PROGRAM

proccessdapcomplete.exe

LOCATION

On the server Earthquake, `f:\DAP\bin\`

DESCRIPTION

A Visual Basic.NET program that runs hourly to process the raw data in flat file format on the server Earthquake in `f:\DAP\wg` and `f:\DAP\sr`

AUTHORS

Eric Anson (eanson@tucson.ars.ag.gov)
Jason Wong (jwong@tucson.ars.ag.gov)

DEPENDENCIES

SQL Server "`DAP`" database on server Sinkhole. Daily directories in `f:\DAP\wg` and `f:\DAP\sr`. A Directory (`f:\DAP\Unprocessed`) to hold unprocessed data. A mail server, `smtp.tucson.ars.ag.gov`

REQUIRED PARAMETERS

--watershed_tag

"`wg`" for Walnut Gulch
"`sr`" for Santa Rita

--process_missing_tag

"`y`" if there are any missing daily directories (data from successive days were combined into one daily directory)
"`n`" if there are no missing daily directories

IMPORTANT SUBROUTINES/FUNCTIONS

--mdlProcessDAP.Main()

Main subroutine for the entire program. Each watershed in the database has a `LastDate` record that identifies when that watershed was last processed by this program. This subroutine queries for that `LastDate` and steps from that date to the current day to processes any daily directories that have yet to be processed.

If the daily directory from Tombstone is not yet completely uploaded, there will not be a `done.txt` (or `srdone.txt` for Santa Rita) file in the directory. The subroutine will not process the directory if this file is absent. Once the daily directory has been processed, the `LastDate` will be updated in the database

--mdlProcessDir.ProcessDir(intWatershed, strCurrentDateDir)

Processes a daily directory. Each watershed has a number of corresponding "sites" in the database. Each of these sites has a flat file that resides in a particular location within the daily directory hierarchy that is reported by the database. The data in flat files are recorded as lines of comma-separated text called "arrays". One of the programming hurdles that needed to be overcome was the possibility of a partial line at the end of the flat file due to loss of communication with the datalogger. For this reason, the last line of a flat file is not processed for the current day. Instead, this last line is recorded to a file identified by the Site ID in `f:\DAP\Unprocessed`. This subroutine processes the flat file in this manner:

It is assumed that each file begins with the literal `"8888"`. Following this literal may be the completion of a partial line. This line fragment is appended to the corresponding file in `f:\DAP\Unprocessed`. The line(s) in this file is processed before processing any lines in the current flat file. Each array in the flat file is time-stamped. Each site in the database records the timestamp of the last array processed. The subroutine checks the timestamp of each array to make sure that it is not earlier than the last recorded timestamp before the array is processed. It also prevents processing arrays whose timestamp reports the future. Each array is then checked for validity and then processed into the database.

--mdlLineOps.ProcessLine(strLine)

Processes a line/array. This function assumes the argument `strLine` is valid (validated by `mdlLineOps.ValidLine`) and returns -1 when it is processed into the database successfully. Each array type has its own routine to be parsed into the database in order to match the corresponding table(s). Maintenance arrays that have calibration information are of particular interest. Arrays that signal a calibration change will update the proper table as well as add an entry into the corresponding history table for the particular sensor/site. This is done so that when the data is calculated from the data array tables, they will be in context with regards to the calibration values. The daily maintenance data is checked with the most recent calibration/site information as a sanity check. If the values do not match, then it is possible that the calibration change array was not recorded. In this case, an e-mail warning is sent to the database administrator, who can then look at the source flat file and take the appropriate action.

--mdlRunoffDepths.CalculateRunoffDepths(intSensorType)

Gathers unprocessed runoff data, defines runoff events, calculates depths based on calibration records, and inserts results into the database. This subroutine first

makes a query for any runoff array records in the database that have yet to be classified as part of a runoff event (where the `QAQC` Boolean record is `false`). The records are sorted by sensor, then date. For each sensor, events are defined in this manner:

Looking at the timestamp of the last record, it checks to see if there exists an `array3` record one hour after the timestamp. A runoff event is defined as a group of breakpoint records where the time gap between two successive records is no more than one hour. The datalogger reports `array3` records every hour. Therefore, if an `array3` record exists one hour after the last timestamp, the entire group of records queried can be classified in at least one runoff event. Otherwise, starting from the last record, we step in descending order of timestamps to see where in the group of records we can cut off the dangling event (where the gap between successive records is greater than an hour) that we cannot process yet.

If there are still breakpoints to process, then we create a calibration factor lookup table queried from the `flume/weir` and `flumeHistory/weirHistory` tables. The runoff depths are then calculated based on this lookup table. The breakpoints are stepped through in ascending order of timestamp, and the events are defined. There may be a series of breakpoints that have the same depth (depths are rounded to the nearest hundredth of a foot). In this case, only those breakpoints which are the endpoints of a series of identical depths are marked to be inserted into the database. An artificial breakpoint of zero depth is added to the end of the event at the proper timestep (defined in the `flume/weir` tables). Event records and breakpoint records are finally put into the database (the `QAQC` Boolean record is set to `true`).

Note: There is a stored procedure within the DAP database called `insertRunoffEvent` that is used by this subroutine to obtain the newly created `eventID` to be used to link the breakpoints to the runoff event.

--mdlPrecipDepths.CalculatePrecipDepths()

Gathers unprocessed precipitation data, defines precipitation events, calculates depths based on calibration records, and inserts results into the database. This subroutine first makes a query for any precipitation array records in the database that have yet to be classified as part of a precipitation event (where the `QAQC` Boolean record is `false`). The records are sorted by sensor, then date. For each sensor, events are defined in this manner:

Looking at the timestamp of the last record, it checks to see if there exists an `array3` record one hour after the timestamp. A precipitation event is defined as a group of breakpoint records where the time gap between two successive records is no more than one hour. The datalogger reports `array3` records every hour. Therefore, if an `array3` record exists one hour after the last timestamp, the entire group of records queried can be classified in at least one precipitation event. Otherwise, starting from the last record, we step in descending order of

timestamps to see where in the group of records we can cut off the dangling event (where the gap between successive records is greater than an hour) that we cannot process yet.

If there are still breakpoints to process, then we create a calibration factor lookup table queried from the `rainGage` and `rgHistory` tables. The precipitation depths are then calculated based on this lookup table (what is recorded in the database is accumulated depth rounded to the nearest five-thousandth of an inch; the accumulation calculation is based on the delta-millivolt reading from the sensor and the calibration factor). The breakpoints are stepped through in ascending order of timestamp, and the events are defined. An artificial breakpoint of zero depth is added to the beginning of the event whose timestamp based on a linear extrapolation of the timestamps of the first two breakpoints (if there is only one breakpoint, the artificial breakpoint is one minute earlier). Event records and breakpoint records are finally put into the database (the `QAQC` Boolean record is set to `true`).

Note: There is a stored procedure within the DAP database called `insertPrecipEvent` that is used by this subroutine to obtain the newly created `eventID` to be used to link the breakpoints to the precipitation event.