retek® Merchandising System™ is a trademark of Retek Inc.

Retek and the Retek logo are registered trademarks of Retek Inc.

©2002 Retek Inc. All rights reserved.

All other product names mentioned are trademarks or registered trademarks of their respective owners and should be treated as such.

Printed in the United States of America.

# *Customer Support*

**Customer Support hours:**

8AM to 5PM Central Standard Time (GMT-6), Monday through Friday, excluding Retek company holidays (in 2002: Jan. 1, May 27, July 4, July 5, Sept. 2, Nov. 28, Nov. 29, and Dec. 25).

**Customer Support emergency hours:**

24 hours a day, 7 days a week.

| Contact Method | Contact Information |
|---|---|
| **Phone** | US & Canada: 1-800-61-RETEK (1-800-617-3835)<br>World: +1 612-587-5000 |
| **Fax** | (+1) 612-587-5100 |
| **E-mail** | support@retek.com |
| **Internet** | www.retek.com/support<br>Retek's secure client Web site to update and view issues |
| **Mail** | Retek Customer Support<br>Retek on the Mall<br>950 Nicollet Mall<br>Minneapolis, MN 55403 |

**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.

# Contents

# Chapter 1 - Publishing designs

## Allocations Message Family Manager Publishing Design

### Functional Area

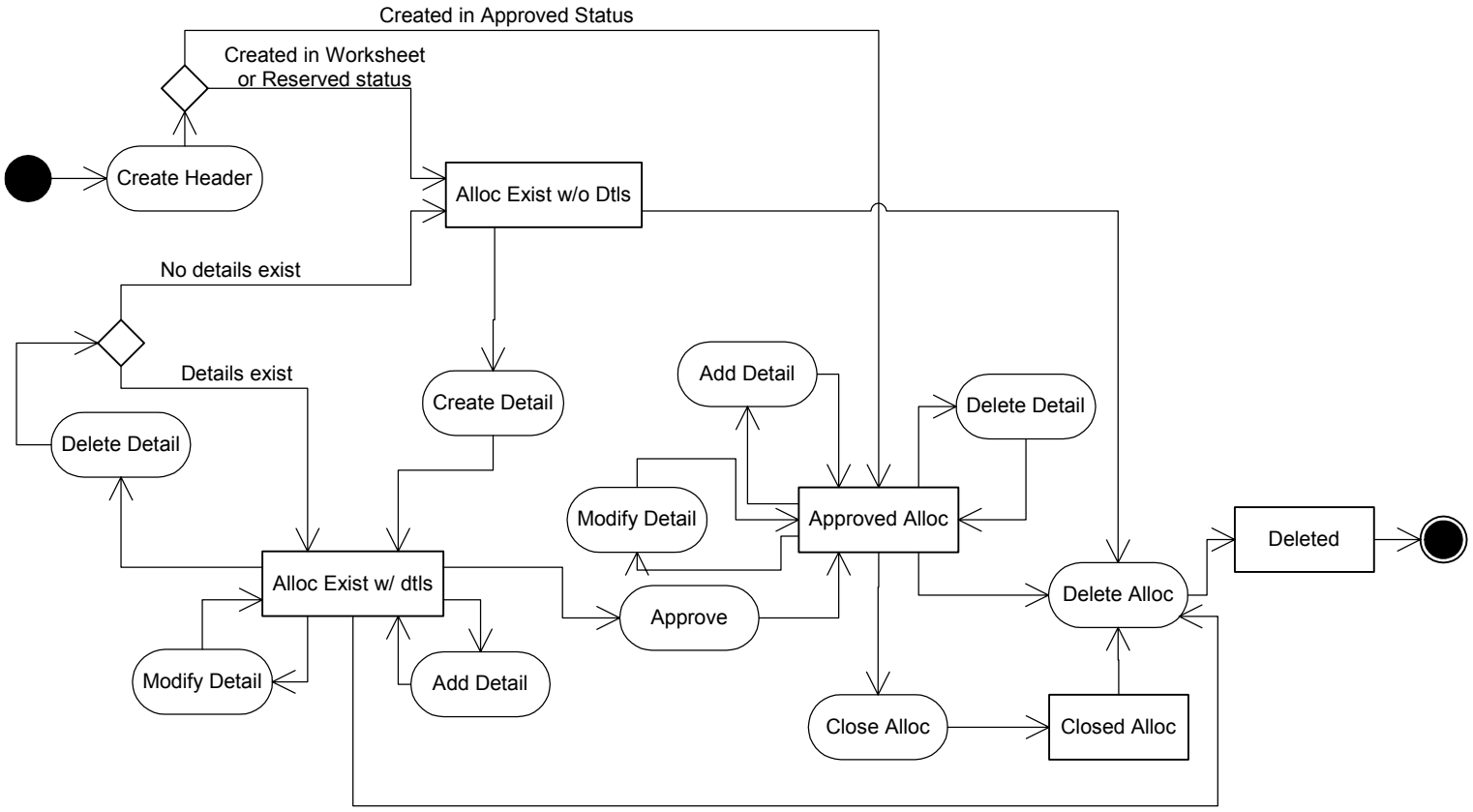
Stock Order Publication – Allocations

### Design Overview

RMS is responsible for communicating allocation information with external systems such as RDM. There are three different ways in which allocations can be entered into RMS. The first method is through the Allocations stand-alone product. These allocations will be 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 will be sent to the integration bus. Detail modification messages for allocations will not be sent to the bus. The second way allocations can be created is through the Semi-automatic ordering option. Via this replenishment method, allocations and orders will be inserted into the alloc_header and alloc_detail tables in worksheet status and will have to be manually approved. In order for allocation messages to be sent to the bus the allocation must at least be in 'A'pproved status. Worksheet messages will remain on the queue and combined until they are approved. Once this occurs one large create message is sent to the bus. The final way which allocations are entered into RMS is via automatic replenishment allocations. These allocations are initially set in worksheet status and are approved by the rplapprv.pc batch program. Once again only messages for approved allocations will be sent to the bus.

Modified and deleted allocation information will also be sent to the bus. 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 will be ignored at the detail level. A header delete message will signify that the complete allocation can be deleted.

# State Diagram




# Description of Activities

## Create Header

1. **Prerequisites**: Allocation can be created in one of three manners: via the stand-alone allocations product, semi-manual ordering, or automatic 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 till 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 my 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 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 an allocation detail is created an "Allocation Detail Created 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). If an Allocation Create message is also in the queue for the same allocation the two messages will be 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 bus.

## 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 a "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.

## Triggers

Trigger Description (EC_TABLE_ALH_AIUDR):

This trigger fires on any insert, update or delete for the alloc_header table.

On Insert: This trigger will call the ALLOC_XML.BUILD_MESSAGE function to determine the necessary values to be included in the published message as well as build the message itself. This trigger also calls the RMSMFM_ALLOC.ADDTOQ function to add the appropriate values to the queue.

action_type: A

message_type : AllocCre

wh: :new.wh

alloc_no: :new.alloc_no

order_no: :new.order_no,

item: :new.item

order_type :new.order_type

status :new.status

On Update: This trigger will call the ALLOC_XML.BUILD_MESSAGE function to determine the necessary values to be included in the published message as well as build the message itself. This trigger also calls the RMSMFM_ALLOC.ADDTOQ function to add the appropriate values to the queue.

action_type: M

message_type : AllocHdrMod

wh: :new.wh

alloc_no: :new.alloc_no

order_no: :new.order_no

item: :new.item

order_type :new.order_type

status :new.status

On Delete: This trigger will call the ALLOC_XML.BUILD_MESSAGE function to determine the necessary values to be included in the published message as well as build the message itself. This trigger also calls the RMSMFM_ALLOC.ADDTOQ function to add the appropriate values to the queue. See specifics on these two functions below. The AllocDesc.dtd should be used to create this message. These variables will need to be set to the following values:

action_type := 'D';

message_type := 'AllocDel';

alloc_no := :old.alloc_no;

L_record_alc.ORDER_NO := :old.order_no;

item := :old.item;

alloc_status := :old.status;

wh_id := :old.wh;

order_type := :old.order_type;

Public Functions:

**ALLOC_XML. BUILD_MESSAGE (O_status, O_text, O_message, IO_alloc_msg, I_action_type)–** This function is called by the trigger EC_TABLE_ALH_AIUDR on insert, update and delete of the ALLOC_HEADER table. This function gathers all the data necessary to build the message that needs to be sent to the Retek Integration Bus.

Trigger Description (EC_TABLE_ALD_AIUDR): Calls ALLOC_XML.BUILD_MESSAGE on insert, update, and delete of table ALLOC_DETAIL and then calls RMSMFM_ALLOC.ADDTOQ to populate the staging table ALLOC_MFQUEUE.

On Insert:

action_type: 'A'

message_type: 'AllocDtlCre'

alloc_no: :new.alloc_no

to_loc: :new.to_loc

qty_allocated: :new.qty_allocated

to_loc_type: :new.to_loc_type

On Update:

action_type: 'M'

message_type: 'AllocDtlMod'

alloc_no: :new.alloc_no

to_loc: :new.to_loc

qty_allocated: :new.qty_allocated

to_loc_type: :new.to_loc_type

On Delete:

action_type: 'D'

message_type: 'AllocDtlCre'

alloc_no: :old.alloc_no

to_loc: :old.to_loc

Public Functions:

ALLOCDTL_XML.BUILD_MESSAGE (L_status, L_text, L_message, L_orig_ind, L_record_alc, L_action_type) – This function is called by the trigger EC_TABLE_ALD_AIUDR.

## Message Family Manager Procedures

Public Procedures:

ADDTOQ (O_status_code, O_error_msg, I_message_type, I_alloc_no, I_virtual_from_loc, I_physical_from_loc I_to_loc_type, I_to_loc, I_alloc_status, I_orig_ind, I_message) – This procedure is called by the capture triggers, ec_table_adr_aiudr.trg and ec_table_alh_aiudr.trg, and takes the message type, family key values (I_alloc_no, I_virtual_from_loc, I_physical_from_loc, I_to_loc_type, I_to_loc) and, the message itself. It inserts a row into the allocation message family queue along with the passed in values (I_alloc_status and I_orig_ind) and the next sequence number from the allocation message family sequence, setting the status to unpublished. It returns error codes and an error message.

GETNXT (O_status_code, O_error_msg, O_message_type, O_message, O_alloc_no, O_virtual_from_loc, O_physical_from_loc, O_loc_type, O_to_loc_type, O_to_loc) – 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 keys (O_alloc_no, O_virtual_from_loc, O_physical_from_loc, O_loc_type, O_to_loc_type, O_to_loc) 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; these codes come from an EAI team defined RIB_CODES package. For more discussion of the status codes, refer to the Error Handling Guidelines document.

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.

Private Procedures:

**CREATE_PREVIOUS (I_queue_rec)** – This procedure determines if a header level create already exists on the queue table for the same key value and with a sequence number less than the current records sequence number.

**CLEAN_QUEUE (I_queue_rec)** – This procedure cleans up the queue by eliminating modification messages. It is only called if CREATE_PREVIOUS returns true. For each modification message type, it finds the previous corresponding create message type. If the message type is AllocDtlMod it then calls REPLACE_QUE_ALLOC_DETAIL to copy the modify detail message into the create message and calls DELETE_QUEUE_REC to delete the modify record. Else if the message type is AllocHdrMod it then calls REPLACE_QUE_ALLOC_HEADER to copy the modify header 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.

**MAKE_CREATE (O_msg, I_queue_rec)** – This procedure 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.

**DELETE_QUEUE_REC (I_seq_no)** – This procedure deletes a specific record from the queue. It deletes based on the sequence number passed in.

**REPLACE_QUEUE_ALLOC_HEADER (I_rec, I_data)** – This procedure replaces the message in the create header record with the message from the modify header record.

**REPLACE_QUEUE_ALLOC_DETAIL (I_rec, I_data)** – This procedure replaces the message in the create detail record with the message from the modify detail record. There will be one of these procedures for each detail level.

**REPLACE_QUEUE_MESSAGE (I_rec, I_data)** – This procedure replaces the message in the message in the create header record with the complete hierarchical message.

# ATP Message Family Manager Publishing Design

## Functional Area

Available to Promise

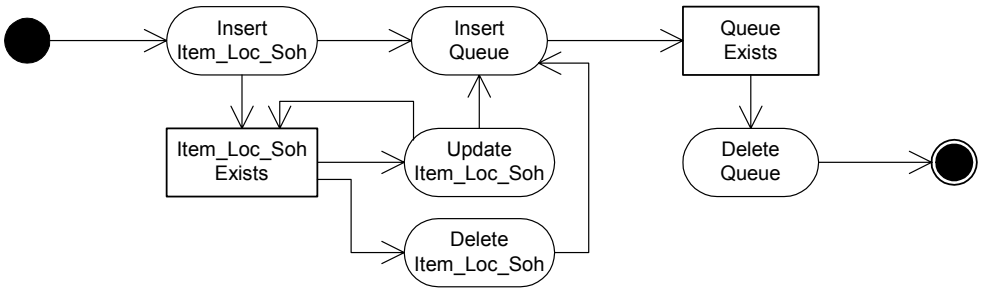
## Design Overview

ATP publication consists of a single flat message containing item/location stock data from the table ITEM_LOC_SOH. A snapshot of the table will be placed in the message queue each time a record is created or modified. The message family manager will create messages from these snapshots, which will be retrieved in the order they were put in the queue. The messages will then be sent to the bus. No delete messages will be sent to the bus. ITEM_LOC_SOH records are deleted if and only if the item is deleted from the system, so publication of deletes will be handled by the Item MFM.

ATP information will not be published until the item on the ATP record has been published. The ATP MFM checks to see if the item has been published by looking at the Item Publishing Queue, ITEM_MFQUEUE. If the queue contains an Item Create message, the item has not been published yet. If the queue does not contain an Item Create message, the item has either been published or deleted. If the item has been deleted, the ITEM_LOC_SOH records for the item will also have been deleted, and no ATP information for the item will be published. If the item has been published, the ITEM_LOC_SOH records for the item will still exist in the system, and the ATP information for the item will be published. ATP records will stay on the queue until the item is published or deleted.

If a record on the ITEM_LOC_SOH table is deleted, a record is added to the ATP queue. The record will contain the ATP's primary key (item and location) along with the message type "ATPDel." This record will never be published. It is used by the ATP MFM to keep track of which ATP records have been deleted. When the ATP MFM has an ATP Create message that it is ready to publish, after checking to see if the item has been published, there will be a second check to see if there is an ATP Delete message further down the queue. If it finds an ATP Delete message, all of the ATP records for that item/location combination on the queue between the Create and Delete are removed, including the Create and Delete. If the ATP MFM does not find any ATP Delete messages, the ATP Create message is published.

## State Diagram




## Description of Activities

### Insert ILS

1 **Prerequisites**: A *relationship* exists between the item and location.

2 **Activity Detail**: Once an ITEM_LOC_SOH record has been created, ATP information is ready to be *inserted* into the queue, along with a message type of "ATPCre."

Messages: None.

### Insert Queue

1 **Prerequisites**: An ITEM_LOC_SOH record has been created.

2 **Activity Detail**: ATP *information* is inserted into the queue. The ATP information includes a snapshot of the ITEM_LOC_SOH record and the message type.

3 Messages: *None*.

### Update ILS

1 **Prerequisites**: An *ITEM*_LOC_SOH record has been created.

2 **Activity Detail**: Once an *ITEM*_LOC_SOH record has been modified, ATP information is ready to be inserted into the queue, along with a message type of "ATPMod."

3 Messages: *None*.

### Delete ILS

1. **Prerequisites**: An ITEM_LOC_*SOH* record has been created.
2. **Activity Detail**: Once an ITEM_LOC_SOH record has been deleted, a message type of "ATPDel" is inserted into the queue, along with the record's primary key (item and location fields.) No other ATP information is inserted into the queue since the "ATPDel" message will not be published.
3. Messages: *None*.

### Delete Queue

1. **Prerequisites**: ATP information has been inserted into the queue.
2. **Activity Detail**: If the ITEM_LOC_SOH record has been deleted, all of the ATP messages for that item/location combination are removed from the queue without being published. Otherwise, if the *ATP's* item has been published, a message is created from the information in the queue. The message and message type are sent to a RIB publication adaptor, and the information in the queue is then deleted.
3. **Messages**: The message is an XML message that contains all of the information from the ITEM_LOC_SOH table.

## Triggers

Trigger Description (EC_TABLE_ILI_AIUDR):

This trigger will capture inserts, updates and deletes to the ITEM_LOC_SOH table and write data into the ATP_MFQUEUE message queue. It will call RMSMFM_ATP.ADDTOQ to insert a snapshot of ITEM_LOC_SOH into the message queue.

On Insert: A message type of "ATPCre" is inserted into the message queue, along with a snapshot of the ITEM_LOC_SOH record.

On Update: A message type of "ATPMod" is inserted into the message queue, along with a snapshot of the ITEM_LOC_SOH record.

On Delete: A message type of "ATPDel" is inserted into the message queue, along with the record's primary key (item and location fields.)

## Message Family Manager Procedures

Public Procedures:

**ADDTOQ(O_status_code, O_error_msg, I_message_type, I_record)** – This procedure is called by EC_TABLE_ILI_AIUDR, and takes the message type and a record that contains all of the values in the ITEM_LOC_SOH table. It inserts a row into the message family queue ATP_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.

**GETNXT (O_status_code, O_error_msg, O_message_type, O_message, O_item, O_loc)** – 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 is the item and location, which will be populated for all message types. The program creates the xml message internally, using values in the message queue. Status code is one of five values, as shown in the following table. For more discussion of the status codes, refer to the Error Handling Guidelines document or the process flow in the following section. These codes come from an EAI team defined 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.

# Banner Message Family Manager Publishing Design

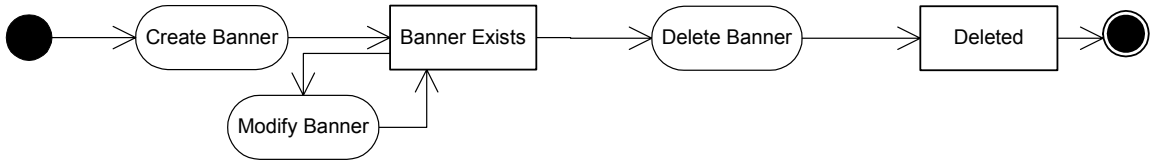
## Functional Area

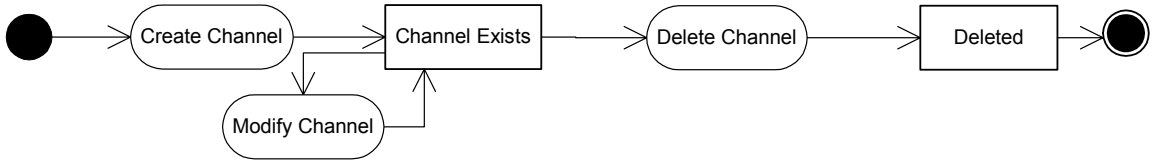
Banner/Channel

## Design Overview

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.

## State Diagram



# Description of Activities

## 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.

# Triggers

Trigger Description (EC_TABLE_BAN_AIUDR):

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, then call RMSMFM_BANNER.ADDTOQ to insert this message into the message queue.

On Insert: A BannerDesc message containing information from the BANNER table is created.

On Update: A BannerDesc message containing information from the BANNER table is created.

On Delete: A BannerRef message containing the banner id is created.

Trigger Description (EC_TABLE_CHN_AIUDR):

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, then call RMSMFM_BANNER.ADDTOQ to insert this message into the message queue.

On Insert: A ChannelDesc message containing information from the CHANNELS table is created.

On Update: A ChannelDesc message containing information from the CHANNELS table is created.

On Delete: A ChannelRef message containing the channel id is created.

## Message Family Manager Procedures

Public Procedures for Banner MFM:

**ADDTOQ(O_status_code, O_error_msg, I_message_type, I_banner_id, I_channel_id, I_message)** – 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.

**GETNXT (O_status_code, O_error_msg, O_message_type, O_message, O_banner_id, O_channel_id)** – 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 consists the banner id), which will be populated for all message types, and the channel id, which can be NULL. Status code is one of five values, as shown in the following table. For more discussion of the status codes, refer to the Error Handling Guidelines document or the process flow in the following section. These codes come from an EAI team defined 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.

# Differentiator Group Message Family Manager Publishing Design

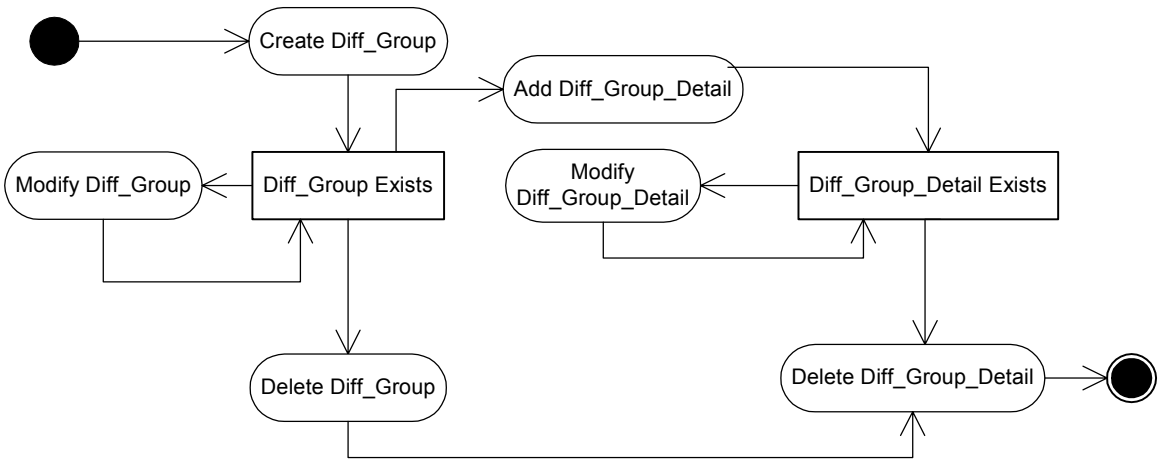
## Functional Area

Differentiator Groups

## Design 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.

## State Diagram



# Description of Activities

## Create Diff_Group

1. **Prerequisites**: Diff_Group doesn't 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 uda 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 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) exists.
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) exists.
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. Since 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.

## Triggers

Triggers should only insert records onto the staging table if no record already exists or if a record does exist but is locked.

Trigger Description (EC_TABLE_DGH_AIUDR): This trigger fires on any insert, update or delete on the Diff_Group_Head table. It captures the new data for inserts and updates. It captures the old data on deletes. It sets the action type and message type and calls the DIFFGRPHDR_XML.BUILD_MESSAGE procedure to build the message. The record is inserted into the DIFFGRP_MFQUEUE table by calling the RMSMFM_DIFFGRP.ADDTOQ procedure (I_diff_id is passed in as NULL).

<u>On Insert:</u>

Sets action_type to ‘A’dd and message_type to ‘DiffGrpHdrCre’.

<u>On Update:</u>

Sets action_type to ‘M’odify and message_type to ‘DiffGrpHdrMod’.

<u>On Delete:</u>

Sets action_type to ‘D’elete and message_type to ‘DiffGrpDel’.

**DIFFGRPHDR_XML. BUILD_MESSAGE (O_status, O_text, O_message, I_record, I_action_type)–** This function is called by the trigger EC_TABLE_DGH_AIUDR on insert, update and delete of the Diff_Group_Head table. This function gathers all the data necessary to build the message that needs to be sent to the Retek Integration Bus. It determines the proper message to build based on the action_type that is sent in the trigger. It builds DiffGrpRef xml messages for delete statements, or DiffGrpHdrDesc xml messages for updates or inserts.

Trigger Description (EC_TABLE_DGD_AIUDR): This trigger fires on any insert, update or delete on the Diff_Group_Detail table. It captures the new data for inserts and updates. It captures the old data on deletes. It sets the action type and message type and calls the DIFFGRPDTL_XML.BUILD_MESSAGE procedure to build the message. The record is inserted into the DIFFGRP_MFQUEUE table by calling the RMSMFM_DIFFGRP.ADDTOQ procedure.

On Insert:

Sets action_type to 'A'dd and message_type to 'DiffGrpDtlCre'.

On Update:

Sets action_type to 'M'odify and message_type to 'DiffGrpDtlMod'.

On Delete:

Sets action_type to 'D'elete and message_type to 'DiffGrpDtlDel'.

**DIFFGRPDTL_XML. BUILD_MESSAGE (O_status, O_text, O_message, I_record, I_action_type)–** This function is called by the trigger EC_TABLE_DGD_AIUDR on insert, update and delete of the Diff_Group_Detail table. This function gathers all the data necessary to build the message that needs to be sent to the Retek Integration Bus. It determines the proper message to build based on the action_type that is sent in the trigger. It builds DiffGrpDtlRef xml messages for delete statements, or DiffGrpDtlDesc xml messages for updates or inserts.

## Message Family Manager Procedures

Public Procedures:

**ADDTOQ(O_status_code, O_error_msg, I_message_type, I_message, I_diff_group_id, I_diff_id, I_message)** – 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 (for more information on consolidation messages refer to the Integration Design Patterns document). It returns error codes and strings according to the standards of the application in which it is being implemented.

**GETNXT (O_status_code, O_error_msg, I_diff_group_id, I_diff_id, O_message_type, O_message)** – This publicly exposed procedure is typically called by a RIB publication adaptor. It'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 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. For more discussion of the status codes, refer to the Error Handling Guidelines document or the process flow in the following section. These codes come from an EAI team defined 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. The facility id is only included in messages coming from RDM.

# Differentiatior ID Message Family Manager Publishing Design

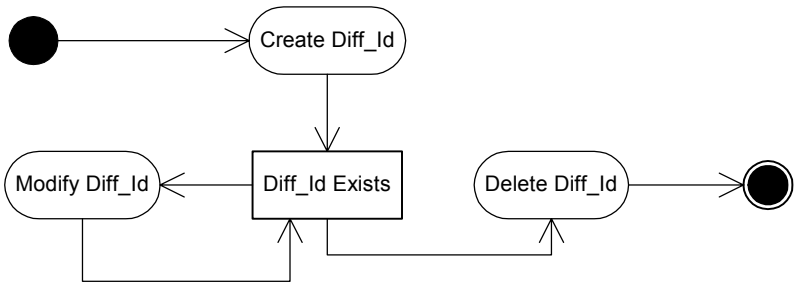
## Functional Area

Differentiator Ids

## Design Overview

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.

## State Diagram




## Description of Activities

### Create Diff_Id

1 **Prerequisites**: Diff_Id doesn't 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_group 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 uda 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.

# Triggers

Triggers should only insert records onto the staging table if no record already exists or if a record does exist but is locked.

Trigger Description (EC_TABLE_DID_AIUDR): This trigger fires on any insert, update or delete on the Diff_Ids table. It captures the new data for inserts and updates. It captures the old data on deletes. It sets the action type and message type and calls the DIFFID_XML.BUILD_MESSAGE procedure to build the message. The record is inserted into the DIFFID_MFQUEUE table by calling the RMSMFM_DIFFID.ADDTOQ procedure.

On Insert:

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

On Update:

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

On Delete:

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

**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 Retek 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.

## Message Family Manager Procedures

Public Procedures:

**ADDTOQ(O_status_code, O_error_msg, I_message_type, I_diff_id, I_message)** – 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 (for more information on consolidation messages refer to the Integration Design Patterns document). It returns error codes and strings according to the standards of the application in which it is being implemented.

**GETNXT (O_status_code, O_error_msg, O_message_type, O_message, O_diff_id)** – This publicly exposed procedure is typically called by a RIB publication adaptor. It'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 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. For more discussion of the status codes, refer to the Error Handling Guidelines document or the process flow in the following section. These codes come from an EAI team defined 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. The facility id is only included in messages coming from RDM.

# Item Message Family Manager Publishing Design

## Functional Area

Items

## 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

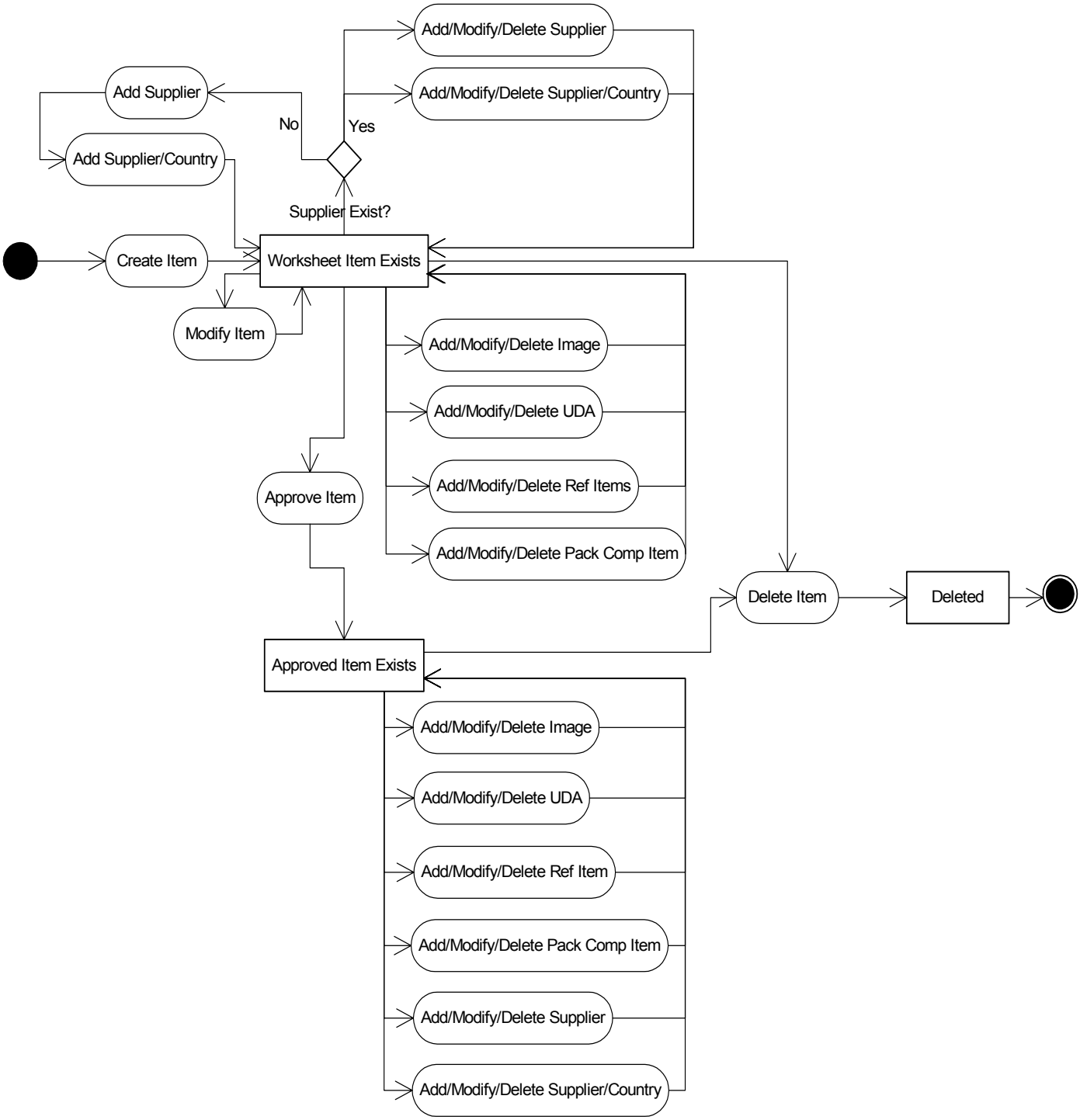
item_image

uda_item_ff

uda_item_lov

uda_item_date

# State Diagram



# Description of Activities

## Create a Worksheet Item

1. Prerequisites: No prerequisites exist for creating an item except that RMS foundation data such as departments and suppliers exist first. Items are created using the RMS online item dialogue.
2. Activity Detail: The creation of the item is the first step of gathering all the hierarchical information needed for publishing the item.
3. Messages: A message for the item creation is placed on the queue for future publishing. This is a flat message that will be collected with the item detail messages to comprise the final hierarchical message. It will not be published until the item is approved. The presence of this message on the queue signals the publishing process that more detail information for the item is forthcoming.

## Approve an Item

1. Prerequisites: An item must exist and be submitted for approval.
2. Activity Detail: The item record is updated in the item_master table. An ItemHdrMod message type record is inserted on the item_mfqueue table. Once the item is approved, it is of interest to other software systems. It can be included in orders, transfers, shipments, etc.
3. Messages: ItemHdrDesc message type is created. It is a flat, synchronous message containing this item record. The approved item create message is a hierarchical message containing the item and all the item family detail records. First an ItemDesc node is created and the ItemHdrDesc message is added to this message. Next, all the child messages are appended to the message until there are no more records in the item_mfqueue table for this item. Then the final item message is formatted.

## Modify an Item

1. Prerequisites: An item can have any status to be modified. Once the item is approved, there are only a few fields that can be modified.
2. Activity Detail: The item record is updated in the item_master table. An ItemHdrMod message type record is inserted on the item_mfqueue table.
3. Messages: ItemHdrDesc message type is created. It is a flat, synchronous message containing this item record. If a record that has an ItemCre message type exists on the item_mfqueue table for this item, this “modify” message is never used in publishing. Only the final “modify” item record message with an ‘A’(Approved) status is published. If no ItemCre record exists on the item_mfqueue table for this item, it is published as a flat message.

## Create Item/Supplier

1. **Prerequisites**: The supplier and the item already exist.
2. **Activity Detail**: The item/supplier combination is inserted into the item_supplier table. An ItemSupCre message type record is also inserted on the item_mfqueue table.
3. **Messages**: ItemSupDesc message type is created. It is a flat, synchronous message containing this item/supplier record. It is published as part of the item creation message when the item is approved, or flat after the original publication of the item creation.

## Modify Item/Supplier

1. **Prerequisites**: The item/supplier combination already exists.
2. **Activity Detail**: The item/supplier record is updated in the item_supplier table. An ItemSupMod message type record is inserted on the item_mfqueue table.
3. **Messages**: ItemSupDesc message type is created. It is a flat, synchronous message containing this item/supplier record. If records that have an ItemCre and an ItemSupCre message type exist on the item_mfqueue table for this item/supplier, the message is updated with the "modify" message and published as part of the item creation message when the item is approved. Otherwise, it is published as a flat message.

## Delete Item/Supplier

1. **Prerequisites**: The item/supplier combination already exists and is not being used somewhere in the system.
2. **Activity Detail**: The item/supplier record is deleted from the item_supplier table and all child records from the item_supp_country and item_supp_country_dim tables. An ItemSupDel message type record is inserted on the item_mfqueue table.
3. **Messages**: ItemSupRef message type is created. It is a flat, synchronous message containing the keys for this item/supplier record. If records that have an ItemCre and an ItemSupCre message type exist on the item_mfqueue table for this item/supplier, the ItemSupCre and any ItemSupMod records are deleted from the item_mfqueue table. Otherwise, it is published as a flat message.

## Create Item/Supplier/Country

1. **Prerequisites**: The supplier, country and the item already exist.
2. **Activity Detail**: The item/supplier/country combination is inserted into the item_supp_country table. An ItemSupCtyCre message type record is also inserted on the item_mfqueue table.
3. **Messages**: ItemSupCtyDesc message type is created. It is a flat, synchronous message containing this item/supplier/country record. It is published as part of the item creation message when the item is approved, or flat after the original publication of the item creation.

## Modify Item/Supplier/Country

1. **Prerequisites**: The item/supplier/country combination already exists.
2. **Activity Detail**: The item/supplier/country record is updated in the item_supp_country table. An ItemSupCtyMod message type record is inserted on the item_mfqueue table.
3. **Messages**: ItemSupCtyDesc message type is created. It is a flat, synchronous message containing this item/supplier/country record. If records that have an ItemCre and an ItemSupCtyCre message type exist on the item_mfqueue table for this item/supplier/country, the message is updated with the "modify" message and published as part of the item creation message when the item is approved. Otherwise, it is published as a flat message.

## Delete Item/Supplier/Country

1. **Prerequisites**: The item/supplier/country combination already exists.
2. **Activity Detail**: The item/supplier/country record is deleted from the item_supp_country table and all child records from the item_supp_country_dim table. An ItemSupCtyDel message type record is inserted on the item_mfqueue table.
3. **Messages**: ItemSupCtyRef message type is created. It is a flat, synchronous message containing the keys for this item/supplier/country record. If records that have an ItemCre and an ItemSupCtyCre message type exist on the item_mfqueue table for this item/supplier/country, the ItemSupCtyCre and any ItemSupCtyMod records are deleted from the item_mfqueue table. Otherwise, it is published as a flat message.

## Create Item/Supplier/Country/Dimension

1. **Prerequisites**: The item/supplier/country already exists.
2. **Activity Detail**: The item/supplier/country/dimension combination is inserted into the item_supp_country_dim table. An ISCDimCre message type record is also inserted on the item_mfqueue table.
3. **Messages**: ISCDimDesc message type is created. It is a flat, synchronous message containing this item/supplier/country/dimension record. It is published as part of the item creation message when the item is approved, or flat after the original publication of the item creation.

## Modify Item/Supplier/Country/Dimension

1. **Prerequisites**: The item/supplier/country/dimension combination already exists.
2. **Activity Detail**: The item/supplier/country/dimension record is updated in the item_supp_country_dim table. An ISCDimMod message type record is inserted on the item_mfqueue table.
3. **Messages**: ISCDimDesc message type is created. It is a flat, synchronous message containing this item/supplier record. If records that have an ItemCre and an ISCDimCre message type exist on the item_mfqueue table for this item/supplier/country/dimension, the message is updated with the "modify" message and published as part of the item creation message when the item is approved. Otherwise, it is published as a flat message.

## Delete Item/Supplier/Country/Dimension

1. **Prerequisites**: The item/supplier/country/dimension combination already exists.
2. **Activity Detail**: The item/supplier/country/dimension record is deleted from the item_supp_country_dim table and all child records from the item_supp_country and item_supp_country_dim tables. An ISCDimDel message type record is inserted on the item_mfqueue table.
3. **Messages**: ISCDimRef message type is created. It is a flat, synchronous message containing the keys for this item/supplier/country/dimension record. If records that have an ItemCre and an ISCDimCre message type exist on the item_mfqueue table for this item/supplier/country/dimension, the ISCDimCre and any ISCDimMod records are deleted from the item_mfqueue table. Otherwise, it is published as a flat message.

## CreateRef_Item

1. **Prerequisites**: The parent item exists.
2. **Activity Detail**: The item is inserted into the item_master table. An ItemUPCCre message type record is also inserted on the item_mfqueue table.
3. **Messages**: ItemUPCDesc message type is created. It is a flat, synchronous message containing this reference item record. It is published as part of the item creation message when the item is approved, or flat after the original publication of the item creation.

## Modify Ref_Item

1. **Prerequisites**: The reference item exists as a child item.
2. **Activity Detail**: The item record is updated in the item_master table. An ItemUPCMod message type record is inserted on the item_mfqueue table.
3. **Messages**: ItemUPCDesc message type is created. It is a flat, synchronous message containing this reference item record. If records that have an ItemCre and an ItemUPCCre message type exist on the item_mfqueue table for this reference item, the message is updated with the "modify" message and published as part of the item creation message when the item is approved. Otherwise, it is published as a flat message.

## Delete Ref_Item

1. **Prerequisites**: The reference item already exists as a child item.
2. **Activity Detail**: The reference item record is deleted from the item_master table. An ItemUPCDel message type record is inserted on the item_mfqueue table.
3. **Messages**: ItemUPCRef message type is created. It is a flat, synchronous message containing the keys for this reference item record. If records that have an ItemCre and an ItemUPCCre message type exist on the item_mfqueue table for this reference item, the ItemUPCCre and any ItemUPCMod records are deleted from the item_mfqueue table. Otherwise, it is published as a flat message.

## Create Pack Comp

1. **Prerequisites**: The pack item exists.
2. **Activity Detail**: The pack comp is inserted into the packitem_breakout table. An ItemBOMCre message type record is also inserted on the item_mfqueue table.
3. **Messages**: ItemBOMDesc message type is created. It is a flat, synchronous message containing this pack comp record. It is published as part of the item creation message when the item is approved, or flat after the original publication of the item creation.

## Modify Pack Comp

1. **Prerequisites**: The pack comp exists for the pack.
2. **Activity Detail**: The pack comp record is updated in the packitem_breakout table. An ItemBOMMod message type record is inserted on the item_mfqueue table.
3. **Messages**: ItemBOMDesc message type is created. It is a flat, synchronous message containing this pack comp record. If records that have an ItemCre and an ItemBOMCre message type exist on the item_mfqueue table for this pack comp, the message is updated with the "modify" message and published as part of the item creation message when the item is approved. Otherwise, it is published as a flat message.

## Delete Pack Comp

1. **Prerequisites**: The pack comp already exists for the pack.
2. **Activity Detail**: The pack comp record is deleted from the packitem_breakout table. The packitem_qty is retrieved from the v_packitem_qty view. If the quantity for the pack comp is 0, an ItemBOMDel message type record is inserted on the item_mfqueue table. If the quantity for the pack comp greater than 0, an ItemBOMMod message type record is inserted on the item_mfqueue table.
3. **Messages**: If the message type is ItemBOMDel, a ItemBOMRef message type is created. It is a flat, synchronous message containing the keys for this pack comp record. If records that have an ItemCre and an ItemBOMCre message type exist on the item_mfqueue table for this pack comp, the ItemBOMCre and any ItemBOMMod records are deleted from the item_mfqueue table. Otherwise, it is published as a flat message. If the message type is ItemBOMMod, a message is create and processed as described in the Modify Pack Comp Messages section.

## Create Item/Image

1. **Prerequisites**: The item already exists.
2. **Activity Detail**: The item/image combination is inserted into the item_image table. An ItemImageCre message type record is also inserted on the item_mfqueue table.
3. **Messages**: ItemImageDesc message type is created. It is a flat, synchronous message containing this item/image record. It is published as part of the item creation message when the item is approved, or flat after the original publication of the item creation.

## Modify Item/Image

1. **Prerequisites**: The item/image combination already exists.
2. **Activity Detail**: The item/image record is updated in the item_image table. An ItemImageMod message type record is inserted on the item_mfqueue table.
3. **Messages**: ItemImageDesc message type is created. It is a flat, synchronous message containing this item/image record. If records that have an ItemCre and an ItemImageCre message type exist on the item_mfqueue table for this item/image, the message is updated with the “modify” message and published as part of the item creation message when the item is approved. Otherwise, it is published as a flat message.

## Delete Item/Image

1. **Prerequisites**: The item/image combination already exists.
2. **Activity Detail**: The item/image record is deleted from the item_image table. An ItemImageDel message type record is inserted on the item_mfqueue table.
3. **Messages**: ItemImageRef message type is created. It is a flat, synchronous message containing the keys for this item/image record. If records that have an ItemCre and an ItemImageCre message type exist on the item_mfqueue table for this item/image, the ItemImageCre and any ItemImageMod records are deleted from the item_mfqueue table. Otherwise, it is published as a flat message.

## Create Item/UDA/FreeFormat

1. **Prerequisites**: The item already exists.
2. **Activity Detail**: The item/uda/freeformat combination is inserted into the uda_item_ff table. An ItemUDAFFCre message type record is also inserted on the item_mfqueue table.
3. **Messages**: ItemUDAFFDesc message type is created. It is a flat, synchronous message containing this item/uda/freeformat record. It is published as part of the item creation message when the item is approved, or flat after the original publication of the item creation.

## Modify Item/UDA/FreeFormat

1. **Prerequisites**: The item/uda/freeformat combination already exists.
2. **Activity Detail**: The item/uda/freeformat record is updated in the uda_item_ff table. An ItemUDAFFMod message type record is inserted on the item_mfqueue table.
3. **Messages**: ItemUDAFFDesc message type is created. It is a flat, synchronous message containing this item/uda/freeformat record. If records that have an ItemCre and an ItemUDAFFCre message type exist on the item_mfqueue table for this item/uda/freeformat, the message is updated with the “modify” message and published as part of the item creation message when the item is approved. Otherwise, it is published as a flat message.

## Delete Item/UDA/FreeFormat

1. **Prerequisites**: The item/uda/freeformat combination already exists.
2. **Activity Detail**: The item/uda/freeformat record is deleted from the uda_item_ff table. An ItemUDAFFDel message type record is inserted on the item_mfqueue table.
3. **Messages**: ItemUDAFFRef message type is created. It is a flat, synchronous message containing the keys for this item/uda/freeformat record. If records that have an ItemCre and an ItemUDAFFCre message type exist on the item_mfqueue table for this item/uda/freeformat, the ItemUDAFFCre and any ItemUDAFFMod records are deleted from the item_mfqueue table. Otherwise, it is published as a flat message.

## Create Item/UDA/LOV

1. **Prerequisites**: The item already exists.
2. **Activity Detail**: The item/uda/lov combination is inserted into the uda_item_lov table. An ItemUDALOVCre message type record is also inserted on the item_mfqueue table.
3. **Messages**: ItemUDALOVDesc message type is created. It is a flat, synchronous message containing this item/uda/lov record. It is published as part of the item creation message when the item is approved, or flat after the original publication of the item creation.

## Modify Item/UDA/LOV

1. **Prerequisites**: The item/uda/lov combination already exists.
2. **Activity Detail**: The item/uda/lov record is updated in the uda_item_lov table. An ItemUDALOVMod message type record is inserted on the item_mfqueue table.
3. **Messages**: ItemUDALOVDesc message type is created. It is a flat, synchronous message containing this item/uda/lov record. If records that have an ItemCre and an ItemUDALOVCre message type exist on the item_mfqueue table for this item/uda/lov, the message is updated with the "modify" message and published as part of the item creation message when the item is approved. Otherwise, it is published as a flat message.

## Delete Item/UDA/LOV

1. **Prerequisites**: The item/uda/lov combination already exists.
2. **Activity Detail**: The item/uda/lov record is deleted from the uda_item_lov table. An ItemUDALOVDel message type record is inserted on the item_mfqueue table.
3. **Messages**: ItemUDALOVRef message type is created. It is a flat, synchronous message containing the keys for this item/uda/lov record. If records that have an ItemCre and an ItemUDALOVCre message type exist on the item_mfqueue table for this item/uda/lov, the ItemUDALOVCre and any ItemUDALOVMod records are deleted from the item_mfqueue table. Otherwise, it is published as a flat message.

## Create Item/UDA/Date

1. **Prerequisites**: The item already exists.
2. **Activity Detail**: The item/uda/date combination is inserted into the uda_item_date table. An ItemUDADateCre message type record is also inserted on the item_mfqueue table.
3. **Messages**: ItemUDADateDesc message type is created. It is a flat, synchronous message containing this item/uda/date record. It is published as part of the item creation message when the item is approved, or flat after the original publication of the item creation.

## Modify Item/UDA/Date

1. **Prerequisites**: The item/uda/date combination already exists.
2. **Activity Detail**: The item/uda/date record is updated in the uda_item_date table. An ItemUDADateMod message type record is inserted on the item_mfqueue table.
3. **Messages**: ItemUDADateDesc message type is created. It is a flat, synchronous message containing this item/uda/date record. If records that have an ItemCre and an ItemUDADateCre message type exist on the item_mfqueue table for this item/uda/lov, the message is updated with the "modify" message and published as part of the item creation message when the item is approved. Otherwise, it is published as a flat message.

## Delete Item/UDA/Date

1. **Prerequisites**: The item/uda/date combination already exists.
2. **Activity Detail**: The item/uda/date record is deleted from the uda_item_lov table. An ItemUDADateDel message type record is inserted on the item_mfqueue table.
3. **Messages**: ItemUDADateRef message type is created. It is a flat, synchronous message containing the keys for this item/uda/date record. If records that have an ItemCre and an ItemUDADateCre message type exist on the item_mfqueue table for this item/uda/date, the ItemUDADateCre and any ItemUDADateMod records are deleted from the item_mfqueue table. Otherwise, it is published as a flat message.

## Delete an Item

1. **Prerequisites**: The item exists. An 'A'(Approved) item can be deleted when the user presses the "Cancel" button in the RMS dialogue after creating and approving the item.
2. **Activity Detail**: The item record is deleted from the item_master table and any child records that exist are deleted from the child tables. An ItemDel message type record is inserted on the item_mfqueue table.
3. **Message**: ItemRef message type is created. It is a flat, synchronous message containing the key for this item record. If a record that has an ItemCre message type exists on the item_mfqueue table for this item, all records for this item are deleted from the item_mfqueue table. Otherwise, it is published as a flat message.

## Triggers

Trigger Description (EC_TABLE_IEM_AIUDR): This trigger fires on any insert, update or delete on the item_master table. It captures the data in the "new" bind variables for inserts and updates. It captures the "old" data on deletes. It sets the action type and message type and calls the ITEM_XML.BUILD_MESSAGE procedure to build the message. The record is inserted into the ITEM_MFQUEUE table by calling the RMSMFM_ITEMS.ADDTOQ procedure.

On Insert:

For transaction level items, sets action_type to 'A'dd and message_type to 'ItemHdrCre'.

For reference level items (below the transaction level), sets action_type to 'A'dd and message_type to 'ItemUPCCre'.

Parent and grandparent items are not published.

On Update:

For transaction level items, sets action_type to 'M'odify and message_type to 'ItemHdrMod'.

For reference level items (below the transaction level), sets action_type to 'M'odify and message_type to 'ItemUPCMod'.

On Delete:

For transaction level items, sends only the item column value for the message.

For reference level items (below the transaction level), sends only the item and item_parent column values for the message.

For transaction level items, sets action_type to 'D'elete and message_type to 'ItemHdrDel'.

For reference level items (below the transaction level), sets action_type to 'D'elete and message_type to 'ItemUPCDel'.

**ITEM_XML.BUILD_MESSAGE(O_status, O_text, O_message, I_record, I_action_type)–** This function is called by the trigger EC_TABLE_ITEM_AIUDR on insert, update and delete of the item_master table. This function gathers all the data necessary to build the message that needs to be sent to the Retek Integration Bus. It determines the proper message to build based on the action_type and the item type (transactional or reference) that is set in the trigger. For transaction level items, it builds ItemRef xml messages for delete statements, or ItemDesc xml messages for updates or inserts. For reference items, it builds ItemUPCRef xml messages for delete statements, or ItemUPCDesc xml messages for updates or inserts.

Trigger Description (EC_TABLE_ISP_AIUDR): This trigger fires on any insert, update or delete on the item_supplier table. It captures the data in the "new" bind variables for inserts and updates. It captures the "old" data on deletes. It sets the action type and message type and calls the ITEMSUPPLIER_XML.BUILD_MESSAGE procedure to build the message. The record is inserted into the ITEM_MFQUEUE table by calling the RMSMFM_ITEMS.ADDTOQ procedure.

On Insert:

Sets action_type to 'A'dd and message_type to 'ItemSupCre'.

On Update:

Sets action_type to 'M'odify and message_type to 'ItemSupMod'.

On Delete:

Sends only the item and supplier column values for the message.

Sets action_type to 'D'elete and message_type to 'ItemSupDel'.

**ITEMSUPPLIER_XML.BUILD_MESSAGE(O_status, O_text, O_message, I_record, I_action_type)–** This function is called by the trigger EC_TABLE_ISP_AIUDR on insert, update and delete of the item_supplier table. This function gathers all the data necessary to build the message that needs to be sent to the Retek Integration Bus. It determines the proper message to build based on the action_type that is set in the trigger. It builds ItemSupRef xml messages for delete statements, or ItemSupDesc xml messages for updates or inserts.

Trigger Description (EC_TABLE_ISC_AIUDR): This trigger fires on any insert, update or delete on the item_supp_country table. It captures the data in the "new" bind variables for inserts and updates. It captures the "old" data on deletes. It sets the action type and message type and calls the ISC_XML.BUILD_MESSAGE procedure to build the message. The record is inserted into the ITEM_MFQUEUE table by calling the RMSMFM_ITEMS.ADDTOQ procedure.

On Insert:

Sets action_type to 'A'dd and message_type to 'ItemSupCtyCre'.

On Update:

Sets action_type to 'M'odify and message_type to 'ItemSupCtyMod'.

On Delete:

Sends only the item, supplier and origin_country_id column values for the message.

Sets action_type to 'D'elete and message_type to 'ItemSupCtyDel'.

**ISC_XML.BUILD_MESSAGE(O_status, O_text, O_message, I_record, I_action_type)–** This function is called by the trigger EC_TABLE_ISC_AIUDR on insert, update and delete of the item_supp_country table. This function gathers all the data necessary to build the message that needs to be sent to the Retek Integration Bus. It determines the proper message to build based on the action_type that is set in the trigger. It builds ItemSupCtyRef xml messages for delete statements, or ItemSupCtyDesc xml messages for updates or inserts.

Trigger Description (EC_TABLE_ISD_AIUDR): This trigger fires on any insert, update or delete on the item_supp_country_dim table. It captures the data in the "new" bind variables for inserts and updates. It captures the "old" data on deletes. It sets the action type and message type and calls the ISCD_XML.BUILD_MESSAGE procedure to build the message. The record is inserted into the ITEM_MFQUEUE table by calling the RMSMFM_ITEMS.ADDTOQ procedure.

On Insert:

Sets action_type to 'A'dd and message_type to 'ISCDimCre'.

On Update:

Sets action_type to 'M'odify and message_type to 'ISCDimMod'.

On Delete:

Sends only the item, supplier, origin_country_id and dim_object column values for the message.

Sets action_type to 'D'elete and message_type to 'ISCDimDel'.

**ISC_XML.BUILD_MESSAGE(O_status, O_text, O_message, I_record, I_action_type)–** This function is called by the trigger EC_TABLE_ISC_AIUDR on insert, update and delete of the item_supp_country_dim table. This function gathers all the data necessary to build the message that needs to be sent to the Retek Integration Bus. It determines the proper message to build based on the action_type that is set in the trigger. It builds ISCDimRef xml messages for delete statements, or ISCDimDesc xml messages for updates or inserts.

Trigger Description (EC_TABLE_PKS_AIUDR: This trigger fires on any insert, update or delete on the packitem_breakout table. It captures the data in the "new" bind variables for inserts and updates. It captures the "old" data on deletes. It populates a PL/SQL table with this data.

Trigger Description (EC_TABLE_PKS_IUDS: This trigger fires on any insert, update or delete on the packitem_breakout table. It loops through the PL/SQL table that was populated in the row trigger and determines the value for the packitem quantity in the message based on what is retrieved from the v_packsku_qty view and the DML event. It calls the ITEMBOM_XML.BUILD_MESSAGE procedure to build the message. The record is inserted into the ITEM_MFQUEUE table by calling the RMSMFM_ITEMS.ADDTOQ procedure.

On Insert:

If the v_packsku_qty quantity is equal to the record just added, it sets action_type to 'A'dd and message_type to 'ItemBOMCre'. If not, it sets action_type to 'M'odify and message_type to 'ItemBOMMod'.

On Update:

Sets action_type to 'M'odify and message_type to 'ItemBOMMod'.

On Delete:

Sends only the pack_no and item column values for the message.

If the packitem quantity is 0, it sets action_type to 'D'elete and message_type to 'ItemBOMDel'.

If the absolute value of packitem quantity is greater than 0, it sets action_type to 'M'odify and message_type to 'ItemBOMMod'.

**ITEMBOM_XML.BUILD_MESSAGE(O_status, O_text, O_message, I_record, I_action_type)–** This function is called by the trigger EC_TABLE_PKS_IUDS on insert, update and delete of the item_supplier table. This function gathers all the data necessary to build the message that needs to be sent to the Retek Integration Bus. It determines the proper message to build based on the action_type that is set in the trigger. It builds ItemBOMRef xml messages for delete statements, or ItemBOMDesc xml messages for updates or inserts.

Trigger Description (EC_TABLE_IIM_AIUDR): This trigger fires on any insert, update or delete on the item_image table. It captures the data in the "new" bind variables for inserts and updates. It captures the "old" data on deletes. It sets the action type and message type and calls the ITEMIMAGE_XML.BUILD_MESSAGE procedure to build the message. The record is inserted into the ITEM_MFQUEUE table by calling the RMSMFM_ITEMS.ADDTOQ procedure.

On Insert:

Sets action_type to 'A'dd and message_type to 'ItemImageCre'.

On Update:

Sets action_type to 'M'odify and message_type to 'ItemImageMod'.

On Delete:

Sends only the item and image_name column values for the message.

Sets action_type to 'D'elete and message_type to 'ItemImageDel'.

**ITEMIMAGE_XML.BUILD_MESSAGE(O_status, O_text, O_message, I_record, I_action_type)–** This function is called by the trigger EC_TABLE_IIM_AIUDR on insert, update and delete of the item_image table. This function gathers all the data necessary to build the message that needs to be sent to the Retek Integration Bus. It determines the proper message to build based on the action_type that is set in the trigger. It builds ItemImageRef xml messages for delete statements, or ItemImageDesc xml messages for updates or inserts.

Trigger Description (EC_TABLE_UIF_AIUDR): This trigger fires on any insert, update or delete on the uda_item_ff table. It captures the data in the "new" bind variables for inserts and updates. It captures the "old" data on deletes. It sets the action type and message type and calls the UDA_ITEM_XML.BUILD_MESSAGE procedure to build the message. The record is inserted into the ITEM_MFQUEUE table by calling the RMSMFM_ITEMS.ADDTOQ procedure.

On Insert:

Sets action_type to 'A'dd and message_type to 'ItemUDAFFCre'.

On Update:

Sets action_type to 'M'odify and message_type to 'ItemUDAFFMod'.

On Delete:

Sends only the item and uda_id column values for the message.

Sets action_type to 'D'elete and message_type to 'ItemUDAFFDel'.

**UDA_ITEM_XML.BUILD_MESSAGE(O_status, O_text, O_message, I_record, I_action_type)–** This function is called by the trigger EC_TABLE_UIF_AIUDR on insert, update and delete of the item_supplier table. This function gathers all the data necessary to build the message that needs to be sent to the Retek Integration Bus. It determines the proper message to build based on the action_type that is set in the trigger. It builds ItemUDAFFRef xml messages for delete statements, or ItemUDAFFDesc xml messages for updates or inserts.

Trigger Description (EC_TABLE_UIL_AIUDR): This trigger fires on any insert, update or delete on the uda_item_lov table. It captures the data in the "new" bind variables for inserts and updates. It captures the "old" data on deletes. It sets the action type and message type and calls the UDA_ITEM_XML.BUILD_MESSAGE procedure to build the message. The record is inserted into the ITEM_MFQUEUE table by calling the RMSMFM_ITEMS.ADDTOQ procedure.

On Insert:

Sets action_type to 'A'dd and message_type to 'ItemUDALOVCre'.

On Update:

Sets action_type to 'M'odify and message_type to 'ItemUDALOVMod'.

On Delete:

Sends only the item, uda_id and uda_value column values for the message.

Sets action_type to 'D'elete and message_type to 'ItemUDALOVDel'.

**UDA_ITEM_XML.BUILD_MESSAGE(O_status, O_text, O_message, I_record, I_action_type)–** This function is called by the trigger EC_TABLE_UIL_AIUDR on insert, update and delete of the item_supplier table. This function gathers all the data necessary to build the message that needs to be sent to the Retek Integration Bus. It determines the proper message to build based on the action_type that is set in the trigger. It builds ItemUDALOVRef xml messages for delete statements, or ItemUDALOVDesc xml messages for updates or inserts.

Trigger Description (EC_TABLE_UID_AIUDR): This trigger fires on any insert, update or delete on the uda_item_date table. It captures the data in the "new" bind variables for inserts and updates. It captures the "old" data on deletes. It sets the action type and message type and calls the UDA_ITEM_XML.BUILD_MESSAGE procedure to build the message. The record is inserted into the ITEM_MFQUEUE table by calling the RMSMFM_ITEMS.ADDTOQ procedure.

On Insert:

Sets action_type to 'A'dd and message_type to 'ItemUDADateCre'.

On Update:

Sets action_type to 'M'odify and message_type to 'ItemUDADateMod'.

On Delete:

Sends only the item and uda_id column values for the message.

Sets action_type to 'D'elete and message_type to 'ItemUDADateDel'.

**UDA_ITEM_XML.BUILD_MESSAGE(O_status, O_text, O_message, I_record, I_action_type)–** This function is called by the trigger EC_TABLE_UID_AIUDR on insert, update and delete of the item_supplier table. This function gathers all the data necessary to build the message that needs to be sent to the Retek Integration Bus. It determines the proper message to build based on the action_type that is set in the trigger. It builds ItemUDADateRef xml messages for delete statements, or ItemUDADateDesc xml messages for updates or inserts.

## Message Family Manager Procedures

Public Procedures:

**ADDTOQ(O_status, O_text, I_queue_rec, I_message)** – This procedure is called by a DML event capture trigger, and takes the message type, family key values and, for synchronously captured messages, the message itself.

First it checks the input parameter for the item status that is part of the I_queue_rec input record. This input record is defined the package specification. If the item status is 'A'(approved), it sets a local variable to 'Y'(yes) and uses this local variable as the value for the approve_ind column in the insert statement. It inserts a record into the item_mfqueue table using the sequence for the table, the values from the input record parameter, the local variable for the approve_ind and the input CLOB parameter that contains the data in XML format.

**GETNXT (O_status_code, O_error_msg, O_message_type, O_message, O_item, O_supplier, O_country_id, O_dim_object, O_upc, O_bom_comp, O_image_name, O_uda_id, O_uda_value, O_sellable_ind)**– 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 procedure produces a message through the following steps:

It loops through the item_mfqueue table records that have a pub_status of 'U'(Unpublished).

If the return from the CREATE_PREVIOUS function is TRUE

- calls the CLEAN_QUEUE procedure.
- if the approve_ind column equals 'Y'(Yes)
  - calls the MAKE_CREATE procedure
  - assigns all the output parameters with the values from the current item_mfqueue row except for O_message which is returned from the MAKE_CREATE procedure and sets O_status to API_CODES.SUCCESS.
- if the CAN_CREATE returns FALSE, sets the pub_status field of the current item_mfqueue row to 'N' and updates the row.

If the return from the CREATE_PREVIOUS function is FALSE

- assigns all the output parameters with the values from the current item_mfqueue row and set O_status to API_CODES.SUCCESS.
- call the DELETE_QUEUE_REC to delete the row from the item_mfqueue table.

If no "publishable" messages are retrieved from the above steps the procedure returns a status of 'N'(No message).

Status code is one of five values, as shown in the following table. For more discussion of the status codes, refer to the Error Handling Guidelines document or the process flow in the following section. These codes come from an EAI team defined RIB_CODES package.

Private Procedures:

These private procedures are only necessary when the initial create message is hierarchical. If all messages in the family are flat, there is no need for these procedures.

**CREATE_PREVIOUS(O_status, O_text, I_queue_rec)** – This function determines if a header level create already exists on the queue table for the same key value and with a sequence number less than the current records sequence number.

It checks the item_mfqueue table for the existence of a row for that has an item equal to the passed in value for item, a message type equal to the value of ItemCre and a seq_no that is less than or equal to the passed in value for seq_no. If such a row exists in the table, it returns TRUE.

**CLEAN_QUEUE(O_status, O_text, I_queue_rec)** – This procedure cleans up the queue by eliminating modification messages. It is only called if CREATE_PREVIOUS returns true. For each modification message type, it finds the previous corresponding create message type. It then calls REPLACE_QUEUE 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 messages. It then calls DELETE_QUEUE_REC to delete the create message record.

The following examples illustrate the flow of the logic in this procedure for the item family:

First it checks the message_type passed to procedure for the value of any of the item delete message types, i.e. ItemSupCtyDel, ItemUPCDel, etc. If the message_type is a "delete" message, it deletes records from the item_mfqueue table for the appropriate key values and for the seq_no less than or equal to the passed in value for seq_no.

Example for the message type ItemSupCtyDel:

```
delete from item_mfqueue
 where supplier = I_queue_rec.supplier
   and item     = I_queue_rec.item
   and country  = I_queue_rec.country
   and seq_no  <= I_queue_rec.seq_no;
```

If the message_type is an "update" message such as ItemSupMod, it assigns the corresponding "Add" message_type to a local variable.

Example for the message type ISCDimMod:

```
L_create_type := ISCDimAdd;
```

If this local variable is not null and if the call to REPLACE_QUEUE returns TRUE, it calls DELETE_QUEUE_REC to delete the row from item_mfqueue.

**REPLACE_QUEUE(O_status, O_text, I_rec, I_message_type)** – This procedure replaces the message in the "create" message type record with the message from a "Modify" message type record.

It locks the item_mfqueue table for all rows that have a seq_no less than the passed in value for seq_no. It updates the message column with the passed in value for message for the row that matches the key values passed in the record to the function and that matches the message_type passed as a parameter. It uses the nvl function for all key columns except item because these key values are optional and dependent on the message_type.

**DELETE_QUEUE_REC(O_status, O_text, I_seq_no)** – This procedure deletes from the item_mfqueue table the row that has the seq_no column value equal to the sequence value passed to the procedure.

**MAKE_CREATE(O_status, O_text, O_msg, I_queue_rec)** – This procedure 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 for that item with a sequence number less than or equal to the current records sequence number. This new message is passed back to the bus.

For the Item this procedure is implemented as follows:

Two cursors are used. One cursor cursor retrieves the row from the item_mfqueue table for the item_master message, the item is equal to the value of the passed in item, the seq_no is less than the passed in seq_no and the message_type is equal to 'ItemCre'. The other cursor retrieves all the item related messages for the item details, the item is equal to the value of the passed in item, the seq_no is less than or equal to the sequence value passed to the procedure and the message_type is equal message_type value passed to the cursor. Order the second cursor by seq_no. A local procedure with parameters for message_type and message_name, adds the detail message to the header message. It loops through the second cursor with the value of the message type parameter and do the following:

- Assigns a local variable to the return of the rib_xml.readRoot function. Pass the value of the message and message name to this function.
- Assigns a local variable to the return of the rib_xml.addElement function. Pass the variable that contains the header XML message and the message name.
- Calls the rib_xml.addElementContents procedure and use these variables as the parameters.

The procedure starts by fetching the header message from the queue in the first cursor. It calls API_LIBRARY.CREATE_MESSAGE to create a root message. Next, it creates the item header root by assigning to the header root variable the value from the rib_xml.readRoot function. Next, it creates the item header element by assigning to the header element variable the value from the rib_xml.addElement function. Finally, it adds the message to the root by calling the rib_xml.addElementContents procedure and passing the header root and the header element variables as parameters.

It adds all the item related detail messages by calling the local procedure described above for each item detail and passing the message type and the message name unique to the item detail. It uses the constants define in the package spec for these values. The order for adding an item detail to the XML message is specified in the item DTD.

Finally, it calls the API_LIBRARY.WRITE_DOCUMENT function to format the XML document and deletes all rows that comprise this message from the item_mfqueue table.

# Ordering Message Family Manager Publishing Design

## Functional Area

Orders

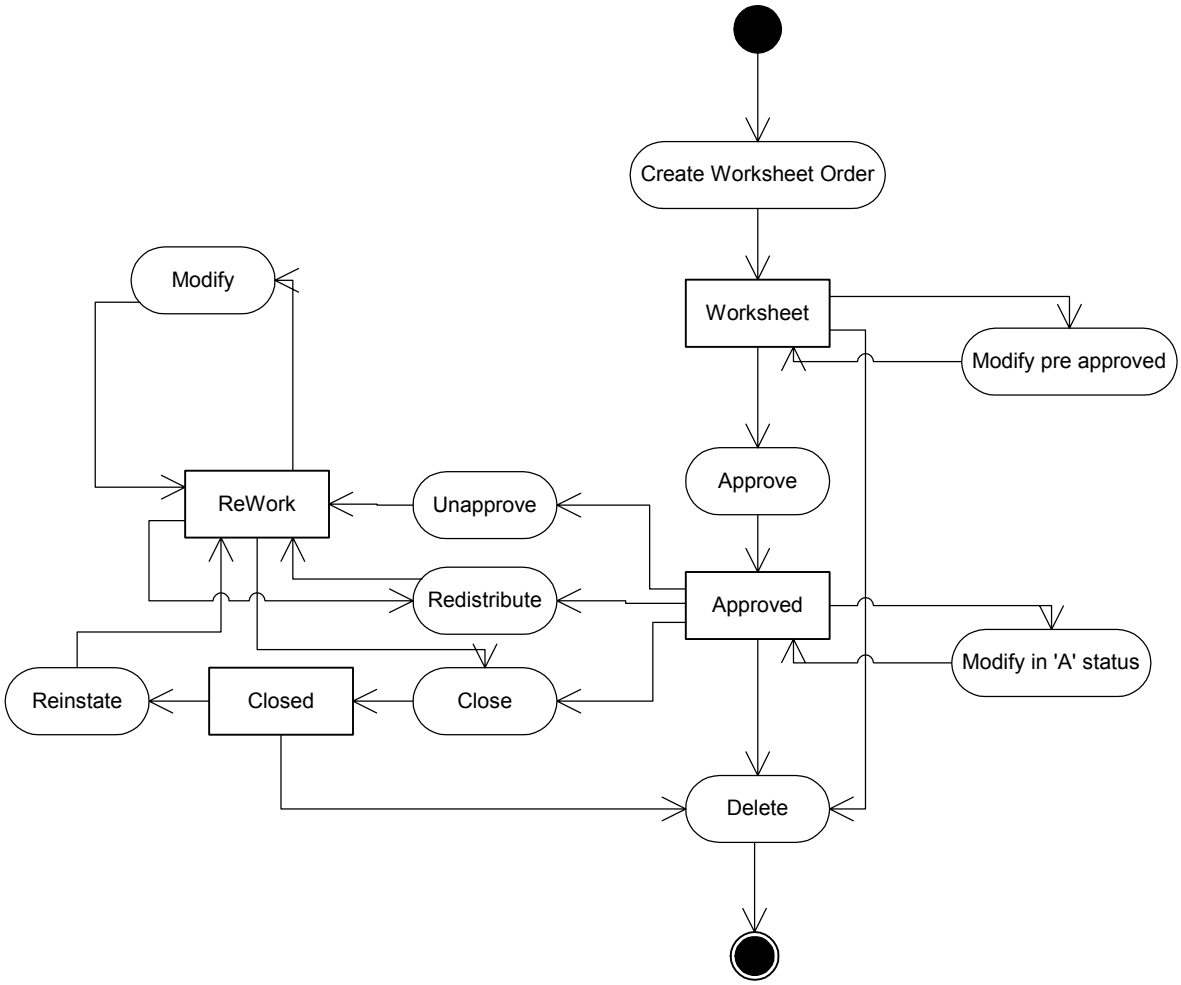
## Design Overview

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, etc. ORDSKU is a child of ORDHEAD and contains the item(s) that are ordered, the size of the pack being ordered, etc. 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 message types will create five different document types when published. Each type is discussed below.

Five different documents types (i.e. published messages) will be published for orders. Two for the header level, two for the detail level, and one that is a combination of header and detail create messages. The initial published message, 'PODesc', will only be published once for each order when an order is initially approved. This message will be a summation of the 'POHdrDesc' message and all of the subsequent 'PODtlDesc' messages. The 'POHdrDesc' message will be created when an insertion or modification to the ORDHEAD table is made. The 'PORef' message will be created when an order is deleted from the ORDHEAD table. 'PODtlDesc' message will be created when a record is inserted or modified on the ORDLOC table. 'PODtlRef' will be created when an ORDLOC record is deleted.

To facilitate publishing order information at the physical location level, two additional ORDLOC triggers, EC_TABLE_OLO2_AIUDR, EC_TABLE_OLO_AIUDS can be installed. In addition, a new family manager, RMSMFM_ORDERPHYS will then control order publishing. See the 'Ordering Physical MFM design.doc' for a more detailed discussion.

# State Diagram



# Description of Activities

## 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 'POHdrdesc' is written to the ordering queue table. Upon detail additions, each will have a 'PODtlDesc' 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 'POHdrDesc' or 'PODtlDesc' 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 'PODesc' 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 'POHdrDesc' 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. This is a flat, synchronous message.

## 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:** 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 'PODtlRef' 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 flat, synchronous message will be published. For each redistributed item, a 'PODtlDesc' 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 'POHdrDesc' 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. This is a flat, synchronous message.

## 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 'POHdrDesc' message will be created for order at the end of the session if the header was modified. A 'PODtlDesc' message will be created for each detail record modified or added.

## 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 un-received quantities will be canceled out. No details will be modifiable while the order is in this status.
3. **Message**: A 'POHdrDesc' message will be created for order at the end of the session the order was modified. A 'PODtlDesc' 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. Each is a flat, synchronous message.

## 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 'POHdrDesc' message will be created for order at the end of the session the order was modified. A 'PODtlDesc' 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. Each is a flat, synchronous message.

## 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 'PORef' and 'PODtlRef' messages created will remove all the previous messages on the ordering queue table. If the order has been approved, then a 'PODtlRef' message will be created for each detail record and a 'PORef' message for the header. Each is a flat, synchronous message.

## Triggers

Trigger Description (EC_TABLE_OHE_AIUDR): This triggers 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 passes the column information into ORDER_XML.BUILD_HEAD_MSG to create the xml message, then calls RMSMFM_ORDER.ADDTOQ to place the message and order onto the ORDER_MFQUEUE table.

On Insert: A 'POCre' message type is created, with all of the non-base table attributes retrieved from ORDER_XML.GET_MSG_HEADER.

On Update: A 'POHdrMod' message type is created. If this is the first time the order has been approved, the approve indicator is set to 'Y'es to signify that the order should be published in it's entirety. In all other instances, the approve indicator will be set to 'N'o. Then all of the non-base table attributes will be retrieved from ORDER_XML.GET_MSG_HEADER.

On Delete: A 'PODel' message type is created with only the order number passed for the xml message.

Trigger Description (EC_TABLE_OLO_AIUDR): This triggers fires when an ordloc record has been inserted, updated or deleted on the qty_ordered column. Each action is detailed below. In general, this trigger passes the column information into ORDER_XML.BUILD_LOC_MSG to create the xml message, then calls RMSMFM_ORDER.ADDTOQ to place the message and order, item, location onto the ORDER_MFQUEUE table.

On Insert and Update: A 'PODtlCre' or 'PODtlMod' message type is created. All of the non-base table attributes are then retrieved from ORDSKU and ITEM_SUPP_COUNTRY for the complete message body.

On Delete: A 'PODtlDel' message type is created with only the order number, item, location, location type passed for the xml message.

## Message Family Manager Procedures

Public Procedures:

**ADDTOQ(O_status_code, O_error_msg, I_pub_status, I_approve_ind, I_message_type, I_order_no, I_item, I_location, I_message)** – 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 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.

**GETNXT (O_status_code, O_error_msg, O_message_type, O_message, O_order_no, O_item, O_location)** – This publicly exposed procedure is typically called by a RIB publication adaptor. It'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 family key(s) (order_no for ORDHEAD table and order_no, item, 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. Status code is one of five values, as defined in the API_CODES package specification. These codes come from an EAI team defined RIB_CODES package.

This program loops through each message on the ORDER_MFQUEUE table. When no messages are found, the program exists returning the 'N'o message found API code. If the approve indicator is 'Y', then BUILD_CREATE_MSG is called to group the 'POHdrDesc' and all 'PODtlDesc' messages for the order into the 'PODesc' message. If the approve indicator is 'N', then CREATE_PREVIOUS is called to determine if the 'POCre' message type has been published. If the 'POCre' message type has been published, then the current message is returned and deleted from the queue in a call to DELETE_QUEUE_REC. If the 'POCre' message type has not been published, then CLEAN_QUEUE is called.

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.

Private Procedures:

These private procedures are only necessary when the initial create message is hierarchical. If all messages in the family are flat, there is no need for these procedures.

**CREATE_PREVIOUS(I_queue_rec)** – This procedure determines if a header level create already exists on the queue table for the same key value and with a sequence number less than the current records sequence number.

**CLEAN_QUEUE(I_queue_rec)** – This procedure cleans up the queue by eliminating modification messages. It is only called if CREATE_PREVIOUS returns true (meaning the create message for the order has not been published). For each delete message type, it finds the previous corresponding create message type. It then deletes the create message record and the delete message itself. For each modification message type, it calls REPLACE_QUEUE to copy the modify message into the create message it is replacing and calls DELETE_QUEUE_REC to delete the modify record. For create message types, it updates the pub status to 'N' to indicate that the record has been processed and is ready for publishing when the order is approved.

**BUILD_CREATE_MSG(O_msg, I_queue_rec)** – This procedure 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 'PODesc' document type. It then gets the header create message, 'POHdrDesc' and adds it to the new create message. The remainder of this procedure gets each of the details grouped by their document type, 'PODtlDesc' and removes the 'POHdrMod' node from the detail message. Then the 'POHdrDesc' messages is added to the new create 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.

**DELETE_QUEUE_REC(I_seq_no)** – This procedure deletes a specific record from the queue. It deletes based on the sequence number passed in.

**REPLACE_QUEUE( I_rec, I_data)** – This procedure replaces the message in the create header or detail record with the message from the modify header or detail record.

OHdrDesc.xls, MAP_PODtlDesc.xls, MAP_PORef.xls, MAP_PODtlRef.xls.

# Ordering Physical Message Family Manager Publishing Design

## Functional Area

ORDERING, PHYSICAL

## Design Overview

In order to accommodate interfacing warehouse management systems and other products subscribing to order publishing, ordering information will be rolled up and stored at a physical level. In general only subscribing to the virtual or physical level is necessary. In a non-multi channel environment, the ordering and ordering physical family managers are the same. In a multi channel environment, if a subscriber requires information at the physical level, then they will need to use this family manager and the EC_TABLE_OLO2_AIUDR.TRG, EC_TABLE_OLO_AIUDS.TRG table triggers. If the subscriber requires the tsf_po_link_no as part of the ordering information, then they will be required to use the ordering family manager and roll up the total quantities within their subscribing system.

Document types vary slightly with the ordering DTD's. They are as follows; 'POPhyDesc' for the initial published message, 'POHdrDesc' for the header message, 'POPhyDtlDesc' for the detail level message, 'PORef" for the header delete message, "POPhyDtlRef" for the detail delete message.

## State Diagram

See 'Ordering MFM design.doc' for this section.

## Description of Activities

See 'Ordering MFM design.doc' for this section.

## Triggers

Triggers should only insert records onto the staging table if no record already exists or if a record does exist but is locked.

Trigger Description (EC_TABLE_OHE2_AIUDR): This triggers 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 passes the column information into ORDER_XML.BUILD_HEAD_MSG to create the xml message, then calls RMSMFM_ORDERPHYS.ADDTOQ to place the message and order onto the ORDERPHYS_MFQUEUE table.

On Insert: A 'POCre' message type is created, with all of the non-base table attributes retrieved from ORDER_XML.GET_MSG_HEADER.

On Update: A 'POHdrMod' message type is created. If this is the first time the order has been approved, the approve indicator is set to 'Y'es to signify that the order should be published in it's entirety. In all other instances, the approve indicator will be set to 'N'o. Then all of the non-base table attributes will be retrieved from ORDER_XML.GET_MSG_HEADER.

On Delete: A 'PODel' message type is created with only the order number passed for the xml message.

Trigger Description (EC_TABLE_OLO2_AIUDR): This triggers fires when an ordloc record has been inserted, updated or deleted on the qty_ordered column. Each action is detailed below. This trigger is used to populate a PL/SQL binary table. This table is eventually used to publish ordloc information at a physical warehouse/store level.

On Insert or Update: The new order number, item, location, location type, quantity ordered, and unit cost are inserted into the PL/SQL table. If the location type is a warehouse, then the physical warehouse is retrieved from the WH table.

On Delete: The old order number, item, location, location type, quantity ordered, and unit cost are inserted into the PL/SQL table. If the location type is a warehouse, then the physical warehouse is retrieved from the WH table.

Trigger Description (EC_TABLE_OLO_AIUDS): This triggers fires after the statement of an inserted, updated or deleted record or group of records. Each action is detailed below. The purpose of this trigger is to create messages at the physical level. This trigger loops the records on the PL/SQL table generated in EC_TABLE_OLO2_AIUDR. The order, item and location are retrieved and used to retrieve the total quantity ordered and total virtual locations, from V_ORDLOC_STORES_PHYS_WH, for the physical location. If no virtual locations exist, then either the last virtual location for a physical was deleted or all of the virtual locations for a physical were deleted in the same statement. To avoid publishing numerous delete messages for the same physical location, a function ALREADY_DELETED loops through the PL/SQL table to determine if the physical location has already been processed for a delete. If the record hasn't been processed for a delete or records were found on V_ORDLOC_STORES_PHYS_WH for the physical location, then the record is processed. .In general, this trigger passes the column information into ORDER_XML.BUILD_PHYS_LOC_MSG to create the xml message, then calls RMSMFM_ ORDERPHYS.ADDTOQ to place the message and order, item, location onto the ORDERPHYS _MFQUEUE table.

On Insert: If this is the first record (i.e. counter = 1) on ordloc for this item/phys location, then a 'PODtlCre' message type is created. If this is not the first record (i.e. counter > 1), then a 'PODtlMod' message type is created. All of the non-base table attributes are then retrieved from ORDSKU and ITEM_SUPP_COUNTRY for the complete message body.

On Update: A 'PODtlMod' message type is created. All of the non-base table attributes are then retrieved from ORDSKU and ITEM_SUPP_COUNTRY for the complete message body.

On Delete: If no records exist on the ORDLOC view, then a 'PODtlDel' message type is created with only the order number, item, location, location type passed for the xml message. If records do exist, then the physical location is not being deleted, and a 'PODtlMod' message type is created.

## Message Family Manager Procedures

See 'Ordering MFM design.doc' for this section.

# Store Message Family Manager Publishing Design

## Functional Area

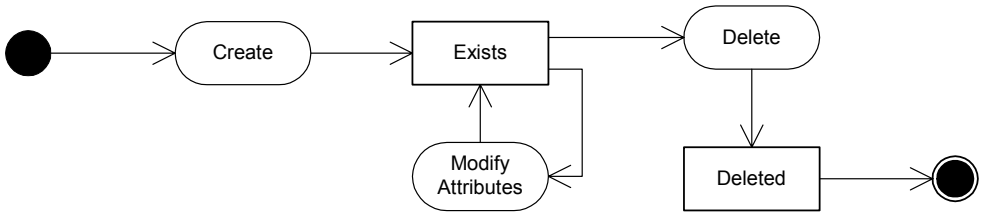
Locations

## Design Overview

Store publication consists of a single flat message containing store attributes from the table STORE. One message will be synchronously created and placed in the message queue each time a store is created, modified, or deleted. When a store is created or modified, the flat message will contain numerous attributes of the store. When a store is deleted, the message will simply contain the unique identifier of the store. Messages are retrieved from the message queue in the order they were created.

Along with the XML message, the message family manager will also send the store type, which specifies whether or not the store is physical or virtual (i.e. stockholding or not.) The store type is used by the RIB publication adaptor for routing messages, since some parts of the system will only have messages from physical locations sent to it. The RIB publication adaptor will also combine store and warehouse messages as location messages for parts of the system that don't differentiate between stores and warehouses.

## State Diagram



# Description of Activities

## Create

1. **Prerequisites**: Every level in the organizational hierarchy above stores must be established before a store can be created.
2. **Activity Detail**: Once a store has been created, it is ready to be published. An initial publication message is made.
3. **Messages**: A "Store Create" message is queued. This message is a flat message that contains a partial snapshot of the attributes on the STORE table.

## Modify

1. **Prerequisites**: Store has been created.
2. **Activity Detail**: The user is allowed to change attributes of the store. 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 "Store Modify" message to be queued. This message contains the same attributes as the "Store Create" message.

## Delete

1. **Prerequisites**: Store has been created.
2. **Activity Detail**: Deleting a store removes it from the system. External systems are notified by a published message.
3. **Messages**: When a store is deleted a "Store Delete" message, which is a flat notification message, is queued. The message contains the store identifier.

# Triggers

Trigger Description (EC_TABLE_STR_AIUDR):

This trigger will capture inserts/updates/deletes to the STORE table and write data into the store_mfqueue message queue. It will call STORE_XML.BUILD_MESSAGE to create the XML message, then call RMSMFM_STORE.ADDTOQ to insert this message into the message queue.

On Insert: A StoreDesc message containing information from the STORE table is created.

On Update: A StoreDesc message containing information from the STORE table is created.

On Delete: A StoreRef message containing the store id is created.

# Message Family Manager Procedures

Public Procedures for Store MFM:

**ADDTOQ(O_status_code, O_error_msg, I_message_type, I_store, I_store_type, I_message)** – This procedure is called by EC_TABLE_STR_AIUDR, and takes the message type, store id, store type and the message itself. It inserts a row into the message family queue STORE_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.

**GETNXT (O_status_code, O_error_msg, O_message_type, O_message, O_store, O_store_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, the store type is an indicator specifying whether the store is physical or virtual, and the family key is the store id, which will be populated for all message types. Status code is one of five values, as shown in the following table. For more discussion of the status codes, refer to the Error Handling Guidelines document or the process flow in the following section. These codes come from an EAI team defined 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.

# Supplier Message Family Manager Publishing Design

## Functional Area

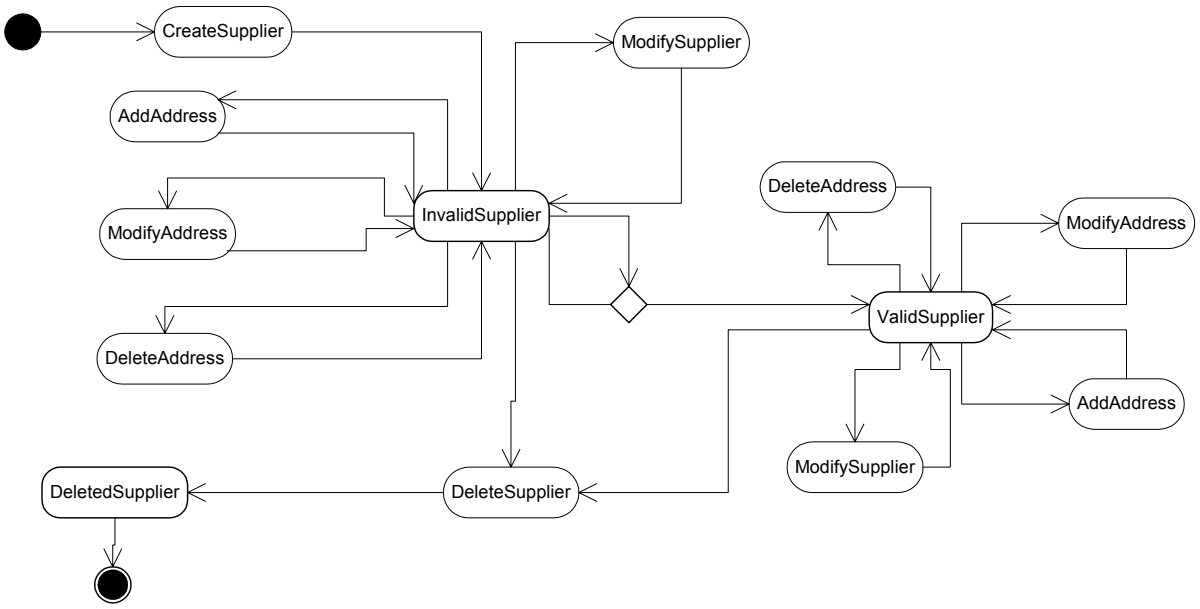
Suppliers and supplier addresses are published.

## Design Overview

New or updated supplier information will be sent from RMS to RDM and RCOM. As suppliers and addresses are added, 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 always being required. Once all the criteria are met for a valid create message the messages will be combined and sent to the Bus. 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.

## State Diagram



# Description of Activities

## <Create Supplier>

1. **Prerequisites**: Supplier doesn't 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 not valid
2. **Activity Detail**: Update the supplier record in the database.
3. **Messages**: VendorHdrMod message is created but message data replaces data in VendorCre and message is deleted from queue.

## <AddAddress>

1. **Prerequisites**: Supplier exists and is not valid
2. **Activity Detail**: Insert the address into the database.
3. **Messages**: VendorAddrCre message is created but not published.

## <ModifyAddress>

1. **Prerequisites**: Supplier exists and is not valid. Address exists.
2. **Activity Detail**: Update the address in the database.
3. **Messages**: VendorAddrMod message is created but message data replaces data in VendorAddrCre and message is deleted from queue.

## <DeleteAddress>

1. **Prerequisites**: Supplier exists and is not valid. Address exists.
2. **Activity Detail**: Delete the address from the database.
3. **Messages**: VendorAddrDel message is created. Message and corresponding VendorAddrCre message are deleted from queue.

## <DeleteSupplier>

1. **Prerequisites**: Supplier exists and is not valid.
2. **Activity Detail**: Delete any existing addresses for the supplier and the supplier from the database.
3. **Messages**: VendorDel message is created. Message and corresponding VendorCre, VendorAddrCre message are deleted from queue.

### <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.

### <DeleteAddress>

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.

## Triggers

Triggers should only insert records onto the staging table.

Trigger Description (EC_TABLE_SUP_AIUDR): 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 key the returns allowed indicator and the invoice match indicator. The message is then inserted into the mfqueue table by calling rmsmfm_supplier.addtoq.

On Insert:

Set event type to ‘A’ and message type to VendorHdrCre.

On Update:

Set event type to ‘D’ and message type to VendorHdrMod.

On Delete:

Set event type to 'D' and message type to VendorDel.

Trigger Description (EC_TABLE_ADR_AIUDR): 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 key the returns allowed indicator and the invoice match indicator. The message is then inserted into the mfqueue table by calling rmsmfm_supplier.addtoq.

On Insert:

Set event type to 'A' and message type to VendorAddrCre.

On Update:

Set event type to 'D' and message type to VendorAddrMod.

On Delete:

Set event type to 'D' and message type to VendorAddrDel.

## Message Family Manager Procedures

Public Procedures:

**ADDTOQ(I_message_type, I_supplier, I_addr_seq_no, I_addr_type, I_ret_allow_ind, I_invc_match_ind, I_message, O_status, O_text)** – 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.

**GETNXT (O_status_code, O_error_msg, O_message_type, O_message, O_supplier, O_addr_seq_no, O_addr_type)** – This publicly exposed procedure is called by a RIB publication adaptor. It'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 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, adder_seq_no, and addr_type. Status code is one of 3 values, as shown in the following table. For more discussion of the status codes, refer to the Error Handling Guidelines document or the process flow in the following section. These codes come from an EAI team defined 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.

| | | |
|---|---|---|
| NO_MSG | 'N' | No more messages to process |
| UNHANDLED_ERROR | 'E' | Unclassified (fatal) Error |
| SUCCESS | 'S' | Success |

Private Procedures:

These private procedures are only used when the initial create message is hierarchical.

**CREATE_PREVIOUS(I_queue_rec)** – 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.

**CLEAN_QUEUE(I_queue_rec)** – 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.

**CAN_CREATE(I_queue_rec)** – 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.

**MAKE_CREATE(O_msg, I_queue_rec)** – 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.

**DELETE_QUEUE_REC(I_seq_no)** – This procedure deletes a specific record from the queue. It deletes based on the sequence number passed in.

**REPLACE_QUEUE_SUP( I_rec, I_data)** – This procedure replaces the message in the create supplier record with the message from the modify supplier record.

**REPLACE_QUEUE_ADR( I_rec, I_data)** – This procedure replaces the message in the create address record with the message from the modify address record.

**REPLACE_QUEUE_MESSAGE( I_rec, I_data)** – This procedure replaces the message in the create supplier record with the complete hierarchical message.

# Transfers Message Family Manager Publishing Design

## Functional Area

Transfers

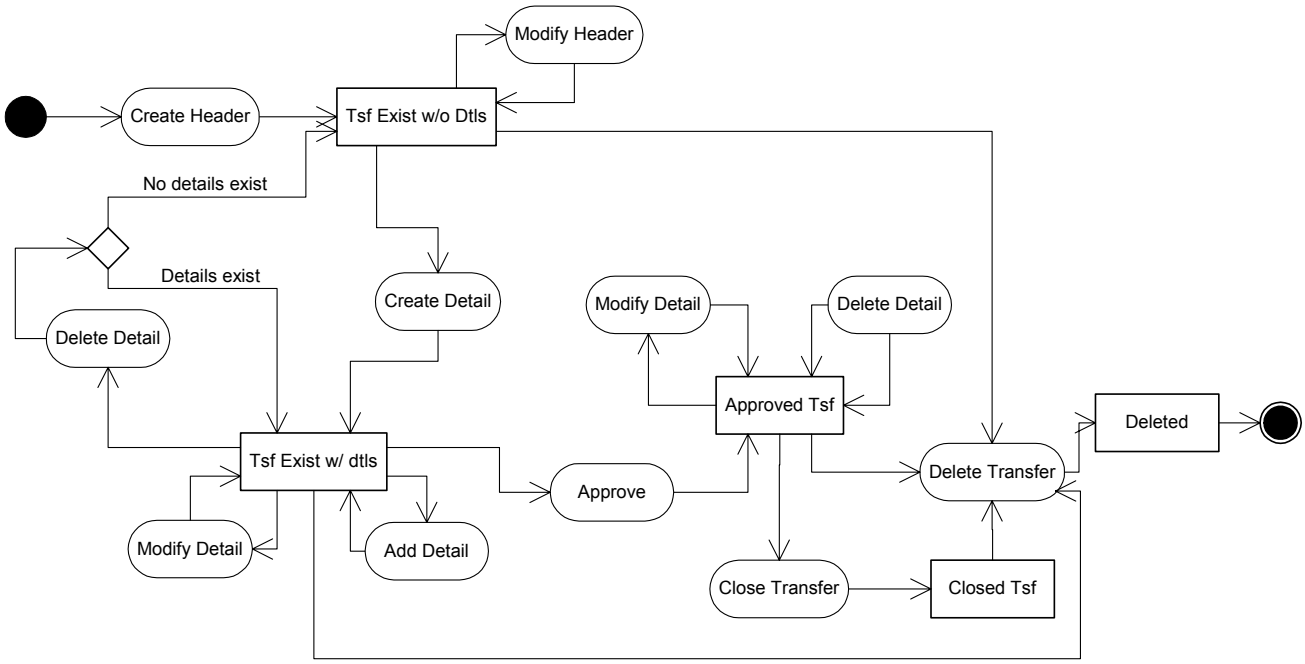
## Design Overview

Rather than waiting for a batch cycle to download transfer information to an external system, transfers will now follow a more real-time process.

Currently, 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, will now have triggers that track inserts, deletes, and modifications. These triggers will insert into the new transfer queue, transfer_mfqueue. The transfer family manager will be responsible for pulling transfer information from this new queue and sending it to the external system(s) at the appropriate time and in the correct order.

The transfer messages that will be published by the family manager will vary. A complete message including header information, detail information, and component ticketing information (if applicable) will be created when a transfer is approved. This is the first message that will be published. After approval, "flat" messages will be sent when modifications are made to the header information or detail information (the transfer quantity is modified or a transfer detail is deleted).

## State Diagram



# Description of Activities

## 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 header is created, a "TransferCre" request is queued. The Transfer Create message is a flat message containing a snapshot of the header at the time the message is processed.

## 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 "TsfHdrMod" 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 (TsfHdrMod) 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**: When a transfer detail is added, a "TsfDtlCre" request is queued. The Transfer Detail Create message is a hierarchical message containing a snapshot of the details at the time the message is processed, and its corresponding ticket component information if applicable.

## Modify Details

1. **Prerequisites**: Only modifications to transfer **quantities** will be 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**: When a transfer quantity is modified a "TsfDtlMod" request is queued. The Transfer Detail Modified message is a hierarchical message containing a snapshot of the details at the time the message is published, and its corresponding ticket component information if applicable.

## Delete Details

1. **Prerequisites**: Only a detail that hasn't 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 hasn't been shipped.
3. **Messages**: When an item is deleted from a transfer, a "TsfDtlDel" Message, which is a flat notification message containing the transfer number, is queued.

## 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) will be performed on it.
3. **Messages**: Closing a transfer queues a "TsfHdrMod" 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.

# Triggers

Trigger Description (EC_TABLE_THD_AIUDR): This trigger will fire when a record is inserted into the TSFHEAD table, when a record is deleted from the TSFHEAD table, or when the status of the transfer has been modified to either 'A'pproved, 'C'losed, or 'D'eleted. This trigger will ignore book transfers and non-salable book transfers (these transfer types are internal to RMS and should not be sent to an external system). The trigger will use the new tsfhead xml builder to create a clob to place on the queue. Then, the transfer family manager's ADDTOQ function will be called so a record is placed on the queue.

On Insert: When a record is inserted into TSFHEAD the trigger will insert a record into the queue by calling the new ADDTOQ function within the family manager. The message type for this action is 'TransferCre'.

On Update: When the status of a transfer is modified, this trigger will fire. There are only three statuses that the trigger should be concerned with. When a transfer is placed in 'A'pproved status , the trigger will insert a record into the queue by calling the new ADDTOQ function within the family manager. The message type for this action will be TransferHdrMod and because the new status is approved, the appr_ind on the queue should be set to 'Y'. When the new status is 'C'losed, the family manager will insert a record into the queue with message type = TransferHdrMod and the appr_ind = 'N'. When a transfer's status is updated to 'D'eleted, the family manager will insert a record into the queue with a message_type = TransferDel.

On Delete: When a record is removed from the TSFHEAD table, the family manager will insert a record into the queue with a message type = TransferDel.

Trigger Description (EC_TABLE_TDT_AIUDR): This trigger will fire when a record is inserted into the TSFDETAIL table, when a record is deleted from the TSFDETAIL table, or when the transfer quantity of one of the items on the transfer has been modified. The trigger will use the new tsfdetail xml builder to create a clob to place on the queue. Then, the transfer family manager's ADDTOQ function will be called so a record is placed on the queue.

On Insert: When a record is inserted into TSFDETAIL, the trigger will insert a record into the queue by calling the new ADDTOQ function within the family manager. The message type for this action is 'TransferDtlCre'.

On Update: When the transfer quantity for one of the items on the transfer is modified, this trigger will fire. The trigger will insert a record into the queue by calling the new ADDTOQ function within the family manager. The message type for this action will be TransferDtlMod and the item being updated should be placed on the queue. **Note: when a transfer quantity is reduced because of an increase in the cancelled quantity, a record should NOT be inserted into the queue.

On Delete: When a record is removed from the TSFDETAIL table, the family manager will insert a record into the queue with a message type = TransferDtlDel for the transfer/item that was deleted.

## Message Family Manager Procedures

Public Procedures:

**ADDTOQ(I_message_type, I_tsf_no, I_item, I_appr_ind, I_physical_from_loc, I_virtual_from_loc, I_from_loc_type, I_to_loc_type, I_message, O_status, O_text)** – This procedure is called by both the tsfhead trigger and the tsfdetail trigger (ec_table_thd_aiudr and ec_table_tdt_aiudr respectively), and takes the message type, family key values (tsf_no, and item) 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, setting the status to unpublished. It returns error codes and strings according to the standards of the application in which it is being implemented.

**GETNXT (O_status_code, O_error_msg, O_message_type, O_message, O_tsf_no, O_item, O_physical_from_loc, O_virtual_from_loc, O_from_loc_type, O_to_loc_type,)** – This public 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 created by the xml builder when either the tsfhead or tsfdetail trigger is executed, and the tsf_no/item is the key for the message as pertains to the transfer family, not all of which will necessarily be populated for all message types (e.g. item may or may not be NULL depending on which trigger inserted the record into the queue). Status code is one of five values, as shown in the following table. For more discussion of the status codes, refer to the Error Handling Guidelines document or the process flow in the following section. These codes come from an EAI team defined 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. The facility id is only included in messages coming from RDM.

Private Procedures:

These private procedures are only necessary when the initial create message is hierarchical.

**CREATE_PREVIOUS(I_queue_rec)** – This procedure determines if a header level create already exists on the queue table for the tsf_no with a sequence number less than the current records sequence number.

**CLEAN_QUEUE(I_queue_rec)** – This procedure cleans up the queue by eliminating modification messages. It is only called if CREATE_PREVIOUS returns true. For each modification message type, it finds the previous corresponding create message type. It then calls REPLACE_QUEUE_TSFHEAD/TSFDETAIL 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.

**MAKE_CREATE(O_msg, I_queue_rec)** – This procedure 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's ready to be published for the first time to the external system(s)).

**DELETE_QUEUE_REC(I_seq_no)** – This procedure deletes a specific record from the queue. It deletes based on the sequence number passed in.

**REPLACE_QUEUE_TSFHEAD( I_rec, I_data)** – This procedure replaces the message in the create header record with the message from the modify header record.

**REPLACE_QUEUE_TSFDETAIL( I_rec, I_data)** – This procedure replaces the message in the create detail record with the message from the modify detail record. There will be one of these procedures for each detail level.

**REPLACE_QUEUE_MESSAGE( I_rec, I_data)** – This procedure replaces the message in the message in the create header record with the complete hierarchical message.

# UDA Message Family Manager Publishing Design

## Functional Area

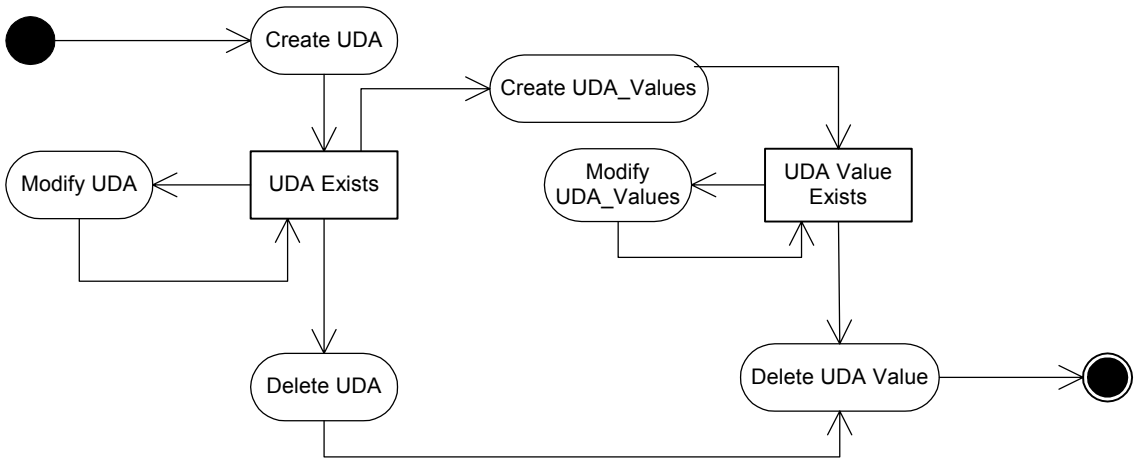
UDA

## Design Overview

As user defined attributes are added, modified or deleted, the event capture trigger creates an xml message in the xml builder and adds the message to the uda_mfqueue table. All the messages on the UDA_MFQUEUE table are published in the same order as they occur in the RMS database.

All values on the UDA and UDA_VALUES tables are affected.

## State Diagram



# Description of Activities

## Create UDA

1. **Prerequisites**: UDA doesn't 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 doesn't 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**: UDAValDesc message type is 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 exists.
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.

**Activity Detail**: Deleting a UDA_value removes it from the UDA_VALUES table and inserts a UDAValDel row to the UDA_MFQUEUE table.

**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. Since 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.

# Triggers

Trigger Description (EC_TABLE_UDA_AIUDR): This trigger fires on any insert, update or delete on the UDA table. It captures the data in new for inserts and updates. It captures the old data on deletes. It sets the action type and message type and calls the UDA_XML.BUILD_UDA_MSG procedure to build the message. The record is inserted into the UDA_MFQUEUE table by calling the RMSMFM_UDA.ADDTOQ procedure.

On Insert:

Sets action_type to ‘A’dd and message_type to ‘UDAHdrCre’.

On Update:

Sets action_type to ‘M’odify and message_type to ‘UDAHdrMod’.

On Delete:

Sets action_type to ‘D’elete and message_type to ‘UDAHdrDel’.

**UDA_XML. BUILD_UDA_MSG (O_status, O_text, O_message, I_record, I_action_type)–** This function is called by the trigger EC_TABLE_UDA_AIUDR on insert, update and delete of the UDA table. This function gathers all the data necessary to build the message that needs to be sent to the Retek Integration Bus. It determines the proper message to build based on the action_type that is set in the trigger. It builds UDARef xml messages for delete statements, or UDADesc xml messages for updates or inserts.

Trigger Description (EC_TABLE_UDV_AIUDR): This trigger fires on any insert, update or delete on the UDA_VALUES table. It captures the data in new for inserts and updates. It captures the old data on deletes. It sets the action type and message type and calls the UDA_XML.BUILD_UDAV_MSG procedure to build the message. The record is inserted into the UDA_MFQUEUE table by calling the RMSMFM_UDA.ADDTOQ procedure

**On Insert:**

**Sets action_type to 'A'dd and message_type to 'UDAValCre'.**

On Update:

Sets action_type to 'M'odify and message_type to 'UDAValMod'.

On Delete:

Sets action_type to 'D'elete and message_type to 'UDAValDel'.

Public Functions:

**UDA_XML. BUILD_UDAV_MSG (O_status, O_text, O_message, I_record, I_action_type)–** This function is called by the trigger EC_TABLE_UDV_AIUDR on insert, update and delete of the UDA_VALUES table. This function gathers all the data necessary to build the message that needs to be sent to the Retek Integration Bus. It determines the proper message to build based on the action_type that is set in the trigger. It builds UDAValRef xml messages for delete statements, or UDAValDesc xml messages for updates or inserts.

## Message Family Manager Procedures

Public Procedures:

**ADDTOQ(O_status_code, O_text, I_message_type, I_uda_id, I_uda_value, I_message)** – 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.

**GETNXT (O_status_code, O_error_msg, O_message_type, O_message, O_uda_id, O_uda_value)** – 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. Status code is one of five values, as shown in the following table. For more discussion of the status codes, refer to the Error Handling Guidelines. These codes come from an EAI team defined 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.

| | | |
|---|---|---|
| NO_MSG | 'N' | No more messages to process |
| UNHANDLED_ERROR | 'E' | Unclassified (fatal) Error |
| SUCCESS | 'S' | Success |

# WH Message Family Manager Publishing Design

## Functional Area

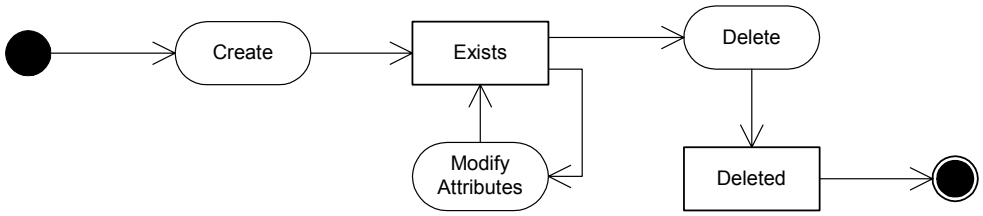
Locations

## Design Overview

Warehouse publication consists of a single flat message containing warehouse attributes from the table WH. One message will be synchronously created and placed in the message queue each time a warehouse is created, modified, or deleted. When a warehouse is created or modified, the flat message will contain numerous attributes of the warehouse. When a warehouse is deleted, the message will simply contain the unique identifier of the warehouse. Messages are retrieved from the message queue in the order they were created.

Along with the XML message, the message family manager will also send the warehouse type, which specifies whether or not the warehouse is physical or virtual. The warehouse type is used by the RIB publication adaptor for routing messages, since some parts of the system will only have messages from physical locations sent to it. The RIB publication adaptor will also combine store and warehouse messages as location messages for parts of the system that don't differentiate between stores and warehouses.

## State Diagram



# Description of Activities

## Create

1. **Prerequisites:** None
2. **Activity Detail**: Once a warehouse has been created, it is ready to be published. An initial publication message is made.
3. **Messages**: A "WH Create" message is queued. This message is a flat message that contains a partial snapshot of the attributes on the WH table.

## Modify

1. **Prerequisites**: Warehouse has been created.
2. **Activity Detail**: The user is allowed to change attributes of the warehouse. 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 "WH Modify" message to be queued. This message contains the same attributes as the "WH Create" message.

## Delete

1. **Prerequisites**: Warehouse has been created.
2. **Activity Detail**: Deleting a warehouse removes it from the system. External systems are notified by a published message.
3. **Messages**: When a warehouse is deleted a "WH Delete" message, which is a flat notification message, is queued. The message contains the warehouse identifier.

# Triggers

Trigger Description (EC_TABLE_WH_AIUDR):

This trigger will capture inserts/updates/deletes to the WH table and write data into the wh_mfqueue message queue. It will call WH_XML.BUILD_MESSAGE to create the XML message, then call RMSMFM_WH.ADDTOQ to insert this message into the message queue.

On Insert: A WHDesc message containing information from the wh table is created.

On Update: A WHDesc message containing information from the wh table is created.

On Delete: A WHRef message containing the wh id is created.

# Message Family Manager Procedures

Public Procedures:

**ADDTOQ(O_status_code, O_error_msg, I_message_type, I_wh, I_wh_type, I_message)** – This procedure is called by EC_TABLE_WH_AIUDR, and takes the message type, wh id, wh type, and the message itself. It inserts a row into the message family queue WH_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.

**GETNXT (O_status_code, O_error_msg, O_message_type, O_message, O_wh, O_wh_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, the wh type is an indicator specifying whether the wh is physical or virtual, and the family key is the wh id, which will be populated for all message types. Status code is one of five values, as shown in the following table. For more discussion of the status codes, refer to the Error Handling Guidelines document or the process flow in the following section. These codes come from an EAI team defined 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.

# Work Order Message Family Manager Publishing Design

## Functional Area

Work Orders

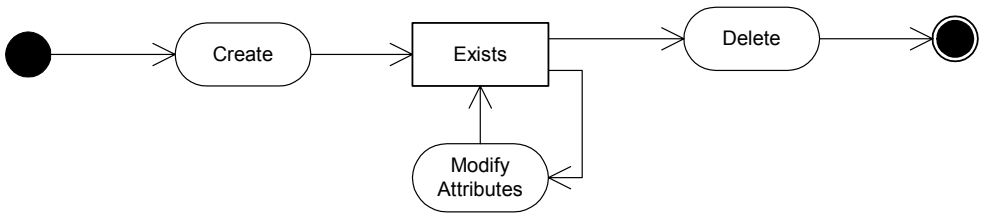
## Design Overview

Work order publication consists of a single flat message containing attributes from the table WO_DETAIL. One message will be synchronously created and placed in the message queue each time a WO_DETAIL record is created, modified, or deleted. The primary key for the WO_DETAIL consists of the work order id, 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 flat message will contain a full snapshot of the WO_DETAIL record. When a WO_DETAIL record is deleted, the message will contain 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. Along with the XML message, the message family manager will also 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.

## State Diagram



# Description of Activities

## 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 is a flat message that 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, which is a flat notification message, is queued. The message contains a partial snapshot of the WO_DETAIL table.

## Triggers

Trigger Description (EC_TABLE_WDL_AIUDR):

This trigger will capture inserts/updates/deletes to the WO_DETAIL table and write data into the WORKORDER_MFQUEUE message queue. It will call WORKORDER_XML.BUILD_MESSAGE to create the XML message, then call RMSMFM_WORKORDER.ADDTOQ to insert this message into the message queue.

On Insert: An 'InBdWOCre' message containing information from the WO_DETAIL table is created.

On Update: An 'InBdWOMod' message containing information from the WO_DETAIL table is created.

On Delete: An 'InBdWODel' message containing information from the WO_DETAIL table is created.

'InBdWO' stands for Inbound Work Order. This helps RDM keep track of which work orders are coming into its system.

## Message Family Manager Procedures

Public Procedures:

**ADDTOQ(O_status_code, O_error_msg, I_message_type, I_wo_id, I_order_no, I_wh, I_seq_no, I_item, I_location, I_message)** – This procedure is called by EC_TABLE_WDL_AIUDR, and takes the message type, the order that the work order is attached to, columns from the WO_DETAIL table that make up its primary key, and the message itself. It inserts a row into the message family queue WORKORDER_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.

**GETNXT (O_status_code, O_error_msg, O_message_type, O_message, O_wo_id, O_order_no, O_wh, O_seq_no, O_item, O_location)** – 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 order_no specifies which order the work order is attached to, and the family key is the wo_id, wh, seq_no, item, and location. All of the columns in the family key will be populated for all message types. Status code is one of five values, as shown in the following table. For more discussion of the status codes, refer to the Error Handling Guidelines document or the process flow in the following section. These codes come from an EAI team defined 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.

# Chapter 2 – Subscription designs

## Appointments Subscription Design

### Functional Area

Appointments

### Design Overview

These APIs are used to organize and process RDM-generated Appointments messages. Appointments records indicate the quantities of particular Items sent to various Locations within the system. In addition, the records indicate the specific document (Purchase Order, Transfer or Allocation) responsible for the movement of the Item in question.

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 Procedures

RMSSUB_APPOINTCRE

This procedure creates one complete Appointment record, consisting of a single Header record and one or more Detail records.

**Public API Procedures:**

**CONSUME (O_STATUS_CODE, O_ERROR_MESSAGE, I_MESSAGE):** This procedure accepts an XML file (I_message) in the form of an Oracle CLOB data type from the RIB. The procedure validates 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 passed to the function APPOINTMENT_PROCESS_SQL.PROC_SCH and PROC_SCHDET, which validate the information and place the data on the database (depending upon the success of the validation).

**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.

**Private Internal Functions and Procedures:**

**HANDLE_ERRORS (O_status_code, IO_**error**_message, I_cause, I_program):** This function is used to put error handling in one place in order to make future error handling enhancements easier to implement. 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.

RMSSUB_APPOINTDTLCRE

This procedure adds a single Detail record to an existing Appointments record.

**Public API Procedures:**

**CONSUME (O_STATUS_CODE, O_ERROR_MESSAGE, I_MESSAGE):** This procedure accepts an XML file (I_message) in the form of an Oracle CLOB data type from the RIB. The procedure validates 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 passed to the function APPOINTMENT_PROCESS_SQL.PROC_SCHDET, which validates the information and places the data on the database (depending upon the success of the validation).

**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.

**Private Internal Functions and Procedures:**

**HANDLE_ERRORS (O_status_code, IO_error_message, I_cause, I_program):** This function is used to put error handling in one place in order to make future error handling enhancements easier to implement. 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.

RMSSUB_APPOINTHDRMOD

This procedure modifies a single Appointment Header record.

**Public API Procedures:**

**CONSUME (O_STATUS_CODE, O_ERROR_MESSAGE, I_MESSAGE):** This procedure accepts an XML file (I_message) in the form of an Oracle CLOB data type from the RIB. The procedure validates 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 passed to the function APPOINTMENT_PROCESS_SQL.PROC_MODHED, which validates the information and modifies the data on the database (depending upon the success of the validation).

**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.

**Private Internal Functions and Procedures:**

**HANDLE_ERRORS (O_status_code, IO_error_message, I_cause, I_program):** This function is used to put error handling in one place in order to make future error handling enhancements easier to implement. 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.

RMSSUB_APPOINTDTLMOD

This procedure modifies a single Appointment Detail record.

**Public API Procedures:**

**CONSUME (O_STATUS_CODE, O_ERROR_MESSAGE, I_MESSAGE):** This procedure accepts an XML file (I_message) in the form of an Oracle CLOB data type from the RIB. The procedure validates 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 passed to the function APPOINTMENT_PROCESS_SQL.PROC_MODDET, which validates the information and modifies the data on the database (depending upon the success of the validation).

**Private Internal Functions and Procedures:**

**HANDLE_ERRORS (O_status_code, IO_error_message, I_cause, I_program):** This function is used to put error handling in one place in order to make future error handling enhancements easier to implement. 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.

RMSSUB_APPOINTDEL

This procedure deletes one complete Appointment record, consisting of a single Header record and one or more Detail records.

**Public API Procedures:**

**CONSUME (O_STATUS_CODE, O_ERROR_MESSAGE, I_MESSAGE):** This procedure accepts an XML file (I_message) in the form of an Oracle CLOB data type from the RIB. The procedure validates 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 passed to the function APPOINTMENT_PROCESS_SQL.PROC_DEL, which validates the information and deletes the data from the database (depending upon the success of the validation).

**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.

**Private Internal Functions and Procedures:**

**HANDLE_ERRORS (O_status_code, IO_error_message, I_cause, I_program):** This function is used to put error handling in one place in order to make future error handling enhancements easier to implement. 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.

RMSSUB_APPOINTDTLDEL

This procedure deletes a single Appointment Detail record.

**Public API Procedures:**

**CONSUME (O_STATUS_CODE, O_ERROR_MESSAGE, I_MESSAGE):** This procedure accepts an XML file (I_message) in the form of an Oracle CLOB data type from the RIB. The procedure validates 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 passed to the function APPOINTMENT_PROCESS_SQL.PROC_DELDET, which validates the information and deletes the data from the database (depending upon the success of the validation).

**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.

**Private Internal Functions and Procedures:**

**HANDLE_ERRORS (O_status_code, IO_error_message, I_cause, I_program):** This function is used to put error handling in one place in order to make future error handling enhancements easier to implement. 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.

APPOINTMENT_PROCESS_SQL

This package is a library of functions that are used to validate, insert, update and delete Appointments Records. The functions accept data in the form of complete Records or Table arrays.

**Public Procedures:**

PROC_SCH (O_error_message, I_appt_head, I_appt_detail): This function processes Scheduled (i.e. newly-created) Appointments messages. It calls VAL_LOC and HEAD_EXISTS to validate the data, inserts the Header record if it doesn't already exist and then loops through each Detail record passed in, calling PROC_SCHDET for each.

PROC_SCHDET (O_error_message, O_record_locked, I_appt_head): This function processes the Detail portion of Scheduled (i.e. newly-created) Appointments messages. It calls VAL_LOC, VAL_DOC, VAL_ITEM and DETAIL_EXISTS to validate the data and then inserts the Detail record if it doesn't already exist.

PROC_MODHED (O_error_message, O_record_locked, I_appt_head): This function processes Header Modified Scheduled (i.e. modified) Appointments messages. It calls VAL_LOC and HEAD_EXISTS to validate the data; it then modifies the Header record, or inserts the Header record if it doesn't exist. If a record cannot be modified becaues it is locked, the O_record_locked indicator is returned as TRUE.

PROC_MODDET (O_error_message, O_record_locked, I_appt_detail_rec): This function processes Detail Modified Scheduled (i.e. modified) Appointments messages. It calls VAL_LOC and DETAIL_EXISTS to validate the data; it then modifies the Detail record, or inserts the Detail record if it doesn't exist. If a record cannot be modified becaues it is locked, the O_record_locked indicator is returned as TRUE.

PROC_DEL (O_error_message, O_record_locked, I_appt_head): This function processes Delete Appointments messages. It calls VAL_LOC and HEAD_EXISTS to validate the data; it then deletes the Header record and all child Detail records. If a record cannot be modified becaues it is locked, the O_record_locked indicator is returned as TRUE.

PROC_DELDET (O_error_message, O_record_locked, I_appt_head): This function processes Delete Detail Appointments messages. It calls VAL_LOC and HEAD_EXISTS to validate the data; it then deletes the Detail record. If a record cannot be modified becaues it is locked, the O_record_locked indicator is returned as TRUE.

Private Internal Functions and Procedures:

VAL_LOC (O_error_message, O_valid_loc, O_loc_type, I_loc): This function verifies that the passed-in Location exists on either the Store or Warehouse tables. It returns the Location Type for the passed-in Location.

VAL_DOC (O_error_message, O_valid_doc, I_doc, I_doc_type): This function verifies that the passed-in Document exists on either the Order, Transfer or Allocation Header tables.

VAL_ITEM (O_error_message, O_valid_item, I_item): This function verifies that the passed-in Item exists on the Item Master table.

HEAD_EXISTS (O_error_message, O_head_exists, O_record_locked, O_rowid, I_appt_head): This function verifies that thepassed-in Appointments Header record exists. It returns the rowid of the record. If the record is locked, the O_record_locked indicator is returned as TRUE.

DETAIL_EXISTS (O_error_message, O_detail_exists, O_record_locked, O_rowid, I_detail_rec): This function verifies that thepassed-in Appointments Header record exists. It returns the rowid of the record. If the record is locked, the O_record_locked indicator is returned as TRUE.

# ASN Subscription Design

## Functional Area

ASN – Advanced Shipping Notice from a supplier.

The structure of the shipment tables will be changed to better show what is being shipped. Previously there was one order or transfer per shipment. We will be changing the table structure to allow multiple transfers or allocations per shipment. This is being done to better mirror what actually happens. 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.

## Design Overview

A supplier or consolidator will send an Advanced Shipping Notice (ASN) to RDM. RDM will publish the information and it will be placed onto the SEEBEYOND Bus. RMS will subscribe to the ASN information as published from the Bus and place 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.

## Subscription Procedures

Subscribing to ASN entails the use of three public consume procedures (in three different packages). These procedures correspond to the types of activities that can be done to an ASN record: create, modify and delete.

## Public API Procedures:

**RMSSUB_ASNINCRE.CONSUME (O_STATUS_CODE, O_ERROR_MESSAGE, I_MESSAGE)** - This procedure accepts a XML file in the form of an Oracle CLOB data type from the RIB (I_message). This message will contain an ASN create message consisting of the aforementioned header and detail records. The procedure will then place a call to the main RMSSUB_ASN.CONSUME function, passing on a message type of 'C'reate, 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 appropriate shipment and ordering database tables depending upon the success of the validation.

**RMSSUB_ASNINMOD.CONSUME (O_STATUS_CODE, O_ERROR_MESSAGE, I_MESSAGE)** - This procedure accepts a XML file in the form of an Oracle CLOB data type from the RIB (I_message). This message will contain an ASN modify message consisting of the aforementioned header and detail records. The procedure will then place a call to the main RMSSUB_ASN.CONSUME function, passing on a message type of 'M'odify, 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 update the appropriate shipment and ordering database tables depending upon the success of the validation.

**RMSSUB_ASNINDEL.CONSUME (O_STATUS_CODE, O_ERROR_MESSAGE, I_MESSAGE)** - This procedure accepts a XML file in the form of an Oracle CLOB data type from the RIB (I_message). This message will contain an ASN modify message consisting of the aforementioned header and detail records. The procedure will then place a call to the main RMSSUB_ASN.CONSUME function, passing on a message type of 'D'elete, 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 a private internal function, which will validate the values and delete the ASN record from the appropriate shipment and invoice database tables depending upon the success of the validation.

**Error Handling (in all of the above packages):**

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, IO_text, I_cause, I_program)-** 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_ASN 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:

All of the following functions exist within RMSSUB_ASN.

**Main Consume Function:**

**RMSSUB_ASN.CONSUME (O_STATUS_CODE, O_ERROR_MESSAGE, I_MESSAGE_TYPE, I_MESSAGE)** - This procedure accepts a XML file in the form of an Oracle CLOB data type from the RIB (I_message) and a message_type (Create, Modify or Delete) from one of the aforementioned public ASN procedures, depending on the type of ASN message being subscribed to. This message will consist of the aforementioned header and detail records.

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 shipment and ordering database tables depending upon the success of the validation.

**XML Parsing:**

**PARSE_ASN (O_TEXT, O_VAL, I_XML_VAL) –** This function will used to extract the header level information from the ASN XML file 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 (O_TEXT, O_VAL, I_XML_VAL) –** This function will used to extract the order level information from the ASN XML file and place that information onto an internal order record.

```
TYPE order_record IS RECORD(asn
SHIPMENT.ASN%TYPE,

po_num           SHIPMENT.ORDER_NO%TYPE,

not_after_date   ORDHEAD.NOT_AFTER_DATE%TYPE);
```

**PARSE_CARTON (O_TEXT, O_VAL, I_XML_VAL) –** 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 ASN XML file and place that information onto an internal arrayed carton record.

```
TYPE carton_record IS RECORD(asn
SHIPMENT.ASN%TYPE,

po_num          SHIPMENT.ORDER_NO%TYPE,

carton_num      CARTON.CARTON%TYPE,

location        CARTON.LOCATION%TYPE);
```

**PARSE_ITEM (O_TEXT, O_VAL, I_XML_VAL) –** This function will used to extract the item level information from the ASN XML file and place that information onto an internal arrayed item record.

```
TYPE item_record IS RECORD(asn
SHIPMENT.ASN%TYPE,

po_num          SHIPMENT.ORDER_NO%TYPE,

carton_num      CARTON.CARTON%TYPE,

item            SHIPSKU.ITEM%TYPE,

ref_item        SHIPSKU.REF_ITEM%TYPE,

                                                          vpn
ITEM_SUPPLIER.VPN%TYPE,

alloc_loc       CARTON.LOCATION%TYPE,

qty_shipped     VARCHAR2(20));
```

**Validation:**

**PROCESS_ASN(O_TEXT, I_HEADER, I_DETAIL_1, I_DETAIL_2…)** – After the values are parsed for a particular order in an ASN record, RMSSUB_ASN.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(O_TEXT, I_ASN_NO) – 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.

# BOL Subscription Design

## Functional Area

BOL – Bill of Lading

When external locations ship products, they send up a BOL 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 allocations 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 structure of the shipment tables will be changed to better show what is being shipped. Previously there was one order or transfer per shipment. We will be changing the table structure to allow multiple transfers or allocations per shipment. This is being done to better mirror what actually happens. 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 a 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 now rather than an order or transfer as it worked previously.

## Design Overview

**Data Flow:**

An external location (store or warehouse) will publish a Bill of Lading, thereby placing the BOL information onto the RIB (Retek Information Bus). RMS will subscribe to the BOL 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 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.

Cartons will be required in all BOL (outbound ASN) messages.

## Subscription Procedures

Subscribing to a BOL message entails the use of one public consume procedure. This procedure corresponds to the type of activity that can be done to a BOL record (in this case create).

Public API Procedures:

**RMSSUB_ASNOUTCRE.CONSUME (O_STATUS_CODE, O_ERROR_MESSAGE, I_MESSAGE)** - 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 BOL create message consisting of the aforementioned header and detail records. The procedure will then place a call to the main RMSSUB_BOL.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 appropriate shipment, transfer or allocation database tables depending upon the success of the validation.

Private Internal Functions and Procedures (rmssub_asnoutcre.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, IO_text, I_cause, I_program)-** 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_BOL package and all errors that occur during subscription in the BOL_SQL 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_BOL.**

**Main Consume Function:**

**RMSSUB_BOL.CONSUME (O_STATUS_CODE, O_ERROR_MESSAGE, I_MESSAGE)** - This procedure accepts a XML file in the form of an Oracle CLOB data type from the RIB (I_message) from the aforementioned public BOL procedure whenever a create message is made available by the RIB. This message will consist of the aforementioned header and detail records.

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 shipment and ordering database tables depending upon the success of the validation.

**XML Parsing:**

**PARSE_BOL (O_TEXT, O_VAL, I_XML_VAL) –** This function will used to extract the header level information from the ASN XML file and place that information onto an internal ASN header record.

Record is based upon the shipment table:

```
SHIPMENT%ROWTYPE;
```

**PARSE_DISTRO (O_TEXT, O_VAL, I_XML_VAL) –** This function will used to extract the order level information from the ASN XML file and place that information onto an internal order record.

```
TYPE distro_record IS RECORD (distro_no
tsfhead.tsf_no%TYPE,
                                distro_type
VARCHAR2(2));
```

**PARSE_ITEM (O_TEXT, O_VAL, I_XML_VAL) –** This function will used to extract the item level information from the ASN XML file and place that information onto an internal arrayed item record.

```
TYPE item_record IS RECORD (item
item_master.item%TYPE,
                              carton
shipsku.carton%TYPE,
                              qty
tsfdetail.tsf_qty%TYPE,
                              from_disposition
inv_status_codes.inv_status_code%TYPE);
```

**Validation:**

**PROCESS_BOL(O_TEXT, I_HEADER, I_DISTRO, I_ITEMS)** – After the values are parsed for a particular distro in a BOL record, RMSSUB_BOL.CONSUME will call this function, which will in turn call various functions inside BOL_SQL in order to validate the values and place them on the appropriate shipment and transfer or allocation database tables depending upon the success of the validation.

Only one BOL and distro record will be passed in at a time, whereas multiple items will be passed in as an array into this function. If one distro, carton or item value is rejected, then current functionality dictates that the entire BOL message will be rejected.

# Customer Reserve Subscription Design

## Functional Area

Customer Reserve

## Design Overview

RMS receives customer order details from RCOM that must be accounted for in RMS inventory. RMS will be modified to view the customer reserve and customer backorder from RCOM. This package will handle the processing for maintaining inventory at the item location level as well as transaction data records and history records when a sale occurs. The following diagram shows the process flow:

1. RCOM looks at inventory levels.

2. Customer Reserve is updated.

3. Customer Order is created.

4a. Success- Shipment Notification is sent

4b. Insufficient Inventory – Shipment discrepancy sent

5a. Sale Transaction to ReSA

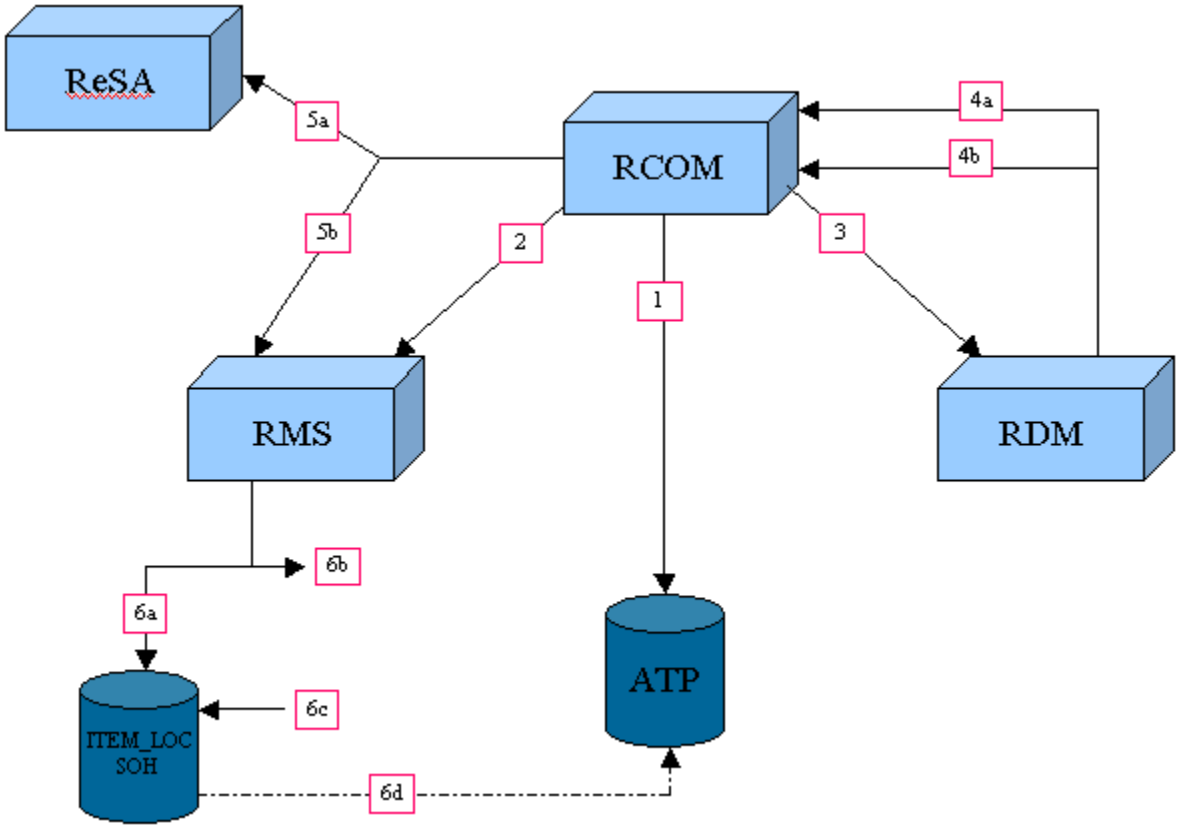
5b. Transfer sent from virtual WH to store

6a. Inventory Updated, WH decremented, store incremented.

6b. RMS creates book transfer

6c. POSUPLD decrements inventory from the store

6d. Inventory Updated


ReSA
5a
RCOM
4a
4b
5b
2
3
1
RMS
RDM
6b
6a
ATP
6c
ITEM_LOC
SOH
6d

# Subscription Procedures

RMSSUB_CORESCRE

RMSSUB_CORESCANCRE

RMSSUB_CUSTBOCRE

RMSSUB_CUSTBOCANCRE

RMSSUB_CUSTRESTOBOCRE

RMSSUB_CUSTBOTORESCRE

RMSSUB_CUSTSALECRE

RMSSUB_CUSTRETSALECRE

Public API Procedures (depending upon the format and types of messages being subscribed to):

**CONSUME (O_STATUS_CODE, O_ERROR_MESSAGE, I_MESSAGE)** - This procedure accepts an XML file in the form of an Oracle CLOB data type from the RIB (I_message). The CONSUME in the main API for customer reserve, RMSSUB_CORESERVE, is called passing in the CLOB as well as the message type.

**Validation:** Validation occurs in the RMSSUB_CORESERVE.PROCESS_HEADER function.

**Error Handling:**

If an error occurs in this procedure a call to HANDLE_ERRORS is made in order to parse a complete error message and pass back a status to the RIB.

**Private Internal Functions and Procedures:**

**HANDLE_ERRORS (O_STATUS_CODE, IO_ERROR_MESSAGE, I_CAUSE, I_PROGRAM)-** 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.

RMSSUB_CORESERVE

**Public API Procedures (depending upon the format and types of messages being subscribed to):**

**CONSUME (O_STATUS_CODE, I_MESSAGE_TYPE, I_MESSAGE)** - This procedure accepts an XML file in the form of an Oracle CLOB data type from the initial API called by the RIB and a message type depending upon which area of customer reserve is being effected. 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 database depending upon the success of the validation.

**Validation:** Validation occurs in the PROCESS_HEADER function to ensure that the message includes an item, quantity and either stock or sell location. Additional validation for customer reserve is done within the package custinvmgmts/b, which is the functional package, associated with customer reserve.

**Error Handling:**

If an error occurs in this procedure or any of the internal functions, this procedure will pass back FALSE to the calling API.

## Private Internal Functions and Procedures:

**PARSE_HEADER (O_ERROR_MESSAGE, I_CUSTRESV_ROOT, I_MESSAGE_TYPE) -** Modularity may neccesitate the use of this internal function to parse the values within the XML file through a series of calls to RIB_XML and pass those values on to private internal functions for processing depending upon the structure of the message being subscribed to and the design of ones consume function.

**PROCESS_HEADER(O_ERROR_MESSAGE, L_STOCK_LOC, L_SELL_LOC, L_ITEM, L_QTY, L_DATE, I_MESSAGE_TYPE) –** This function handles validation and calls the main processing function for customer reserve, CUST_INV_MGMT.CUST_RESV.

# Inventory Adjustment Subscription Design

## Functional Area

Inventory Adjustment

## Design Overview

Inventory adjustments will no longer be handled as a batch operation. RMS will subscribe to inventory adjustment messages published by RDM. The messages will primarily contain information about the item, the physical warehouse, the quantity the specific disposition change, and the reason for the adjustment.

Inventory adjustment messages received will be processed and inventory adjustments will be made at a stock holding location level. Only disposition changes that alter an inventory status within RMS will be processed. The disposition changes that alter inventory status are those in which there is a difference between the INV_STATUS for each INV_STATUS_CODE on INV_STATUS_CODES. All other messages will simply be ignored at this time. Also, if either the *to* or the *from* disposition is null, this will indicate a simple addition or subtraction of stock to the non-null disposition's inventory status.

## Subscription Procedures

**RMSSUB_INVADJUSTCRE**

Public API Procedures (depending upon the format and types of messages being subscribed to):

**CONSUME (O_STATUS_CODE, O_ERROR_MESSAGE, I_MESSAGE)** - This procedure accepts an XML file in the form of an Oracle CLOB data type from the RIB (I_message). The procedure will then pass the clob onto the package RMSSUB_INVADJUST for parsing and processing.

**Validation:**

All validation will be done in the INVADJ_SQL.PROCESS_INVADJ function. The following fields cannot be NULL: item, location, adj_qty, user_id, adj_date. The location type must be either 'S' or 'W'. The doc_type must be NULL, 'P', or 'T'. And either the to_disposition or from_dispostion or both fields must be populated, both cannot be NULL.

**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.

Private Internal Functions and Procedures:

**HANDLE_ERRORS (O_status_code, IO_error_message, I_cause, I_program)-** 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.

**RMSSUB_INVADJUST**

**CONSUME (O_ERROR_MESSAGE, I_MESSAGE)** - This procedure accepts an XML file in the form of an Oracle CLOB data type from the package RMSSUB_INVADJUSTCRE. 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 database depending upon the success of the validation.

**PARSE_HEADER (O_error_message OUT VARCHAR2, O_invadj_record OUT invadj_record, I_invadj_root IN OUT xmldom.DOMElement) -** Modularity may neccesitate the use of this internal function to parse the values within the XML file through a series of calls to RIB_XML and pass those values on to private internal functions for processing depending upon the structure of the message being subscribed to and the design of ones consume function.

**FUNCTION PROCESS_HEADER (O_error_message IN OUT VARCHAR2,** I_invadj_record IN invadj_record) – This function will call the processing package for inventory adjustment INVADJ_SQL.PROCESS_INVADJ.

# Receipts Subscription Design

## Functional Area

Receipts:

Purchase Order Receiving (PO)

Transfer Receiving (TSF)

Allocation Receiving (ALLOC)

## Design Overview

When a transfer, PO or allocation is received at a location, the receipt information will be published by the external location and placed onto the SEEBEYOND Bus. RMS will subscribe to the receipt information as published from the Bus and place the information onto RMS tables depending upon the validity of the records enclosed within the message.

The receipt message is a flat message.

## Subscription Procedures

Subscribing to a receipt message entails the use of two public consume procedures. These procedures correspond to the type of activities that can be done to a receipt record (in this case create and adjust).

**Public API Procedures:**

**RMSSUB_RECEIPTCRE.CONSUME (O_STATUS_CODE, O_ERROR_MESSAGE, I_MESSAGE)** - 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 receipt create message consisting of the aforementioned flat receipt record. The procedure will then place a call to the main RMSSUB_RECEIVE.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 appropriate ordering, transfer or allocation database tables depending upon the success of the validation.

**RMSSUB_RECEIPTMOD.CONSUME (O_STATUS_CODE, O_ERROR_MESSAGE, I_MESSAGE)** - 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 receipt modify message consisting of the aforementioned flat receipt file. The procedure will then place a call to the main RMSSUB_RECEIVE.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 appropriate ordering, transfer or allocation database tables depending upon the success of the validation.

**Private Internal Functions and Procedures (rmssub_receiptcre.pls, rmssub_receiptmod.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, IO_text, I_cause, I_program)-** 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_BOL package and all errors that occur during subscription in the BOL_SQL 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_RECEIVE.

Main Consume Function:

RMSSUB_RECEIVE.CONSUME (O_STATUS_CODE, O_ERROR_MESSAGE, I_MESSAGE) - This procedure accepts a XML file in the form of an Oracle CLOB data type from the RIB (I_message) from one of the aforementioned public receipt procedures whenever a create or adjustment message is made available by the RIB. This message will consist of the aforementioned flat receipt 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 ordering, transfer or allocation database tables depending upon the success of the validation.

**XML Parsing:**

**PARSE_RECEIPT (O_TEXT, O_VAL, I_XML_VAL) –** This function will used to extract the receipt information from the receipt XML file and place that information onto an internal receipt record.

```
TYPE receipt_record IS RECORD (loc
item_loc.loc%TYPE,
                                    appt_no
appt_head.appt%TYPE,
                                    order_no
ordhead.order_no%TYPE,
                                    item
item_master.item%TYPE,
                                    qty
tran_data.units%TYPE,
                                    doc_type
VARCHAR2(1),
                                    tran_type
VARCHAR2(1),
```

```
                                    tran_date
   tran_data.tran_date%TYPE,
                                    receipt_no
   NUMBER(12),
                                    asn_no
   shipment.asn%TYPE,
                                    destination_id
   item_loc.loc%TYPE,
                                    carton
   shipsku.carton%TYPE,
                                    distro_no
   alloc_header.alloc_no%TYPE,
                                    distro_type
   VARCHAR2(1),
                                    disposition
   inv_status_codes.inv_status_code%TYPE);
```

**Validation:**

**PROCESS_RECEIPT(O_TEXT, I_HEADER, I_DETAIL_1)** – After the values are parsed for a particular receipt, RMSSUB_RECEIVE.CONSUME will call this function, which will in turn call various functions inside STOCK_ORDER_RCV_SQL or PO_RCV_SQL in order to validate the values and place them on the appropriate ordering, transfer or allocation database tables depending upon the success of the validation.

If the doc_type passed into RMSSUB_RECEIVE.CONSUME is 'A', then STOCK_ORDER_RCV_SQL.ALLOC_LINE_ITEM. If the doc_type passed into RMSSUB_RECEIVE.CONSUME is 'T', then STOCK_ORDER_RCV_SQL.TSF_LINE_ITEM. And if the doc_type passed into RMSSUB_RECEIVE.CONSUME is 'P', then PO_RCV_SQL.PO_LINE_ITEM.

# RMS SOStatus Subscription Design

## Functional Area

The functional area of this design is Stock Order Status. 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

## Design Overview

Stock order status upload will receive a message from the RIB, published from RDM, communicating the status of a specific stock order. This communication will allow the synchronization of data between RDM and RMS. The information from RDM will have only one level, in other words no detail records. This information will be used to update tsfdetail, alloc_detail and item_loc_soh.

## Subscription Procedures

Public API Procedures (depending upon the format and types of messages being subscribed to):

**CONSUME (O_STATUS, O_TEXT, I_MESSAGE_TYPE, I_DOCUMENT)** - This procedure accepts an XML file in the form of an Oracle CLOB data type from the RIB (I_document) and a message type of update, delete, insert etc, depending upon the types of messages being subscribed to in a particular functional area. 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 database depending upon the success of the validation.

**Validation:**

Validate the distro is valid. A distro refers to either a transfer or an allocation.

**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.

Private Internal Functions and Procedures:

**HANDLE_ERRORS (O_status, IO_text, I_cause, I_program)-** 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 (O_TEXT, O_RECORD_LOCKED, I_XML_VAL) –** This function breaks down the message into its companant parts and sends these parts into PROCESS_SOS.

**PROCESS_SOS (O_error_message, O_record_locked, I_type, I_distro_number, I_to_location, I_item, I_qty,**

I_status) - Based on the status sent from RDM, quantity fields on either tsfdetail or alloc_detail and item_loc_soh will be updated. A breakdown of what fields are updated for each status can be found in Stock Order Status Upload Functional Spec.doc.

**UPDATE_TSF(O_error_message, I_distro_number, I_item, I_tsf_qty, I_selected_qty,**

**I_distro_qty, I_cancelled_qty) -** Updates the record on tsf_detail if the distro is a tsf.

**UPDATE_ALLOC(O_error_message, I_distro_number, I_location, I_qty_allocated, I_selected_qty, I_distro_qty, I_cancelled_qty) –** Updates the record on alloc_detail if the distro is an allocation.

**UPD_FROM_ITEM_LOC(O_error_message, I_from_location, I_item, I_reserved_qty, I_comp_level_upd) –** Updates item_loc_soh.tsf_reserved_qty for the from location if the comp_level_upd indacator is 'N'. If this ind is 'Y' then it will also update the item_loc_soh.pack_comp_resv field for the item passed in.

**UPD_TO_ITEM_LOC(O_error_message, I_to_location, I_item, I_expected_qty, I_comp_level_upd) –** Updates item_loc_soh.tsf_expected_qty for the to location if the comp_level_upd indacator is 'N'. If this ind is 'Y' then it will also update the item_loc_soh.pack_comp_exp field for the item passed in.

# RTV Subscription Design

## Functional Area

RTV – Return to Vendor

## Design Overview

When a RTV is shipped out from the warehouse, the RTV information will be published by the external system and placed on the SEEBEYOND Bus. RMS will subscribe to the RTV information as published from the Bus and place the information onto RMS tables depending on the validity of the records enclosed within the message.

The RTV message is a flat message.

## Subscription Procedures

Subscribing to a RTV message entails the use of two public consume procedures. These procedures correspond to the type of activities that can be done to a RTV record (in this case create and adjust).

Public API Procedures:

**RMSSUB_RTVCRE.CONSUME (O_STATUS_CODE, O_ERROR_MESSAGE, I_MESSAGE)** - 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 RTV create message consisting of the flat RTV record. The procedure will then place a call to the main RMSSUB_RTV.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 perform desired functionality depending upon the success of the validation.

**Private Internal Functions and Procedures (rmssub_rtvcre.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, IO_text, I_cause, I_program)-** 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_BOL package and all errors that occur during subscription in the BOL_SQL 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_RTV.**

**Main Consume Function:**

**RMSSUB_RTV.CONSUME (O_STATUS_CODE, O_ERROR_MESSAGE, I_MESSAGE)** - This procedure accepts an XML file in the form of an Oracle CLOB data type from the RIB (I_message) from one of the public RTV procedures whenever a create message is made available by the RIB. This message will consist of the flat RTV 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 perform desired functionality depending upon the success of the validation.

**XML Parsing:**

**PARSE_RTV (O_TEXT, O_VAL, I_XML_VAL) –** This function will be used to extract the RTV information from the RTV XML file and place that information onto an internal RTV record.

```
TYPE rtv_record IS RECORD (loc
item_loc.loc%TYPE,
                           ext_ref_no
rtv_head.ext_ref_no%TYPE,
                           item
rtv_detail.item%TYPE,
                           ret_auth_num
rtv_head.ret_auth_num%TYPE,
                           unit_qty
rtv_detail.qty_returned%TYPE,
                           supplier
rtv_head.supplier%TYPE,
                           ship_addr1
rtv_head.ship_to_add_1%TYPE,
                           ship_addr2
rtv_head.ship_to_add_2%TYPE,
                           ship_addr3
rtv_head.ship_to_add_3%TYPE,
                           state
rtv_head.state%TYPE,
                           city
rtv_head.ship_to_city%TYPE,
                           pcode
rtv_head.ship_to_pcode%TYPE,
                           country
rtv_head.ship_to_country_id%TYPE,
                           from_disp
inv_status_codes.inv_status_code%TYPE,
                           tran_date
tran_data.tran_date%TYPE,
                           unit_cost
rtv_detail.unit_cost%TYPE);
```

**Validation:**

PROCESS_RTV(O_TEXT, I_HEADER, I_DETAIL_1) – After the values are parsed for a particular RTV, RMSSUB_RTV.CONSUME will call this function, which will in turn call various functions inside RTV_SQL.APPLY_PROCESS which will call several internal functions that will perform desired functionality depending upon the success of the validation.