Retek® Merchandising System 10.1.2

Addendum to Operations Guide

Retek

The software described in this documentation is furnished under a license agreement, is the confidential information of Retek Inc., 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, and the copyright notice may not be removed without the consent of Retek Inc.

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

Printed in the United States of America.

Customer Support

Customer Support hours:

Customer Support is available 7x24x365 via e-mail, phone, and Web access.

Depending on the Support option chosen by a particular client (Standard, Plus, or Premium), the times that certain services are delivered may be restricted. Severity 1 (Critical) issues are addressed on a 7x24 basis and receive continuous attention until resolved, for all clients on active maintenance.

Contact Method	Contact Information
Internet (ROCS)	www.retek.com/support Retek's secure client Web site to update and view issues
E-mail	support@retek.com
Phone	US & Canada: 1-800-61-RETEK (1-800-617-3835) World: +1 612-587-5800 EMEA: 011 44 1223 703 444 Asia Pacific: 61 425 792 927
Mail	Retek Customer Support Retek on the Mall 950 Nicollet Mall 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 – Introduction

This addendum to the Retek Merchandising System (RMS) 10.1.2 Operations Guide contains updates to the following batch designs:

- Daily record deletion (dlyprg)
- Contract replenishment (cntrordb)
- Time hierarchy download (ftmednld)
- Upload stock count results (stkupld)

Chapter 2 - Daily record deletion [dlyprg]

Design overview

The purpose of this program is to delete all of the records in the system marked for delete (by having a record on the DAILY_PURGE table) during the day. Before deleting the records, all relations will be checked to ensure that the record can be deleted. For example, if an item has been marked for delete, this program checks that the item was not put on order later in the day. If relations are found to exist, a record is written to the DAILY_PURGE_ERROR_LOG table. Records on this table will be used to generate a report itemizing any problems found when running this program. If a record is written to the DAILY_PURGE_ERROR_LOG table, meaning that relations exist, the record will not be deleted that night.

Tables affected

TABLE	INDEX	SELECT	INSERT	UPDATE	DELETE
DAILY_PURGE	No	Yes	No	Yes	No
DAILY_PURGE_ERROR_LOG	No	No	No	Yes	Yes

Scheduling constraints

Processing Cycle: PHASE 0 (daily)

Scheduling Diagram: This program must run first to avoid processing deleted entities.

Pre-Processing: N/A

Post-Processing: N/A

Threading Scheme: N/A (single threaded)

Restart recovery

```
EXEC SQL DECLARE c_daily_purge CURSOR FOR
        SELECT      key_value,
                    table_name,
                    delete_type
        FROM        daily_purge
```

ORDER BY delete_order;

Program flow

N/A

Shared modules

N/A

Function level description

N/A

I/O specification

N/A

Technical issues

N/A

Chapter 3 - Contract replenishment – Type ‘B’ contracts [cntrordb]

Design overview

This batch module automatically creates replenished orders for type B contracts. Orders will be created for all type B contract items that are ready to have orders raised against them. Contract, item, and location information are selected from the contracting tables where production dates are ready to be met. An order will be written for each contract and all of the items and locations on the contract.

Scheduling constraints

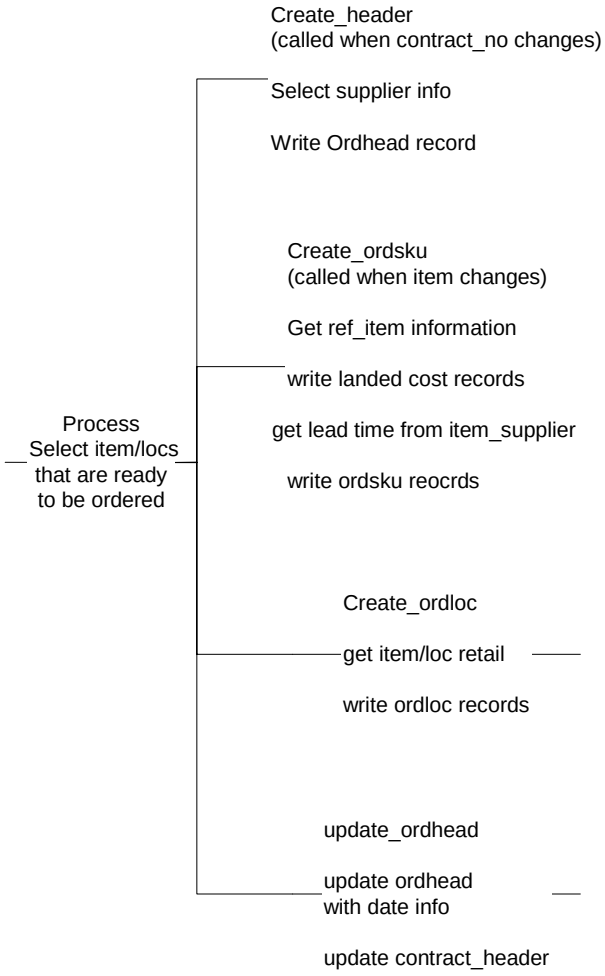
Processing Cycle:	Phase 3. Must be run after repladj
Scheduling Diagram:	N/A
Pre-Processing:	N/A
Post-Processing:	Update of system_variables, set last_cont_order_date = vdate
Threading Scheme:	Contract_no (questionable whether this is necessary)

Restart recovery

The logical unit of work is a unique contract. A commit will take place after the number of contracts processed is equal to the max counter from the restart_control table.

Multi-threading issue: If the contract_header table is sizable, the threading mechanism may have to be re-thought or omitted.

Program flow



Shared modules

N/A

Function level description

Init

System level variables are selected here including the internal RMS date, the date of the last type B contract ordering run, the minimum number of days before a contracted ready date before an order can be raised, and landed cost information. The restart/recovery process should be initialized.

Process

Contracted item/locations that are to have orders raised against them are selected from the contracting tables. These are only for contracts of type B that are within the production plan time frame. An order will be created for each contract, and the create_header function will perform the inserts into the RMS ordhead table. After the order has completed, the order header and contract header information is updated by calling the update_ordhead function.

The add_ordsku function is called to insert item level information into the RMS ordsku table. If the ELC indicator is "Y", then each item on the order should be sent to the add_cost_comp function.

The add_ordloc function is called for every record to insert location quantity and retail information into the RMS ordloc table.

create_ordhead

Insert the order header information into the ordhead table. The NEXT_ORDER_NUMBER stored function is called to retrieve the next available order number for the insert. The package CURRENCY_SQL.GET_RATE is also called to get the exchange rate to use in the insert. The insert is in the form of an insert/select. It joins with the sup_import_attr and addr tables to get the information necessary to create the order.

Add_ordsku

This function adds item level order records. If the contract does not include reference item information, the supplier's primary reference item value is retrieved for inserting into the ordering table. The item information, including ref_item, is inserted into the ordsku table. The get_dates function is called to retrieve the lead time for the item and supplier. These dates will be used to determine the not before and not after dates on ordhead. This function calls the item_defaults function.

Item_defaults

This function calls the stored procedure ORDER_SETUP_SQL.DEFAULT_ORDSKU_DOCS. This package calls defaults all the required documents that are needed when creating ordsku_records.

Update_ordsku

This function is called whenever processing for a given item, on a given contract is completed. It updates the latest_ship_date with the correct value given the latest ready_date for the item on the contract. The item could have many different ready_dates and the latest ship date should reflect the latest one.

Add_ordloc

This function first checks if an ordloc record has already been inserted. If a record exists, it updates the ordered qty. If a record does not exist, this function adds location level order records. The item/location retail value is retrieved by the get_retail function. The item/location information will be inserted into ordloc. The contracting item-level tables will be updated to reflect that the ordered quantity was ordered against that contract, item, and production plan date. The stored procedure CONTRACT_SQL.GET_UNIT_COST is called to get the unit cost to use in the insert.

Get_retail

The retail value should be selected from the appropriate item/location (item_loc) table.

Get_dates

This function determines the earliest and latest ready dates. These dates are then used when updating ordhead and ordsku.

Add_cost_comp

This function updates the landed cost tables with the new order information. The stored package ORDER_EXPENSE_SQL.INSERT_COST_COMP is called to update the system tables. The package ORDER_HTS_SQL.DEFAULT_CALC_HTS is called to default the hts information for item.

ELC_CALC_SQL.CALC_COMP is called to recalculate expenses.

Update_ordhead

The information (on ordhead) is updated with the derived not before and not after dates (determined by the item lead times).

Update_ordsku

The latest ship_date is updated based on the greatest ready_date determined by get_dates.

I/O specification

N/A

Technical issues

N/A

Chapter 4 – Time hierarchy download [ftmednld.pc)

Design overview

Currently, no extracts exist for the time dimension. So as to not maintain the calendar in multiple places (e.g. RMS, RDF and RPP), a time dimension extract is required that downloads the RMS calendar, including the following fields: year, half, quarter, month, week, day and date (in a yyyymmdd format). The downloaded information would only use the 454 calendar format. The download includes the entire calendar in the RMS. The extract must account for a fiscal year that could be different than the standard year in the calendar table. A field on the System Options table indicates the month in which the fiscal year begins. For example, if the fiscal year begins on the 3rd month, the following chart highlights how this impacts the extract. The following is a subset of the data on the Calendar table.

First Day	Year	Month	# of Wks in Month
25-OCT-99	1999	11	4
22-NOV-99	1999	12	5
27-DEC-99	2000	1	4
24-JAN-00	2000	2	4
21-FEB-00	2000	3	5
27-MAR-00	2000	4	4

If the fiscal year followed standard calendar, the first day of the year 2000 would be 27-Dec-99. However, for the fiscal year, which starts on the 3rd month, the first day of the year 2000 would be 21-FEB-00. Therefore, 20-FEB-00 would be extracted as 1999 (year), 2 (half), 4 (quarter), 12 (month), 4 (week), 7 (day), 20000220 (date). If the year followed the regular calendar, 20-FEB-00 would be extracted as 2000 (year), 1 (half), 1 (quarter), 2 (month), 4 (week), 7 (day), 20000220 (date).

Input specifications

Table-To-File

This program captures the earliest and latest dates from the calendar. It also incorporates the start_of_half_month from the system_options table when capturing the earliest date.

Driving Cursor

NA

Output specifications

Output Files

The file outputted will be named ftmehier.dat.

Record Name	Field Name	Field Type	Default Value	Description
	Year	Char(4)		The 454 year
	Half	Char(1)		The 454 half of the year, valid values are 1 or 2
	Quarter	Char(1)		The 454 quarter of the year, valid values 1-4
	Month	Char(2)		The 454 month of the year, valid values 1-12
	Week	Char(2)		The 454 week of the year, valid values 1-53
	Day	Char(1)		The 454 day of the current week, valid values 1-7
	Date	Char(8)		The date from which the 454 data was derived, in YYYYMMDD format

Function level description

main

The standard Retek main() function. Calls init(), process(), and final().

init

Initialize restart recovery by calling retek_init() and set up the output file.

format_buffer

Formats the string that will be used to write to the output file.

get_dates

Mutates output arguments with first and last calendar date captured from the calendar.first_day field. First calendar date is determined by taking the first record from the calendar table ordered in chronological order with respect to first_date. The fetched record's month_454 field must match the absolute value of the system_options.start_of_half_month field. Last calendar date is determined by capturing the first record while the calendar table records are ordered in a descending order with respect to the first_day field.

increment_date

Mutates input/output arguments to hold incremented date.

increment_454

Mutates input/output arguments to hold incremented 454 date. Also captures the calendar.no_of_weeks field from the calendar table's row whose first day field corresponds to the present date.

To determine the 454 day, month, weeks, yearly weeks, quarter, half and year, simply increment their current values by one if the corresponding date counters justify the incrementation. If any one of them turns over, reset them to 1. Note that weeks turn over when the no_of_weeks value fetched from the calendar table is no longer greater than or equal to the current week value.

init_454

Initializes argument 454 date instance's date fields.

write_fdetl

Writes data from argument to output file.

process

This function first makes a call to format_buffer. It then allocates a date struct to hold the 454 date. It then calls the init_454 function with a pointer to the 454 date struct as an argument. It then captures the earliest and latest values of calendar.first_day into local variables by calling get_dates. For each day in the date range (including the earliest and latest dates), the current calendar and 454 dates are calculated by calling increment_date and increment_454.

A record containing all of the 454 values for the date, in addition to the date itself (in YYYYMMDD format) is then written to the file by calling write_fdetl.

final

Take care of file clean up and complete the restart recovery process by calling retek_close.

Scheduling considerations

This program can be run ad hoc.

Restart/recovery

Due to the relatively small amount of processing this program performs, restart recovery will not be used. The calls to retek_init() and retek_close() are used in the program only for logging purposes (to prevent double-runs).

Chapter 5 - Upload stock count results [stkupld]

Design overview

The purpose of this batch module is to accept cycle count details from an external system. The cycle count transactions will be compared with Retek system snapshots of stock on hand at the time of the cycle count to determine the stock and/or dollar adjustments to be made. The following common functions are performed on each stock record read from the input file:

- if record exists on STAKE_SKU_LOC then update it
- if record doesn't exist on STAKE_SKU_LOC:
  - validate that item/location exists in system
  - insert a record into STAKE_SKU_LOC
  - insert stock take record into STAKE_SKU_LOC.
- if record is a pack - update/insert information on STAKE_SKU_LOC for all component items

TABLE	SELECT	INSERT	UPDATE	DELETE
stake_qty	No	Yes	No	No
stake_sku_loc	No	Yes	Yes	No
item_loc_soh	Yes	No	No	No
item_loc	Yes	No	No	No
item_master	Yes	No	No	No
wh	Yes	No	No	No
stake_head	Yes	No	No	No
stake_location	Yes	Yes	No	No
stake_prod_loc	Yes	No	No	No
v_packsku_qty	Yes	No	No	No
system_options	Yes	No	No	No

This program reads a user-created interface file of cycle counts. Files will be unique to location and cycle count ID. All records will be validated for layout. Invalid layouts will produce fatal errors. Fields will be validated for content. Invalid contents will produce non-fatal errors. Valid records will update the physical_count_qty field on STAKE_SKU_LOC for a given item/location/ cycle count combination. If the item is a pack, component items will have their component quantity added to the pack_comp_qty field on STAKE_SKU_LOC. If an item does not exist on STAKE_SKU_LOC, the item/location combination will be validated on the item/location tables and a new record will be inserted to STAKE_SKU_LOC.

Fatal errors will terminate file processing. Non-fatal errors will discontinue record processing and will write invalid record to a reject file.

File layout will be verified by interface library routines:

- get_record: validates common fields in file head record and fills structure of remaining fields that are passed from this program.
- process_dtl_ftail: called after end-of-file is reached. Will process file trailer record by validating its layout and verifying that the file record counter is set properly.

Re-run:

- If this program terminates normally, restart without recovery.
- If this program terminates abnormally, restart without recovery.

Scheduling constraints

Processing Cycle:	PHASE 3 (Daily)
Scheduling Diagram:	This program will probably be run at the start of the batch cycle during POS polling, or possibly at the end of the batch run if pending warehouse transactions exist. It can be scheduled to run multiple times throughout the day, as WMS or POS data becomes available.
Pre-Processing:	N/A
Post-Processing	N/A
Threading Scheme:	N/A

Restart recovery

The logical unit of work for the stock take upload module will be a count of discrete inventory transactions. Each record will be uniquely identified by a location and item. The logical unit of work will be defined as a number of these transaction records, determined by the commit_max_ctr field on the restart_control table.

The file records will be grouped in numbers equal to the commit_max_ctr. After all records in a given read are processed (or rejected), the restart commit logic and restart file writing logic will be called, after which the following group of file records will be read and processed. The commit logic will save the current file pointer position in the input file and any application image information (e.g. record and reject counters) and commit all database transactions. The file writing logic will append the temporary holding files to the final output files.

The commit_max_ctr field should be set to prevent excessive rollback space usage and to reduce the overhead of file I/O. The recommended commit counter setting is 10,000 records (subject to change based on experimentation).

Error handling will recognize three levels of record processing: process success, non-fatal errors, and fatal errors. Item level validation will occur on all fields before table processes are initiated. If all field-level validations return successfully, inserts and updates will be allowed. If a non-fatal error is produced, the remaining fields will be validated but the record will be rejected and written to the reject file. If a fatal error is returned, file processing will end immediately. A restart will be initiated from the file pointer position saved in the restart_bookmark string at the time of the last commit point that was reached during file processing.

Program flow

N/A

Shared modules

validate_all_numeric: interface library function.

validate_all_numeric_signed: interface library function.

valid_date: interface library function.

Function level description

init

initialize restart recovery

open input file

- file should be specified as input parameter to program

declare final output filename (used in restart_write_file logic)

open reject file (as a temporary file for restart)

- file should be specified as input parameter to program

call restart_file_init logic

assign application image array variables- line counter (g_l_rec_cnt), reject counter (g_l_rej_cnt), cycle_count, stocktake date

if fresh start (l_file_start = 0)

read file header record (get_record)

if (record type <> 'FHEAD') Fatal Error

validate file type = 'STKU'

else fseek to l_file_start location

validate head (validate_head())

validate cycle count id & cycle count date are valid on stake_head

process

loop - fread rows (equal to commit counter) of input file

if end of file encountered, decrement for loop counter and set end of file flag to true

for loop to process all records read

copy input detail structure elements to stake_sku_loc structure elements

validate elements (validate_detail())

if non-fatal error occurs write detail structure to reject file (write_to_rej_file) and continue at the top of the for-loop

update stake_sku_loc

if record doesn't exist, validate that item/location is valid

if invalid then non-fatal error -write record & continue

insert to stake_sku_loc (if display pack also insert component items)

end loop for loop to process individual records

insert structure of arrays (for valid record counter) into stake_sku_loc

restart file commit - save current input file position, and application image (cnt, cycle count & date)

restart write file function

if end of file reached then break from while loop

end outer loop to read from file

restart commit final

validate_head

if file type != 'STKU' then fatal file type error

copy stocktake_date into variable

nullpad stocktake_date

copy loc_type into variable (value will always be warehouse 'W') nullpad stocktake_dat

nullpad loc_type

copy loc_value into variable

nullpad loc_value

copy store_value, wh_value, and loc_value into variables (store will always be –1)

get cycle count for location and stocktake_date.

validate cycle count.

validate_detail

if record type != FDETL then fatal file layout error

do standard string validations - if any return non-fatal error then set non-fatal error flag to true

nullpad all fields

left shift item and qty

check that store and qty are all numeric

place decimal in qty field

ON Fatal Error

- Exit Function with -1 return code

ON Non-Fatal Error

- write out rejected record to the reject file using write_to_rej_file functionn, pass pointer to detail record structure, number of bytes in structure, and reject file pointer

I/O specification

Input File

The input file should be accepted as a runtime parameter at the command line.

Record Name	Field Name	Field Type	Description
File Header	file type record descriptor	Char(5)	hardcode 'FHEAD'
	file line identifier	Number(10)	Id of current line being processed., hardcode '000000001'
	file type	Char(4)	hardcode 'STKU'
	file create date	Date(14) YYYYMMDDHHMISS	date written by convert program
	stocktake_date	Date(14) YYYYMMDDHHMISS	stake_head.stocktake_date
	cycle count	Number(8)	stake_head.cycle_count
	loc_type	Char(1)	hardcode 'W' or 'S'
	location	Number(10)	stake_location.wh
Transaction record	file type record descriptor	Char(5)	hardcode 'FDETL'
	file line identifier	Number(10)	Id of current line being processed, internally incremented
	item type	Char(3)	hardcode 'ITM'

Record Name	Field Name	Field Type	Description
	item value	Char(25)	item id
	inventory quantity	Number(12,4)	total units or total weight
	location description	Char(30)	Where in the location the item exists. Ex: Back Stockroom or Front Window Display
File trailer	file type record descriptor	Char(5)	hardcode 'FTAIL'
	file line identifier	Number(10)	Id of current line being processed, internally incremented
	file record count	Number(10)	Number of detail records.

Reject File

The reject file should be able to be re-processed directly. The file format will therefore be identical to the input file layout. The file header and trailer records will be created by the interface library routines and the detail records will be created using the write_to_rej_file function. A reject line counter will be kept in the program and is required to ensure that the file line count in the trailer record matches the number of rejected records. A reject file will be created in all cases. If no errors occur, the reject file will consist only of a file header and trailer record and the file line count will be equal to 0.

A final reject file name, a temporary reject file name, and a reject file pointer should be declared. The reject file pointer will identify the temporary reject file. This is for the purposes of restart recovery. When a commit event takes place, the restart_write_function should be called (passing the file pointer, the temporary name and the final name). This will append all of the information that has been written to the temp file since the last commit to the final file. Therefore, in the event of a restart, the reject file will be in synch with the input file.

Error File

Standard Retek batch error handling modules will be used and all errors (fatal & non-fatal) will be written to an error log for the program execution instance. These errors can be viewed on-line with the batch error handling report.

Technical issues

N/A