

# VPM-B Program Update Explanation

This post explains the changes to the original VPM code for implementing the Boyle's Law compensation algorithm (VPM-B).

All changes to the original code are highlighted in yellow in the program code for VPM-B that follows in this pdf file.

The main difference between the original VPM algorithm and this new one is that during the decompression calculation, the allowable gradients for each compartment are modified (reduced) at each stop by the Boyle's Law compensation subroutine. In the original VPM, the allowable gradients remained fixed throughout each deco schedule iteration and were only modified by the Critical Volume subroutine for the next deco schedule iteration.

Summary of changes:

1. In the main program, the variable Deco\_Ceiling\_Depth has been changed to Ascent\_Ceiling\_Depth. This is because there are now two (2) distinct subroutines for determining the deco ceiling. For the initial ascent to the first stop, this is now calculated by a new subroutine called CALC\_ASCENT\_CEILING which returns the value of the new variable Ascent\_Ceiling\_Depth. This subroutine does the same thing as the former CALC\_DECO\_CEILING only variable name has been changed to Ascent\_Ceiling\_Depth.
2. There is still a sub called CALC\_DECO\_CEILING, but it now uses the values of two new array variables Deco\_Gradient\_He and Deco\_Gradient\_N2 which are modified (reduced) at each deco stop by the new BOYLES\_LAW\_COMPENSATION subroutine.
3. In the main program you will see that where the former CALC\_DECO\_CEILING subroutine was called, it now calls CALC\_ASCENT\_CEILING which determines the first deco stop only.
4. In the Deco Stop Loop Block within Critical Volume Loop (main program) and in the Deco Stop Loop Block for Final Decompression Schedule (main program), you will see that subroutine BOYLES\_LAW\_COMPENSATION is called just before the DECOMPRESSION\_STOP subroutine in each case. This is to compute the adjustment (reduction) of Deco\_Gradient\_He and Deco\_Gradient\_N2 at each stop due to the Boyle expansion of a gas bubble with reduction in pressure.
5. The DECOMPRESSION\_STOP subroutine has been changed to use the new Deco\_Gradient\_He and Deco\_Gradient\_N2 variables now instead of the former Allowable\_Gradient\_He and Allowable\_Gradient\_N2 variables for each deco stop. The Allowable\_Gradient\_He and Allowable\_Gradient\_N2 variables are still used in the program, but only when setting the first stop (initial ascent ceiling).
6. A completely new subroutine BOYLES\_LAW\_COMPENSATION is in the program. This new subroutine uses many of the common variable and arrays and the subroutine RADIUS\_ROOT\_FINDER which are already in the program. The new subroutine solves the cubic equation to determine the expansion of a gas bubble due to reduction in pressure between each deco stop. With a Boyle expansion of a gas bubble, the gradient for bubble formation changes (gets reduced) because the Laplace condition,  $\text{Deco\_Gradient} = 2 \cdot \gamma / \text{radius}$ , changes (gets smaller) as the bubble gets larger.
7. Other than those changes, the main program and all the other subroutines are the same.
8. The new program tends to be somewhat conservative and the Critical Radii may need to be dialed down a little (try for example 0.55 to 0.7 for N2 and 0.45 to 0.6 for He). Also the Critical Volume parameter Lambda may need to be reduced a little to say 6500.

# PROGRAM VPM-B

```
C=====
C   Varying Permeability Model (VPM) Decompression Program in FORTRAN
C   with Boyle's Law compensation algorithm (VPM-B)
C
C   Author:   Erik C. Baker
C
C   "DISTRIBUTE FREELY - CREDIT THE AUTHORS"
C
C   This program extends the 1986 VPM algorithm (Yount & Hoffman) to include
C   mixed gas, repetitive, and altitude diving.  Developments to the algorithm
C   were made by David E. Yount, Eric B. Maiken, and Erik C. Baker over a
C   period from 1999 to 2001.  This work is dedicated in remembrance of
C   Professor David E. Yount who passed away on April 27, 2000.
C
C   Notes:
C   1.  This program uses the sixteen (16) half-time compartments of the
C       Buhlmann ZH-L16 model.  The optional Compartment 1b is used here with
C       half-times of 1.88 minutes for helium and 5.0 minutes for nitrogen.
C
C   2.  This program uses various DEC, IBM, and Microsoft extensions which
C       may not be supported by all FORTRAN compilers.  Comments are made with
C       a capital "C" in the first column or an exclamation point "!" placed
C       in a line after code.  An asterisk "*" in column 6 is a continuation
C       of the previous line.  All code, except for line numbers, starts in
C       column 7.
C
C   3.  Comments and suggestions for improvements are welcome.  Please
C       respond by e-mail to:  EBaker@se.aeieng.com
C
C   Acknowledgment:  Thanks to Kurt Spaugh for recommendations on how to clean
C   up the code.
C=====
C   IMPLICIT NONE
C=====
C   LOCAL VARIABLES - MAIN PROGRAM
C=====
C   CHARACTER M*1, OS_Command*3, Word*7, Units*3
C   CHARACTER Line1*70, Critical_Volume_Algorithm*3
C   CHARACTER Units_Word1*4, Units_Word2*7, Altitude_Dive_Algorithm*3
C
C   INTEGER I, J                                !loop counters
C   INTEGER*2 Month, Day, Year, Clock_Hour, Minute
C   INTEGER Number_of_Mixes, Number_of_Changes, Profile_Code
C   INTEGER Segment_Number_Start_of_Ascent, Repetitive_Dive_Flag
C
C   LOGICAL Schedule_Converged, Critical_Volume_Algorithm_Off
C   LOGICAL Altitude_Dive_Algorithm_Off
C
C   REAL Ascent_Ceiling_Depth, Deco_Stop_Depth, Step_Size
C   REAL Sum_of_Fractions, Sum_Check
C   REAL Depth, Ending_Depth, Starting_Depth
C   REAL Rate, Rounding_Operation1, Run_Time_End_of_Segment
C   REAL Last_Run_Time, Stop_Time, Depth_Start_of-Deco_Zone
C   REAL Rounding_Operation2, Deepest_Possible_Stop_Depth
C   REAL First_Stop_Depth, Critical_Volume_Comparison
C   REAL Next_Stop, Run_Time_Start_of-Deco_Zone
C   REAL Critical_Radius_N2_Microns, Critical_Radius_He_Microns
C   REAL Run_Time_Start_of_Ascent, Altitude_of_Dive
C   REAL Deco_Phase_Volume_Time, Surface_Interval_Time
C   REAL Pressure_Other_Gases_mmmHg
C=====
C   LOCAL ARRAYS - MAIN PROGRAM
C=====
C   INTEGER Mix_Change(10)
```

```

REAL Fraction_Oxygen(10)
REAL Depth_Change (10)
REAL Rate_Change(10), Step_Size_Change(10)
REAL Helium_Half_Time(16), Nitrogen_Half_Time(16)
REAL He_Pressure_Start_of_Ascent(16)
REAL N2_Pressure_Start_of_Ascent(16)
REAL He_Pressure_Start_of-Deco_Zone(16)
REAL N2_Pressure_Start_of-Deco_Zone(16)
REAL Phase_Volume_Time (16)
REAL Last_Phase_Volume_Time(16)
C=====
C   GLOBAL CONSTANTS IN NAMED COMMON BLOCKS
C=====
      REAL Water_Vapor_Pressure
      COMMON /Block_8/ Water_Vapor_Pressure

      REAL Surface_Tension_Gamma, Skin_Compression_GammaC
      COMMON /Block_19/ Surface_Tension_Gamma, Skin_Compression_GammaC

      REAL Crit_Volume_Parameter_Lambda
      COMMON /Block_20/ Crit_Volume_Parameter_Lambda

      REAL Minimum-Deco_Stop_Time
      COMMON /Block_21/ Minimum-Deco_Stop_Time

      REAL Regeneration_Time_Constant
      COMMON /Block_22/ Regeneration_Time_Constant

      REAL Constant_Pressure_Other_Gases
      COMMON /Block_17/ Constant_Pressure_Other_Gases

      REAL Gradient_Onset_of_Imperm_Atm
      COMMON /Block_14/ Gradient_Onset_of_Imperm_Atm
C=====
C   GLOBAL VARIABLES IN NAMED COMMON BLOCKS
C=====
      INTEGER Segment_Number
      REAL Run_Time, Segment_Time
      COMMON /Block_2/ Run_Time, Segment_Number, Segment_Time

      REAL Ending_Ambient_Pressure
      COMMON /Block_4/ Ending_Ambient_Pressure

      INTEGER Mix_Number
      COMMON /Block_9/ Mix_Number

      REAL Barometric_Pressure
      COMMON /Block_18/ Barometric_Pressure

      LOGICAL Units_Equal_Fsw, Units_Equal_Msw
      COMMON /Block_15/ Units_Equal_Fsw, Units_Equal_Msw

      REAL Units_Factor
      COMMON /Block_16/ Units_Factor
C=====
C   GLOBAL ARRAYS IN NAMED COMMON BLOCKS
C=====
      REAL Helium_Time_Constant(16)
      COMMON /Block_1A/ Helium_Time_Constant

      REAL Nitrogen_Time_Constant(16)
      COMMON /Block_1B/ Nitrogen_Time_Constant

      REAL Helium_Pressure(16), Nitrogen_Pressure(16)
      COMMON /Block_3/ Helium_Pressure, Nitrogen_Pressure

```

```

REAL Fraction_Helium(10), Fraction_Nitrogen(10)
COMMON /Block_5/ Fraction_Helium, Fraction_Nitrogen

REAL Initial_Critical_Radius_He(16)
REAL Initial_Critical_Radius_N2(16)
COMMON /Block_6/ Initial_Critical_Radius_He,
*       Initial_Critical_Radius_N2

REAL Adjusted_Critical_Radius_He(16)
REAL Adjusted_Critical_Radius_N2(16)
COMMON /Block_7/ Adjusted_Critical_Radius_He,
*       Adjusted_Critical_Radius_N2

REAL Max_Crushing_Pressure_He(16), Max_Crushing_Pressure_N2(16)
COMMON /Block_10/ Max_Crushing_Pressure_He,
*       Max_Crushing_Pressure_N2

REAL Surface_Phase_Volume_Time(16)
COMMON /Block_11/ Surface_Phase_Volume_Time

REAL Max_Actual_Gradient(16)
COMMON /Block_12/ Max_Actual_Gradient

REAL Amb_Pressure_Onset_of_Imperm(16)
REAL Gas_Tension_Onset_of_Imperm(16)
COMMON /Block_13/ Amb_Pressure_Onset_of_Imperm,
*       Gas_Tension_Onset_of_Imperm
C=====
C   NAMELIST FOR PROGRAM SETTINGS (READ IN FROM ASCII TEXT FILE)
C=====
C   NAMELIST /Program_Settings/ Units, Altitude_Dive_Algorithm,
*       Minimum-Deco_Stop_Time, Critical_Radius_N2_Microns,
*       Critical_Radius_He_Microns, Critical_Volume_Algorithm,
*       Crit_Volume_Parameter_Lambda,
*       Gradient_Onset_of_Imperm_Atm,
*       Surface_Tension_Gamma, Skin_Compression_GammaC,
*       Regeneration_Time_Constant, Pressure_Other_Gases_mmHg
C=====
C   ASSIGN HALF-TIME VALUES TO BUHLMANN COMPARTMENT ARRAYS
C=====
C   DATA Helium_Half_Time(1)/1.88/,Helium_Half_Time(2)/3.02/,
*       Helium_Half_Time(3)/4.72/,Helium_Half_Time(4)/6.99/,
*       Helium_Half_Time(5)/10.21/,Helium_Half_Time(6)/14.48/,
*       Helium_Half_Time(7)/20.53/,Helium_Half_Time(8)/29.11/,
*       Helium_Half_Time(9)/41.20/,Helium_Half_Time(10)/55.19/,
*       Helium_Half_Time(11)/70.69/,Helium_Half_Time(12)/90.34/,
*       Helium_Half_Time(13)/115.29/,Helium_Half_Time(14)/147.42/,
*       Helium_Half_Time(15)/188.24/,Helium_Half_Time(16)/240.03/
C   DATA Nitrogen_Half_Time(1)/5.0/,Nitrogen_Half_Time(2)/8.0/,
*       Nitrogen_Half_Time(3)/12.5/,Nitrogen_Half_Time(4)/18.5/,
*       Nitrogen_Half_Time(5)/27.0/,Nitrogen_Half_Time(6)/38.3/,
*       Nitrogen_Half_Time(7)/54.3/,Nitrogen_Half_Time(8)/77.0/,
*       Nitrogen_Half_Time(9)/109.0/,Nitrogen_Half_Time(10)/146.0/,
*       Nitrogen_Half_Time(11)/187.0/,Nitrogen_Half_Time(12)/239.0/,
*       Nitrogen_Half_Time(13)/305.0/,Nitrogen_Half_Time(14)/390.0/,
*       Nitrogen_Half_Time(15)/498.0/,Nitrogen_Half_Time(16)/635.0/
C=====
C   OPEN FILES FOR PROGRAM INPUT/OUTPUT
C=====
C   OPEN (UNIT = 7, FILE = 'VPMDECO.IN', STATUS = 'UNKNOWN',
*       ACCESS = 'SEQUENTIAL', FORM = 'FORMATTED')
C   OPEN (UNIT = 8, FILE = 'VPMDECO.OUT', STATUS = 'UNKNOWN',
*       ACCESS = 'SEQUENTIAL', FORM = 'FORMATTED')
C   OPEN (UNIT = 10, FILE = 'VPMDECO.SET', STATUS = 'UNKNOWN',
*       ACCESS = 'SEQUENTIAL', FORM = 'FORMATTED')
C=====

```

```

C      BEGIN PROGRAM EXECUTION WITH OUTPUT MESSAGE TO SCREEN
C=====
      OS_Command = 'CLS'
      CALL SYSTEMQQ (OS_Command)                !Pass "clear screen" command
      PRINT *, ' '                               !to MS operating system
      PRINT *, 'PROGRAM VPMDECO'
      PRINT *, ' '                               !asterisk indicates print to screen
C=====
C      READ IN PROGRAM SETTINGS AND CHECK FOR ERRORS
C      IF THERE ARE ERRORS, WRITE AN ERROR MESSAGE AND TERMINATE PROGRAM
C=====
      READ (10,Program_Settings)

      IF ((Units .EQ. 'fsw').OR.(Units .EQ. 'FSW')) THEN
        Units_Equal_Fsw = (.TRUE.)
        Units_Equal_Msw = (.FALSE.)
      ELSE IF ((Units .EQ. 'msw').OR.(Units .EQ. 'MSW')) THEN
        Units_Equal_Fsw = (.FALSE.)
        Units_Equal_Msw = (.TRUE.)
      ELSE
        CALL SYSTEMQQ (OS_Command)
        WRITE (*,901)
        WRITE (*,900)
        STOP 'PROGRAM TERMINATED'
      END IF

      IF ((Altitude_Dive_Algorithm .EQ. 'ON') .OR.
*        (Altitude_Dive_Algorithm .EQ. 'on')) THEN
        Altitude_Dive_Algorithm_Off = (.FALSE.)
      ELSE IF ((Altitude_Dive_Algorithm .EQ. 'OFF') .OR.
*        (Altitude_Dive_Algorithm .EQ. 'off')) THEN
        Altitude_Dive_Algorithm_Off = (.TRUE.)
      ELSE
        WRITE (*,902)
        WRITE (*,900)
        STOP 'PROGRAM TERMINATED'
      END IF

      IF ((Critical_Radius_N2_Microns .LT. 0.2) .OR.
*        (Critical_Radius_N2_Microns .GT. 1.35)) THEN
        CALL SYSTEMQQ (OS_Command)
        WRITE (*,903)
        WRITE (*,900)
        STOP 'PROGRAM TERMINATED'
      END IF

      IF ((Critical_Radius_He_Microns .LT. 0.2) .OR.
*        (Critical_Radius_He_Microns .GT. 1.35)) THEN
        CALL SYSTEMQQ (OS_Command)
        WRITE (*,903)
        WRITE (*,900)
        STOP 'PROGRAM TERMINATED'
      END IF

      IF ((Critical_Volume_Algorithm .EQ. 'ON').OR.
*        (Critical_Volume_Algorithm .EQ. 'on')) THEN
        Critical_Volume_Algorithm_Off = (.FALSE.)
      ELSE IF ((Critical_Volume_Algorithm .EQ. 'OFF').OR.
*        (Critical_Volume_Algorithm .EQ. 'off')) THEN
        Critical_Volume_Algorithm_Off = (.TRUE.)
      ELSE
        WRITE (*,904)
        WRITE (*,900)
        STOP 'PROGRAM TERMINATED'
      END IF
C=====

```

```

C      INITIALIZE CONSTANTS/VARIABLES BASED ON SELECTION OF UNITS - FSW OR MSW
C      fsw = feet of seawater, a unit of pressure
C      msw = meters of seawater, a unit of pressure
C=====
C      IF (Units_Equal_Fsw) THEN
C          WRITE (*,800)
C          Units_Word1 = 'fswg'
C          Units_Word2 = 'fsw/min'
C          Units_Factor = 33.0
C          Water_Vapor_Pressure = 1.607      !based on respiratory quotient of 0.8
C                                          !(Schreiner value)
C      END IF
C      IF (Units_Equal_Msw) THEN
C          WRITE (*,801)
C          Units_Word1 = 'mswg'
C          Units_Word2 = 'msw/min'
C          Units_Factor = 10.1325
C          Water_Vapor_Pressure = 0.493      !based on respiratory quotient of 0.8
C                                          !(Schreiner value)
C      END IF
C=====
C      INITIALIZE CONSTANTS/VARIABLES
C=====
C      Constant_Pressure_Other_Gases = (Pressure_Other_Gases_mmHg/760.0)
C      *                               * Units_Factor
C      Run_Time = 0.0
C      Segment_Number = 0
C
C      DO I = 1,16
C          Helium_Time_Constant(I) = ALOG(2.0)/Helium_Half_Time(I)
C          Nitrogen_Time_Constant(I) = ALOG(2.0)/Nitrogen_Half_Time(I)
C          Max_Crushing_Pressure_He(I) = 0.0
C          Max_Crushing_Pressure_N2(I) = 0.0
C          Max_Actual_Gradient(I) = 0.0
C          Surface_Phase_Volume_Time(I) = 0.0
C          Amb_Pressure_Onset_of_Imperm(I) = 0.0
C          Gas_Tension_Onset_of_Imperm(I) = 0.0
C          Initial_Critical_Radius_N2(I) = Critical_Radius_N2_Microns
C      *          * 1.0E-6
C          Initial_Critical_Radius_He(I) = Critical_Radius_He_Microns
C      *          * 1.0E-6
C      END DO
C=====
C      INITIALIZE VARIABLES FOR SEA LEVEL OR ALTITUDE DIVE
C      See subroutines for explanation of altitude calculations.  Purposes are
C      1) to determine barometric pressure and 2) set or adjust the VPM critical
C      radius variables and gas loadings, as applicable, based on altitude,
C      ascent to altitude before the dive, and time at altitude before the dive
C=====
C      IF (Altitude_Dive_Algorithm_Off) THEN
C          Altitude_of_Dive = 0.0
C          CALL CALC_BAROMETRIC_PRESSURE (Altitude_of_Dive)      !subroutine
C          WRITE (*,802) Altitude_of_Dive, Barometric_Pressure
C          DO I = 1,16
C          Adjusted_Critical_Radius_N2(I) = Initial_Critical_Radius_N2(I)
C          Adjusted_Critical_Radius_He(I) = Initial_Critical_Radius_He(I)
C          Helium_Pressure(I) = 0.0
C          Nitrogen_Pressure(I) = (Barometric_Pressure -
C      *          Water_Vapor_Pressure)*0.79
C          END DO
C      ELSE
C          CALL VPM_ALTITUDE_DIVE_ALGORITHM      !subroutine
C      END IF
C=====
C      START OF REPETITIVE DIVE LOOP
C      This is the largest loop in the main program and operates between Lines

```

```

C      30 and 330.  If there is one or more repetitive dives, the program will
C      return to this point to process each repetitive dive.
C=====
30      DO 330, WHILE (.TRUE.)                                !loop will run continuously until
                                                         !there is an exit statement
C=====
C      INPUT DIVE DESCRIPTION AND GAS MIX DATA FROM ASCII TEXT INPUT FILE
C      BEGIN WRITING HEADINGS/OUTPUT TO ASCII TEXT OUTPUT FILE
C      See separate explanation of format for input file.
C=====
      READ (7,805) Line1
      CALL CLOCK (Year, Month, Day, Clock_Hour, Minute, M)      !subroutine
      WRITE (8,811)
      WRITE (8,812)
      WRITE (8,813)
      WRITE (8,813)
      WRITE (8,814) Month, Day, Year, Clock_Hour, Minute, M
      WRITE (8,813)
      WRITE (8,815) Line1
      WRITE (8,813)
      READ (7,*) Number_of_Mixes                                !check for errors in gasmixes
      DO I = 1, Number_of_Mixes
        READ (7,*) Fraction_Oxygen(I), Fraction_Helium(I),
*          Fraction_Nitrogen(I)
        Sum_of_Fractions = Fraction_Oxygen(I) + Fraction_Helium(I) +
*          Fraction_Nitrogen(I)
        Sum_Check = Sum_of_Fractions
        IF (Sum_Check .NE. 1.0) THEN
          CALL SYSTEMQQ (OS_Command)
          WRITE (*,906)
          WRITE (*,900)
          STOP 'PROGRAM TERMINATED'
        END IF
      END DO
      WRITE (8,820)
      DO J = 1, Number_of_Mixes
        WRITE (8,821) J, Fraction_Oxygen(J), Fraction_Helium(J),
*          Fraction_Nitrogen(J)
      END DO
      WRITE (8,813)
      WRITE (8,813)
      WRITE (8,830)
      WRITE (8,813)
      WRITE (8,831)
      WRITE (8,832)
      WRITE (8,833) Units_Word1, Units_Word1, Units_Word2, Units_Word1
      WRITE (8,834)
C=====
C      DIVE PROFILE LOOP - INPUT DIVE PROFILE DATA FROM ASCII TEXT INPUT FILE
C      AND PROCESS DIVE AS A SERIES OF ASCENT/DESCENT AND CONSTANT DEPTH
C      SEGMENTS.  THIS ALLOWS FOR MULTI-LEVEL DIVES AND UNUSUAL PROFILES.  UPDATE
C      GAS LOADINGS FOR EACH SEGMENT.  IF IT IS A DESCENT SEGMENT, CALC CRUSHING
C      PRESSURE ON CRITICAL RADII IN EACH COMPARTMENT.
C      "Instantaneous" descents are not used in the VPM.  All ascent/descent
C      segments must have a realistic rate of ascent/descent.  Unlike Haldanian
C      models, the VPM is actually more conservative when the descent rate is
C      slower because the effective crushing pressure is reduced.  Also, a
C      realistic actual supersaturation gradient must be calculated during
C      ascents as this affects critical radii adjustments for repetitive dives.
C      Profile codes: 1 = Ascent/Descent, 2 = Constant Depth, 99 = Decompress
C=====
      DO WHILE (.TRUE.)                                !loop will run continuously until
                                                         !there is an exit statement

      READ (7,*) Profile_Code
      IF (Profile_Code .EQ. 1) THEN
        READ (7,*) Starting_Depth, Ending_Depth, Rate, Mix_Number

```

```

        CALL GAS_LOADINGS_ASCENT_DESCENT (Starting_Depth,           !subroutine
*                                     Ending_Depth, Rate)
        IF (Ending_Depth .GT. Starting_Depth) THEN
            CALL CALC_CRUSHING_PRESSURE (Starting_Depth,           !subroutine
*                                     Ending_Depth, Rate)
        END IF
        IF (Ending_Depth .GT. Starting_Depth) THEN
            Word = 'Descent'
        ELSE IF (Starting_Depth .GT. Ending_Depth) THEN
            Word = 'Ascent '
        ELSE
            Word = 'ERROR'
        END IF
        WRITE (8,840) Segment_Number, Segment_Time, Run_Time,
*                               Mix_Number, Word, Starting_Depth, Ending_Depth,
*                               Rate
        ELSE IF (Profile_Code .EQ. 2) THEN
            READ (7,*) Depth, Run_Time_End_of_Segment, Mix_Number
            CALL GAS_LOADINGS_CONSTANT_DEPTH (Depth,               !subroutine
*                               Run_Time_End_of_Segment)
            WRITE (8,845) Segment_Number, Segment_Time, Run_Time,
*                               Mix_Number, Depth
        ELSE IF (Profile_Code .EQ. 99) THEN
            EXIT
        ELSE
            CALL SYSTEMQQ (OS_Command)
            WRITE (*,907)
            WRITE (*,900)
            STOP 'PROGRAM TERMINATED'
        END IF
    END DO

C=====
C   BEGIN PROCESS OF ASCENT AND DECOMPRESSION
C   First, calculate the regeneration of critical radii that takes place over
C   the dive time. The regeneration time constant has a time scale of weeks
C   so this will have very little impact on dives of normal length, but will
C   have major impact for saturation dives.
C=====
        CALL NUCLEAR_REGENERATION (Run_Time)                       !subroutine

C=====
C   CALCULATE INITIAL ALLOWABLE GRADIENTS FOR ASCENT
C   This is based on the maximum effective crushing pressure on critical radii
C   in each compartment achieved during the dive profile.
C=====
        CALL CALC_INITIAL_ALLOWABLE_GRADIENT                       !subroutine

C=====
C   SAVE VARIABLES AT START OF ASCENT (END OF BOTTOM TIME) SINCE THESE WILL
C   BE USED LATER TO COMPUTE THE FINAL ASCENT PROFILE THAT IS WRITTEN TO THE
C   OUTPUT FILE.
C   The VPM uses an iterative process to compute decompression schedules so
C   there will be more than one pass through the decompression loop.
C=====
        DO I = 1,16
            He_Pressure_Start_of_Ascent(I) = Helium_Pressure(I)
            N2_Pressure_Start_of_Ascent(I) = Nitrogen_Pressure(I)
        END DO
        Run_Time_Start_of_Ascent = Run_Time
        Segment_Number_Start_of_Ascent = Segment_Number

C=====
C   INPUT PARAMETERS TO BE USED FOR STAGED DECOMPRESSION AND SAVE IN ARRAYS.
C   ASSIGN INITIAL PARAMETERS TO BE USED AT START OF ASCENT
C   The user has the ability to change mix, ascent rate, and step size in any
C   combination at any depth during the ascent.
C=====

```



```

      READ (7,*) Number_of_Changes
      DO I = 1, Number_of_Changes
        READ (7,*) Depth_Change(I), Mix_Change(I), Rate_Change(I),
*          Step_Size_Change(I)
      END DO
      Starting_Depth = Depth_Change(1)
      Mix_Number = Mix_Change(1)
      Rate = Rate_Change(1)
      Step_Size = Step_Size_Change(1)
C=====
C   CALCULATE THE DEPTH WHERE THE DECOMPRESSION ZONE BEGINS FOR THIS PROFILE
C   BASED ON THE INITIAL ASCENT PARAMETERS AND WRITE THE DEEPEST POSSIBLE
C   DECOMPRESSION STOP DEPTH TO THE OUTPUT FILE
C   Knowing where the decompression zone starts is very important. Below
C   that depth there is no possibility for bubble formation because there
C   will be no supersaturation gradients. Deco stops should never start
C   below the deco zone. The deepest possible stop deco stop depth is
C   defined as the next "standard" stop depth above the point where the
C   leading compartment enters the deco zone. Thus, the program will not
C   base this calculation on step sizes larger than 10 fsw or 3 msw. The
C   deepest possible stop depth is not used in the program, per se, rather
C   it is information to tell the diver where to start putting on the brakes
C   during ascent. This should be prominently displayed by any deco program.
C=====
      CALL CALC_START_OF_DECO_ZONE (Starting_Depth, Rate,           !subroutine
*                                Depth_Start_of_Deco_Zone)
      IF (Units_Equal_Fsw) THEN
        IF (Step_Size .LT. 10.0) THEN
          Rounding_Operation1 =
*            (Depth_Start_of_Deco_Zone/Step_Size) - 0.5
          Deepest_Possible_Stop_Depth = ANINT(Rounding_Operation1)
*            * Step_Size
        ELSE
          Rounding_Operation1 = (Depth_Start_of_Deco_Zone/10.0)
*            - 0.5
          Deepest_Possible_Stop_Depth = ANINT(Rounding_Operation1)
*            * 10.0
        END IF
      END IF
      IF (Units_Equal_Msw) THEN
        IF (Step_Size .LT. 3.0) THEN
          Rounding_Operation1 =
*            (Depth_Start_of_Deco_Zone/Step_Size) - 0.5
          Deepest_Possible_Stop_Depth = ANINT(Rounding_Operation1)
*            * Step_Size
        ELSE
          Rounding_Operation1 = (Depth_Start_of_Deco_Zone/3.0)
*            - 0.5
          Deepest_Possible_Stop_Depth = ANINT(Rounding_Operation1)
*            * 3.0
        END IF
      END IF
      WRITE (8,813)
      WRITE (8,813)
      WRITE (8,850)
      WRITE (8,813)
      WRITE (8,857) Depth_Start_of_Deco_Zone, Units_Word1
      WRITE (8,858) Deepest_Possible_Stop_Depth, Units_Word1
      WRITE (8,813)
      WRITE (8,851)
      WRITE (8,852)
      WRITE (8,853) Units_Word1, Units_Word2, Units_Word1
      WRITE (8,854)
C=====
C   TEMPORARILY ASCEND PROFILE TO THE START OF THE DECOMPRESSION ZONE, SAVE
C   VARIABLES AT THIS POINT, AND INITIALIZE VARIABLES FOR CRITICAL VOLUME LOOP

```



```

IF (Deco_Stop_Depth .GT. Depth_Start_of_Deco_Zone) THEN
  WRITE (*,905)
  WRITE (*,900)
  STOP 'PROGRAM TERMINATED'
END IF
C=====
C   HANDLE THE SPECIAL CASE WHEN NO DECO STOPS ARE REQUIRED - ASCENT CAN BE
C   MADE DIRECTLY TO THE SURFACE
C   Write ascent data to output file and exit the Critical Volume Loop.
C=====
  IF (Deco_Stop_Depth .EQ. 0.0) THEN
    DO I = 1,16
      Helium_Pressure(I) = He_Pressure_Start_of_Ascent(I)
      Nitrogen_Pressure(I) = N2_Pressure_Start_of_Ascent(I)
    END DO
    Run_Time = Run_Time_Start_of_Ascent
    Segment_Number = Segment_Number_Start_of_Ascent
    Starting_Depth = Depth_Change(1)
    Ending_Depth = 0.0
    CALL GAS_LOADINGS_ASCENT_DESCENT (Starting_Depth,           !subroutine
*                                     Ending_Depth, Rate)
    WRITE (8,860) Segment_Number, Segment_Time, Run_Time,
*                                     Mix_Number, Deco_Stop_Depth, Rate

    EXIT                                     !exit the critical volume loop at Line 100
  END IF
C=====
C   ASSIGN VARIABLES FOR ASCENT FROM START OF DECO ZONE TO FIRST STOP.  SAVE
C   FIRST STOP DEPTH FOR LATER USE WHEN COMPUTING THE FINAL ASCENT PROFILE
C=====
  Starting_Depth = Depth_Start_of_Deco_Zone
  First_Stop_Depth = Deco_Stop_Depth
C=====
C   DECO STOP LOOP BLOCK WITHIN CRITICAL VOLUME LOOP
C   This loop computes a decompression schedule to the surface during each
C   iteration of the critical volume loop.  No output is written from this
C   loop, rather it computes a schedule from which the in-water portion of the
C   total phase volume time (Deco_Phase_Volume_Time) can be extracted.  Also,
C   the gas loadings computed at the end of this loop are used the subroutine
C   which computes the out-of-water portion of the total phase volume time
C   (Surface_Phase_Volume_Time) for that schedule.
C
C   Note that exit is made from the loop after last ascent is made to a deco
C   stop depth that is less than or equal to zero.  A final deco stop less
C   than zero can happen when the user makes an odd step size change during
C   ascent - such as specifying a 5 msw step size change at the 3 msw stop!
C=====
  DO WHILE (.TRUE.)                                     !loop will run continuously until
                                                         !there is an exit statement

    CALL GAS_LOADINGS_ASCENT_DESCENT (Starting_Depth,           !subroutine
*                                     Deco_Stop_Depth, Rate)

    IF (Deco_Stop_Depth .LE. 0.0) EXIT                     !exit at Line 60
    IF (Number_of_Changes .GT. 1) THEN
      DO I = 2, Number_of_Changes
        IF (Depth_Change(I) .GE. Deco_Stop_Depth) THEN
          Mix_Number = Mix_Change(I)
          Rate = Rate_Change(I)
          Step_Size = Step_Size_Change(I)
        END IF
      END DO
    END IF

    CALL BOYLES_LAW_COMPENSATION (First_Stop_Depth,
*                                Deco_Stop_Depth, Step_Size)    !subroutine

```

```

        CALL DECOMPRESSION_STOP (Deco_Stop_Depth, Step_Size)          !subroutine
        Starting_Depth = Deco_Stop_Depth
        Next_Stop = Deco_Stop_Depth - Step_Size
        Deco_Stop_Depth = Next_Stop
        Last_Run_Time = Run_Time
60      END DO                                                         !end of deco stop loop block
C=====
C      COMPUTE TOTAL PHASE VOLUME TIME AND MAKE CRITICAL VOLUME COMPARISON
C      The deco phase volume time is computed from the run time.  The surface
C      phase volume time is computed in a subroutine based on the surfacing gas
C      loadings from previous deco loop block.  Next the total phase volume time
C      (in-water + surface) for each compartment is compared against the previous
C      total phase volume time.  The schedule is converged when the difference is
C      less than or equal to 1 minute in any one of the 16 compartments.
C
C      Note: the "phase volume time" is somewhat of a mathematical concept.
C      It is the time divided out of a total integration of supersaturation
C      gradient x time (in-water and surface).  This integration is multiplied
C      by the excess bubble number to represent the amount of free-gas released
C      as a result of allowing a certain number of excess bubbles to form.
C=====
        Deco_Phase_Volume_Time = Run_Time - Run_Time_Start_of_Deco_Zone

        CALL CALC_SURFACE_PHASE_VOLUME_TIME                          !subroutine

        DO I = 1,16
            Phase_Volume_Time(I) = Deco_Phase_Volume_Time +
*               Surface_Phase_Volume_Time(I)
            Critical_Volume_Comparison = ABS(Phase_Volume_Time(I) -
*               Last_Phase_Volume_Time(I))
            IF (Critical_Volume_Comparison .LE. 1.0) THEN
                Schedule_Converged = (.TRUE.)
            END IF
        END DO

C=====
C      CRITICAL VOLUME DECISION TREE BETWEEN LINES 70 AND 99
C      There are two options here.  If the Critical Volume Algorithm setting is
C      "on" and the schedule is converged, or the Critical Volume Algorithm
C      setting was "off" in the first place, the program will re-assign variables
C      to their values at the start of ascent (end of bottom time) and process
C      a complete decompression schedule once again using all the same ascent
C      parameters and first stop depth.  This decompression schedule will match
C      the last iteration of the Critical Volume Loop and the program will write
C      the final deco schedule to the output file.
C
C      Note: if the Critical Volume Algorithm setting was "off", the final deco
C      schedule will be based on "Initial Allowable Supersaturation Gradients."
C      If it was "on", the final schedule will be based on "Adjusted Allowable
C      Supersaturation Gradients" (gradients that are "relaxed" as a result of
C      the Critical Volume Algorithm).
C
C      If the Critical Volume Algorithm setting is "on" and the schedule is not
C      converged, the program will re-assign variables to their values at the
C      start of the deco zone and process another trial decompression schedule.
C=====
70      IF ((Schedule_Converged) .OR.
*          (Critical_Volume_Algorithm_Off)) THEN
        DO I = 1,16
            Helium_Pressure(I) = He_Pressure_Start_of_Ascent(I)
            Nitrogen_Pressure(I) = N2_Pressure_Start_of_Ascent(I)
        END DO
        Run_Time = Run_Time_Start_of_Ascent
        Segment_Number = Segment_Number_Start_of_Ascent
        Starting_Depth = Depth_Change(1)
        Mix_Number = Mix_Change(1)

```

```

Rate = Rate_Change(1)
Step_Size = Step_Size_Change(1)
Deco_Stop_Depth = First_Stop_Depth
Last_Run_Time = 0.0
C=====
C      DECO STOP LOOP BLOCK FOR FINAL DECOMPRESSION SCHEDULE
C=====
C      DO WHILE (.TRUE.)                                !loop will run continuously until
C                                                         !there is an exit statement

C      CALL GAS_LOADINGS_ASCENT_DESCENT (Starting_Depth,      !subroutine
*                                                         Deco_Stop_Depth, Rate)
C=====
C      DURING FINAL DECOMPRESSION SCHEDULE PROCESS, COMPUTE MAXIMUM ACTUAL
C      SUPERSATURATION GRADIENT RESULTING IN EACH COMPARTMENT
C      If there is a repetitive dive, this will be used later in the VPM
C      Repetitive Algorithm to adjust the values for critical radii.
C=====
C      CALL CALC_MAX_ACTUAL_GRADIENT (Deco_Stop_Depth)        !subroutine

C      WRITE (8,860) Segment_Number, Segment_Time, Run_Time,
*                  Mix_Number, Deco_Stop_Depth, Rate
C      IF (Deco_Stop_Depth .LE. 0.0) EXIT                    !exit at Line 80
C      IF (Number_of_Changes .GT. 1) THEN
C        DO I = 2, Number_of_Changes
C          IF (Depth_Change(I) .GE. Deco_Stop_Depth) THEN
C            Mix_Number = Mix_Change(I)
C            Rate = Rate_Change(I)
C            Step_Size = Step_Size_Change(I)
C          END IF
C        END DO
C      END IF

C      CALL BOYLES LAW COMPENSATION (First Stop Depth,
*                  Deco_Stop_Depth, Step_Size)                !subroutine

C      CALL DECOMPRESSION_STOP (Deco_Stop_Depth, Step_Size)  !subroutine
C=====
C      This next bit just rounds up the stop time at the first stop to be in
C      whole increments of the minimum stop time (to make for a nice deco table).
C=====
C      IF (Last_Run_Time .EQ. 0.0) THEN
C        Stop_Time =
*          ANINT((Segment_Time/Minimum-Deco-Stop-Time) + 0.5) *
*          Minimum-Deco-Stop-Time
C      ELSE
C        Stop_Time = Run_Time - Last_Run_Time
C      END IF
C=====
C      DURING FINAL DECOMPRESSION SCHEDULE, IF MINIMUM STOP TIME PARAMETER IS A
C      WHOLE NUMBER (i.e. 1 minute) THEN WRITE DECO SCHEDULE USING INTEGER
C      NUMBERS (looks nicer). OTHERWISE, USE DECIMAL NUMBERS.
C      Note: per the request of a noted exploration diver(!), program now allows
C      a minimum stop time of less than one minute so that total ascent time can
C      be minimized on very long dives. In fact, with step size set at 1 fsw or
C      0.2 msw and minimum stop time set at 0.1 minute (6 seconds), a near
C      continuous decompression schedule can be computed.
C=====
C      IF (AINT(Minimum-Deco-Stop-Time) .EQ.
*          Minimum-Deco-Stop-Time) THEN
C        WRITE (8,862) Segment_Number, Segment_Time, Run_Time,
*          Mix_Number, INT(Deco_Stop_Depth),
*          INT(Stop_Time), INT(Run_Time)
C      ELSE
C        WRITE (8,863) Segment_Number, Segment_Time, Run_Time,
*          Mix_Number, Deco_Stop_Depth, Stop_Time,

```

```

*                               Run_Time
    END IF
    Starting_Depth = Deco_Stop_Depth
    Next_Stop = Deco_Stop_Depth - Step_Size
    Deco_Stop_Depth = Next_Stop
    Last_Run_Time = Run_Time
80    END DO                               !end of deco stop loop block
                                           !for final deco schedule

    EXIT                               !exit critical volume loop at Line 100

                                           !final deco schedule written
ELSE
C=====
C    IF SCHEDULE NOT CONVERGED, COMPUTE RELAXED ALLOWABLE SUPERSATURATION
C    GRADIENTS WITH VPM CRITICAL VOLUME ALGORITHM AND PROCESS ANOTHER
C    ITERATION OF THE CRITICAL VOLUME LOOP
C=====
    CALL CRITICAL_VOLUME (Deco_Phase_Volume_Time)           !subroutine
    Deco_Phase_Volume_Time = 0.0
    Run_Time = Run_Time_Start_of_Deco_Zone
    Starting_Depth = Depth_Start_of_Deco_Zone
    Mix_Number = Mix_Change(1)
    Rate = Rate_Change(1)
    Step_Size = Step_Size_Change(1)
    DO I = 1,16
        Last_Phase_Volume_Time(I) = Phase_Volume_Time(I)
        Helium_Pressure(I) = He_Pressure_Start_of_Deco_Zone(I)
        Nitrogen_Pressure(I) = N2_Pressure_Start_of_Deco_Zone(I)
    END DO

    CYCLE                               !Return to start of critical volume loop
                                           !(Line 50) to process another iteration

99    END IF                               !end of critical volume decision tree

100   CONTINUE                           !end of critical volume loop
C=====
C    PROCESSING OF DIVE COMPLETE.  READ INPUT FILE TO DETERMINE IF THERE IS A
C    REPETITIVE DIVE.  IF NONE, THEN EXIT REPETITIVE LOOP.
C=====
    READ (7,*) Repetitive_Dive_Flag
    IF (Repetitive_Dive_Flag .EQ. 0) THEN
        EXIT                               !exit repetitive dive loop
                                           !at Line 330
C=====
C    IF THERE IS A REPETITIVE DIVE, COMPUTE GAS LOADINGS (OFF-GASSING) DURING
C    SURFACE INTERVAL TIME.  ADJUST CRITICAL RADII USING VPM REPETITIVE
C    ALGORITHM.  RE-INITIALIZE SELECTED VARIABLES AND RETURN TO START OF
C    REPETITIVE LOOP AT LINE 30.
C=====
    ELSE IF (Repetitive_Dive_Flag .EQ. 1) THEN
        READ (7,*) Surface_Interval_Time

        CALL GAS_LOADINGS_SURFACE_INTERVAL (Surface_Interval_Time) !subroutine

        CALL VPM_REPETITIVE_ALGORITHM (Surface_Interval_Time)      !subroutine

        DO I = 1,16
            Max_Crushing_Pressure_He(I) = 0.0
            Max_Crushing_Pressure_N2(I) = 0.0
            Max_Actual_Gradient(I) = 0.0
        END DO
        Run_Time = 0.0
        Segment_Number = 0
        WRITE (8,880)

```

```

        WRITE (8,890)
        WRITE (8,813)

        CYCLE      !Return to start of repetitive loop to process another dive
C=====
C      WRITE ERROR MESSAGE AND TERMINATE PROGRAM IF THERE IS AN ERROR IN THE
C      INPUT FILE FOR THE REPETITIVE DIVE FLAG
C=====
        ELSE
            CALL SYSTEMQQ (OS_Command)
            WRITE (*,908)
            WRITE (*,900)
            STOP 'PROGRAM TERMINATED'
        END IF
330    CONTINUE                                     !End of repetitive loop
C=====
C      FINAL WRITES TO OUTPUT AND CLOSE PROGRAM FILES
C=====
        WRITE (*,813)
        WRITE (*,871)
        WRITE (*,872)
        WRITE (*,813)
        WRITE (8,880)
        CLOSE (UNIT = 7, STATUS = 'KEEP')
        CLOSE (UNIT = 8, STATUS = 'KEEP')
        CLOSE (UNIT = 10, STATUS = 'KEEP')
C=====
C      FORMAT STATEMENTS - PROGRAM INPUT/OUTPUT
C=====
800    FORMAT ('0UNITS = FEET OF SEAWATER (FSW)')
801    FORMAT ('0UNITS = METERS OF SEAWATER (MSW)')
802    FORMAT ('0ALTITUDE = ',1X,F7.1,4X,'BAROMETRIC PRESSURE = ',
*F6.3)
805    FORMAT (A70)
811    FORMAT (26X,'DECOMPRESSION CALCULATION PROGRAM')
812    FORMAT (24X,'Developed in FORTRAN by Erik C. Baker')
814    FORMAT ('Program Run:',4X,I2.2,'-',I2.2,'-',I4,1X,'at',1X,I2.2,
*      ':',I2.2,1X,A1,'m',23X,'Model: VPM-B')
815    FORMAT ('Description:',4X,A70)
813    FORMAT (' ')
820    FORMAT ('Gasmix Summary:',24X,'FO2',4X,'FHe',4X,'FN2')
821    FORMAT (26X,'Gasmix #',I2,2X,F5.3,2X,F5.3,2X,F5.3)
830    FORMAT (36X,'DIVE PROFILE')
831    FORMAT ('Seg-',2X,'Segm.',2X,'Run',3X,'|',1X,'Gasmix',1X,'|',1X,
*      'Ascent',4X,'From',5X,'To',6X,'Rate',4X,'|',1X,'Constant')
832    FORMAT ('ment',2X,'Time',3X,'Time',2X,'|',2X,'Used',2X,'|',3X,
*      'or',5X,'Depth',3X,'Depth',4X,'+Dn/-Up',2X,'|',2X,'Depth')
833    FORMAT (2X,'#',3X,'(min)',2X,'(min)',1X,'|',4X,'#',3X,'|',1X,
*      'Descent',2X,'(',A4,')',2X,'(',A4,')',2X,'(',A7,')',1X,
*      '|',2X,'(',A4,')')
834    FORMAT ('-----',1X,'-----',2X,'-----',1X,'|',1X,'-----',1X,'|',
*      1X,'-----',2X,'-----',2X,'-----',2X,'-----',1X,
*      '|',1X,'-----')
840    FORMAT (I3,3X,F5.1,1X,F6.1,1X,'|',3X,I2,3X,'|',1X,A7,F7.0,
*      1X,F7.0,3X,F7.1,3X,'|')
845    FORMAT (I3,3X,F5.1,1X,F6.1,1X,'|',3X,I2,3X,'|',36X,'|',F7.0)
850    FORMAT (31X,'DECOMPRESSION PROFILE')
851    FORMAT ('Seg-',2X,'Segm.',2X,'Run',3X,'|',1X,'Gasmix',1X,'|',1X,
*      'Ascent',3X,'Ascent',3X,'Col',3X,'|',2X,'DECO',3X,'STOP',
*      3X,'RUN')
852    FORMAT ('ment',2X,'Time',3X,'Time',2X,'|',2X,'Used',2X,'|',3X,
*      'To',6X,'Rate',4X,'Not',3X,'|',2X,'STOP',3X,'TIME',3X,
*      'TIME')
853    FORMAT (2X,'#',3X,'(min)',2X,'(min)',1X,'|',4X,'#',3X,'|',1X,
*      '(',A4,')',1X,'(',A7,')',2X,'Used',2X,'|',1X,'(',A4,')',
*      2X,'(min)',2X,'(min)')

```

```

854  FORMAT ('-----',1X,'-----',2X,'-----',1X,'|',1X,'-----',1X,'|',
*        1X,'-----',1X,'-----',1X,'-----',1X,'|',1X,
*        '-----',2X,'-----',2X,'-----')
857  FORMAT (10X,'Leading compartment enters the decompression zone',
*        1X,'at',F7.1,1X,A4)
858  FORMAT (17X,'Deepest possible decompression stop is',F7.1,1X,A4)
860  FORMAT (I3,3X,F5.1,1X,F6.1,1X,'|',3X,I2,3X,'|',2X,F4.0,3X,F6.1,
*        10X,'|')
862  FORMAT (I3,3X,F5.1,1X,F6.1,1X,'|',3X,I2,3X,'|',25X,'|',2X,I4,3X,
*        I4,2X,I5)
863  FORMAT (I3,3X,F5.1,1X,F6.1,1X,'|',3X,I2,3X,'|',25X,'|',2X,F5.0,1X,
*        F6.1,1X,F7.1)
871  FORMAT (' PROGRAM CALCULATIONS COMPLETE')
872  FORMAT ('0Output data is located in the file VPMDECO.OUT')
880  FORMAT (' ')
890  FORMAT ('REPETITIVE DIVE:')
C=====
C    FORMAT STATEMENTS - ERROR MESSAGES
C=====
900  FORMAT (' ')
901  FORMAT ('0ERROR! UNITS MUST BE FSW OR MSW')
902  FORMAT ('0ERROR! ALTITUDE DIVE ALGORITHM MUST BE ON OR OFF')
903  FORMAT ('0ERROR! RADIUS MUST BE BETWEEN 0.2 AND 1.35 MICRONS')
904  FORMAT ('0ERROR! CRITICAL VOLUME ALGORITHM MUST BE ON OR OFF')
905  FORMAT ('0ERROR! STEP SIZE IS TOO LARGE TO DECOMPRESS')
906  FORMAT ('0ERROR IN INPUT FILE (GASMIX DATA)')
907  FORMAT ('0ERROR IN INPUT FILE (PROFILE CODE)')
908  FORMAT ('0ERROR IN INPUT FILE (REPETITIVE DIVE CODE)')
C=====
C    END OF MAIN PROGRAM
C=====
END

C=====
C    NOTE ABOUT PRESSURE UNITS USED IN CALCULATIONS:
C    It is the convention in decompression calculations to compute all gas
C    loadings, absolute pressures, partial pressures, etc., in the units of
C    depth pressure that you are diving - either feet of seawater (fsw) or
C    meters of seawater (msw). This program follows that convention with the
C    the exception that all VPM calculations are performed in SI units (by
C    necessity). Accordingly, there are several conversions back and forth
C    between the diving pressure units and the SI units.
C=====

C=====
C    FUNCTION SUBPROGRAM FOR GAS LOADING CALCULATIONS - ASCENT AND DESCENT
C=====
FUNCTION SCHREINER_EQUATION (Initial_Inspired_Gas_Pressure,
*Rate_Change_Insp_Gas_Pressure, Interval_Time, Gas_Time_Constant,
*Initial_Gas_Pressure)
C=====
C    ARGUMENTS
C=====
REAL Initial_Inspired_Gas_Pressure           !input
REAL Rate_Change_Insp_Gas_Pressure           !input
REAL Interval_Time, Gas_Time_Constant         !input
REAL Initial_Gas_Pressure                     !input
REAL SCHREINER_EQUATION                      !output
C=====
C    Note: The Schreiner equation is applied when calculating the uptake or
C    elimination of compartment gases during linear ascents or descents at a
C    constant rate. For ascents, a negative number for rate must be used.
C=====
SCHREINER_EQUATION =

```



```

*Initial_Inspired_Gas_Pressure + Rate_Change_Insp_Gas_Pressure*
*(Interval_Time - 1.0/Gas_Time_Constant) -
*(Initial_Inspired_Gas_Pressure - Initial_Gas_Pressure -
*Rate_Change_Insp_Gas_Pressure/Gas_Time_Constant)*
*EXP (-Gas_Time_Constant*Interval_Time)
RETURN
END

C=====
C    FUNCTION SUBPROGRAM FOR GAS LOADING CALCULATIONS - CONSTANT DEPTH
C=====
C    FUNCTION HALDANE_EQUATION (Initial_Gas_Pressure,
C    *Inspired_Gas_Pressure, Gas_Time_Constant, Interval_Time)
C=====
C    ARGUMENTS
C=====
C    REAL Initial_Gas_Pressure, Inspired_Gas_Pressure           !input
C    REAL Gas_Time_Constant, Interval_Time                     !input
C    REAL HALDANE_EQUATION                                     !output
C=====
C    Note: The Haldane equation is applied when calculating the uptake or
C    elimination of compartment gases during intervals at constant depth (the
C    outside ambient pressure does not change).
C=====
C    HALDANE_EQUATION = Initial_Gas_Pressure +
C    *(Inspired_Gas_Pressure - Initial_Gas_Pressure)*
C    *(1.0 - EXP(-Gas_Time_Constant * Interval_Time))
C    RETURN
C    END

C=====
C    SUBROUTINE GAS_LOADINGS_ASCENT_DESCENT
C    Purpose: This subprogram applies the Schreiner equation to update the
C    gas loadings (partial pressures of helium and nitrogen) in the half-time
C    compartments due to a linear ascent or descent segment at a constant rate.
C=====
C    SUBROUTINE GAS_LOADINGS_ASCENT_DESCENT (Starting_Depth,
C    *                                     Ending_Depth, Rate)
C    IMPLICIT NONE
C=====
C    ARGUMENTS
C=====
C    REAL Starting_Depth, Ending_Depth, Rate                   !input
C=====
C    LOCAL VARIABLES
C=====
C    INTEGER I                                                  !loop counter
C    INTEGER Last_Segment_Number

C    REAL Initial_Inspired_He_Pressure
C    REAL Initial_Inspired_N2_Pressure
C    REAL Last_Run_Time
C    REAL Helium_Rate, Nitrogen_Rate, Starting_Ambient_Pressure

C    REAL SCHREINER_EQUATION                                   !function subprogram
C=====
C    GLOBAL CONSTANTS IN NAMED COMMON BLOCKS
C=====
C    REAL Water_Vapor_Pressure
C    COMMON /Block_8/ Water_Vapor_Pressure
C=====
C    GLOBAL VARIABLES IN NAMED COMMON BLOCKS
C=====
C    INTEGER Segment_Number                                   !both input

```

```

REAL Run_Time, Segment_Time                                !and output
COMMON /Block_2/ Run_Time, Segment_Number, Segment_Time

REAL Ending_Ambient_Pressure                                !output
COMMON /Block_4/ Ending_Ambient_Pressure

INTEGER Mix_Number
COMMON /Block_9/ Mix_Number

REAL Barometric_Pressure
COMMON /Block_18/ Barometric_Pressure
C=====
C   GLOBAL ARRAYS IN NAMED COMMON BLOCKS
C=====
REAL Helium_Time_Constant(16)
COMMON /Block_1A/ Helium_Time_Constant

REAL Nitrogen_Time_Constant(16)
COMMON /Block_1B/ Nitrogen_Time_Constant

REAL Helium_Pressure(16), Nitrogen_Pressure(16)             !both input
COMMON /Block_3/ Helium_Pressure, Nitrogen_Pressure          !and output

REAL Fraction_Helium(10), Fraction_Nitrogen(10)

REAL Initial_Helium_Pressure(16), Initial_Nitrogen_Pressure(16) !output
COMMON /Block_23/ Initial_Helium_Pressure,
*                   Initial_Nitrogen_Pressure
C=====
C   CALCULATIONS
C=====
Segment_Time = (Ending_Depth - Starting_Depth)/Rate
Last_Run_Time = Run_Time
Run_Time = Last_Run_Time + Segment_Time
Last_Segment_Number = Segment_Number
Segment_Number = Last_Segment_Number + 1
Ending_Ambient_Pressure = Ending_Depth + Barometric_Pressure
Starting_Ambient_Pressure = Starting_Depth + Barometric_Pressure
Initial_Inspired_He_Pressure = (Starting_Ambient_Pressure -
*                               Water_Vapor_Pressure)*Fraction_Helium(Mix_Number)
Initial_Inspired_N2_Pressure = (Starting_Ambient_Pressure -
*                               Water_Vapor_Pressure)*Fraction_Nitrogen(Mix_Number)
Helium_Rate = Rate*Fraction_Helium(Mix_Number)
Nitrogen_Rate = Rate*Fraction_Nitrogen(Mix_Number)
DO I = 1,16
  Initial_Helium_Pressure(I) = Helium_Pressure(I)
  Initial_Nitrogen_Pressure(I) = Nitrogen_Pressure(I)

  Helium_Pressure(I) = SCHREINER_EQUATION
*   (Initial_Inspired_He_Pressure, Helium_Rate,
*   Segment_Time, Helium_Time_Constant(I),
*   Initial_Helium_Pressure(I))

  Nitrogen_Pressure(I) = SCHREINER_EQUATION
*   (Initial_Inspired_N2_Pressure, Nitrogen_Rate,
*   Segment_Time, Nitrogen_Time_Constant(I),
*   Initial_Nitrogen_Pressure(I))
END DO
C=====
C   END OF SUBROUTINE
C=====
RETURN
END

```

```

C=====
C  SUBROUTINE CALC_CRUSHING_PRESSURE
C  Purpose: Compute the effective "crushing pressure" in each compartment as
C  a result of descent segment(s). The crushing pressure is the gradient
C  (difference in pressure) between the outside ambient pressure and the
C  gas tension inside a VPM nucleus (bubble seed). This gradient acts to
C  reduce (shrink) the radius smaller than its initial value at the surface.
C  This phenomenon has important ramifications because the smaller the radius
C  of a VPM nucleus, the greater the allowable supersaturation gradient upon
C  ascent. Gas loading (uptake) during descent, especially in the fast
C  compartments, will reduce the magnitude of the crushing pressure. The
C  crushing pressure is not cumulative over a multi-level descent. It will
C  be the maximum value obtained in any one discrete segment of the overall
C  descent. Thus, the program must compute and store the maximum crushing
C  pressure for each compartment that was obtained across all segments of
C  the descent profile.
C
C  The calculation of crushing pressure will be different depending on
C  whether or not the gradient is in the VPM permeable range (gas can diffuse
C  across skin of VPM nucleus) or the VPM impermeable range (molecules in
C  skin of nucleus are squeezed together so tight that gas can no longer
C  diffuse in or out of nucleus; the gas becomes trapped and further resists
C  the crushing pressure). The solution for crushing pressure in the VPM
C  permeable range is a simple linear equation. In the VPM impermeable
C  range, a cubic equation must be solved using a numerical method.
C
C  Separate crushing pressures are tracked for helium and nitrogen because
C  they can have different critical radii. The crushing pressures will be
C  the same for helium and nitrogen in the permeable range of the model, but
C  they will start to diverge in the impermeable range. This is due to
C  the differences between starting radius, radius at the onset of
C  impermeability, and radial compression in the impermeable range.
C=====
C  SUBROUTINE CALC_CRUSHING_PRESSURE (Starting_Depth, Ending_Depth,
C  *                               Rate)
C  IMPLICIT NONE
C=====
C  ARGUMENTS
C=====
C  REAL Starting_Depth, Ending_Depth, Rate                                !input
C=====
C  LOCAL VARIABLES
C=====
C  INTEGER I                                                              !loop counter
C
C  REAL Starting_Ambient_Pressure, Ending_Ambient_Pressure
C  REAL Starting_Gas_Tension, Ending_Gas_Tension
C  REAL Crushing_Pressure_He, Crushing_Pressure_N2
C  REAL Gradient_Onset_of_Imperm, Gradient_Onset_of_Imperm_Pa
C  REAL Ending_Ambient_Pressure_Pa, Amb_Press_Onset_of_Imperm_Pa
C  REAL Gas_Tension_Onset_of_Imperm_Pa
C  REAL Crushing_Pressure_Pascals_He, Crushing_Pressure_Pascals_N2
C  REAL Starting_Gradient, Ending_Gradient
C  REAL A_He, B_He, C_He, Ending_Radius_He, High_Bound_He
C  REAL Low_Bound_He
C  REAL A_N2, B_N2, C_N2, Ending_Radius_N2, High_Bound_N2
C  REAL Low_Bound_N2
C  REAL Radius_Onset_of_Imperm_He, Radius_Onset_of_Imperm_N2
C=====
C  GLOBAL CONSTANTS IN NAMED COMMON BLOCKS
C=====
C  REAL Gradient_Onset_of_Imperm_Atm
C  COMMON /Block_14/ Gradient_Onset_of_Imperm_Atm
C
C  REAL Constant_Pressure_Other_Gases
C  COMMON /Block_17/ Constant_Pressure_Other_Gases

```

```

      REAL Surface_Tension_Gamma, Skin_Compression_GammaC
      COMMON /Block_19/ Surface_Tension_Gamma, Skin_Compression_GammaC
C=====
C   GLOBAL VARIABLES IN NAMED COMMON BLOCKS
C=====
      REAL Units_Factor
      COMMON /Block_16/ Units_Factor

      REAL Barometric_Pressure
      COMMON /Block_18/ Barometric_Pressure
C=====
C   GLOBAL ARRAYS IN NAMED COMMON BLOCKS
C=====
      REAL Helium_Pressure(16), Nitrogen_Pressure(16)           !input
      COMMON /Block_3/ Helium_Pressure, Nitrogen_Pressure

      REAL Adjusted_Critical_Radius_He(16)                     !input
      REAL Adjusted_Critical_Radius_N2(16)
      COMMON /Block_7/ Adjusted_Critical_Radius_He,
*       Adjusted_Critical_Radius_N2

      REAL Max_Crushing_Pressure_He(16), Max_Crushing_Pressure_N2(16) !output
      COMMON /Block_10/ Max_Crushing_Pressure_He,
*       Max_Crushing_Pressure_N2

      REAL Amb_Pressure_Onset_of_Imperm(16)                   !input
      REAL Gas_Tension_Onset_of_Imperm(16)
      COMMON /Block_13/ Amb_Pressure_Onset_of_Imperm,
*       Gas_Tension_Onset_of_Imperm

      REAL Initial_Helium_Pressure(16), Initial_Nitrogen_Pressure(16) !input
      COMMON /Block_23/ Initial_Helium_Pressure,
*       Initial_Nitrogen_Pressure
C=====
C   CALCULATIONS
C   First, convert the Gradient for Onset of Impermeability from units of
C   atmospheres to diving pressure units (either fsw or msw) and to Pascals
C   (SI units). The reason that the Gradient for Onset of Impermeability is
C   given in the program settings in units of atmospheres is because that is
C   how it was reported in the original research papers by Yount and
C   colleauges.
C=====
      Gradient_Onset_of_Imperm = Gradient_Onset_of_Imperm_Atm           !convert to
*       * Units_Factor                                           !diving units

      Gradient_Onset_of_Imperm_Pa = Gradient_Onset_of_Imperm_Atm       !convert to
*       * 101325.0                                           !Pascals
C=====
C   Assign values of starting and ending ambient pressures for descent segment
C=====
      Starting_Ambient_Pressure = Starting_Depth + Barometric_Pressure
      Ending_Ambient_Pressure = Ending_Depth + Barometric_Pressure
C=====
C   MAIN LOOP WITH NESTED DECISION TREE
C   For each compartment, the program computes the starting and ending
C   gas tensions and gradients. The VPM is different than some dissolved gas
C   algorithms, Buhlmann for example, in that it considers the pressure due to
C   oxygen, carbon dioxide, and water vapor in each compartment in addition to
C   the inert gases helium and nitrogen. These "other gases" are included in
C   the calculation of gas tensions and gradients.
C=====
      DO I = 1,16
          Starting_Gas_Tension = Initial_Helium_Pressure(I) +
*       Initial_Nitrogen_Pressure(I) + Constant_Pressure_Other_Gases

          Starting_Gradient = Starting_Ambient_Pressure -

```

```

*                               Starting_Gas_Tension

Ending_Gas_Tension = Helium_Pressure(I) + Nitrogen_Pressure(I)
*                               + Constant_Pressure_Other_Gases

Ending_Gradient = Ending_Ambient_Pressure - Ending_Gas_Tension
C=====
C    Compute radius at onset of impermeability for helium and nitrogen
C    critical radii
C=====
    Radius_Onset_of_Imperm_He = 1.0/(Gradient_Onset_of_Imperm_Pa/
*        (2.0*(Skin_Compression_GammaC-Surface_Tension_Gamma)) +
*        1.0/Adjusted_Critical_Radius_He(I))

    Radius_Onset_of_Imperm_N2 = 1.0/(Gradient_Onset_of_Imperm_Pa/
*        (2.0*(Skin_Compression_GammaC-Surface_Tension_Gamma)) +
*        1.0/Adjusted_Critical_Radius_N2(I))
C=====
C    FIRST BRANCH OF DECISION TREE - PERMEABLE RANGE
C    Crushing pressures will be the same for helium and nitrogen
C=====
    IF (Ending_Gradient .LE. Gradient_Onset_of_Imperm) THEN

        Crushing_Pressure_He = Ending_Ambient_Pressure -
*                               Ending_Gas_Tension

        Crushing_Pressure_N2 = Ending_Ambient_Pressure -
*                               Ending_Gas_Tension

    END IF
C=====
C    SECOND BRANCH OF DECISION TREE - IMPERMEABLE RANGE
C    Both the ambient pressure and the gas tension at the onset of
C    impermeability must be computed in order to properly solve for the ending
C    radius and resultant crushing pressure. The first decision block
C    addresses the special case when the starting gradient just happens to be
C    equal to the gradient for onset of impermeability (not very likely!).
C=====
    IF (Ending_Gradient .GT. Gradient_Onset_of_Imperm) THEN

        IF(Starting_Gradient .EQ. Gradient_Onset_of_Imperm) THEN
            Amb_Pressure_Onset_of_Imperm(I) =
*                               Starting_Ambient_Pressure
            Gas_Tension_Onset_of_Imperm(I) = Starting_Gas_Tension
        END IF
C=====
C    In most cases, a subroutine will be called to find these values using a
C    numerical method.
C=====
    IF(Starting_Gradient .LT. Gradient_Onset_of_Imperm) THEN

        CALL ONSET_OF_IMPERMEABILITY                                !subroutine
*                               (Starting_Ambient_Pressure,
*                               Ending_Ambient_Pressure, Rate, I)

    END IF
C=====
C    Next, using the values for ambient pressure and gas tension at the onset
C    of impermeability, the equations are set up to process the calculations
C    through the radius root finder subroutine. This subprogram will find the
C    root (solution) to the cubic equation using a numerical method. In order
C    to do this efficiently, the equations are placed in the form
C     $Ar^3 - Br^2 - C = 0$ , where r is the ending radius after impermeable
C    compression. The coefficients A, B, and C for helium and nitrogen are
C    computed and passed to the subroutine as arguments. The high and low
C    bounds to be used by the numerical method of the subroutine are also
C    computed (see separate page posted on Deco List ftp site entitled
C    "VPM: Solving for radius in the impermeable regime"). The subprogram

```

```

C      will return the value of the ending radius and then the crushing
C      pressures for helium and nitrogen can be calculated.
C=====
      Ending_Ambient_Pressure_Pa =
*          (Ending_Ambient_Pressure/Units_Factor) * 101325.0

      Amb_Press_Onset_of_Imperm_Pa =
*          (Amb_Pressure_Onset_of_Imperm(I)/Units_Factor)
*          * 101325.0

      Gas_Tension_Onset_of_Imperm_Pa =
*          (Gas_Tension_Onset_of_Imperm(I)/Units_Factor)
*          * 101325.0

      B_He = 2.0*(Skin_Compression_GammaC-Surface_Tension_Gamma)

      A_He = Ending_Ambient_Pressure_Pa -
*          Amb_Press_Onset_of_Imperm_Pa +
*          Gas_Tension_Onset_of_Imperm_Pa +
*          (2.0*(Skin_Compression_GammaC-Surface_Tension_Gamma))
*          /Radius_Onset_of_Imperm_He

      C_He = Gas_Tension_Onset_of_Imperm_Pa *
*          Radius_Onset_of_Imperm_He**3

      High_Bound_He = Radius_Onset_of_Imperm_He
      Low_Bound_He = B_He/A_He

      CALL RADIUS_ROOT_FINDER (A_He,B_He,C_He,                      !subroutine
*          Low_Bound_He, High_Bound_He, Ending_Radius_He)

      Crushing_Pressure_Pascals_He =
*          Gradient_Onset_of_Imperm_Pa +
*          Ending_Ambient_Pressure_Pa -
*          Amb_Press_Onset_of_Imperm_Pa +
*          Gas_Tension_Onset_of_Imperm_Pa *
*          (1.0-Radius_Onset_of_Imperm_He**3/Ending_Radius_He**3)

      Crushing_Pressure_He =
*          (Crushing_Pressure_Pascals_He/101325.0) * Units_Factor

      B_N2 = 2.0*(Skin_Compression_GammaC-Surface_Tension_Gamma)

      A_N2 = Ending_Ambient_Pressure_Pa -
*          Amb_Press_Onset_of_Imperm_Pa +
*          Gas_Tension_Onset_of_Imperm_Pa +
*          (2.0*(Skin_Compression_GammaC-Surface_Tension_Gamma))
*          /Radius_Onset_of_Imperm_N2

      C_N2 = Gas_Tension_Onset_of_Imperm_Pa *
*          Radius_Onset_of_Imperm_N2**3

      High_Bound_N2 = Radius_Onset_of_Imperm_N2
      Low_Bound_N2 = B_N2/A_N2

      CALL RADIUS_ROOT_FINDER (A_N2,B_N2,C_N2,                      !subroutine
*          Low_Bound_N2,High_Bound_N2, Ending_Radius_N2)

      Crushing_Pressure_Pascals_N2 =
*          Gradient_Onset_of_Imperm_Pa +
*          Ending_Ambient_Pressure_Pa -
*          Amb_Press_Onset_of_Imperm_Pa +
*          Gas_Tension_Onset_of_Imperm_Pa *
*          (1.0-Radius_Onset_of_Imperm_N2**3/Ending_Radius_N2**3)

      Crushing_Pressure_N2 =

```

```

      *      (Crushing_Pressure_Pascals_N2/101325.0) * Units_Factor
      END IF
C=====
C      UPDATE VALUES OF MAX CRUSHING PRESSURE IN GLOBAL ARRAYS
C=====
      Max_Crushing_Pressure_He(I) = MAX(Max_Crushing_Pressure_He(I),
      *                               Crushing_Pressure_He)

      Max_Crushing_Pressure_N2(I) = MAX(Max_Crushing_Pressure_N2(I),
      *                               Crushing_Pressure_N2)
      END DO
C=====
C      END OF SUBROUTINE
C=====
      RETURN
      END

C=====
C      SUBROUTINE ONSET_OF_IMPERMEABILITY
C      Purpose: This subroutine uses the Bisection Method to find the ambient
C      pressure and gas tension at the onset of impermeability for a given
C      compartment. Source: "Numerical Recipes in Fortran 77",
C      Cambridge University Press, 1992.
C=====
      SUBROUTINE ONSET_OF_IMPERMEABILITY (Starting_Ambient_Pressure,
      *                               Ending_Ambient_Pressure, Rate, I)
      IMPLICIT NONE
C=====
C      ARGUMENTS
C=====
      INTEGER I                                !input - array subscript for compartment

      REAL Starting_Ambient_Pressure, Ending_Ambient_Pressure, Rate      !input
C=====
C      LOCAL VARIABLES
C=====
      INTEGER J                                !loop counter

      REAL Initial_Inspired_He_Pressure
      REAL Initial_Inspired_N2_Pressure, Time
      REAL Helium_Rate, Nitrogen_Rate
      REAL Low_Bound, High_Bound, High_Bound_Helium_Pressure
      REAL High_Bound_Nitrogen_Pressure, Mid_Range_Helium_Pressure
      REAL Mid_Range_Nitrogen_Pressure, Last_Diff_Change
      REAL Function_at_High_Bound, Function_at_Low_Bound
      REAL Mid_Range_Time, Function_at_Mid_Range, Differential_Change
      REAL Mid_Range_Ambient_Pressure, Gas_Tension_at_Mid_Range
      REAL Gradient_Onset_of_Imperm
      REAL Starting_Gas_Tension, Ending_Gas_Tension

      REAL SCHREINER_EQUATION                                !function subprogram
C=====
C      GLOBAL CONSTANTS IN NAMED COMMON BLOCKS
C=====
      REAL Water_Vapor_Pressure
      COMMON /Block_8/ Water_Vapor_Pressure

      REAL Gradient_Onset_of_Imperm_Atm
      COMMON /Block_14/ Gradient_Onset_of_Imperm_Atm

      REAL Constant_Pressure_Other_Gases
      COMMON /Block_17/ Constant_Pressure_Other_Gases
C=====
C      GLOBAL VARIABLES IN NAMED COMMON BLOCKS
C=====

```

```

        INTEGER Mix_Number
        COMMON /Block_9/ Mix_Number

        REAL Units_Factor
        COMMON /Block_16/ Units_Factor
C=====
C    GLOBAL ARRAYS IN NAMED COMMON BLOCKS
C=====
        REAL Helium_Time_Constant(16)
        COMMON /Block_1A/ Helium_Time_Constant

        REAL Nitrogen_Time_Constant(16)
        COMMON /Block_1B/ Nitrogen_Time_Constant

        REAL Fraction_Helium(10), Fraction_Nitrogen(10)
        COMMON /Block_5/ Fraction_Helium, Fraction_Nitrogen

        REAL Amb_Pressure_Onset_of_Imperm(16)                                !output
        REAL Gas_Tension_Onset_of_Imperm(16)
        COMMON /Block_13/ Amb_Pressure_Onset_of_Imperm,
*           Gas_Tension_Onset_of_Imperm

        REAL Initial_Helium_Pressure(16), Initial_Nitrogen_Pressure(16)      !input
        COMMON /Block_23/ Initial_Helium_Pressure,
*           Initial_Nitrogen_Pressure
C=====
C    CALCULATIONS
C    First convert the Gradient for Onset of Impermeability to the diving
C    pressure units that are being used
C=====
        Gradient_Onset_of_Imperm = Gradient_Onset_of_Imperm_Atm
*                               * Units_Factor
C=====
C    ESTABLISH THE BOUNDS FOR THE ROOT SEARCH USING THE BISECTION METHOD
C    In this case, we are solving for time - the time when the ambient pressure
C    minus the gas tension will be equal to the Gradient for Onset of
C    Impermeability. The low bound for time is set at zero and the high
C    bound is set at the elapsed time (segment time) it took to go from the
C    starting ambient pressure to the ending ambient pressure. The desired
C    ambient pressure and gas tension at the onset of impermeability will
C    be found somewhere between these endpoints. The algorithm checks to
C    make sure that the solution lies in between these bounds by first
C    computing the low bound and high bound function values.
C=====
        Initial_Inspired_He_Pressure = (Starting_Ambient_Pressure -
*           Water_Vapor_Pressure)*Fraction_Helium(Mix_Number)

        Initial_Inspired_N2_Pressure = (Starting_Ambient_Pressure -
*           Water_Vapor_Pressure)*Fraction_Nitrogen(Mix_Number)

        Helium_Rate = Rate*Fraction_Helium(Mix_Number)
        Nitrogen_Rate = Rate*Fraction_Nitrogen(Mix_Number)
        Low_Bound = 0.0

        High_Bound = (Ending_Ambient_Pressure - Starting_Ambient_Pressure)
*           /Rate

        Starting_Gas_Tension = Initial_Helium_Pressure(I) +
*           Initial_Nitrogen_Pressure(I) + Constant_Pressure_Other_Gases

        Function_at_Low_Bound = Starting_Ambient_Pressure -
*           Starting_Gas_Tension - Gradient_Onset_of_Imperm

        High_Bound_Helium_Pressure = SCHREINER_EQUATION
*           (Initial_Inspired_He_Pressure, Helium_Rate,
*           High_Bound, Helium_Time_Constant(I),

```



```

*      Initial_Helium_Pressure(I))

      High_Bound_Nitrogen_Pressure = SCHREINER_EQUATION
*      (Initial_Inspired_N2_Pressure, Nitrogen_Rate,
*      High_Bound, Nitrogen_Time_Constant(I),
*      Initial_Nitrogen_Pressure(I))

      Ending_Gas_Tension = High_Bound_Helium_Pressure +
*      High_Bound_Nitrogen_Pressure + Constant_Pressure_Other_Gases

      Function_at_High_Bound = Ending_Ambient_Pressure -
*      Ending_Gas_Tension - Gradient_Onset_of_Imperm

      IF ((Function_at_High_Bound*Function_at_Low_Bound) .GE. 0.0) THEN
        PRINT *, 'ERROR! ROOT IS NOT WITHIN BRACKETS'
        PAUSE
      END IF

C=====
C      APPLY THE BISECTION METHOD IN SEVERAL ITERATIONS UNTIL A SOLUTION WITH
C      THE DESIRED ACCURACY IS FOUND
C      Note: the program allows for up to 100 iterations. Normally an exit will
C      be made from the loop well before that number. If, for some reason, the
C      program exceeds 100 iterations, there will be a pause to alert the user.
C=====
      IF (Function_at_Low_Bound .LT. 0.0) THEN
        Time = Low_Bound
        Differential_Change = High_Bound - Low_Bound
      ELSE
        Time = High_Bound
        Differential_Change = Low_Bound - High_Bound
      END IF
      DO J = 1, 100
        Last_Diff_Change = Differential_Change
        Differential_Change = Last_Diff_Change*0.5
        Mid_Range_Time = Time + Differential_Change

        Mid_Range_Ambient_Pressure = (Starting_Ambient_Pressure +
*      Rate*Mid_Range_Time)

        Mid_Range_Helium_Pressure = SCHREINER_EQUATION
*      (Initial_Inspired_He_Pressure, Helium_Rate,
*      Mid_Range_Time, Helium_Time_Constant(I),
*      Initial_Helium_Pressure(I))

        Mid_Range_Nitrogen_Pressure = SCHREINER_EQUATION
*      (Initial_Inspired_N2_Pressure, Nitrogen_Rate,
*      Mid_Range_Time, Nitrogen_Time_Constant(I),
*      Initial_Nitrogen_Pressure(I))

        Gas_Tension_at_Mid_Range = Mid_Range_Helium_Pressure +
*      Mid_Range_Nitrogen_Pressure + Constant_Pressure_Other_Gases

        Function_at_Mid_Range = Mid_Range_Ambient_Pressure -
*      Gas_Tension_at_Mid_Range - Gradient_Onset_of_Imperm

        IF (Function_at_Mid_Range .LE. 0.0) Time = Mid_Range_Time

        IF ((ABS(Differential_Change) .LT. 1.0E-3) .OR.
*      (Function_at_Mid_Range .EQ. 0.0)) GOTO 100

      END DO
      PRINT *, 'ERROR! ROOT SEARCH EXCEEDED MAXIMUM ITERATIONS'
      PAUSE
C=====
C      When a solution with the desired accuracy is found, the program jumps out
C      of the loop to Line 100 and assigns the solution values for ambient

```

```

C      pressure and gas tension at the onset of impermeability.
C=====
100    Amb_Pressure_Onset_of_Imperm(I) = Mid_Range_Ambient_Pressure
      Gas_Tension_Onset_of_Imperm(I) = Gas_Tension_at_Mid_Range
C=====
C      END OF SUBROUTINE
C=====
      RETURN
      END

C=====
C      SUBROUTINE RADIUS_ROOT_FINDER
C      Purpose: This subroutine is a "fail-safe" routine that combines the
C      Bisection Method and the Newton-Raphson Method to find the desired root.
C      This hybrid algorithm takes a bisection step whenever Newton-Raphson would
C      take the solution out of bounds, or whenever Newton-Raphson is not
C      converging fast enough. Source: "Numerical Recipes in Fortran 77",
C      Cambridge University Press, 1992.
C=====
      SUBROUTINE RADIUS_ROOT_FINDER (A,B,C, Low_Bound, High_Bound,
*                                     Ending_Radius)
      IMPLICIT NONE

C=====
C      ARGUMENTS
C=====
      REAL A, B, C, Low_Bound, High_Bound          !input
      REAL Ending_Radius                          !output
C=====
C      LOCAL VARIABLES
C=====
      INTEGER I                                     !loop counter

      REAL Function, Derivative_of_Function, Differential_Change
      REAL Last_Diff_Change, Last_Ending_Radius
      REAL Radius_at_Low_Bound, Radius_at_High_Bound
      REAL Function_at_Low_Bound, Function_at_High_Bound
C=====
C      BEGIN CALCULATIONS BY MAKING SURE THAT THE ROOT LIES WITHIN BOUNDS
C      In this case we are solving for radius in a cubic equation of the form,
C       $Ar^3 - Br^2 - C = 0$ . The coefficients A, B, and C were passed to this
C      subroutine as arguments.
C=====
      Function_at_Low_Bound =
*      Low_Bound*(Low_Bound*(A*Low_Bound - B)) - C

      Function_at_High_Bound =
*      High_Bound*(High_Bound*(A*High_Bound - B)) - C

      IF ((Function_at_Low_Bound .GT. 0.0).AND.
*      (Function_at_High_Bound .GT. 0.0)) THEN
        PRINT *, 'ERROR! ROOT IS NOT WITHIN BRACKETS'
        PAUSE
      END IF

C=====
C      Next the algorithm checks for special conditions and then prepares for
C      the first bisection.
C=====
      IF ((Function_at_Low_Bound .LT. 0.0).AND.
*      (Function_at_High_Bound .LT. 0.0)) THEN
        PRINT *, 'ERROR! ROOT IS NOT WITHIN BRACKETS'
        PAUSE
      END IF
      IF (Function_at_Low_Bound .EQ. 0.0) THEN
        Ending_Radius = Low_Bound
        RETURN

```

```

ELSE IF (Function_at_High_Bound .EQ. 0.0) THEN
    Ending_Radius = High_Bound
    RETURN
ELSE IF (Function_at_Low_Bound .LT. 0.0) THEN
    Radius_at_Low_Bound = Low_Bound
    Radius_at_High_Bound = High_Bound
ELSE
    Radius_at_High_Bound = Low_Bound
    Radius_at_Low_Bound = High_Bound
END IF
Ending_Radius = 0.5*(Low_Bound + High_Bound)
Last_Diff_Change = ABS(High_Bound-Low_Bound)
Differential_Change = Last_Diff_Change
C=====
C    At this point, the Newton-Raphson Method is applied which uses a function
C    and its first derivative to rapidly converge upon a solution.
C    Note: the program allows for up to 100 iterations. Normally an exit will
C    be made from the loop well before that number. If, for some reason, the
C    program exceeds 100 iterations, there will be a pause to alert the user.
C    When a solution with the desired accuracy is found, exit is made from the
C    loop by returning to the calling program. The last value of ending
C    radius has been assigned as the solution.
C=====
    Function = Ending_Radius*(Ending_Radius*(A*Ending_Radius - B)) - C

    Derivative_of_Function =
*      Ending_Radius*(Ending_Radius*3.0*A - 2.0*B)

    DO I = 1,100
        IF((((Ending_Radius-Radius_at_High_Bound)*
*          Derivative_of_Function-Function)*
*          ((Ending_Radius-Radius_at_Low_Bound)*
*          Derivative_of_Function-Function).GE.0.0) .OR.
*          (ABS(2.0*Function).GT.
*          (ABS(Last_Diff_Change*Derivative_of_Function)))) THEN

            Last_Diff_Change = Differential_Change

            Differential_Change = 0.5*(Radius_at_High_Bound -
*              Radius_at_Low_Bound)

            Ending_Radius = Radius_at_Low_Bound + Differential_Change
            IF (Radius_at_Low_Bound .EQ. Ending_Radius) RETURN
        ELSE
            Last_Diff_Change = Differential_Change
            Differential_Change = Function/Derivative_of_Function
            Last_Ending_Radius = Ending_Radius
            Ending_Radius = Ending_Radius - Differential_Change
            IF (Last_Ending_Radius .EQ. Ending_Radius) RETURN
        END IF
        IF (ABS(Differential_Change) .LT. 1.0E-12) RETURN
        Function =
*          Ending_Radius*(Ending_Radius*(A*Ending_Radius - B)) - C

        Derivative_of_Function =
*          Ending_Radius*(Ending_Radius*3.0*A - 2.0*B)

        IF (Function .LT. 0.0) THEN
            Radius_at_Low_Bound = Ending_Radius
        ELSE
            Radius_at_High_Bound = Ending_Radius
        END IF
    END DO
    PRINT *, 'ERROR! ROOT SEARCH EXCEEDED MAXIMUM ITERATIONS'
    PAUSE
C=====

```

```

C      END OF SUBROUTINE
C=====
C      END

C=====
C      SUBROUTINE GAS_LOADINGS_CONSTANT_DEPTH
C      Purpose: This subprogram applies the Haldane equation to update the
C      gas loadings (partial pressures of helium and nitrogen) in the half-time
C      compartments for a segment at constant depth.
C=====
C      SUBROUTINE GAS_LOADINGS_CONSTANT_DEPTH (Depth,
C      *                               Run_Time_End_of_Segment)
C      IMPLICIT NONE
C=====
C      ARGUMENTS
C=====
C      REAL Depth, Run_Time_End_of_Segment                                !input
C=====
C      LOCAL VARIABLES
C=====
C      INTEGER I                                                         !loop counter
C      INTEGER Last_Segment_Number

C      REAL Initial_Helium_Pressure, Initial_Nitrogen_Pressure
C      REAL Inspired_Helium_Pressure, Inspired_Nitrogen_Pressure
C      REAL Ambient_Pressure, Last_Run_Time

C      REAL HALDANE_EQUATION                                           !function subprogram
C=====
C      GLOBAL CONSTANTS IN NAMED COMMON BLOCKS
C=====
C      REAL Water_Vapor_Pressure
C      COMMON /Block_8/ Water_Vapor_Pressure
C=====
C      GLOBAL VARIABLES IN NAMED COMMON BLOCKS
C=====
C      INTEGER Segment_Number                                           !both input
C      REAL Run_Time, Segment_Time                                     !and output
C      COMMON /Block_2/ Run_Time, Segment_Number, Segment_Time

C      REAL Ending_Ambient_Pressure                                     !output
C      COMMON /Block_4/ Ending_Ambient_Pressure

C      INTEGER Mix_Number
C      COMMON /Block_9/ Mix_Number

C      REAL Barometric_Pressure
C      COMMON /Block_18/ Barometric_Pressure
C=====
C      GLOBAL ARRAYS IN NAMED COMMON BLOCKS
C=====
C      REAL Helium_Time_Constant(16)
C      COMMON /Block_1A/ Helium_Time_Constant

C      REAL Nitrogen_Time_Constant(16)
C      COMMON /Block_1B/ Nitrogen_Time_Constant

C      REAL Helium_Pressure(16), Nitrogen_Pressure(16)                 !both input
C      COMMON /Block_3/ Helium_Pressure, Nitrogen_Pressure             !and output

C      REAL Fraction_Helium(10), Fraction_Nitrogen(10)
C      COMMON /Block_5/ Fraction_Helium, Fraction_Nitrogen
C=====
C      CALCULATIONS
C=====

```

```

Segment_Time = Run_Time_End_of_Segment - Run_Time
Last_Run_Time = Run_Time_End_of_Segment
Run_Time = Last_Run_Time
Last_Segment_Number = Segment_Number
Segment_Number = Last_Segment_Number + 1
Ambient_Pressure = Depth + Barometric_Pressure

Inspired_Helium_Pressure = (Ambient_Pressure -
*      Water_Vapor_Pressure)*Fraction_Helium(Mix_Number)

Inspired_Nitrogen_Pressure = (Ambient_Pressure -
*      Water_Vapor_Pressure)*Fraction_Nitrogen(Mix_Number)

Ending_Ambient_Pressure = Ambient_Pressure
DO I = 1,16
    Initial_Helium_Pressure = Helium_Pressure(I)
    Initial_Nitrogen_Pressure = Nitrogen_Pressure(I)

    Helium_Pressure(I) = HALDANE_EQUATION
*      (Initial_Helium_Pressure, Inspired_Helium_Pressure,
*      Helium_Time_Constant(I), Segment_Time)

    Nitrogen_Pressure(I) = HALDANE_EQUATION
*      (Initial_Nitrogen_Pressure, Inspired_Nitrogen_Pressure,
*      Nitrogen_Time_Constant(I), Segment_Time)
END DO
C=====
C      END OF SUBROUTINE
C=====
      RETURN
      END

C=====
C      SUBROUTINE NUCLEAR_REGENERATION
C      Purpose: This subprogram calculates the regeneration of VPM critical
C      radii that takes place over the dive time. The regeneration time constant
C      has a time scale of weeks so this will have very little impact on dives of
C      normal length, but will have a major impact for saturation dives.
C=====
      SUBROUTINE NUCLEAR_REGENERATION (Dive_Time)

      IMPLICIT NONE

C=====
C      ARGUMENTS
C=====
      REAL Dive_Time                                     !input
C=====
C      LOCAL VARIABLES
C=====
      INTEGER I                                         !loop counter

      REAL Crushing_Pressure_Pascals_He, Crushing_Pressure_Pascals_N2
      REAL Ending_Radius_He, Ending_Radius_N2
      REAL Crush_Pressure_Adjust_Ratio_He
      REAL Crush_Pressure_Adjust_Ratio_N2
      REAL Adj_Crush_Pressure_He_Pascals, Adj_Crush_Pressure_N2_Pascals
C=====
C      GLOBAL CONSTANTS IN NAMED COMMON BLOCKS
C=====
      REAL Surface_Tension_Gamma, Skin_Compression_GammaC
      COMMON /Block_19/ Surface_Tension_Gamma, Skin_Compression_GammaC

      REAL Regeneration_Time_Constant
      COMMON /Block_22/ Regeneration_Time_Constant
C=====

```

```

C      GLOBAL VARIABLES IN NAMED COMMON BLOCKS
C=====
C      REAL Units_Factor
C      COMMON /Block_16/ Units_Factor
C=====
C      GLOBAL ARRAYS IN NAMED COMMON BLOCKS
C=====
C      REAL Adjusted_Critical_Radius_He(16)                                !input
C      REAL Adjusted_Critical_Radius_N2(16)
C      COMMON /Block_7/ Adjusted_Critical_Radius_He,
C      *          Adjusted_Critical_Radius_N2

C      REAL Max_Crushing_Pressure_He(16), Max_Crushing_Pressure_N2(16)    !input
C      COMMON /Block_10/ Max_Crushing_Pressure_He,
C      *          Max_Crushing_Pressure_N2

C      REAL Regenerated_Radius_He(16), Regenerated_Radius_N2(16)          !output
C      COMMON /Block_24/ Regenerated_Radius_He, Regenerated_Radius_N2

C      REAL Adjusted_Crushing_Pressure_He(16)                             !output
C      REAL Adjusted_Crushing_Pressure_N2(16)
C      COMMON /Block_25/ Adjusted_Crushing_Pressure_He,
C      *          Adjusted_Crushing_Pressure_N2
C=====
C      CALCULATIONS
C      First convert the maximum crushing pressure obtained for each compartment
C      to Pascals. Next, compute the ending radius for helium and nitrogen
C      critical nuclei in each compartment.
C=====
C      DO I = 1,16
C          Crushing_Pressure_Pascals_He =
C      *          (Max_Crushing_Pressure_He(I)/Units_Factor) * 101325.0

C          Crushing_Pressure_Pascals_N2 =
C      *          (Max_Crushing_Pressure_N2(I)/Units_Factor) * 101325.0

C          Ending_Radius_He = 1.0/(Crushing_Pressure_Pascals_He/
C      *          (2.0*(Skin_Compression_GammaC - Surface_Tension_Gamma)) +
C      *          1.0/Adjusted_Critical_Radius_He(I))

C          Ending_Radius_N2 = 1.0/(Crushing_Pressure_Pascals_N2/
C      *          (2.0*(Skin_Compression_GammaC - Surface_Tension_Gamma)) +
C      *          1.0/Adjusted_Critical_Radius_N2(I))
C=====
C      A "regenerated" radius for each nucleus is now calculated based on the
C      regeneration time constant. This means that after application of
C      crushing pressure and reduction in radius, a nucleus will slowly grow
C      back to its original initial radius over a period of time. This
C      phenomenon is probabilistic in nature and depends on absolute temperature.
C      It is independent of crushing pressure.
C=====
C      Regenerated_Radius_He(I) = Adjusted_Critical_Radius_He(I) +
C      *          (Ending_Radius_He - Adjusted_Critical_Radius_He(I)) *
C      *          EXP(-Dive_Time/Regeneration_Time_Constant)

C      Regenerated_Radius_N2(I) = Adjusted_Critical_Radius_N2(I) +
C      *          (Ending_Radius_N2 - Adjusted_Critical_Radius_N2(I)) *
C      *          EXP(-Dive_Time/Regeneration_Time_Constant)
C=====
C      In order to preserve reference back to the initial critical radii after
C      regeneration, an "adjusted crushing pressure" for the nuclei in each
C      compartment must be computed. In other words, this is the value of
C      crushing pressure that would have reduced the original nucleus to the
C      to the present radius had regeneration not taken place. The ratio
C      for adjusting crushing pressure is obtained from algebraic manipulation
C      of the standard VPM equations. The adjusted crushing pressure, in lieu

```

```

C      of the original crushing pressure, is then applied in the VPM Critical
C      Volume Algorithm and the VPM Repetitive Algorithm.

```

```

C=====
      Crush_Pressure_Adjust_Ratio_He =
      *      (Ending_Radius_He*(Adjusted_Critical_Radius_He(I) -
      *      Regenerated_Radius_He(I))) / (Regenerated_Radius_He(I) *
      *      (Adjusted_Critical_Radius_He(I) - Ending_Radius_He))

      Crush_Pressure_Adjust_Ratio_N2 =
      *      (Ending_Radius_N2*(Adjusted_Critical_Radius_N2(I) -
      *      Regenerated_Radius_N2(I))) / (Regenerated_Radius_N2(I) *
      *      (Adjusted_Critical_Radius_N2(I) - Ending_Radius_N2))

      Adj_Crush_Pressure_He_Pascals = Crushing_Pressure_Pascals_He *
      *      Crush_Pressure_Adjust_Ratio_He

      Adj_Crush_Pressure_N2_Pascals = Crushing_Pressure_Pascals_N2 *
      *      Crush_Pressure_Adjust_Ratio_N2

      Adjusted_Crushing_Pressure_He(I) =
      *      (Adj_Crush_Pressure_He_Pascals / 101325.0) * Units_Factor

      Adjusted_Crushing_Pressure_N2(I) =
      *      (Adj_Crush_Pressure_N2_Pascals / 101325.0) * Units_Factor

      END DO
C=====
C      END OF SUBROUTINE
C=====
      RETURN
      END

```

```

C=====
C      SUBROUTINE CALC_INITIAL_ALLOWABLE_GRADIENT
C      Purpose: This subprogram calculates the initial allowable gradients for
C      helium and nitrogen in each compartment. These are the gradients that
C      will be used to set the deco ceiling on the first pass through the deco
C      loop. If the Critical Volume Algorithm is set to "off", then these
C      gradients will determine the final deco schedule. Otherwise, if the
C      Critical Volume Algorithm is set to "on", these gradients will be further
C      "relaxed" by the Critical Volume Algorithm subroutine. The initial
C      allowable gradients are referred to as "PssMin" in the papers by Yount
C      and colleagues, i.e., the minimum supersaturation pressure gradients
C      that will probe bubble formation in the VPM nuclei that started with the
C      designated minimum initial radius (critical radius).
C
C      The initial allowable gradients are computed directly from the
C      "regenerated" radii after the Nuclear Regeneration subroutine. These
C      gradients are tracked separately for helium and nitrogen.
C=====
      SUBROUTINE CALC_INITIAL_ALLOWABLE_GRADIENT

      IMPLICIT NONE

C=====
C      LOCAL VARIABLES
C=====
      INTEGER I !loop counter

      REAL Initial_Allowable_Grad_He_Pa, Initial_Allowable_Grad_N2_Pa
C=====
C      GLOBAL CONSTANTS IN NAMED COMMON BLOCKS
C=====
      REAL Surface_Tension_Gamma, Skin_Compression_GammaC
      COMMON /Block_19/ Surface_Tension_Gamma, Skin_Compression_GammaC
C=====

```

```

C      GLOBAL VARIABLES IN NAMED COMMON BLOCKS
C=====
C      REAL Units_Factor
C      COMMON /Block_16/ Units_Factor
C=====
C      GLOBAL ARRAYS IN NAMED COMMON BLOCKS
C=====
C      REAL Regenerated_Radius_He(16), Regenerated_Radius_N2(16)          !input
C      COMMON /Block_24/ Regenerated_Radius_He, Regenerated_Radius_N2

C      REAL Allowable_Gradient_He(16), Allowable_Gradient_N2 (16)        !output
C      COMMON /Block_26/ Allowable_Gradient_He, Allowable_Gradient_N2

C      REAL Initial_Allowable_Gradient_He(16)                            !output
C      REAL Initial_Allowable_Gradient_N2(16)
C      COMMON /Block_27/
C      *      Initial_Allowable_Gradient_He, Initial_Allowable_Gradient_N2
C=====
C      CALCULATIONS
C      The initial allowable gradients are computed in Pascals and then converted
C      to the diving pressure units. Two different sets of arrays are used to
C      save the calculations - Initial Allowable Gradients and Allowable
C      Gradients. The Allowable Gradients are assigned the values from Initial
C      Allowable Gradients however the Allowable Gradients can be changed later
C      by the Critical Volume subroutine. The values for the Initial Allowable
C      Gradients are saved in a global array for later use by both the Critical
C      Volume subroutine and the VPM Repetitive Algorithm subroutine.
C=====
C      DO I = 1,16

C          Initial_Allowable_Grad_N2_Pa = ((2.0*Surface_Tension_Gamma*
C      *      (Skin_Compression_GammaC - Surface_Tension_Gamma)) /
C      *      (Regenerated_Radius_N2(I)*Skin_Compression_GammaC))

C          Initial_Allowable_Grad_He_Pa = ((2.0*Surface_Tension_Gamma*
C      *      (Skin_Compression_GammaC - Surface_Tension_Gamma)) /
C      *      (Regenerated_Radius_He(I)*Skin_Compression_GammaC))

C          Initial_Allowable_Gradient_N2(I) =
C      *      (Initial_Allowable_Grad_N2_Pa / 101325.0) * Units_Factor

C          Initial_Allowable_Gradient_He(I) =
C      *      (Initial_Allowable_Grad_He_Pa / 101325.0) * Units_Factor

C          Allowable_Gradient_He(I) = Initial_Allowable_Gradient_He(I)
C          Allowable_Gradient_N2(I) = Initial_Allowable_Gradient_N2(I)
C      END DO
C=====
C      END OF SUBROUTINE
C=====
C      RETURN
C      END

```

```

C=====
C      SUBROUTINE CALC_ASCENT_CEILING
C      Purpose: This subprogram calculates the ascent ceiling (the safe ascent
C      depth) in each compartment, based on the allowable gradients, and then
C      finds the deepest ascent ceiling across all compartments.
C=====

```

```

      SUBROUTINE CALC_ASCENT_CEILING (Ascent_Ceiling_Depth)

```

```

      IMPLICIT NONE

```

```

C=====
C      ARGUMENTS
C=====

```



```

      REAL Ascent_Ceiling_Depth                                     !output
=====
C
C   LOCAL VARIABLES
=====
C
      INTEGER I                                                    !loop counter

      REAL Gas_Loading, Weighted_Allowable_Gradient
      REAL Tolerated_Ambient_Pressure
=====
C
C   LOCAL ARRAYS
=====
C
      REAL Compartment_Ascent_Ceiling(16)
=====
C
C   GLOBAL CONSTANTS IN NAMED COMMON BLOCKS
=====
C
      REAL Constant_Pressure_Other_Gases
      COMMON /Block_17/ Constant_Pressure_Other_Gases
=====
C
C   GLOBAL VARIABLES IN NAMED COMMON BLOCKS
=====
C
      REAL Barometric_Pressure
      COMMON /Block_18/ Barometric_Pressure
=====
C
C   GLOBAL ARRAYS IN NAMED COMMON BLOCKS
=====
C
      REAL Helium_Pressure(16), Nitrogen_Pressure(16)             !input
      COMMON /Block_3/ Helium_Pressure, Nitrogen_Pressure

      REAL Allowable_Gradient_He(16), Allowable_Gradient_N2 (16)   !input
      COMMON /Block_26/ Allowable_Gradient_He, Allowable_Gradient_N2
=====
C
C   CALCULATIONS
C   Since there are two sets of allowable gradients being tracked, one for
C   helium and one for nitrogen, a "weighted allowable gradient" must be
C   computed each time based on the proportions of helium and nitrogen in
C   each compartment. This proportioning follows the methodology of
C   Buhlmann/Keller. If there is no helium and nitrogen in the compartment,
C   such as after extended periods of oxygen breathing, then the minimum value
C   across both gases will be used. It is important to note that if a
C   compartment is empty of helium and nitrogen, then the weighted allowable
C   gradient formula cannot be used since it will result in division by zero.
=====
C
      DO I = 1,16
         Gas_Loading = Helium_Pressure(I) + Nitrogen_Pressure(I)

         IF (Gas_Loading .GT. 0.0) THEN
            Weighted_Allowable_Gradient =
*            (Allowable_Gradient_He(I)* Helium_Pressure(I) +
*            Allowable_Gradient_N2(I)* Nitrogen_Pressure(I)) /
*            (Helium_Pressure(I) + Nitrogen_Pressure(I))

            Tolerated_Ambient_Pressure = (Gas_Loading +
*            Constant_Pressure_Other_Gases) - Weighted_Allowable_Gradient

         ELSE
            Weighted_Allowable_Gradient =
*            MIN(Allowable_Gradient_He(I), Allowable_Gradient_N2(I))

            Tolerated_Ambient_Pressure =
*            Constant_Pressure_Other_Gases - Weighted_Allowable_Gradient
         END IF
      END DO
=====
C
C   The tolerated ambient pressure cannot be less than zero absolute, i.e.,
C   the vacuum of outer space!
=====
C
      IF (Tolerated_Ambient_Pressure .LT. 0.0) THEN

```

```

      Tolerated_Ambient_Pressure = 0.0
    END IF
C=====
C   The Ascent Ceiling Depth is computed in a loop after all of the individual
C   compartment ascent ceilings have been calculated.  It is important that
C   the Ascent Ceiling Depth (max ascent ceiling across all compartments) only
C   be extracted from the compartment values and not be compared against some
C   initialization value.  For example, if MAX(Ascent Ceiling Depth . .) was
C   compared against zero, this could cause a program lockup because sometimes
C   the Ascent Ceiling Depth needs to be negative (but not less than zero
C   absolute ambient pressure) in order to decompress to the last stop at zero
C   depth.
C=====
      Compartment Ascent Ceiling(I) =
    *      Tolerated_Ambient_Pressure - Barometric_Pressure
    END DO
      Ascent_Ceiling_Depth = Compartment_Ascent_Ceiling(1)
      DO I = 2,16
        Ascent Ceiling Depth =
    *      MAX(Ascent_Ceiling_Depth, Compartment_Ascent_Ceiling(I))
    END DO
C=====
C   END OF SUBROUTINE
C=====
      RETURN
      END

```

```

C=====
C   SUBROUTINE CALC_MAX_ACTUAL_GRADIENT
C   Purpose: This subprogram calculates the actual supersaturation gradient
C   obtained in each compartment as a result of the ascent profile during
C   decompression.  Similar to the concept with crushing pressure, the
C   supersaturation gradients are not cumulative over a multi-level, staged
C   ascent.  Rather, it will be the maximum value obtained in any one discrete
C   step of the overall ascent.  Thus, the program must compute and store the
C   maximum actual gradient for each compartment that was obtained across all
C   steps of the ascent profile.  This subroutine is invoked on the last pass
C   through the deco stop loop block when the final deco schedule is being
C   generated.
C
C   The max actual gradients are later used by the VPM Repetitive Algorithm to
C   determine if adjustments to the critical radii are required.  If the max
C   actual gradient did not exceed the initial allowable gradient, then no
C   adjustment will be made.  However, if the max actual gradient did exceed
C   the initial allowable gradient, such as permitted by the Critical Volume
C   Algorithm, then the critical radius will be adjusted (made larger) on the
C   repetitive dive to compensate for the bubbling that was allowed on the
C   previous dive.  The use of the max actual gradients is intended to prevent
C   the repetitive algorithm from being overly conservative.
C=====
      SUBROUTINE CALC_MAX_ACTUAL_GRADIENT (Deco_Stop_Depth)

      IMPLICIT NONE
C=====
C   ARGUMENTS
C=====
      REAL Deco_Stop_Depth                                !input
C=====
C   LOCAL VARIABLES
C=====
      INTEGER I                                           !loop counter

      REAL Compartment_Gradient
C=====
C   GLOBAL CONSTANTS IN NAMED COMMON BLOCKS

```

```

C=====
      REAL Constant_Pressure_Other_Gases
      COMMON /Block_17/ Constant_Pressure_Other_Gases
C=====
C      GLOBAL VARIABLES IN NAMED COMMON BLOCKS
C=====
      REAL Barometric_Pressure
      COMMON /Block_18/ Barometric_Pressure
C=====
C      GLOBAL ARRAYS IN NAMED COMMON BLOCKS
C=====
      REAL Helium_Pressure(16), Nitrogen_Pressure(16)                !input
      COMMON /Block_3/ Helium_Pressure, Nitrogen_Pressure

      REAL Max_Actual_Gradient(16)
      COMMON /Block_12/ Max_Actual_Gradient                          !output
C=====
C      CALCULATIONS
C      Note: negative supersaturation gradients are meaningless for this
C      application, so the values must be equal to or greater than zero.
C=====
      DO I = 1,16
        Compartment_Gradient = (Helium_Pressure(I) +
*          Nitrogen_Pressure(I) + Constant_Pressure_Other_Gases)
*          - (Deco_Stop_Depth + Barometric_Pressure)
        IF (Compartment_Gradient .LE. 0.0) THEN
          Compartment_Gradient = 0.0
        END IF
        Max_Actual_Gradient(I) =
*          MAX(Max_Actual_Gradient(I), Compartment_Gradient)
      END DO
C=====
C      END OF SUBROUTINE
C=====
      RETURN
      END

C=====
C      SUBROUTINE CALC_SURFACE_PHASE_VOLUME_TIME
C      Purpose: This subprogram computes the surface portion of the total phase
C      volume time. This is the time factored out of the integration of
C      supersaturation gradient x time over the surface interval. The VPM
C      considers the gradients that allow bubbles to form or to drive bubble
C      growth both in the water and on the surface after the dive.
C
C      This subroutine is a new development to the VPM algorithm in that it
C      computes the time course of supersaturation gradients on the surface
C      when both helium and nitrogen are present. Refer to separate write-up
C      for a more detailed explanation of this algorithm.
C=====
      SUBROUTINE CALC_SURFACE_PHASE_VOLUME_TIME

      IMPLICIT NONE
C=====
C      LOCAL VARIABLES
C=====
      INTEGER I                                                        !loop counter

      REAL Integral_Gradient_x_Time, Decay_Time_to_Zero_Gradient
      REAL Surface_Inspired_N2_Pressure
C=====
C      GLOBAL CONSTANTS IN NAMED COMMON BLOCKS
C=====
      REAL Water_Vapor_Pressure
      COMMON /Block_8/ Water_Vapor_Pressure

```

```

C=====
C    GLOBAL VARIABLES IN NAMED COMMON BLOCKS
C=====
C    REAL Barometric_Pressure
C    COMMON /Block_18/ Barometric_Pressure
C=====
C    GLOBAL ARRAYS IN NAMED COMMON BLOCKS
C=====
C    REAL Helium_Time_Constant(16)
C    COMMON /Block_1A/ Helium_Time_Constant

C    REAL Nitrogen_Time_Constant(16)
C    COMMON /Block_1B/ Nitrogen_Time_Constant

C    REAL Helium_Pressure(16), Nitrogen_Pressure(16)           !input
C    COMMON /Block_3/ Helium_Pressure, Nitrogen_Pressure

C    REAL Surface_Phase_Volume_Time(16)                       !output
C    COMMON /Block_11/ Surface_Phase_Volume_Time
C=====
C    CALCULATIONS
C=====
C    Surface_Inspired_N2_Pressure = (Barometric_Pressure -
*      Water_Vapor_Pressure)*0.79
C    DO I = 1,16
C      IF (Nitrogen_Pressure(I) .GT. Surface_Inspired_N2_Pressure)
*                                     THEN
C        Surface_Phase_Volume_Time(I)=
*      (Helium_Pressure(I)/Helium_Time_Constant(I)+
*      (Nitrogen_Pressure(I)-Surface_Inspired_N2_Pressure)/
*      Nitrogen_Time_Constant(I))
*      /(Helium_Pressure(I) + Nitrogen_Pressure(I) -
*      Surface_Inspired_N2_Pressure)

C      ELSE IF ((Nitrogen_Pressure(I) .LE.
*      Surface_Inspired_N2_Pressure).AND.
*      (Helium_Pressure(I)+Nitrogen_Pressure(I) .GE.
*      Surface_Inspired_N2_Pressure)) THEN

C        Decay_Time_to_Zero_Gradient =
*      1.0/(Nitrogen_Time_Constant(I)-Helium_Time_Constant(I))
*      *ALOG((Surface_Inspired_N2_Pressure -
*      Nitrogen_Pressure(I))/Helium_Pressure(I))

C        Integral_Gradient_x_Time =
*      Helium_Pressure(I)/Helium_Time_Constant(I)*
*      (1.0-EXP(-Helium_Time_Constant(I)*
*      Decay_Time_to_Zero_Gradient))+
*      (Nitrogen_Pressure(I)-Surface_Inspired_N2_Pressure)/
*      Nitrogen_Time_Constant(I)*
*      (1.0-EXP(-Nitrogen_Time_Constant(I)*
*      Decay_Time_to_Zero_Gradient))

C        Surface_Phase_Volume_Time(I) =
*      Integral_Gradient_x_Time/(Helium_Pressure(I) +
*      Nitrogen_Pressure(I) - Surface_Inspired_N2_Pressure)

C      ELSE
C        Surface_Phase_Volume_Time(I) = 0.0
C      END IF
C    END DO
C=====
C    END OF SUBROUTINE
C=====
C    RETURN
C    END

```

```

C=====
C  SUBROUTINE CRITICAL_VOLUME
C  Purpose: This subprogram applies the VPM Critical Volume Algorithm.  This
C  algorithm will compute "relaxed" gradients for helium and nitrogen based
C  on the setting of the Critical Volume Parameter Lambda.
C=====
C  SUBROUTINE CRITICAL_VOLUME (Deco_Phase_Volume_Time)

C  IMPLICIT NONE

C=====
C  ARGUMENTS
C=====
C  REAL Deco_Phase_Volume_Time !input
C=====
C  LOCAL VARIABLES
C=====
C  INTEGER I !loop counter

C  REAL Parameter_Lambda_Pascals
C  REAL Adj_Crush_Pressure_He_Pascals, Adj_Crush_Pressure_N2_Pascals
C  REAL Initial_Allowable_Grad_He_Pa, Initial_Allowable_Grad_N2_Pa
C  REAL New_Allowable_Grad_He_Pascals, New_Allowable_Grad_N2_Pascals
C  REAL B, C

C=====
C  LOCAL ARRAYS
C=====
C  REAL Phase_Volume_Time(16)

C=====
C  GLOBAL CONSTANTS IN NAMED COMMON BLOCKS
C=====
C  REAL Surface_Tension_Gamma, Skin_Compression_GammaC
C  COMMON /Block_19/ Surface_Tension_Gamma, Skin_Compression_GammaC

C  REAL Crit_Volume_Parameter_Lambda
C  COMMON /Block_20/ Crit_Volume_Parameter_Lambda

C=====
C  GLOBAL VARIABLES IN NAMED COMMON BLOCKS
C=====
C  REAL Units_Factor
C  COMMON /Block_16/ Units_Factor

C=====
C  GLOBAL ARRAYS IN NAMED COMMON BLOCKS
C=====
C  REAL Adjusted_Critical_Radius_He(16) !input
C  REAL Adjusted_Critical_Radius_N2(16)
C  COMMON /Block_7/ Adjusted_Critical_Radius_He,
C  * Adjusted_Critical_Radius_N2

C  REAL Surface_Phase_Volume_Time(16) !input
C  COMMON /Block_11/ Surface_Phase_Volume_Time

C  REAL Adjusted_Crushing_Pressure_He(16) !input
C  REAL Adjusted_Crushing_Pressure_N2(16)
C  COMMON /Block_25/ Adjusted_Crushing_Pressure_He,
C  * Adjusted_Crushing_Pressure_N2

C  REAL Allowable_Gradient_He(16), Allowable_Gradient_N2 (16) !output
C  COMMON /Block_26/ Allowable_Gradient_He, Allowable_Gradient_N2

C  REAL Initial_Allowable_Gradient_He(16) !input
C  REAL Initial_Allowable_Gradient_N2(16)
C  COMMON /Block_27/
C  * Initial_Allowable_Gradient_He, Initial_Allowable_Gradient_N2
C=====
C  CALCULATIONS

```

C Note: Since the Critical Volume Parameter Lambda was defined in units of  
C fsw-min in the original papers by Yount and colleagues, the same  
C convention is retained here. Although Lambda is adjustable only in units  
C of fsw-min in the program settings (range from 6500 to 8300 with default  
C 7500), it will convert to the proper value in Pascals-min in this  
C subroutine regardless of which diving pressure units are being used in  
C the main program - feet of seawater (fsw) or meters of seawater (msw).  
C The allowable gradient is computed using the quadratic formula (refer to  
C separate write-up posted on the Deco List web site).

```

C=====
  Parameter_Lambda_Pascals = (Crit_Volume_Parameter_Lambda/33.0)
  *                               * 101325.0

  DO I = 1,16
    Phase_Volume_Time(I) = Deco_Phase_Volume_Time +
  *   Surface_Phase_Volume_Time(I)
  END DO

  DO I = 1,16
    Adj_Crush_Pressure_He_Pascals =
  *   (Adjusted_Crushing_Pressure_He(I)/Units_Factor) * 101325.0

    Initial_Allowable_Grad_He_Pa =
  *   (Initial_Allowable_Gradient_He(I)/Units_Factor) * 101325.0

    B = Initial_Allowable_Grad_He_Pa +
  *   (Parameter_Lambda_Pascals*Surface_Tension_Gamma)/
  *   (Skin_Compression_GammaC*Phase_Volume_Time(I))

    C = (Surface_Tension_Gamma*(Surface_Tension_Gamma*
  *   (Parameter_Lambda_Pascals*
  *   Adj_Crush_Pressure_He_Pascals)))
  *   /(Skin_Compression_GammaC*(Skin_Compression_GammaC*
  *   Phase_Volume_Time(I)))

    New_Allowable_Grad_He_Pascals = (B + SQRT(B**2
  *   - 4.0*C))/2.0

    Allowable_Gradient_He(I) =
  *   (New_Allowable_Grad_He_Pascals/101325.0)*Units_Factor
  END DO

  DO I = 1,16
    Adj_Crush_Pressure_N2_Pascals =
  *   (Adjusted_Crushing_Pressure_N2(I)/Units_Factor) * 101325.0

    Initial_Allowable_Grad_N2_Pa =
  *   (Initial_Allowable_Gradient_N2(I)/Units_Factor) * 101325.0

    B = Initial_Allowable_Grad_N2_Pa +
  *   (Parameter_Lambda_Pascals*Surface_Tension_Gamma)/
  *   (Skin_Compression_GammaC*Phase_Volume_Time(I))

    C = (Surface_Tension_Gamma*(Surface_Tension_Gamma*
  *   (Parameter_Lambda_Pascals*
  *   Adj_Crush_Pressure_N2_Pascals)))
  *   /(Skin_Compression_GammaC*(Skin_Compression_GammaC*
  *   Phase_Volume_Time(I)))

    New_Allowable_Grad_N2_Pascals = (B + SQRT(B**2
  *   - 4.0*C))/2.0

    Allowable_Gradient_N2(I) =
  *   (New_Allowable_Grad_N2_Pascals/101325.0)*Units_Factor
  END DO
C=====
C   END OF SUBROUTINE

```

```

C=====
C      RETURN
C      END

C=====
C      SUBROUTINE CALC_START_OF_DECO_ZONE
C      Purpose: This subroutine uses the Bisection Method to find the depth at
C      which the leading compartment just enters the decompression zone.
C      Source: "Numerical Recipes in Fortran 77", Cambridge University Press,
C      1992.
C=====
C      SUBROUTINE CALC_START_OF_DECO_ZONE (Starting_Depth, Rate,
C      *                               Depth_Start_of_Deco_Zone)

C      IMPLICIT NONE

C=====
C      ARGUMENTS
C=====
C      REAL Starting_Depth, Rate, Depth_Start_of_Deco_Zone          !input
C=====
C      LOCAL VARIABLES
C=====
C      INTEGER I, J                                              !loop counters

C      REAL Initial_Helium_Pressure, Initial_Nitrogen_Pressure
C      REAL Initial_Inspired_He_Pressure
C      REAL Initial_Inspired_N2_Pressure
C      REAL Time_to_Start_of_Deco_Zone, Helium_Rate, Nitrogen_Rate
C      REAL Starting_Ambient_Pressure
C      REAL Cpt_Depth_Start_of_Deco_Zone, Low_Bound, High_Bound
C      REAL High_Bound_Helium_Pressure, High_Bound_Nitrogen_Pressure
C      REAL Mid_Range_Helium_Pressure, Mid_Range_Nitrogen_Pressure
C      REAL Function_at_High_Bound, Function_at_Low_Bound, Mid_Range_Time
C      REAL Function_at_Mid_Range, Differential_Change, Last_Diff_Change

C      REAL SCHREINER_EQUATION                                !function subprogram
C=====
C      GLOBAL CONSTANTS IN NAMED COMMON BLOCKS
C=====
C      REAL Water_Vapor_Pressure
C      COMMON /Block_8/ Water_Vapor_Pressure

C      REAL Constant_Pressure_Other_Gases
C      COMMON /Block_17/ Constant_Pressure_Other_Gases
C=====
C      GLOBAL VARIABLES IN NAMED COMMON BLOCKS
C=====
C      INTEGER Mix_Number
C      COMMON /Block_9/ Mix_Number

C      REAL Barometric_Pressure
C      COMMON /Block_18/ Barometric_Pressure
C=====
C      GLOBAL ARRAYS IN NAMED COMMON BLOCKS
C=====
C      REAL Helium_Time_Constant(16)
C      COMMON /Block_1A/ Helium_Time_Constant

C      REAL Nitrogen_Time_Constant(16)
C      COMMON /Block_1B/ Nitrogen_Time_Constant

C      REAL Helium_Pressure(16), Nitrogen_Pressure(16)
C      COMMON /Block_3/ Helium_Pressure, Nitrogen_Pressure

C      REAL Fraction_Helium(10), Fraction_Nitrogen(10)

```

```

COMMON /Block_5/ Fraction_Helium, Fraction_Nitrogen
C=====
C    CALCULATIONS
C    First initialize some variables
C=====
    Depth_Start_of-Deco_Zone = 0.0
    Starting_Ambient_Pressure = Starting_Depth + Barometric_Pressure

    Initial_Inspired_He_Pressure = (Starting_Ambient_Pressure -
*      Water_Vapor_Pressure)*Fraction_Helium(Mix_Number)

    Initial_Inspired_N2_Pressure = (Starting_Ambient_Pressure -
*      Water_Vapor_Pressure)*Fraction_Nitrogen(Mix_Number)

    Helium_Rate = Rate * Fraction_Helium(Mix_Number)
    Nitrogen_Rate = Rate * Fraction_Nitrogen(Mix_Number)
C=====
C    ESTABLISH THE BOUNDS FOR THE ROOT SEARCH USING THE BISECTION METHOD
C    AND CHECK TO MAKE SURE THAT THE ROOT WILL BE WITHIN BOUNDS.  PROCESS
C    EACH COMPARTMENT INDIVIDUALLY AND FIND THE MAXIMUM DEPTH ACROSS ALL
C    COMPARTMENTS (LEADING COMPARTMENT)
C    In this case, we are solving for time - the time when the gas tension in
C    the compartment will be equal to ambient pressure.  The low bound for time
C    is set at zero and the high bound is set at the time it would take to
C    ascend to zero ambient pressure (absolute).  Since the ascent rate is
C    negative, a multiplier of -1.0 is used to make the time positive.  The
C    desired point when gas tension equals ambient pressure is found at a time
C    somewhere between these endpoints.  The algorithm checks to make sure that
C    the solution lies in between these bounds by first computing the low bound
C    and high bound function values.
C=====
    Low_Bound = 0.0
    High_Bound = -1.0*(Starting_Ambient_Pressure/Rate)
    DO 200 I = 1,16
        Initial_Helium_Pressure = Helium_Pressure(I)
        Initial_Nitrogen_Pressure = Nitrogen_Pressure(I)

        Function_at_Low_Bound = Initial_Helium_Pressure +
*          Initial_Nitrogen_Pressure + Constant_Pressure_Other_Gases
*          - Starting_Ambient_Pressure

        High_Bound_Helium_Pressure = SCHREINER_EQUATION
*          (Initial_Inspired_He_Pressure, Helium_Rate,
*          High_Bound, Helium_Time_Constant(I),
*          Initial_Helium_Pressure)

        High_Bound_Nitrogen_Pressure = SCHREINER_EQUATION
*          (Initial_Inspired_N2_Pressure, Nitrogen_Rate,
*          High_Bound, Nitrogen_Time_Constant(I),
*          Initial_Nitrogen_Pressure)

        Function_at_High_Bound = High_Bound_Helium_Pressure +
*          High_Bound_Nitrogen_Pressure+Constant_Pressure_Other_Gases

        IF ((Function_at_High_Bound * Function_at_Low_Bound) .GE. 0.0)
*          THEN
            PRINT *, 'ERROR! ROOT IS NOT WITHIN BRACKETS'
            PAUSE
        END IF
C=====
C    APPLY THE BISECTION METHOD IN SEVERAL ITERATIONS UNTIL A SOLUTION WITH
C    THE DESIRED ACCURACY IS FOUND
C    Note: the program allows for up to 100 iterations.  Normally an exit will
C    be made from the loop well before that number.  If, for some reason, the
C    program exceeds 100 iterations, there will be a pause to alert the user.
C=====

```



```

      IF (Function_at_Low_Bound .LT. 0.0) THEN
        Time_to_Start_of_Deco_Zone = Low_Bound
        Differential_Change = High_Bound - Low_Bound
      ELSE
        Time_to_Start_of_Deco_Zone = High_Bound
        Differential_Change = Low_Bound - High_Bound
      END IF
      DO 150 J = 1, 100
        Last_Diff_Change = Differential_Change
        Differential_Change = Last_Diff_Change*0.5

        Mid_Range_Time = Time_to_Start_of_Deco_Zone +
*                               Differential_Change

        Mid_Range_Helium_Pressure = SCHREINER_EQUATION
*                               (Initial_Inspired_He_Pressure, Helium_Rate,
*                               Mid_Range_Time, Helium_Time_Constant(I),
*                               Initial_Helium_Pressure)

        Mid_Range_Nitrogen_Pressure = SCHREINER_EQUATION
*                               (Initial_Inspired_N2_Pressure, Nitrogen_Rate,
*                               Mid_Range_Time, Nitrogen_Time_Constant(I),
*                               Initial_Nitrogen_Pressure)

        Function_at_Mid_Range =
*                               Mid_Range_Helium_Pressure +
*                               Mid_Range_Nitrogen_Pressure +
*                               Constant_Pressure_Other_Gases -
*                               (Starting_Ambient_Pressure + Rate*Mid_Range_Time)

        IF (Function_at_Mid_Range .LE. 0.0)
*           Time_to_Start_of_Deco_Zone = Mid_Range_Time

        IF ((ABS(Differential_Change) .LT. 1.0E-3) .OR.
*          (Function_at_Mid_Range .EQ. 0.0)) GOTO 170
150    CONTINUE
        PRINT *, 'ERROR! ROOT SEARCH EXCEEDED MAXIMUM ITERATIONS'
        PAUSE

C=====
C    When a solution with the desired accuracy is found, the program jumps out
C    of the loop to Line 170 and assigns the solution value for the individual
C    compartment.
C=====
170    Cpt_Depth_Start_of_Deco_Zone = (Starting_Ambient_Pressure +
*          Rate*Time_to_Start_of_Deco_Zone) - Barometric_Pressure
C=====
C    The overall solution will be the compartment with the maximum depth where
C    gas tension equals ambient pressure (leading compartment).
C=====
        Depth_Start_of_Deco_Zone = MAX(Depth_Start_of_Deco_Zone,
*          Cpt_Depth_Start_of_Deco_Zone)
200    CONTINUE
C=====
C    END OF SUBROUTINE
C=====
        RETURN
        END

C=====
C    SUBROUTINE PROJECTED_ASCENT
C    Purpose: This subprogram performs a simulated ascent outside of the main
C    program to ensure that a deco ceiling will not be violated due to unusual
C    gas loading during ascent (on-gassing). If the deco ceiling is violated,
C    the stop depth will be adjusted deeper by the step size until a safe
C    ascent can be made.

```

```

C=====
      SUBROUTINE PROJECTED_ASCENT (Starting_Depth, Rate,
*                               Deco_Stop_Depth, Step_Size)
      IMPLICIT NONE
C=====
C      ARGUMENTS
C=====
      REAL Starting_Depth, Rate, Step_Size                !input
      REAL Deco_Stop_Depth                                !input and output
C=====
C      LOCAL VARIABLES
C=====
      INTEGER I                                           !loop counter

      REAL Initial_Inspired_He_Pressure, Initial_Inspired_N2_Pressure
      REAL Helium_Rate, Nitrogen_Rate
      REAL Starting_Ambient_Pressure, Ending_Ambient_Pressure
      REAL New_Ambient_Pressure, Segment_Time
      REAL Temp_Helium_Pressure, Temp_Nitrogen_Pressure
      REAL Weighted_Allowable_Gradient

      REAL SCHREINER_EQUATION                            !function subprogram
C=====
C      LOCAL ARRAYS
C=====
      REAL Initial_Helium_Pressure(16), Initial_Nitrogen_Pressure(16)
      REAL Temp_Gas>Loading(16), Allowable_Gas>Loading (16)
C=====
C      GLOBAL CONSTANTS IN NAMED COMMON BLOCKS
C=====
      REAL Water_Vapor_Pressure
      COMMON /Block_8/ Water_Vapor_Pressure

      REAL Constant_Pressure_Other_Gases
      COMMON /Block_17/ Constant_Pressure_Other_Gases
C=====
C      GLOBAL VARIABLES IN NAMED COMMON BLOCKS
C=====
      INTEGER Mix_Number
      COMMON /Block_9/ Mix_Number

      REAL Barometric_Pressure
      COMMON /Block_18/ Barometric_Pressure
C=====
C      GLOBAL ARRAYS IN NAMED COMMON BLOCKS
C=====
      REAL Helium_Time_Constant(16)
      COMMON /Block_1A/ Helium_Time_Constant

      REAL Nitrogen_Time_Constant(16)
      COMMON /Block_1B/ Nitrogen_Time_Constant

      REAL Helium_Pressure(16), Nitrogen_Pressure(16)      !input
      COMMON /Block_3/ Helium_Pressure, Nitrogen_Pressure

      REAL Fraction_Helium(10), Fraction_Nitrogen(10)
      COMMON /Block_5/ Fraction_Helium, Fraction_Nitrogen

      REAL Allowable_Gradient_He(16), Allowable_Gradient_N2 (16) !input
      COMMON /Block_26/ Allowable_Gradient_He, Allowable_Gradient_N2
C=====
C      CALCULATIONS
C=====
      New_Ambient_Pressure = Deco_Stop_Depth + Barometric_Pressure
      Starting_Ambient_Pressure = Starting_Depth + Barometric_Pressure

```

```

Initial_Inspired_He_Pressure = (Starting_Ambient_Pressure -
*   Water_Vapor_Pressure) * Fraction_Helium(Mix_Number)

Initial_Inspired_N2_Pressure = (Starting_Ambient_Pressure -
*   Water_Vapor_Pressure) * Fraction_Nitrogen(Mix_Number)

Helium_Rate = Rate * Fraction_Helium(Mix_Number)
Nitrogen_Rate = Rate * Fraction_Nitrogen(Mix_Number)
DO I = 1,16
    Initial_Helium_Pressure(I) = Helium_Pressure(I)
    Initial_Nitrogen_Pressure(I) = Nitrogen_Pressure(I)
END DO
665 Ending_Ambient_Pressure = New_Ambient_Pressure

Segment_Time = (Ending_Ambient_Pressure -
*   Starting_Ambient_Pressure) / Rate

DO 670 I = 1,16
    Temp_Helium_Pressure = SCHREINER_EQUATION
*   (Initial_Inspired_He_Pressure, Helium_Rate,
*   Segment_Time, Helium_Time_Constant(I),
*   Initial_Helium_Pressure(I))

    Temp_Nitrogen_Pressure = SCHREINER_EQUATION
*   (Initial_Inspired_N2_Pressure, Nitrogen_Rate,
*   Segment_Time, Nitrogen_Time_Constant(I),
*   Initial_Nitrogen_Pressure(I))

    Temp_Gas>Loading(I) = Temp_Helium_Pressure +
*   Temp_Nitrogen_Pressure

    IF (Temp_Gas>Loading(I) .GT. 0.0) THEN
        Weighted_Allowable_Gradient =
*   (Allowable_Gradient_He(I) * Temp_Helium_Pressure +
*   Allowable_Gradient_N2(I) * Temp_Nitrogen_Pressure) /
*   Temp_Gas>Loading(I)
    ELSE
        Weighted_Allowable_Gradient =
*   MIN(Allowable_Gradient_He(I), Allowable_Gradient_N2(I))
    END IF

    Allowable_Gas>Loading(I) = Ending_Ambient_Pressure +
*   Weighted_Allowable_Gradient - Constant_Pressure_Other_Gases

670 CONTINUE
DO 671 I = 1,16
    IF (Temp_Gas>Loading(I) .GT. Allowable_Gas>Loading(I)) THEN
        New_Ambient_Pressure = Ending_Ambient_Pressure + Step_Size
        Deco_Stop_Depth = Deco_Stop_Depth + Step_Size
        GOTO 665
    END IF
671 CONTINUE
C=====
C   END OF SUBROUTINE
C=====
RETURN
END

```

```

C=====
C   SUBROUTINE BOYLES_LAW_COMPENSATION
C   Purpose: This subprogram calculates the reduction in allowable gradients
C   with decreasing ambient pressure during the decompression profile based
C   on Boyle's Law considerations.
C=====
C   SUBROUTINE BOYLES_LAW_COMPENSATION (First_Stop_Depth,

```

```

*                               Deco_Stop_Depth, Step_Size)

IMPLICIT NONE
=====
C ARGUMENTS
=====
C   REAL First_Stop_Depth, Deco_Stop_Depth, Step_Size           !input
=====
C LOCAL VARIABLES
=====
C   INTEGER I                                                    !loop counter

REAL Next_Stop
REAL Ambient_Pressure_First_Stop, Ambient_Pressure_Next_Stop
REAL Amb_Press_First_Stop_Pascals, Amb_Press_Next_Stop_Pascals
REAL A, B, C, Low_Bound, High_Bound, Ending_Radius
REAL Deco_Gradient_Pascals
REAL Allow_Grad_First_Stop_He_Pa, Radius_First_Stop_He
REAL Allow_Grad_First_Stop_N2_Pa, Radius_First_Stop_N2

=====
C LOCAL ARRAYS
=====
C   REAL Radius1_He(16), Radius2_He(16)
C   REAL Radius1_N2(16), Radius2_N2(16)

=====
C GLOBAL CONSTANTS IN NAMED COMMON BLOCKS
=====
C   REAL Surface_Tension_Gamma, Skin_Compression_GammaC
C   COMMON /Block_19/ Surface_Tension_Gamma, Skin_Compression_GammaC

=====
C GLOBAL VARIABLES IN NAMED COMMON BLOCKS
=====
C   REAL Barometric_Pressure
C   COMMON /Block_18/ Barometric_Pressure

REAL Units_Factor
COMMON /Block_16/ Units_Factor

=====
C GLOBAL ARRAYS IN NAMED COMMON BLOCKS
=====
C   REAL Allowable_Gradient_He(16), Allowable_Gradient_N2(16)
C   COMMON /Block_26/ Allowable_Gradient_He, Allowable_Gradient_N2

REAL Deco_Gradient_He(16), Deco_Gradient_N2(16)
COMMON /Block_34/ Deco_Gradient_He, Deco_Gradient_N2

=====
C CALCULATIONS
=====
C   Next_Stop = Deco_Stop_Depth - Step_Size

C   Ambient_Pressure_First_Stop = First_Stop_Depth +
*                               Barometric_Pressure

Ambient_Pressure_Next_Stop = Next_Stop + Barometric_Pressure

Amb_Press_First_Stop_Pascals =
*   (Ambient_Pressure_First_Stop/Units_Factor) * 101325.0

Amb_Press_Next_Stop_Pascals =
*   (Ambient_Pressure_Next_Stop/Units_Factor) * 101325.0

DO I = 1,16

```

```
Allow_Grad_First_Stop_He_Pa =  
* (Allowable_Gradient_He(I)/Units_Factor) * 101325.0
```

```
Radius_First_Stop_He = (2.0 * Surface_Tension_Gamma) /  
* Allow_Grad_First_Stop_He_Pa
```

```
Radius1_He(I) = Radius_First_Stop_He  
A = Amb_Press_Next_Stop_Pascals  
B = -2.0 * Surface_Tension_Gamma  
C = (Amb_Press_First_Stop_Pascals + (2.0*Surface_Tension_Gamma)/  
* Radius_First_Stop_He)* Radius_First_Stop_He*  
* (Radius_First_Stop_He*(Radius_First_Stop_He))  
Low_Bound = Radius_First_Stop_He  
High_Bound = Radius_First_Stop_He*(Amb_Press_First_Stop_Pascals/  
* Amb_Press_Next_Stop_Pascals)**(1.0/3.0)
```

```
CALL RADIUS_ROOT_FINDER (A,B,C, Low_Bound, High_Bound,  
* Ending_Radius)
```

```
Radius2_He(I) = Ending_Radius  
Deco_Gradient_Pascals = (2.0 * Surface_Tension_Gamma) /  
* Ending_Radius
```

```
Deco_Gradient_He(I) = (Deco_Gradient_Pascals / 101325.0)*  
* Units_Factor
```

```
END DO
```

```
DO I = 1,16  
Allow_Grad_First_Stop_N2_Pa =  
* (Allowable_Gradient_N2(I)/Units_Factor) * 101325.0
```

```
Radius_First_Stop_N2 = (2.0 * Surface_Tension_Gamma) /  
* Allow_Grad_First_Stop_N2_Pa
```

```
Radius1_N2(I) = Radius_First_Stop_N2  
A = Amb_Press_Next_Stop_Pascals  
B = -2.0 * Surface_Tension_Gamma  
C = (Amb_Press_First_Stop_Pascals + (2.0*Surface_Tension_Gamma)/  
* Radius_First_Stop_N2)* Radius_First_Stop_N2*  
* (Radius_First_Stop_N2*(Radius_First_Stop_N2))  
Low_Bound = Radius_First_Stop_N2  
High_Bound = Radius_First_Stop_N2*(Amb_Press_First_Stop_Pascals/  
* Amb_Press_Next_Stop_Pascals)**(1.0/3.0)
```

```
CALL RADIUS_ROOT_FINDER (A,B,C, Low_Bound, High_Bound,  
* Ending_Radius)
```

```
Radius2_N2(I) = Ending_Radius  
Deco_Gradient_Pascals = (2.0 * Surface_Tension_Gamma) /  
* Ending_Radius
```

```
Deco_Gradient_N2(I) = (Deco_Gradient_Pascals / 101325.0)*  
* Units_Factor
```

```
END DO
```

```
C=====
```

```
C      END OF SUBROUTINE
```

```
C=====
```

```
      RETURN
```

```
      END
```

```
C=====
```

```
C      SUBROUTINE DECOMPRESSION STOP
```

```
C      Purpose: This subprogram calculates the required time at each
```

```
C      decompression stop.
```

```

C=====
SUBROUTINE DECOMPRESSION_STOP (Deco_Stop_Depth, Step_Size)

IMPLICIT NONE

C=====
C ARGUMENTS
C=====
REAL Deco_Stop_Depth, Step_Size !input
C=====
C LOCAL VARIABLES
C=====
CHARACTER OS_Command*3

INTEGER I !loop counter
INTEGER Last_Segment_Number

REAL Ambient_Pressure
REAL Inspired_Helium_Pressure, Inspired_Nitrogen_Pressure
REAL Last_Run_Time
REAL Deco_Ceiling_Depth, Next_Stop
REAL Round_Up_Operation, Temp_Segment_Time, Time_Counter
REAL Weighted_Allowable_Gradient

REAL HALDANE_EQUATION !function subprogram
C=====
C LOCAL ARRAYS
C=====
REAL Initial_Helium_Pressure(16)
REAL Initial_Nitrogen_Pressure(16)
C=====
C GLOBAL CONSTANTS IN NAMED COMMON BLOCKS
C=====
REAL Water_Vapor_Pressure
COMMON /Block_8/ Water_Vapor_Pressure

REAL Constant_Pressure_Other_Gases
COMMON /Block_17/ Constant_Pressure_Other_Gases

REAL Minimum_Deco_Stop_Time
COMMON /Block_21/ Minimum_Deco_Stop_Time
C=====
C GLOBAL VARIABLES IN NAMED COMMON BLOCKS
C=====
INTEGER Segment_Number
REAL Run_Time, Segment_Time
COMMON /Block_2/ Run_Time, Segment_Number, Segment_Time

REAL Ending_Ambient_Pressure
COMMON /Block_4/ Ending_Ambient_Pressure

INTEGER Mix_Number
COMMON /Block_9/ Mix_Number

REAL Barometric_Pressure
COMMON /Block_18/ Barometric_Pressure
C=====
C GLOBAL ARRAYS IN NAMED COMMON BLOCKS
C=====
REAL Helium_Time_Constant(16)
COMMON /Block_1A/ Helium_Time_Constant

REAL Nitrogen_Time_Constant(16)
COMMON /Block_1B/ Nitrogen_Time_Constant

REAL Helium_Pressure(16), Nitrogen_Pressure(16) !both input
COMMON /Block_3/ Helium_Pressure, Nitrogen_Pressure !and output

```

```
REAL Fraction_Helium(10), Fraction_Nitrogen(10)
COMMON /Block_5/ Fraction_Helium, Fraction_Nitrogen
```

```
REAL Deco_Gradient_He(16), Deco_Gradient_N2(16)
```

```
COMMON /Block_34/ Deco_Gradient_He, Deco_Gradient_N2
```

```
C=====
C  CALCULATIONS
C=====
```

```
OS_Command = 'CLS'
Last_Run_Time = Run_Time
Round_Up_Operation = ANINT((Last_Run_Time/Minimum_Deco_Stop_Time)
*                          + 0.5) * Minimum_Deco_Stop_Time
Segment_Time = Round_Up_Operation - Run_Time
Run_Time = Round_Up_Operation
Temp_Segment_Time = Segment_Time
Last_Segment_Number = Segment_Number
Segment_Number = Last_Segment_Number + 1
Ambient_Pressure = Deco_Stop_Depth + Barometric_Pressure
Ending_Ambient_Pressure = Ambient_Pressure
Next_Stop = Deco_Stop_Depth - Step_Size
```

```
Inspired_Helium_Pressure = (Ambient_Pressure -
*   Water_Vapor_Pressure)*Fraction_Helium(Mix_Number)
```

```
Inspired_Nitrogen_Pressure = (Ambient_Pressure -
*   Water_Vapor_Pressure)*Fraction_Nitrogen(Mix_Number)
```

```
C=====
C  Check to make sure that program won't lock up if unable to decompress
C  to the next stop.  If so, write error message and terminate program.
C=====
```

```
DO I = 1,16
IF ((Inspired_Helium_Pressure + Inspired_Nitrogen_Pressure)
*   .GT. 0.0) THEN
    Weighted_Allowable_Gradient =
*   (Deco_Gradient_He(I)* Inspired_Helium_Pressure +
*   Deco_Gradient_N2(I)* Inspired_Nitrogen_Pressure) /
*   (Inspired_Helium_Pressure + Inspired_Nitrogen_Pressure)
```

```
IF ((Inspired_Helium_Pressure + Inspired_Nitrogen_Pressure +
*   Constant_Pressure_Other_Gases - Weighted_Allowable_Gradient)
*   .GT. (Next_Stop + Barometric_Pressure)) THEN
    CALL SYSTEMQQ (OS_Command)
    WRITE (*,905) Deco_Stop_Depth
    WRITE (*,906)
    WRITE (*,907)
    STOP 'PROGRAM TERMINATED'
END IF
END IF
END DO
```

```
700 DO 720 I = 1,16
    Initial_Helium_Pressure(I) = Helium_Pressure(I)
    Initial_Nitrogen_Pressure(I) = Nitrogen_Pressure(I)
```

```
    Helium_Pressure(I) = HALDANE_EQUATION
*   (Initial_Helium_Pressure(I), Inspired_Helium_Pressure,
*   Helium_Time_Constant(I), Segment_Time)
```

```
    Nitrogen_Pressure(I) = HALDANE_EQUATION
*   (Initial_Nitrogen_Pressure(I), Inspired_Nitrogen_Pressure,
*   Nitrogen_Time_Constant(I), Segment_Time)
```

```
720 CONTINUE
    CALL CALC_DECO_CEILING (Deco_Ceiling_Depth)
    IF (Deco_Ceiling_Depth .GT. Next_Stop) THEN
```

```

        Segment_Time = Minimum_Deco_Stop_Time
        Time_Counter = Temp_Segment_Time
        Temp_Segment_Time = Time_Counter + Minimum_Deco_Stop_Time
        Last_Run_Time = Run_Time
        Run_Time = Last_Run_Time + Minimum_Deco_Stop_Time
        GOTO 700
    END IF
    Segment_Time = Temp_Segment_Time

RETURN
=====
C    FORMAT STATEMENTS - ERROR MESSAGES
=====
905    FORMAT ('0ERROR! OFF-GASSING GRADIENT IS TOO SMALL TO DECOMPRESS'
*1X, 'AT THE', F6.1, 1X, 'STOP')
906    FORMAT ('0REDUCE STEP SIZE OR INCREASE OXYGEN FRACTION')
907    FORMAT (' ')
=====
C    END OF SUBROUTINE
=====
C    END
=====

=====
C
C    SUBROUTINE CALC DECO CEILING
C    Purpose: This subprogram calculates the deco ceiling (the safe ascent
C    depth) in each compartment, based on the allowable "deco gradients"
C    computed in the Boyle's Law Compensation subroutine, and then finds the
C    deepest deco ceiling across all compartments. This deepest value
C    (Deco Ceiling Depth) is then used by the Decompression Stop subroutine
C    to determine the actual deco schedule.
=====
C    SUBROUTINE CALC_DECO_CEILING (Deco_Ceiling_Depth)
=====

IMPLICIT NONE
=====
C    ARGUMENTS
=====
C    REAL Deco Ceiling Depth                                !output
=====
C    LOCAL VARIABLES
=====
C    INTEGER I                                              !loop counter
=====

REAL Gas>Loading, Weighted Allowable_Gradient
REAL>Tolerated Ambient Pressure
=====
C    LOCAL ARRAYS
=====
C    REAL Compartment Deco Ceiling(16)
=====
C    GLOBAL CONSTANTS IN NAMED COMMON BLOCKS
=====
C    REAL Constant Pressure Other Gases
COMMON /Block_17/ Constant Pressure Other Gases
=====
C    GLOBAL VARIABLES IN NAMED COMMON BLOCKS
=====
C    REAL Barometric Pressure
COMMON /Block_18/ Barometric Pressure
=====
C    GLOBAL ARRAYS IN NAMED COMMON BLOCKS
=====
C    REAL Helium Pressure(16), Nitrogen Pressure(16)        !input
COMMON /Block_3/ Helium_Pressure, Nitrogen_Pressure

```



```

      REAL Deco_Gradient_He(16), Deco_Gradient_N2(16)                                !input
      COMMON /Block 34/ Deco_Gradient_He, Deco_Gradient_N2
C=====
C      CALCULATIONS
C      Since there are two sets of deco gradients being tracked, one for
C      helium and one for nitrogen, a "weighted allowable gradient" must be
C      computed each time based on the proportions of helium and nitrogen in
C      each compartment. This proportioning follows the methodology of
C      Buhlmann/Keller. If there is no helium and nitrogen in the compartment,
C      such as after extended periods of oxygen breathing, then the minimum value
C      across both gases will be used. It is important to note that if a
C      compartment is empty of helium and nitrogen, then the weighted allowable
C      gradient formula cannot be used since it will result in division by zero.
C=====
      DO I = 1,16
        Gas>Loading = Helium_Pressure(I) + Nitrogen_Pressure(I)
        IF (Gas>Loading .GT. 0.0) THEN
          Weighted Allowable Gradient =
          * (Deco_Gradient_He(I)* Helium_Pressure(I) +
          * Deco_Gradient_N2(I)* Nitrogen_Pressure(I)) /
          * (Helium_Pressure(I) + Nitrogen_Pressure(I))
          Tolerated Ambient Pressure = (Gas>Loading +
          * Constant_Pressure_Other_Gases) - Weighted Allowable Gradient
        ELSE
          Weighted Allowable Gradient =
          * MIN(Deco_Gradient_He(I), Deco_Gradient_N2(I))
          Tolerated Ambient Pressure =
          * Constant_Pressure_Other_Gases - Weighted Allowable Gradient
        END IF
C=====
C      The tolerated ambient pressure cannot be less than zero absolute, i.e.,
C      the vacuum of outer space!
C=====
      IF (Tolerated Ambient Pressure .LT. 0.0) THEN
        Tolerated Ambient Pressure = 0.0
      END IF
C=====
C      The Deco Ceiling Depth is computed in a loop after all of the individual
C      compartment deco ceilings have been calculated. It is important that the
C      Deco Ceiling Depth (max deco ceiling across all compartments) only be
C      extracted from the compartment values and not be compared against some
C      initialization value. For example, if MAX(Deco_Ceiling_Depth . .) was
C      compared against zero, this could cause a program lockup because sometimes
C      the Deco Ceiling Depth needs to be negative (but not less than absolute
C      zero) in order to decompress to the last stop at zero depth.
C=====
      Compartment_Deco_Ceiling(I) =
      * Tolerated Ambient Pressure - Barometric Pressure
      END DO
      Deco_Ceiling_Depth = Compartment_Deco_Ceiling(1)
      DO I = 2,16
        Deco_Ceiling_Depth =
        * MAX(Deco_Ceiling_Depth, Compartment_Deco_Ceiling(I))
      END DO
C=====
C      END OF SUBROUTINE
C=====
      RETURN
      END

```

```

C=====
C  SUBROUTINE GAS_LOADINGS_SURFACE_INTERVAL
C  Purpose: This subprogram calculates the gas loading (off-gassing) during
C  a surface interval.
C=====
C  SUBROUTINE GAS_LOADINGS_SURFACE_INTERVAL (Surface_Interval_Time)

      IMPLICIT NONE

C=====
C  ARGUMENTS
C=====
      REAL Surface_Interval_Time                                !input
C=====
C  LOCAL VARIABLES
C=====
      INTEGER I                                                !loop counter

      REAL Inspired_Helium_Pressure, Inspired_Nitrogen_Pressure
      REAL Initial_Helium_Pressure, Initial_Nitrogen_Pressure

      REAL HALDANE_EQUATION                                    !function subprogram
C=====
C  GLOBAL CONSTANTS IN NAMED COMMON BLOCKS
C=====
      REAL Water_Vapor_Pressure
      COMMON /Block_8/ Water_Vapor_Pressure
C=====
C  GLOBAL VARIABLES IN NAMED COMMON BLOCKS
C=====
      REAL Barometric_Pressure
      COMMON /Block_18/ Barometric_Pressure
C=====
C  GLOBAL ARRAYS IN NAMED COMMON BLOCKS
C=====
      REAL Helium_Time_Constant(16)
      COMMON /Block_1A/ Helium_Time_Constant

      REAL Nitrogen_Time_Constant(16)
      COMMON /Block_1B/ Nitrogen_Time_Constant

      REAL Helium_Pressure(16), Nitrogen_Pressure(16)          !both input
      COMMON /Block_3/ Helium_Pressure, Nitrogen_Pressure      !and output
C=====
C  CALCULATIONS
C=====
      Inspired_Helium_Pressure = 0.0
      Inspired_Nitrogen_Pressure = (Barometric_Pressure -
*      Water_Vapor_Pressure)*0.79
      DO I = 1,16
        Initial_Helium_Pressure = Helium_Pressure(I)
        Initial_Nitrogen_Pressure = Nitrogen_Pressure(I)

        Helium_Pressure(I) = HALDANE_EQUATION
*      (Initial_Helium_Pressure, Inspired_Helium_Pressure,
*      Helium_Time_Constant(I), Surface_Interval_Time)

        Nitrogen_Pressure(I) = HALDANE_EQUATION
*      (Initial_Nitrogen_Pressure, Inspired_Nitrogen_Pressure,
*      Nitrogen_Time_Constant(I), Surface_Interval_Time)
      END DO
C=====
C  END OF SUBROUTINE
C=====
      RETURN
      END

```

```

C=====
C  SUBROUTINE VPM_REPETITIVE_ALGORITHM
C  Purpose: This subprogram implements the VPM Repetitive Algorithm that was
C  envisioned by Professor David E. Yount only months before his passing.
C=====
C  SUBROUTINE VPM_REPETITIVE_ALGORITHM (Surface_Interval_Time)

      IMPLICIT NONE

C=====
C  ARGUMENTS
C=====
      REAL Surface_Interval_Time                                !input
C=====
C  LOCAL VARIABLES
C=====
      INTEGER I                                                !loop counter

      REAL Max_Actual_Gradient_Pascals
      REAL Adj_Crush_Pressure_He_Pascals, Adj_Crush_Pressure_N2_Pascals
      REAL Initial_Allowable_Grad_He_Pa, Initial_Allowable_Grad_N2_Pa
      REAL New_Critical_Radius_He, New_Critical_Radius_N2

C=====
C  GLOBAL CONSTANTS IN NAMED COMMON BLOCKS
C=====
      REAL Surface_Tension_Gamma, Skin_Compression_GammaC
      COMMON /Block_19/ Surface_Tension_Gamma, Skin_Compression_GammaC

      REAL Regeneration_Time_Constant
      COMMON /Block_22/ Regeneration_Time_Constant

C=====
C  GLOBAL VARIABLES IN NAMED COMMON BLOCKS
C=====
      REAL Units_Factor
      COMMON /Block_16/ Units_Factor

C=====
C  GLOBAL ARRAYS IN NAMED COMMON BLOCKS
C=====
      REAL Initial_Critical_Radius_He(16)                      !input
      REAL Initial_Critical_Radius_N2(16)
      COMMON /Block_6/ Initial_Critical_Radius_He,
      *      Initial_Critical_Radius_N2

      REAL Adjusted_Critical_Radius_He(16)                     !output
      REAL Adjusted_Critical_Radius_N2(16)
      COMMON /Block_7/ Adjusted_Critical_Radius_He,
      *      Adjusted_Critical_Radius_N2

      REAL Max_Actual_Gradient(16)                             !input
      COMMON /Block_12/ Max_Actual_Gradient

      REAL Adjusted_Crushing_Pressure_He(16)                   !input
      REAL Adjusted_Crushing_Pressure_N2(16)
      COMMON /Block_25/ Adjusted_Crushing_Pressure_He,
      *      Adjusted_Crushing_Pressure_N2

      REAL Initial_Allowable_Gradient_He(16)                   !input
      REAL Initial_Allowable_Gradient_N2(16)
      COMMON /Block_27/
      *      Initial_Allowable_Gradient_He, Initial_Allowable_Gradient_N2

C=====
C  CALCULATIONS
C=====
      DO I = 1,16
      *      Max_Actual_Gradient_Pascals =
      *          (Max_Actual_Gradient(I)/Units_Factor) * 101325.0

```

```

      Adj_Crush_Pressure_He_Pascals =
*      (Adjusted_Crushing_Pressure_He(I)/Units_Factor) * 101325.0

      Adj_Crush_Pressure_N2_Pascals =
*      (Adjusted_Crushing_Pressure_N2(I)/Units_Factor) * 101325.0

      Initial_Allowable_Grad_He_Pa =
*      (Initial_Allowable_Gradient_He(I)/Units_Factor) * 101325.0

      Initial_Allowable_Grad_N2_Pa =
*      (Initial_Allowable_Gradient_N2(I)/Units_Factor) * 101325.0

      IF (Max_Actual_Gradient(I)
*      .GT. Initial_Allowable_Gradient_N2(I)) THEN

          New_Critical_Radius_N2 = ((2.0*Surface_Tension_Gamma*
*      (Skin_Compression_GammaC - Surface_Tension_Gamma)) /
*      (Max_Actual_Gradient_Pascals*Skin_Compression_GammaC -
*      Surface_Tension_Gamma*Adj_Crush_Pressure_N2_Pascals)

          Adjusted_Critical_Radius_N2(I) =
*      Initial_Critical_Radius_N2(I) +
*      (Initial_Critical_Radius_N2(I)-New_Critical_Radius_N2)*
*      EXP(-Surface_Interval_Time/Regeneration_Time_Constant)
      ELSE
          Adjusted_Critical_Radius_N2(I) =
*      Initial_Critical_Radius_N2(I)
      END IF

      IF (Max_Actual_Gradient(I)
*      .GT. Initial_Allowable_Gradient_He(I)) THEN

          New_Critical_Radius_He = ((2.0*Surface_Tension_Gamma*
*      (Skin_Compression_GammaC - Surface_Tension_Gamma)) /
*      (Max_Actual_Gradient_Pascals*Skin_Compression_GammaC -
*      Surface_Tension_Gamma*Adj_Crush_Pressure_He_Pascals)

          Adjusted_Critical_Radius_He(I) =
*      Initial_Critical_Radius_He(I) +
*      (Initial_Critical_Radius_He(I)-New_Critical_Radius_He)*
*      EXP(-Surface_Interval_Time/Regeneration_Time_Constant)
      ELSE
          Adjusted_Critical_Radius_He(I) =
*      Initial_Critical_Radius_He(I)
      END IF
  END DO

C=====
C      END OF SUBROUTINE
C=====

      RETURN
      END

C=====
C      SUBROUTINE CALC_BAROMETRIC_PRESSURE
C      Purpose: This sub calculates barometric pressure at altitude based on the
C      publication "U.S. Standard Atmosphere, 1976", U.S. Government Printing
C      Office, Washington, D.C. The source for this code is a Fortran 90 program
C      written by Ralph L. Carmichael (retired NASA researcher) and endorsed by
C      the National Geophysical Data Center of the National Oceanic and
C      Atmospheric Administration. It is available for download free from
C      Public Domain Aeronautical Software at: http://www.pdas.com/atmos.htm
C=====
C      SUBROUTINE CALC_BAROMETRIC_PRESSURE (Altitude)

```

```

      IMPLICIT NONE
C=====
C      ARGUMENTS
C=====
      REAL Altitude                                     !input
C=====
C      LOCAL CONSTANTS
C=====
      REAL Radius_of_Earth, Acceleration_of_Gravity
      REAL Molecular_weight_of_Air, Gas_Constant_R
      REAL Temp_at_Sea_Level, Temp_Gradient
      REAL Pressure_at_Sea_Level_Fsw, Pressure_at_Sea_Level_Msw
C=====
C      LOCAL VARIABLES
C=====
      REAL Pressure_at_Sea_Level, GMR_Factor
      REAL Altitude_Feet, Altitude_Meters
      REAL Altitude_Kilometers, Geopotential_Altitude
      REAL Temp_at_Geopotential_Altitude
C=====
C      GLOBAL VARIABLES IN NAMED COMMON BLOCKS
C=====
      LOGICAL Units_Equal_Fsw, Units_Equal_Msw
      COMMON /Block_15/ Units_Equal_Fsw, Units_Equal_Msw

      REAL Barometric_Pressure                         !output
      COMMON /Block_18/ Barometric_Pressure
C=====
C      CALCULATIONS
C=====
      Radius_of_Earth = 6369.0                         !kilometers
      Acceleration_of_Gravity = 9.80665                !meters/second^2
      Molecular_weight_of_Air = 28.9644                !mols
      Gas_Constant_R = 8.31432                         !Joules/mol*deg Kelvin
      Temp_at_Sea_Level = 288.15                      !degrees Kelvin

      Pressure_at_Sea_Level_Fsw = 33.0                 !feet of seawater based on 101325 Pa
                                                    !at sea level (Standard Atmosphere)

      Pressure_at_Sea_Level_Msw = 10.0                 !meters of seawater based on 100000 Pa
                                                    !at sea level (European System)

      Temp_Gradient = -6.5                            !Change in Temp deg Kelvin with
                                                    !change in geopotential altitude,
                                                    !valid for first layer of atmosphere
                                                    !up to 11 kilometers or 36,000 feet

      GMR_Factor = Acceleration_of_Gravity *
*              Molecular_weight_of_Air / Gas_Constant_R

      IF (Units_Equal_Fsw) THEN
        Altitude_Feet = Altitude
        Altitude_Kilometers = Altitude_Feet / 3280.839895
        Pressure_at_Sea_Level = Pressure_at_Sea_Level_Fsw
      END IF
      IF (Units_Equal_Msw) THEN
        Altitude_Meters = Altitude
        Altitude_Kilometers = Altitude_Meters / 1000.0
        Pressure_at_Sea_Level = Pressure_at_Sea_Level_Msw
      END IF

      Geopotential_Altitude = (Altitude_Kilometers * Radius_of_Earth) /
*              (Altitude_Kilometers + Radius_of_Earth)

      Temp_at_Geopotential_Altitude = Temp_at_Sea_Level
*              + Temp_Gradient * Geopotential_Altitude

```

```

        Barometric_Pressure = Pressure_at_Sea_Level *
*      EXP(ALOG(Temp_at_Sea_Level / Temp_at_Geopotential_Altitude) *
*      GMR_Factor / Temp_Gradient)
C=====
C      END OF SUBROUTINE
C=====
      RETURN
      END

C=====
C      SUBROUTINE VPM_ALTITUDE_DIVE_ALGORITHM
C      Purpose: This subprogram updates gas loadings and adjusts critical radii
C      (as required) based on whether or not diver is acclimatized at altitude or
C      makes an ascent to altitude before the dive.
C=====
      SUBROUTINE VPM_ALTITUDE_DIVE_ALGORITHM

      IMPLICIT NONE

C=====
C      LOCAL VARIABLES
C=====
      CHARACTER Diver_Acclimatized_at_Altitude*3, OS_Command*3

      INTEGER I                                     !loop counter

      LOGICAL Diver_Acclimatized

      REAL Altitude_of_Dive, Starting_Acclimatized_Altitude
      REAL Ascent_to_Altitude_Hours, Hours_at_Altitude_Before_Dive
      REAL Ascent_to_Altitude_Time, Time_at_Altitude_Before_Dive
      REAL Starting_Ambient_Pressure, Ending_Ambient_Pressure
      REAL Initial_Inspired_N2_Pressure, Rate, Nitrogen_Rate
      REAL Inspired_Nitrogen_Pressure, Initial_Nitrogen_Pressure
      REAL Compartment_Gradient, Compartment_Gradient_Pascals
      REAL Gradient_He_Bubble_Formation, Gradient_N2_Bubble_Formation
      REAL New_Critical_Radius_He, New_Critical_Radius_N2
      REAL Ending_Radius_He, Ending_Radius_N2
      REAL Regenerated_Radius_He, Regenerated_Radius_N2

      REAL HALDANE_EQUATION                        !function subprogram

      REAL SCHREINER_EQUATION                      !function subprogram
C=====
C      GLOBAL CONSTANTS IN NAMED COMMON BLOCKS
C=====
      REAL Water_Vapor_Pressure
      COMMON /Block_8/ Water_Vapor_Pressure

      REAL Constant_Pressure_Other_Gases
      COMMON /Block_17/ Constant_Pressure_Other_Gases

      REAL Surface_Tension_Gamma, Skin_Compression_GammaC
      COMMON /Block_19/ Surface_Tension_Gamma, Skin_Compression_GammaC

      REAL Regeneration_Time_Constant
      COMMON /Block_22/ Regeneration_Time_Constant
C=====
C      GLOBAL VARIABLES IN NAMED COMMON BLOCKS
C=====
      LOGICAL Units_Equal_Fsw, Units_Equal_Msw
      COMMON /Block_15/ Units_Equal_Fsw, Units_Equal_Msw

      REAL Units_Factor
      COMMON /Block_16/ Units_Factor

```

```

      REAL Barometric_Pressure
      COMMON /Block_18/ Barometric_Pressure
C=====
C      GLOBAL ARRAYS IN NAMED COMMON BLOCKS
C=====
      REAL Nitrogen_Time_Constant(16)
      COMMON /Block_1B/ Nitrogen_Time_Constant

      REAL Helium_Pressure(16), Nitrogen_Pressure(16)           !both input
      COMMON /Block_3/ Helium_Pressure, Nitrogen_Pressure       !and output

      REAL Initial_Critical_Radius_He(16)                       !both input

      REAL Initial_Critical_Radius_N2(16)                       !and output
      COMMON /Block_6/ Initial_Critical_Radius_He,
*           Initial_Critical_Radius_N2

      REAL Adjusted_Critical_Radius_He(16)                     !output
      REAL Adjusted_Critical_Radius_N2(16)
      COMMON /Block_7/ Adjusted_Critical_Radius_He,
*           Adjusted_Critical_Radius_N2
C=====
C      NAMELIST FOR PROGRAM SETTINGS (READ IN FROM ASCII TEXT FILE)
C=====
      NAMELIST /Altitude_Dive_Settings/ Altitude_of_Dive,
*           Diver_Acclimatized_at_Altitude,
*           Starting_Acclimatized_Altitude, Ascent_to_Altitude_Hours,
*           Hours_at_Altitude_Before_Dive
C=====
C      CALCULATIONS
C=====

      OS_Command = 'CLS'
      OPEN (UNIT = 12, FILE = 'ALTITUDE.SET', STATUS = 'UNKNOWN',
*           ACCESS = 'SEQUENTIAL', FORM = 'FORMATTED')

      READ (12,Altitude_Dive_Settings)

      IF ((Units_Equal_Fsw) .AND. (Altitude_of_Dive .GT. 30000.0)) THEN
        CALL SYSTEMQQ (OS_Command)
        WRITE (*,900)
        WRITE (*,901)
        STOP 'PROGRAM TERMINATED'
      END IF
      IF ((Units_Equal_Msw) .AND. (Altitude_of_Dive .GT. 9144.0)) THEN
        CALL SYSTEMQQ (OS_Command)
        WRITE (*,900)
        WRITE (*,901)
        STOP 'PROGRAM TERMINATED'
      END IF

      IF ((Diver_Acclimatized_at_Altitude .EQ. 'YES') .OR.
*           (Diver_Acclimatized_at_Altitude .EQ. 'yes')) THEN
        Diver_Acclimatized = (.TRUE.)
      ELSE IF ((Diver_Acclimatized_at_Altitude .EQ. 'NO') .OR.
*           (Diver_Acclimatized_at_Altitude .EQ. 'no')) THEN
        Diver_Acclimatized = (.FALSE.)
      ELSE
        CALL SYSTEMQQ (OS_Command)
        WRITE (*,902)
        WRITE (*,901)
        STOP 'PROGRAM TERMINATED'
      END IF

```

```

Ascent_to_Altitude_Time = Ascent_to_Altitude_Hours * 60.0
Time_at_Altitude_Before_Dive = Hours_at_Altitude_Before_Dive*60.0

IF (Diver_Acclimatized) THEN
    CALL CALC_BAROMETRIC_PRESSURE (Altitude_of_Dive) !subroutine
    WRITE (*,802) Altitude_of_Dive, Barometric_Pressure
    DO I = 1,16
        Adjusted_Critical_Radius_N2(I) = Initial_Critical_Radius_N2(I)
        Adjusted_Critical_Radius_He(I) = Initial_Critical_Radius_He(I)
        Helium_Pressure(I) = 0.0
        Nitrogen_Pressure(I) = (Barometric_Pressure -
*           Water_Vapor_Pressure)*0.79
    END DO
ELSE
    IF ((Starting_Acclimatized_Altitude .GE. Altitude_of_Dive)
*       .OR. (Starting_Acclimatized_Altitude .LT. 0.0)) THEN
        CALL SYSTEMQQ (OS_Command)
        WRITE (*,903)
        WRITE (*,904)
        WRITE (*,901)
        STOP 'PROGRAM TERMINATED'
    END IF
    CALL CALC_BAROMETRIC_PRESSURE (Starting_Acclimatized_Altitude) !subroutine
*
    Starting_Ambient_Pressure = Barometric_Pressure
    DO I = 1,16
        Helium_Pressure(I) = 0.0
        Nitrogen_Pressure(I) = (Barometric_Pressure -
*           Water_Vapor_Pressure)*0.79
    END DO
    CALL CALC_BAROMETRIC_PRESSURE (Altitude_of_Dive) !subroutine
    WRITE (*,802) Altitude_of_Dive, Barometric_Pressure
    Ending_Ambient_Pressure = Barometric_Pressure
    Initial_Inspired_N2_Pressure = (Starting_Ambient_Pressure
*       - Water_Vapor_Pressure)*0.79
    Rate = (Ending_Ambient_Pressure - Starting_Ambient_Pressure)
*       / Ascent_to_Altitude_Time
    Nitrogen_Rate = Rate*0.79

    DO I = 1,16
        Initial_Nitrogen_Pressure = Nitrogen_Pressure(I)

        Nitrogen_Pressure(I) = SCHREINER_EQUATION
*           (Initial_Inspired_N2_Pressure, Nitrogen_Rate,
*           Ascent_to_Altitude_Time, Nitrogen_Time_Constant(I),
*           Initial_Nitrogen_Pressure)

        Compartment_Gradient = (Nitrogen_Pressure(I)
*           + Constant_Pressure_Other_Gases)
*           - Ending_Ambient_Pressure

        Compartment_Gradient_Pascals =
*           (Compartment_Gradient / Units_Factor) * 101325.0

        Gradient_He_Bubble_Formation =
*           ((2.0*Surface_Tension_Gamma*
*           (Skin_Compression_GammaC - Surface_Tension_Gamma)) /
*           (Initial_Critical_Radius_He(I)*Skin_Compression_GammaC))

        IF (Compartment_Gradient_Pascals .GT.
*           Gradient_He_Bubble_Formation) THEN

            New_Critical_Radius_He = ((2.0*Surface_Tension_Gamma*
*           (Skin_Compression_GammaC - Surface_Tension_Gamma))) /
*           (Compartment_Gradient_Pascals*Skin_Compression_GammaC)

```



```

        Adjusted_Critical_Radius_He(I) =
*       Initial_Critical_Radius_He(I) +
*       (Initial_Critical_Radius_He(I)-
*       New_Critical_Radius_He) *
*       EXP(-Time_at_Altitude_Before_Dive/
*       Regeneration_Time_Constant)

        Initial_Critical_Radius_He(I) =
*       Adjusted_Critical_Radius_He(I)
ELSE
        Ending_Radius_He = 1.0/(Compartment_Gradient_Pascals/
*       (2.0*(Surface_Tension_Gamma-Skin_Compression_GammaC))
*       + 1.0/Initial_Critical_Radius_He(I))

        Regenerated_Radius_He =
*       Initial_Critical_Radius_He(I) +
*       (Ending_Radius_He - Initial_Critical_Radius_He(I)) *
*       EXP(-Time_at_Altitude_Before_Dive/
*       Regeneration_Time_Constant)

        Initial_Critical_Radius_He(I) =
*       Regenerated_Radius_He

        Adjusted_Critical_Radius_He(I) =
*       Initial_Critical_Radius_He(I)
END IF

        Gradient_N2_Bubble_Formation =
*       ((2.0*Surface_Tension_Gamma*
*       (Skin_Compression_GammaC - Surface_Tension_Gamma)) /
*       (Initial_Critical_Radius_N2(I)*Skin_Compression_GammaC))

        IF (Compartment_Gradient_Pascals .GT.
*       Gradient_N2_Bubble_Formation) THEN

                New_Critical_Radius_N2 = ((2.0*Surface_Tension_Gamma*
*       (Skin_Compression_GammaC - Surface_Tension_Gamma))) /
*       (Compartment_Gradient_Pascals*Skin_Compression_GammaC)

                Adjusted_Critical_Radius_N2(I) =
*       Initial_Critical_Radius_N2(I) +
*       (Initial_Critical_Radius_N2(I)-
*       New_Critical_Radius_N2) *
*       EXP(-Time_at_Altitude_Before_Dive/
*       Regeneration_Time_Constant)

                Initial_Critical_Radius_N2(I) =
*       Adjusted_Critical_Radius_N2(I)
ELSE
                Ending_Radius_N2 = 1.0/(Compartment_Gradient_Pascals/
*       (2.0*(Surface_Tension_Gamma-Skin_Compression_GammaC))
*       + 1.0/Initial_Critical_Radius_N2(I))

                Regenerated_Radius_N2 =
*       Initial_Critical_Radius_N2(I) +
*       (Ending_Radius_N2 - Initial_Critical_Radius_N2(I)) *
*       EXP(-Time_at_Altitude_Before_Dive/
*       Regeneration_Time_Constant)

                Initial_Critical_Radius_N2(I) =
*       Regenerated_Radius_N2

                Adjusted_Critical_Radius_N2(I) =
*       Initial_Critical_Radius_N2(I)
END IF
END DO

```

```

        Inspired_Nitrogen_Pressure = (Barometric_Pressure -
*      Water_Vapor_Pressure)*0.79
        DO I = 1,16
            Initial_Nitrogen_Pressure = Nitrogen_Pressure(I)

            Nitrogen_Pressure(I) = HALDANE_EQUATION
*          (Initial_Nitrogen_Pressure, Inspired_Nitrogen_Pressure,
*          Nitrogen_Time_Constant(I), Time_at_Altitude_Before_Dive)
        END DO
    END IF
    CLOSE (UNIT = 12, STATUS = 'KEEP')
    RETURN
C=====
C      FORMAT STATEMENTS - PROGRAM OUTPUT
C=====
802  FORMAT ('0ALTITUDE = ',1X,F7.1,4X,'BAROMETRIC PRESSURE = ',
*F6.3)
C=====
C      FORMAT STATEMENTS - ERROR MESSAGES
C=====
900  FORMAT ('0ERROR! ALTITUDE OF DIVE HIGHER THAN MOUNT EVEREST')
901  FORMAT (' ')
902  FORMAT ('0ERROR! DIVER ACCLIMATIZED AT ALTITUDE',
*1X,'MUST BE YES OR NO')
903  FORMAT ('0ERROR! STARTING ACCLIMATIZED ALTITUDE MUST BE LESS',
*1X,'THAN ALTITUDE OF DIVE')
904  FORMAT (' AND GREATER THAN OR EQUAL TO ZERO')
C=====
C      END OF SUBROUTINE
C=====
        END

C=====
C      SUBROUTINE CLOCK
C      Purpose: This subprogram retrieves clock information from the Microsoft
C      operating system so that date and time stamp can be included on program
C      output.
C=====
        SUBROUTINE CLOCK (Year, Month, Day, Clock_Hour, Minute, M)

        IMPLICIT NONE

C=====
C      ARGUMENTS
C=====
        CHARACTER M*1                                     !output
        INTEGER*2 Month, Day, Year                       !output
        INTEGER*2 Minute, Clock_Hour                     !output
C=====
C      LOCAL VARIABLES
C=====
        INTEGER*2 Hour, Second, Hundredth

C=====
C      CALCULATIONS
C=====
        CALL GETDAT (Year, Month, Day)                   !Microsoft run-time
        CALL GETTIM (Hour, Minute, Second, Hundredth)     !subroutines

        IF (Hour .GT. 12) THEN
            Clock_Hour = Hour - 12
            M = 'p'
        ELSE
            Clock_Hour = Hour
            M = 'a'
        ENDIF
C=====

```

C        END OF SUBROUTINE

C=====

      RETURN

      END