Appendix 6D

Procedures for Calculating Drainage from Pipeline

Procedures for Calculating Drainage from Pipeline

Longhorn Pipeline

Introduction

Determination of degree of risk from a pipeline break depends in part on estimating the volume of product that could be released from a break anywhere along the pipeline. There are 43 block valves located along the pipeline, and their effect is to isolate any releases between valves. The inside diameter (i.d.) of the pipeline is 20" from its start in Galena Park to the Satsuma Station and 18" thereafter. Since the diameter is constant between any two valves, the problem of calculating releases is simplified somewhat to that of determining the length of pipe contributing fluid to a break at any given point.

The assumption is made that any fluid releases are dictated strictly by gravity flow of product, with no siphoning effect. Since the block valves act as no flow boundaries, this assumption is reasonable. This reduced problem can be solved by determining the gravitational flow potential at every point along the pipeline.

The methodology described below calculates the flow potential in a series of three steps or phases. All of the positive inflections (peaks, plateaus and steps) are first delineated. In phase I, incremental potentials (potentials measured only between inflections) are calculated at each inflection, in both the forward and backward directions, thereby determining flow contributions for each segment of pipeline between inflections. In phase 2 the incremental, contributions are integrated for the entire section of pipeline between valves, resulting in forward and backward integrated potentials at each inflection. Finally, in phase 3, total flow potentials are calculated for each 100 feet of pipeline, incorporating incremental potentials between inflections and integrated potentials from all other segments between valves. Finer resolution can be obtained by interpolating flow potentials within the 100-foot intervals.

Data Source

Pipeline elevation data were obtained from an Access database acquired from Longhorn Pipeline. The table LH -PROFILE contained 8363 points with elevations at selected footage locations along the pipeline. These points were apparently derived from digitization of contour crossings from 1:24,000 scale topographic maps. Elevation data were spot checked against topographic maps and no discrepancies were found.

Elevations of the 100-foot intervals were calculated by interpolating the profile data. Elevations for all valves were also calculated by interpolating the profile data.

Software

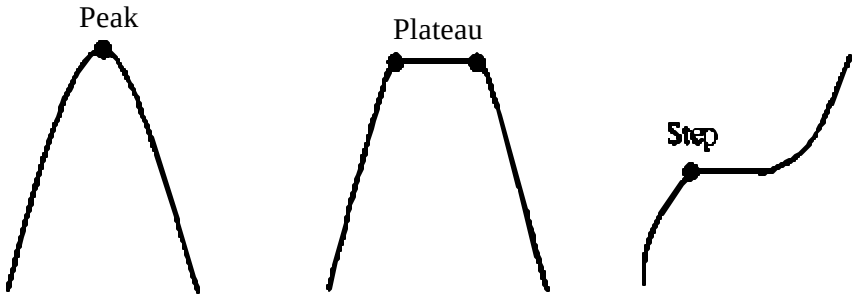
All calculations were performed using AWK scripts. AWK is a powerful list processing language well suited to the types of calculations performed here. The advantages are ease of programming and portability AWK is a standard application in

UNIX operating systems, and complete GNU implementations are available in the public domain for IBM-compatible PCs.

A 3-phased approach was taken in the solution of the problem. This significantly improved the amount of time required to perform the calculations and permitted simplified checking of accuracy of the model.

Inflections

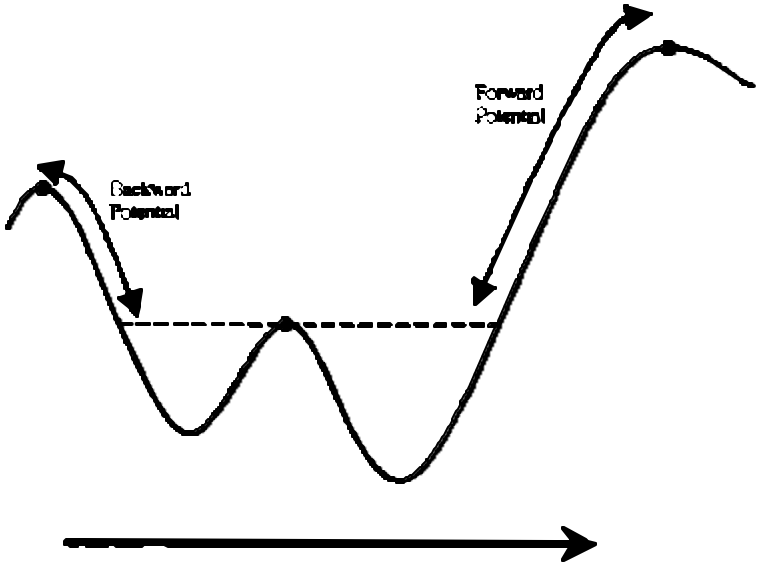
In the strictest sense, anywhere a line changes direction is an inflection. The sense used here is restricted to those inflections where the line changes from a positive to a negative direction (turns downward) or a positive to a neutral direction (flattens). The changes in direction are evaluated in both upstream and downstream directions. The points of inflection, then, delineate peaks, plateaus, and steps. Peaks and plateaus are significant, because they have a higher potential than the surrounding pipeline segments, and thus act to restrain flow.



Phase Calculations

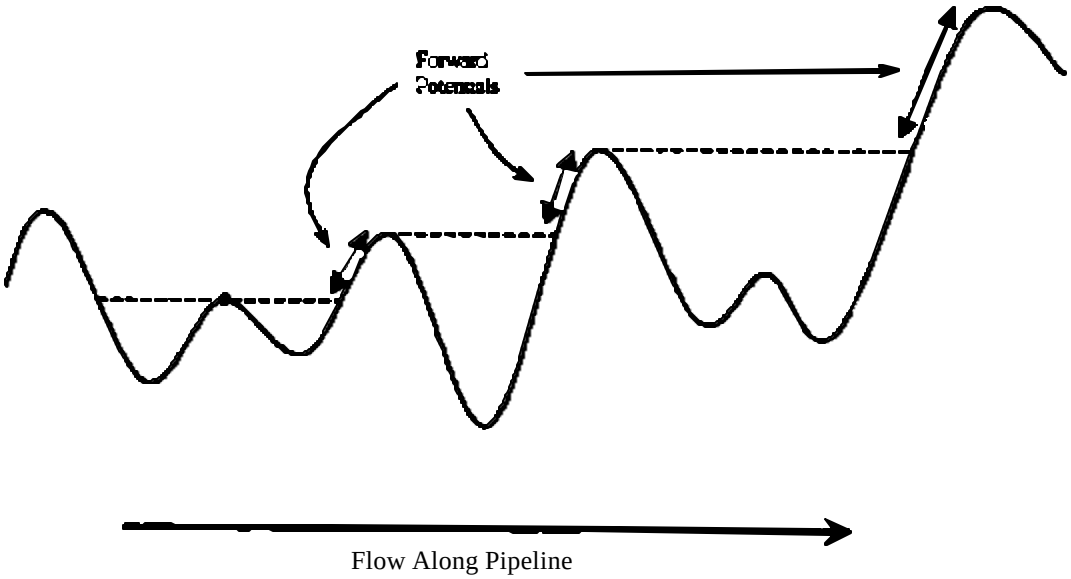
Phase I calculations are concerned with calculating potentials between inflections. For each point of inflection, the forward and backward potentials are calculated using the 100-foot elevation points, interpolated as necessary. Forward potentials are added for each 100-foot segment, up to the next inflection point, higher than or level with the inflection point. Backward potentials are calculated in the same manner back to the previous inflection point. An AWK script that performs these Phase I calculations is shown in Appendix A.

The algorithm can be described simply: starting at an inflection point, if the next point is higher or level, add the distance to the next point to the forward potential. Because the calculation is performed for a segment between inflections, there is no concern with flow restrictions across a peak, and the calculations can proceed from point to point.



Phase 2 Calculation

Phase 2 calculations are concerned with summation at each inflection of potentials between block valves. For each point of inflection, the forward and backward potentials are calculated using inflections at the same height or higher, stepping up as new peaks are encountered, until a maximum inflection or block valve is encountered. An AWK script that performs these Phase 2 calculations is shown in Appendix A.



The algorithm can be described simply: Starting at an inflection point, the forward potential is the sum of all potentials at downstream inflections of equal or increasingly greater height until either a maximum height (regional peak) or block valve is encountered. The back potential is calculated in a similar manner, and the sum of the two is the total potential at the inflection.

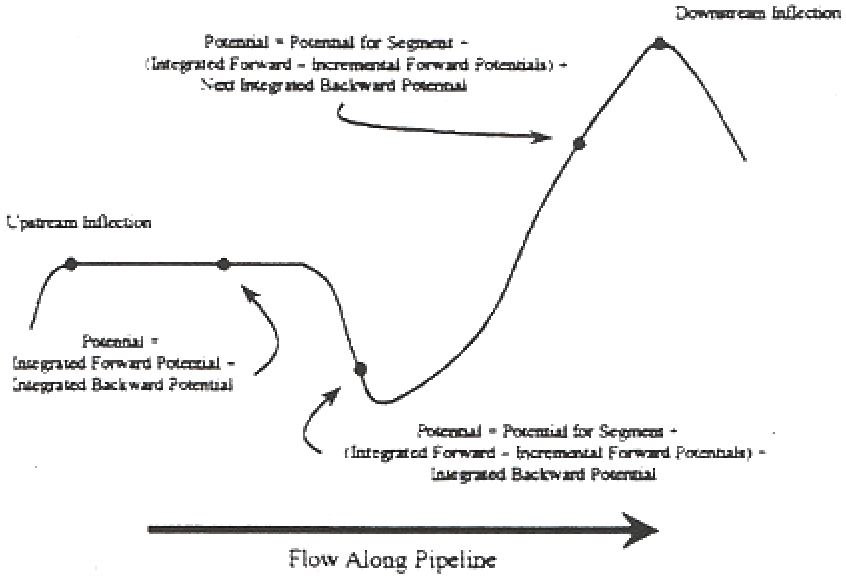
Phase 3 Calculation

The final Phase 3 calculations are concerned with calculating potentials at regular intervals along the pipeline - in this case an interval of 100 feet. The potential for each point is calculated in a manner similar to that used in Phase 1, and initially incorporates only the potential associated with the pipeline segment between inflections. The potentials from the bounding inflections are incorporated according to the following logic:

(1) If the elevation of the point is greater than the elevation of the upstream inflection point (the back point) then the potential at the point is equal to the calculated (incremental) potential plus the integrated forward potential (from Phase 2) of the inflection minus the incremental forward potential (from Phase 1) of the inflection plus the integrated backward potential of the next downstream inflection.

(2) If the elevation of the point is equal to the elevation of the upstream inflection point, then the potential at the point is equal to the sum of the integrated forward and back-ward potentials at the inflection.

(3) If the elevation is less than the elevation of the upstream inflection then the potential at the point is equal to the calculated (incremental) potential plus the integrated forward potential (from Phase 2) of the inflection minus the incremental forward potential (from Phase 1) of the inflection plus the



These rules are simplified where valves are encountered (no flow boundaries), since contributing upstream and downstream flows beyond valves are zero.

An AWK script that performs these Phase 3calculations is shown in Appendix A.

Results

The results of the calculations are summarized in a table of points, defined by their footage location every 100 feet along the pipeline as measured from the beginning at Galena Park (in excess of 36,600 individual points). Each point is associated with a flow potential value expressed in feet. This value represents the length of pipe that is calculated to drain to the point if product is released. This table was used to create a GIS layer (represented as an ArcView shapefile), which allows the information to be queried spatially.

Attachment A

Procedure for Estimating Velocity of Draining Liquid in the Pipeline

Attachment A

Estimating Flow Velocities at Leak Locations along the Pipeline

Since the maximum drained volume often comes from a long stretch of pipe, the velocity of the draining liquid in the pipeline can be estimated to project the approximate time needed to fully drain the line. The velocity of the liquid draining from the pipeline segment immediately downstream of a leak site was estimated using the following procedure:

- The high point along the line between the location of the leak and the nearest downstream valve was found from the pipeline elevation profile.
- The distance between the high point and the leak point was defined as the difference in stationing values of the two locations. This distance was assumed to be the total length of pipe, L, from which drainage occurred.
- The head driving the drainage was assumed to be only due to the difference in elevation, ΔE, of the high point and the leak site.
- The system was assumed to be a straight pipe of length L with a slope of $\Delta E/L$.
- The velocity was estimated using the Fanning equation, shown below:

$$V^2 = \frac{\Delta P g_c D}{2fL\rho}$$

where

V = volumetric fluid velocity, feet/sec
ΔP = pressure drop due to friction (head loss), lb/ft^2
g_c = 32.17 lb(mass) ft / lb(force) sec^2
D = inside diameter of pipe, ft
f = friction factor
L = length of pipe, ft
ρ = fluid density, lb/ft^3

- The friction factor, f, is obtained from a chart that shows the friction factor curves for a variety of pipe materials as a function of the Reynolds Number, N_{Re}, where

$$N_{Re} = \frac{DV\rho}{\mu}$$

where μ = absolute viscosity of the liquid, lb / ft sec

- The estimated liquid draining velocity through the pipe is determined through an iterative calculation:

  - A fluid velocity is first estimated
  - The value of N_{Re} is calculated
  - Using the N_{Re}, the friction factor is found from the friction factor chart
  - The velocity is calculated from the Fanning equation
  - If the calculated velocity is not equal to the estimated velocity, a new velocity is assumed, and the procedure is repeated until the calculated velocity matches the assumed velocity.

- The length of pipe to be traversed by draining liquid is equal to the drained volume divided by the amount of liquid contained in a 1-foot length of pipe.

- The total drain time is then estimated as the length of pipe traversed divided by the liquid velocity.

- The volume of liquid that would drain from the pipeline in 2 hours was then estimated as:

Volume (gal/2-hr) = velocity (fps) x 7200 (sec/2-hr) x 12.495 (gal/ft)

Attachment B

```
#     Phase 1
#     Input file is the inflection set

BEGIN {
      FS = ","

      getline
      startx = $2
      starty = $3
      start_type = $5

}
{
#     Read the inflection points one by one (valves are
#       just a special case of inflection)

#     Read the interval points between current inflection
#       and the next inflection into an array

#     Find the crossing point of the inflection elevation
#       with the interval points, interpolating if necessary

#     The spill spillen at the inflection is the spillen between
#       the next inflection and the crossing point.  If there
#       is no crossing point the spill spillen is zero.  The
#       measures for each point between this croosing and the next
#       inflection is the spill spillen, which will approach zero
#       at the next inflection.

#     Step through the array one point at a time, find the
#       2 crossing points, and calculate the spill spillens

#     When a minimum is reached, the spill spillen is the entire
#       spillen  (or, when the soil spillen is the entire spillen,
#       a minimum has been reached)

#     This will result in incremental spill spillens for each
#       segment between inflections.  Spill spillens for any point
#       between valves will sum the spillens of all enclosed
#       segments.

      endx = $2
      endy = $3
      end_type = $5

      getline < "All_interval.txt"
      ptx = $1
      pty = $4

      while (ptx < startx)
      {
            getline < "All_interval.txt"
                  ptx = $1
                  pty = $4
      }

      x_array[0] = ptx
```

```
      y_array[0] = pty

      count = 1
      while (ptx <= endx)
      {
            if (getline < "All_interval.txt" > 0)
            {
                  # the array consists only of the interval
                  # points between the inflections
                  ptx = $1
                  if (ptx <= endx)
                  {
                        x_array[count] = ptx
                        y_array[count] = $4
                        count ++
                  }
            }
            else
            { ptx = endx +1 }
      }
      array_len = count - 1

      close "All_interval.txt"


# Forward calculations

      calc_length(x_array,y_array,array_len,startx,starty,endx,endy)

      if (start_type == "V")
      {
            #print "Valve"
      }
      #print "Forward calculation: "startx","spillen
      print "F,"startx","spillen

# if the spillen after calculations is zero, then a backward model needs to
#  be calculated - this is accomplished by reversing the array indeices and
# running the same algorithm over the selected segment

      reverse the arrays
      for (i = 0; i <= array_len; i++)
      {
            bx_array[array_len - i] = -(x_array[i])
            by_array[array_len - i] = y_array[i]
      }

      nendx = -endx
      nstartx = -startx

      calc_length(bx_array,by_array,array_len,nendx,endy,nstartx,starty)

      #print "Backward calculation: "endx","spillen
      print "B,"endx","spillen
      if (end_type == "V")
      {
            #print "Valve"
```

```
      }
      #print "........................................."

#------------------------------

      startx = endx
      starty = endy
      start_type = end_type
}

##############################################################################

function calc_length(x_array,y_array,array_len,startx,starty,endx,endy)
{

      spillen = 0
      # upflag is: 0 for down, 1 for level, and 2 for up
      upflag = -1
      for (i = 0; i < count; i++)
      {
            if (y_array[i] < starty)
            {
                  upflag = 0
            }
            # must check for the case of a downturn before a
            # level run with no following upturn
            if (y_array[i] == starty && upflag != 0)
            {
                  # need to make sure equal points
                  # are contiguous
                  if (i > 0 && (y_array[i-1] == starty))
                  {
                    spillen = spillen + x_array[i] - x_array[i-1]
                  }
                  if (i == 0)
                  {
                    spillen = x_array[i] - startx
                  }
                  upflag = 1
            }
            if (y_array[i] > starty)
            {
                  if (i > 0)
                  {
                    interpy = (starty - y_array[i-1]) / (y_array[i] -
y_array[i-1])
                    xdiff = x_array[i] - x_array[i-1]
                    interpx = x_array[i-1] + (xdiff * interpy)
                    spillen = spillen + endx - interpx
                  }
                  else
                  {
                    spillen = spillen + endx - startx
                  }
                  # if it goes up, it can't come back down
                  # before the next inflection
                  upflag = 2
```

```
                    break
              }
        }


        if (upflag == 1)
        {
              spillen = spillen + endx - x_array[array_len]
        }

        return spillen
}
```

```
# Phase 2
# No input file required (use dummy)
#
BEGIN {
      FS = ","
      indx = 0
      place = 0
      NREC = 0
      int_farray[0] = 0
      int_barray[0] = 0

#-------------------------------------------------------


      while (getline < "comb_inflections.txt" > 0) {
            while (++NREC < place)
            {
                  getline < "comb_inflections.txt"

            }

      station = $2
      elevation = $3
      type = $5
      forward = $6
      backward = $7

      if (type == "V")
      {
            x_array[indx] = station
            y_array[indx] = elevation
            f_array[indx] = forward
            b_array[indx] = backward

            #print "New Valve"

      }

      type = ""
      while (type != "V")
      {
            getline < "comb_inflections.txt"
            NREC++
            station = $2
            elevation = $3
            type = $5
            forward = $6
            backward = $7

            x_array[++indx] = station
            y_array[indx] = elevation
            f_array[indx] = forward
            b_array[indx] = backward

            if (type == "V") break
      }
```

```
# at this point we have an array filled from
# valve to valve
# we know our total number of reads by NR

array_len = indx

# Now, we need to step up the line until we reach the
# next highest point, at which point we stop adding
# spill length until we find a higher point

#forward pass
for (i=0; i<array_len; i++)
{
      int_farray[i] = f_array[i]
      highpoint = y_array[i]
      for (j=i+1; j<array_len;j++)
      {
        if (highpoint <= y_array[j])
        {
            int_farray[i] = int_farray[i] + f_array[j]
            highpoint = y_array[j]
        }
      }
}
# guarantee no forward flow at the end valve
int_farray[array_len] = 0

# Now, we need to step down the line until we reach the
# next highest point, at which point we stop adding
# spill length until we find a higher point

#backward pass
for (i=array_len; i>0; i--)
{
      int_barray[i] = b_array[i]
      highpoint = y_array[i]
      # Don't iterate to j==0, b_array[0] is zero
      # for this segment
      for (j=i-1; j>0;j--)
      {
        if (highpoint <= y_array[j])
        {
            int_barray[i] = int_barray[i] + b_array[j]
            highpoint = y_array[j]
        }
      }
}

for (i=0; i<=array_len; i++)
{
      vflag = "I"
      if (i == 0) {vflag = "S"}
      if (i == array_len) {vflag = "E"}

print x_array[i]","y_array[i]","int_farray[i]","int_barray[i]","vflag
}
```

```
        # now, reset the file
        place = NREC
        NREC = 0

        # reset the integration arrays
        for (i=0; i<=array_len; i++0)
        {
              int_farray[i] = 0
              int_barray[i] = 0
        }
        indx = 0
        if (getline < "comb_inflections.txt" == 0) exit
        close ("comb_inflections.txt")

}
}
```

```
# Phase 3 model
# Input is the interval file
#
BEGIN {
      FS = ","

      # Load an array with the inflection data
      indx = 0
      while (getline < "comb_inflections.txt" > 0)
      {
        inf_x[indx] = $2
        inf_y[indx] = $3
        inf_type[indx] = $5
        incr_frwd[indx] = $6
        incr_bkwd[indx] = $7
        inf_frwd[indx] = $8
        inf_bkwd[indx] = $9
        inf_elev[indx++] = $10
      }
      inf_arraylen = indx - 1
      inf_indx = 0

      lastxpoint = 0
      lastypoint = 0
      lastflag = 0

      pt_indx = 0

      startflag = 0
}
{
      # Read the interval points one at a time
      x = $1
      y = $4

      if (x >= inf_x[inf_indx])
      {
         # Add the inflection to the front
         if (pt_indx == 0)
         {
            #print inf_type[inf_indx]
            if (x == inf_x[inf_indx])
            {
               x_array[0] = x
               y_array[0] = y
               s_array[0] = 0
               pt_indx++
            }
            else
            {

               x_array[0] = inf_x[inf_indx]
               y_array[0] = inf_y[inf_indx]
               s_array[0] = 0
               pt_indx++

               if (lastflag == 1)
```

```
                {
                   x_array[pt_indx] = lastxpoint
                   y_array[pt_indx] = lastypoint
                   s_array[pt_indx] = 0
                   pt_indx++
                   lastflag = 0
                }
                if ( x < inf_x[inf_indx + 1])
                {
                   x_array[pt_indx] = x
                   y_array[pt_indx] = y
                   s_array[pt_indx] = 0
                   pt_indx++
                }
             }
          }
          if (x >= inf_x[inf_indx + 1])
          {
             if (x > inf_x[inf_indx + 1])
             {
                  lastxpoint = x
                lastypoint = y
                lastflag = 1
             }

             x_array[pt_indx] = inf_x[inf_indx + 1]
             y_array[pt_indx] = inf_y[inf_indx + 1]
             s_array[pt_indx] = 0
             pt_indx++
             calculate(x_array,y_array,s_array,pt_indx)

             pt_indx = 0
             inf_indx++
          }
          else
          {
             x_array[pt_indx] = x
             y_array[pt_indx] = y
             s_array[pt_indx] = 0
             pt_indx++
          }
       }
}
# Finally, catch the last segment
END {
       x_array[pt_indx] = inf_x[inf_indx + 1]
       y_array[pt_indx] = inf_y[inf_indx + 1]
       s_array[pt_indx] = 0
       pt_indx++

       calculate(x_array,y_array,s_array,pt_indx)
}

function calculate(x_array,y_array,s_array,array_len)
{

       # This is simplified because we are guaranteed to
```

```
      # be between inflections

      # Forward pass
      for (i=0; i<(array_len-1); i++)
      {
         for (j=i+1; j<array_len; j++)
         {
            if (y_array[i] <= y_array[j])
            {
               s_array[i] = s_array[i] + (x_array[j] - x_array[j-1])
            }

         }
        }

      # Backward pass
      for (i=(array_len - 1); i>0; i--)
      {
         for (j=i-1; j>=0; j--)
         {
            if (y_array[i] <= y_array[j])
            {
               s_array[i] = s_array[i] + (x_array[j+1] - x_array[j])
            }
         }
        }

      for (i=startflag; i<array_len; i++)
      {
         if (y_array[i] > inf_elev[inf_indx])
         {
            s_array[i] = s_array[i] + (inf_frwd[inf_indx] -
incr_frwd[inf_indx]) + inf_bkwd[inf_indx + 1]
         }
         else if (y_array[i] == inf_elev[inf_indx])
         {
            if (inf_type[inf_indx] != "V")
            {
            s_array[i] = inf_frwd[inf_indx] + inf_bkwd[inf_indx]
            }
            else
            {
            s_array[i] = inf_frwd[inf_indx]
            }
         }
         else if (y_array[i] < inf_elev[inf_indx])
         {
            if (inf_type[inf_indx] != "V")
            {
            s_array[i] = s_array[i] + (inf_frwd[inf_indx]-
incr_frwd[inf_indx]) + inf_bkwd[inf_indx]
            }
            else
            {
            s_array[i] = s_array[i] + (inf_frwd[inf_indx]-
incr_frwd[inf_indx])
            }
```

```
         }

         print x_array[i]","y_array[i]","s_array[i]
      }
      startflag = 1

      return
}
```