97158-01

Non Standard Day Effects In ATM Trajectory Simulation

Mark Peters
George Hunter
Frank McLoughlin

Prepared for:

National Aeronautics and Space Administration
Ames Research Center, Moffett Field, CA 94035

Under:

FAA Prime Contract No. DTFA03-97-00004
University of California, Berkeley
Agreement No. 1635JB, Amendment No. 1
Seagull Project No. C158

August 1997

The authors would like to acknowledge the contributions of Steve Green and the Center-TRACON Automation System software engineering team of NASA Ames Research Center and Susan Dorsky of Seagull Technology for their assistance in this effort.

Preface

This report documents research undertaken by the National Center of Excellence for Aviation Operations Research, under Federal Aviation Administration Contract Number DTFA03-97-00004. This document has not been reviewed by the Federal Aviation Administration (FAA). Any opinions expressed herein do not necessarily reflect those of the FAA or the U.S. Department of Transportation.

Foreword iii
Table of Contents v
List of Figures vii
1 Introduction 1
2 Analysis of Trajectory Calculations 5
3 Implementation 11

Figure 1.1 Illustration of how the pilot compensates for atmospheric pressure variations to maintain constant climb rate........ 2
Figure 2.1 Simplified Flowchart of TS Operation........ 12

1 Introduction

Air traffic control is an extremely demanding control systems engineering problem. There are several reasons for this: it has a stringent safety constraint, it is highly nonlinear with multiple people in the loop, it is real time and the system must be continuously operational, and it involves complicated models, control strategies, and scenarios. We may view the ATC problem as controlling several different quantities, such as aircraft position, traffic spacing, traffic sequence, and so forth. Probably the most basic quantity to be controlled, however, is flight time. It is natural to view traffic scheduling and separation maintenance as handled by outer loops in the ATC topology hierarchy, with flight time as the inner loop plant. Therefore, a significant amount of research and development has been committed to the problem of flight time prediction for transport aircraft. This report discusses a solution to this problem that meets the growing flight time prediction modeling requirements by today's advanced ATC automation tools.

When creating a flight time prediction model, there are several design trade offs to be made. These are driven by the accuracy requirements, availability of models such as aircraft performance and atmospheric state, and computer resource constraints. Perhaps the most basic trade off is (i) whether the trajectory is to be divided into small time increments starting at either the initial or final conditions and working either forward or backwards in time, respectively; or (ii) whether the trajectory is to be divided into its major flight segments with a flight time approximated for each segment and summed to obtain a total. The first case is applied to problems with more demanding flight time prediction accuracy, say on the order of seconds or tens of seconds. The second case is applied to problems spanning most or all of the flight which do not require high accuracy. Increasing computer resources have made the first case the more popular choice. This report deals with the first case.

The next design trade off is the number of degrees-of-freedom (DOF) to be modeled in the aircraft trajectory. Of course, aircraft flight has six DOF, but they are highly coupled. This coupling levies high demands on the modeling effort because a significant amount of aircraft aerodynamic data are involved. The complicated equations of motion also make this option computer resource intensive. While six DOF models provide the aircraft rotational state variable histories, they do little to enhance the flight time accuracy over three DOF models. The six DOF model does provide a more accurate trajectory description, especially in turn maneuvers, because the aircraft roll and pitch angles do affect the translational states to a limited degree. Four DOF models, which incorporate the aircraft roll angle, have the advantage of combining the simplicity of the 3 DOF model with the added fidelity of the 6

DOF model. Of the rotational states, the roll state has by far the most influence on the translational states. The 3 DOF model is preferred, however, when only the flight time is required. This report deals with the 3 DOF case, although the approach is equally applicable to the 4 DOF case.

The next design trade off is how the equations of motion are developed. Two basic approaches are to (i) describe how the forces determine the aircraft trajectory and (ii) describe the speed profiles of the trajectory segments. These are sometimes referred to as *force-based* and *kinematic* approaches, respectively. The kinematic approach requires less computer resources and modeling effort, but is less accurate, both in flight time and in the trajectory position history. The kinematic approach is especially useful when the piloting variations dominate over the aircraft performance variations. This, for example, is the case on final approach as aircraft capture and maintain the runway localizer. But phases of flight that cannot be accurately modeled without including the aircraft performance require the force-based approach. An example of this is the descent phase from the en route flight. This report deals with the force-based approach.

The next design trade off is the frame of reference. This is especially important for the vertical dimension where one must choose between geometric and pressure altitudes. Geometric altitude refers to the height above mean sea level (MSL) and is independent of the atmospheric state. Pressure altitude refers to the altitude as determined by the aircraft altimeter and is very much dependent on the atmospheric state. The problem here is that these two references are both desirable for different phases of flight. Specifically, when the aircraft flies a certain constant descent rate then the pressure altitude is preferred. This is because the descent rate is, of course, a *pressure altitude* rate since this is what the aircraft air data computer measures. Figure 1.1 illustrates this process. If geometric altitude were to be modeled, then the specified descent rate would need to be transformed into the equivalent geometric altitude rate.

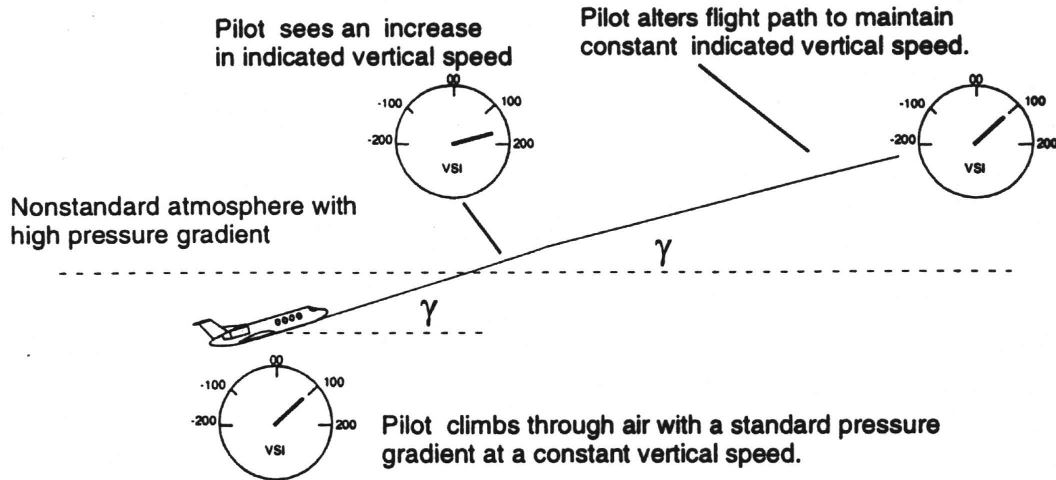

Figure 1.1 Illustration of how the pilot compensates for atmospheric pressure variations to maintain constant climb rate.

On the other hand, when the aircraft flies with a certain throttle setting, then the geometric altitude is preferred. This is because the pilot is not concerned with the descent rate in this case. The descent rate which results is not constant and is a geometric altitude rate. Therefore, if pressure altitude were to be modeled, then the descent rate would need to be transformed into the equivalent pressure altitude rate.

This tradeoff is most important when modeling the cruise and descent portions of the Center airspace flight. In the TRACON airspace, the differences between these two approaches is less significant. In the Center airspace, the descent phase is typically flown with an idle throttle setting, so the geometric altitude reference is preferred. In the cruise phase, on the other hand, a pressure altitude is maintained, so the aircraft is flown with a zero descent rate, or equivalently, a zero flight path angle. So here, the pressure altitude reference is preferred.

In the Center TRACON Automation System (CTAS), early development focused on the extended terminal area — the TRACON and the descent portion of the Center airspace flight. Thus, the natural choice for the vertical dimension modeling was the geometric altitude reference. The CTAS trajectory models reside in its TS (trajectory synthesizer) module. More recently, CTAS testing has included portions of the cruise flight as well. Initially, no corrections were made for the use of TS geometric altitude reference. This meant that in cruise segments, aircraft flew at constant geometric altitude instead of constant pressure altitude. This produced altitude errors which in turn displaced the top of descent (TOD) point causing flight time and descent altitude profile prediction errors. The altitude errors also caused conflict probe problems.

The solution to this problem, as described above, is to apply the appropriate correction factors, depending on which reference frame is used. But the correction factor depends on the reference frame and the phase of flight. This report describes a solution to this problem that emphasized minimum computer execution time. It is to use the pressure altitude reference and apply the correction factor when the aircraft flies with a specified throttle setting — during the Center airspace descent phase. Because the TS was previously using geometric altitude reference, a number of software modifications were required.

Two different correction factors are derived in Section 2. The TS code modifications and test runs are discussed in Section 3. Conclusions are given in Section 4.

2 Analysis of Trajectory Calculations

This section derives two different correction factors based on different assumptions. First, we review the aircraft dynamics and atmospheric properties.

Equations of Motion

Our approach will be to model the vertical dimension using the pressure-altitude reference frame. But we first express the aircraft trajectory equations of motion (EOMs) using the geometric-altitude reference frame because it is more natural. Our next step will then be to convert to the pressure-altitude frame.

The vertical-plane state variables originally modeled in the TS are the true airspeed and geometric altitude. The true airspeed EOM is[1]:

$$\frac{dV_T}{dt} = \frac{T - D}{m} - g \sin \ {}_{a} - \frac{dU_w}{dt} \cos \ {}_{a} \tag{2.1}$$

where,

t	=	Time,
V_T	=	True airspeed,
T	=	Propulsion thrust force,
D	=	Aerodynamic drag force,
m	=	Aircraft mass,
g	=	Gravitational acceleration,
${}_{a}$	=	Flight path angle,
U_w	=	Wind speed.

The geometric altitude EOM is:

$$\frac{dh_g}{dt} = V_T \sin \ {}_{a} \tag{2.2}$$

where,

h_g	=	Geometric altitude.

Our approach is to express Eq. (2.2) in the pressure-altitude frame by converting the geometric altitude derivative to the pressure altitude derivative:

$$\frac{dh_p}{dt} = \left(\frac{dh_p}{dh_g}\right) \frac{dh_g}{dt} \tag{2.3}$$

where,

[1]Slattery, Rhonda A. "Terminal Area Trajectory Synthesis for Air Traffic Control Automation," *Proceedings of the American Control conference*, Seattle, WA, 1995.

$$\frac{dh_p}{dh_g} \quad = \text{ Pressure altitude conversion factor.}$$

We insert Eq. (2.2) into Eq. (2.3) to obtain the new pressure-altitude EOM:

$$\frac{dh_p}{dt} = \left(\frac{dh_p}{dh_g}\right) V_T \sin \quad _a \tag{2.4}$$

The TS uses the small-angle approximation for the flight path angle so Eq. (2.4) reduces to:

$$\frac{dh_p}{dt} = \left(\frac{dh_p}{dh_g}\right) V_T \quad _a \tag{2.5}$$

This linearized form allows for the pressure altitude conversion factor to be incorporated into the EOM by preprocessing it with the flight path angle calculation:

$$\frac{dh_p}{dt} = V_T \quad _a \tag{2.6}$$

where,

$$_a \quad = \text{ Modified flight path angle, } \left(\frac{dh_p}{dh_g}\right) \quad _a \, .$$

There are several advantages to preprocessing the pressure altitude conversion factor as in Eq. (2.6). First, The EOMs are not affected — the true airspeed, flight path angle, and altitude derivatives are computed and integrated as if the geometric altitude reference frame was used. This is convenient because EOMs are typically expressed in the geometric altitude frame, so existing algorithms and software need not be modified. Instead, the flight path angle is modified just prior to the calculation and integration of the altitude derivative.

A second advantage of this approach is that it facilitates appropriate implementation. As discussed in Section 1, the correction factor is to be applied only for flight segments where a throttle setting is specified. It is not to be applied when a descent rate or flight path angle is specified. Fortunately, the flight path angle is calculated differently for these different segment types, so the correction factor can be imbedded precisely in the code where it is required, and omitted from the code where it is not.

Standard Day Atmosphere

Next we derive the pressure altitude conversion factor. First, we must have the expression for the standard day pressure altitude. We will use it to express the pressure altitude as a function of the pressure. This expression is then used in the following two sections to derive the pressure altitude conversion factor. We start with the perfect gas law:

$$P = \quad RT \tag{2.7}$$

where,

P	=	Pressure of gas,
	=	Density of gas,
R	=	Specific gas constant, 287 [J/kg/°K]
T	=	Temperature of gas.

For the gradient region of the standard atmosphere, roughly 36,000 ft and below, temperature varies linearly with pressure altitude:

$$T(h_p) = T_s - a_s h_p \tag{2.8}$$

where,

h_p	=	Pressure altitude, the altitude corresponding to a given atmospheric pressure according to the standard day model,
T_s	=	Standard atmospheric temperature at MSL,
a_s	=	Standard atmospheric lapse rate, [0.00651 °K/m].

The standard atmosphere model also assumes static vertical equilibrium. This assumption, along with Eqs. (2.7) and (2.8) yield[2]:

$$\frac{P}{P_s} \quad = \left(1 - \frac{a_s h_p}{T_s}\right)^{\frac{g_0}{a_s R}} \tag{2.9}$$

where,

	=	Pressure ratio,
P_s	=	Standard atmospheric pressure at MSL.

We now rearrange Eq. (2.9) to solve for the pressure altitude:

$$h_p = \frac{T_s}{a_s}\left[1 - \quad {}^{\frac{a_s R}{g_0}}\right] \tag{2.10}$$

We now have an expression for the pressure altitude that is used in the next two sections to derive two different versions of the pressure altitude conversion factor.

Constant Lapse Rate Solution

This section derives the pressure altitude conversion factor by assuming a constant atmospheric lapse rate, *a*. The advantages of this approach are that it incorporates non standard day temperature and pressure offsets and lapse rate. Also, atmospheric properties aloft are required. This latter feature alleviates the meteorological data requirements; however, it also the reason why a constant lapse rate must be assumed. The constant lapse rate assumption may lead to excessive inaccuracies on particularly non standard weather days.

We begin by rewriting Eq. (2.9) for the geometric altitude frame:

---

[2]Lan, C.E., Roskam, J., *Airplane Aerodynamics and Performance*, RAEC, 1980.

$$\frac{P}{P_g} = \left(1 - \frac{a_g\, h_g}{T_g}\right)^{\frac{g_o}{a_g R}} \tag{2.11}$$

where,

P_g = Non standard atmospheric pressure at MSL,

T_g = Non standard atmospheric temperature at MSL,

a_g = Non standard atmospheric lapse rate.

Eqs. (2.9) and (2.11) are combined to remove P and produce an expression for the pressure altitude as a function of the geometric altitude:

$$h_p = \frac{T_s}{a_s}\left[1 - \left(\frac{P_g}{P_s}\right)^{\frac{a_s R}{g_o}} \left(1 - \frac{a_g h_g}{T_g}\right)^{\frac{a_s}{a_g}}\right] \tag{2.12}$$

We now solve for the pressure altitude conversion factor by taking the derivative of Eq. (2.12) with respect to the geometric altitude:

$$\frac{\mathrm{d}h_p}{\mathrm{d}h_g} = \frac{T_s}{T_g}\left(\frac{P_g}{P_s}\right)^{\frac{a_s R}{g_o}} \left(1 - \frac{a_g\, h_g}{T_g}\right)^{\frac{a_s - a_g}{a_g}} \tag{2.13}$$

Equation (2.13) is the constant-atmospheric lapse rate altitude conversion factor. It requires the non standard atmospheric sea level temperature, pressure and lapse rate. Therefore, it is useful when sea level properties are available but aloft properties are not.

Temperature Aloft Solution

This section derives the pressure altitude conversion factor by assuming that temperature aloft measurement is available. Given this, then no assumption about the atmospheric lapse rate is necessary. Therefore, the accuracy of this solution is not determined by atmospheric modeling assumptions, but rather by the atmospheric measurement and forecast model accuracy.

We begin by rewriting Eq. (2.10):

$$T_s - h_p a_s = T_s \quad \frac{a_s R}{g_o} \tag{2.14}$$

We next take the derivative of Eq. (2.10) with respect to pressure:

$$\frac{\mathrm{d}h_p}{\mathrm{d}P} = -\frac{T_s R}{P_s\, g_o} \quad \left(\frac{a_s R}{g_o} - 1\right) \tag{2.15}$$

This approach also assumes static vertical equilibrium. This provides the hydro static relationship:

$$\frac{dP}{dh_g} = -\frac{P\,g_o}{T\,R} \tag{2.16}$$

Combining Eqs. (2.14) – (2.16), we obtain the pressure altitude conversion factor:

$$\frac{dh_p}{dh_g} = \frac{T_s - h_p a_s}{T} \tag{2.17}$$

But the numerator of Eq. (2.17) is the standard atmosphere sea level temperature less the temperature reduction aloft according to the standard lapse model. That is, it is the standard temperature at the pressure altitude of interest:

$$\frac{dh_p}{dh_g} = \frac{T_s(h_p)}{T} \tag{2.18}$$

where,

$T_s(h_p)$ = Standard atmospheric temperature at the pressure altitude,

T = Actual atmospheric temperature at the pressure altitude.

Equation (2.18) is the temperature aloft solution, and it was preferred over the constant lapse rate solution because it is more practical to implement and is more appropriate since the TS has available near real-time temperature aloft forecasts.

3 Implementation

This section describes the implementation of the pressure altitude conversion factor derived in Section 2 in the CTAS TS module. This implementation served to convert the TS EOMs from the geometric altitude to pressure altitude.

Summary of TS Modifications and Impact

The TS software modifications fall into three categories. First, the pressure altitude conversion factor calculation was implemented and used to compute the modified flight path angle. Second, the initial value of the altitude state variable was modified so the aircraft is initiated at the correct pressure altitude. Third, weather macros were modified such they treat the altitude state variable as pressure altitude instead of geometric altitude. Consider the following simplified flowchart of the current TS operation in Figure 2.1.

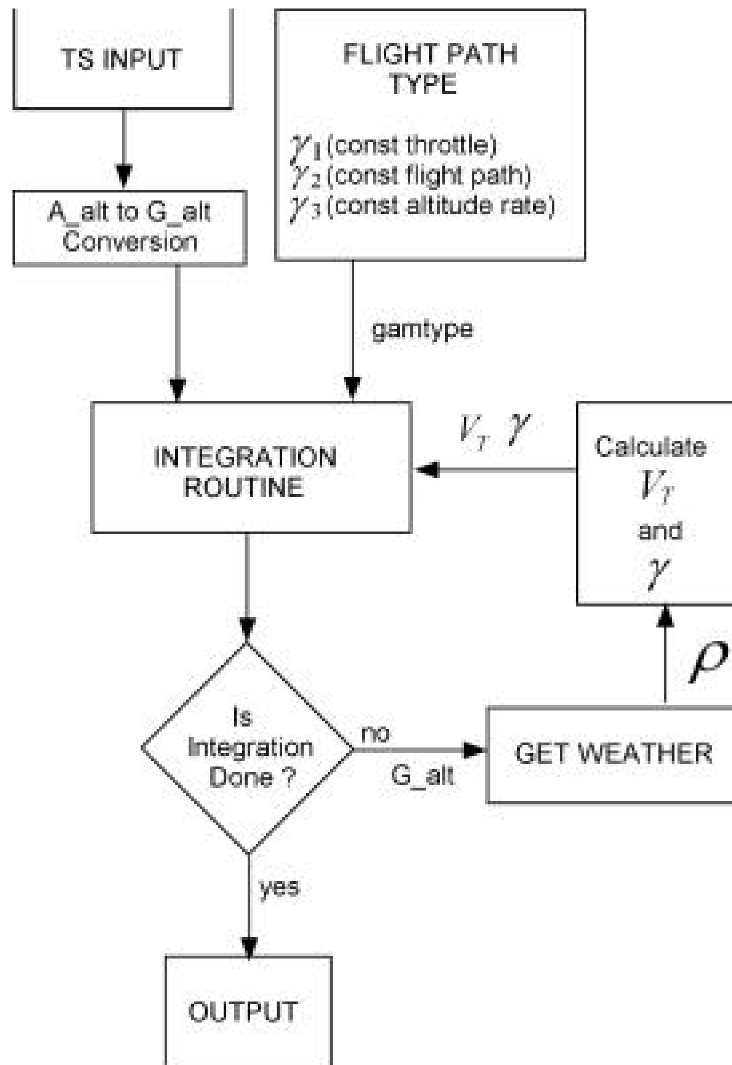

Figure 2.1 Simplified Flowchart of TS Operation

This flowchart highlights the functions which are relevant to the TS modification. Initially, altitude is entered as altimeter altitude (`A_alt`) which is converted to geometric altitude, `G_alt`. This serves as the initial altitude passed to the integration routine. Before integration, the flight path type routines determine which of the three gamma routines will be called and store the information in the flag `gamtype`.

When the integration proceeds, the integration routine calls functions to determine the true airspeed and flight path angle. These functions in turn need ambient density, , which is calculated in the weather functions. The weather functions use geometric altitude as the primary input.

The whole process is simplified on the flow chart by showing geometric altitude commanding density which in turn is used to calculate true airspeed and flight path angle. Once the integration is completed, the data are written to the output in the form of geometric altitude. Altimeter altitude is available if desired via a conversion function found locally in the output source code.

Code Modifications

There are three major changes which must be made to the TS so that pressure altitude is the integration altitude instead of geometric altitude. These changes are as follows:

- Remove the initial altitude conversion from altimeter altitude to geometric altitude and replace it with a conversion from altimeter altitude to pressure altitude. (For conditions above 18000 ft MSL pressure altitude and altimeter altitude are equal).
- Change the weather functions so that they use pressure altitude as an input instead of geometric altitude.
- Modify the `gamma_1` function which calculates the flight path angle under constant throttle settings.

Initial Altitude Conversion

The TS code initially converts the input altitude to geometric altitude using the function `alt_altimeter_to_geometric`. This function is changed so that it returns pressure altitude instead of geometric altitude and is a global change so that geometric altitude is removed from the algorithm entirely. Changing the `alt_altimeter_to_geometric` function has the effect of adding the h term in the following equation. This is the first step to modifying the differential equation:

$$h_p = (h_{go} + \quad h) + \frac{dh_p}{dh_g}\,\frac{dh_g}{dt} \quad t$$

Occasionally, geometric altitude is needed as an output. When it is desired to make the conversion to geometric altitude for output, a renamed copy of the same function is used, `alt_altimeter_to_geometric_MEP`.

Weather Function Modifications

Since pressure altitude is the altitude used in the modified integration routine, it is important to update the weather functions so that the expected input is pressure altitude. This is especially important so that the true airspeed is calculated correctly. True airspeed, critical to the calculation of ground track and altitude rate, relies on accurate values of ambient density. Originally, it was thought that the global removal of the `alt_altimeter_to_geometric` function calls would fix this problem but it does not. In particular, the functions which calculate ambient temperature and pressure had to be modified. This required the writing of new weather macros.

Equation of Motion and Flight Path Angle Modifications

As discussed above, it is not necessary to make any changes to the actual equations of motion in the code. The only required addition to the code is in the `gamma_1` function where the return value gamma is multiplied by a factor of `dhp_dhg`. The following lines now calculate `dhp_dhg`.

```
/* SEAGULL CHANGES */
Temp_Actual= ts_air_temperature(Da_airplane->altitude);
Temp_Standard = STANDARD_SL_TEMP - LAPSE_RATE*(Da_airplane->altitude);
dhp_dhg = Temp_Standard / Temp_Actual;
```

The first line change calculates the actual air temperature using an existing function. The standard temperature is calculated using the sea level temperature and the lapse rate for the standard atmosphere. As discussed earlier, the ratio between the standard temperature and the actual ambient temperature is equal to the ratio (dhp / dhg). The factor `dhp_dhg` is used in `gamma_1` to adjust the output for differences between pressure altitude and geometric altitude:

```
gamma *= dhp_dhg;
```

Results

The TS is the fundamental building block of the CTAS. The TS provides CTAS with accurate 4D trajectory prediction. Prior to this work, the TS algorithm was based on an inertially fixed reference frame which measured altitude as the actual distance above MSL, correctly named geometric altitude. The use of the inertial reference frame is adequate for modeling the descent portions of the flight phases, but the inertial reference frame is not adequate for modeling the cruise portiions of the trajectory. In effect, rather than tracking a constant geometric altitude, an aircraft in cruise will tend to track constant pressure levels in the atmosphere. As the ambient pressure levels vary along a cruise segment, so will the aircraft geometric altitude. In contrast to what actually happens, the TS with its inertially fixed reference frame generated aircraft trajectories which held geometric altitude constant while their altimeter altitude varied. The problem is best illustrated by a portion of TS data which is simulating an aircraft cruising at 33000’ pressure altitude (Flight level 330):

```
>>>Segment of profile:  Step size is  300.0 sec
   -Mach is held constant at 0.6968
   -Vertical speed is held constant at     0.0 ft/min
   -Capture   541.770 nmi path distance    -Allowable capture error of +-  30.38058 internal units
  BACKWARD integration:  ICAPBAK=4     CAPPARBAK=   541.7703   MODEBAK=102    EPSBAK= 30.38058    17 steps

   Time   Dist G_Alt  Mach    V tr  V cal  InFPA   ENGC  Thrst     Wf   Fuel dAltdt Vgnd TCour WSpd WDir  Temp  Pres A_alt
  (sec) (n.mi)  (ft)         (kts)  (kts)  (deg)         (lbs) (lb/hr) (lbs) (ft/m) (kt) (deg) (kt)(deg)(degR)(lb/ft2)(FL)
*  1965  257.9 32701 0.697  408.1  244.5   0.00   0.00      0      0      0      0  471 137.7  123  265 406.6  546.8 330
*  2265  218.5 32701 0.697  408.1  244.8   0.00   0.00      0      0      0      0  473 137.7  127  265 406.5  548.2 330
*  2565  179.0 32701 0.697  407.8  245.3   0.00   0.00      0      0      0      0  474 137.7  132  265 406.0  550.4 329
*  2594  175.1 32701 0.697  407.8  245.3   0.00   0.00      0      0      0      0  491 130.0  133  265 406.0  550.6 329
*  2654  166.9 32701 0.697  407.8  245.4   0.00   0.00      0      0      0      0  491 130.0  133  265 406.0  551.0 328
*  2954  125.9 32701 0.697  407.9  245.9   0.00   0.00      0      0      0      0  492 130.0  136  265 406.2  553.3 328
*  3254   84.8 32701 0.697  408.4  246.5   0.00   0.00      0      0      0      0  494 130.0  134  266 407.3  555.8 327
```

The TS initially made a correction for the difference between geometric altitude(`G_alt`) and altimeter altitude (`A_alt`) and correctly assigned the aircraft to the corresponding geometric altitude for the given ambient conditions. However, from that point on, the integration routine held geometric altitude constant. As the ambient pressure changed, the altimeter altitude (shown in the far right column in terms of flight levels) varied from flight level 330 down to flight level 327. The actual aircraft would hold the flight level constant and a slight variation would be seen in geometric altitude. The correct trend is seen in a portion of data from the modified TS code:

```
>>>Segment of profile:  Step size is  300.0 sec
   -Mach is held constant at 0.6968
   -Vertical speed is held constant at     0.0 ft/min
   -Capture   551.821 nmi path distance    -Allowable capture error of +-  30.38058 internal units
  BACKWARD integration:  ICAPBAK=4     CAPPARBAK=   551.8208   MODEBAK=102    EPSBAK= 30.38058    17 steps

   Time   Dist G_Alt  Mach    V tr  V cal  InFPA   ENGC  Thrst     Wf   Fuel dAltdt Vgnd TCour WSpd WDir  Temp  Pres A_alt
  (sec) (n.mi)  (ft)         (kts)  (kts)  (deg)         (lbs) (lb/hr) (lbs) (ft/m) (kt) (deg) (kt)(deg)(degR)(lb/ft2)(FL)
*  1742  298.2 32701 0.697  408.2  244.6   0.00   0.00      0      0      0      0  510 115.0  123  265 406.8  547.0 33000
*  1771  294.3 32701 0.697  408.2  244.6   0.00   0.00      0      0      0      0  470 137.7  123  265 406.8  547.0 33000
         FINISH OF TURN   ET#=8
*  2049  257.9 32694 0.697  408.1  244.6   0.00   0.00      0      0      0      0  471 137.7  123  265 406.6  547.0 33000
*  2349  218.5 32752 0.697  408.0  244.6   0.00   0.00      0      0      0      0  473 137.7  127  265 406.5  547.0 33000
*  2649  179.0 32835 0.697  407.6  244.6   0.00   0.00      0      0      0      0  474 137.7  133  265 405.7  547.0 33000
*  2678  175.1 32845 0.697  407.6  244.6   0.00   0.00      0      0      0      0  491 130.0  133  265 405.7  547.0 33000
*  2732  167.7 32860 0.697  407.6  244.6   0.00   0.00      0      0      0      0  491 130.0  134  265 405.6  546.9 33000
*  3032  126.7 32949 0.697  407.5  244.6   0.00   0.00      0      0      0      0  493 130.0  137  265 405.5  546.9 33000
```

No significant impact has been observed in the TS execution time due to these modifications.

Listing of TS Modifications

This section contains the actual code changes made. The changes are organized into 5 categories which are

- Initial Altitude Conversion
- Equation of Motion Changes
- Weather Function Changes
- Flight Path Angle Changes
- Miscellaneous Changes

This section contains actual source code with comments interjected. The difference in the fonts will key the reader to what is source code and what is a comment.

Initial Altitude Conversion

The changes which make the initial altitude conversion from altimeter altitude to pressure altitude are contained in the `wthr_utils.c` file (in the lib directory). The `alt_altimeter_to_geometric` function is the function which is changed. This change acts globally.

```
/*
 * NOTE:
 * For geometric-altimeter altitude conversion routines, the actual
 * altimeter setting procedure is more complicated than the algorithm
 * used here.  This algorithm doesn't handle extreme weather
 * conditions or other sophisticated cases.  The extreme altimeter setting
 * was screened out before calling these routines. See section 2. Altimeter
 * Setting Procedures at p7-530, p7-531 in AIM for details.
 */
/*
 *------------------------------------------------------------------------
 * alt_altimeter_to_geometric:
 *     This function converts altimeter altitude to geometric altitude.
 *
 * arguments:
 *     x,y                  Plane's coordination.
 *     altimeter_alt        altimeter altitude.
 *     altimeter_setting    altimeter setting used.
 *
 * return:
 *     geometric altitude.
 *
 * side effects:
 *     none.
 *------------------------------------------------------------------------
 */
double
alt_altimeter_to_geometric (
float           x,
float           y,          /* in Nautical Mile MM */
float           altimeter_alt,     /* altitude is expected in feet and it will
                             * be returned in feet */
float          t,           /* time, seconds */
float          altimeter_setting /* airport altimeter setting */)
{
    float     pressure_alt  = 0.0;
    float     geometric_alt = 0.0;

    Wthr_data_st   *wthr_data;
    wthr_data = get_static_wthr_data_st_ptr ();


       if (altimeter_alt < FLIGHT_LEVEL_TRANSMISSION_ALTITUDE)
```

```
            pressure_alt = altimeter_alt -
                (altimeter_setting - STD_ALTIMETER_SETTING) * 1000.0;
        else
            pressure_alt = altimeter_alt;
```

The conversion to geometric altitude is commented out and the function is changed to return `pressure_alt` instead.

```
/* COMMENTED OUT THE CONVERSION HERE -MEP */

/*      geometric_alt = INTERP_GEOM_ALT_XY_NM_P_ALT(wthr_data,
                             x, y, pressure_alt, t);

*/

      /* MARK CHANGES HERE */

         /* Originally returns geometric altitude */
        return (pressure_alt);
}
```

The original functionality of the function is needed for the data output so it is renamed here for use as needed.

```
/*
 *-----------------------------------------------------------------------
 * alt_altimeter_to_geometric_MEP:
 *      This function converts altimeter altitude to geometric altitude.
 *      MARK COMMENT -> This is basically the same as the original except
 *      that it has been renamed so that it can be used without effecting
 *      the orignal code.
 * arguments:
 *      x,y                     Plane's coordination.
 *      altimeter_alt           altimeter altitude.
 *      altimeter_setting       altimeter setting used.
 *
 * return:
 *      geometric altitude.
 *
 * side effects:
 *      none.
 *-----------------------------------------------------------------------
 */
double
alt_altimeter_to_geometric_MEP (
float           x,
float           y,              /* in Nautical Mile MM */
float           altimeter_alt,  /* altitude is expected in feet and it will
                                 * be returned in feet */
float           t,              /* time, seconds */
float           altimeter_setting /* airport altimeter setting */)
{
    float       pressure_alt    = 0.0;
    float       geometric_alt   = 0.0;

    Wthr_data_st   *wthr_data;
    wthr_data = get_static_wthr_data_st_ptr ();


    if (altimeter_alt < FLIGHT_LEVEL_TRANSMISSION_ALTITUDE)
            pressure_alt = altimeter_alt -
            (altimeter_setting - STD_ALTIMETER_SETTING) * 1000.0;
    else
            pressure_alt = altimeter_alt;
```

```
    geometric_alt = INTERP_GEOM_ALT_XY_NM_P_ALT(wthr_data,
                                x, y, pressure_alt, t);

    return (geometric_alt);
}
```

Since geometric altitude is completely removed from the inner workings of the TS, the conversion from geometric altitude to altimeter altitude is changed to a conversion between pressure altitude and altimeter altitude.

```
/*
 *-------------------------------------------------------------------------
 * alt_geometric_to_altimeter:
 *     This function converts geometric altitude to altimeter altitude.
 *
 * arguments:
 *     x,y                  Plane's coordination.
 *     geometric_alt        geometric altitude.
 *     altimeter_setting    altimeter setting used.
 *
 * return:
 *     altimeter altitude.
 *
 * side effects:
 *     none.
 *-------------------------------------------------------------------------
 */
double
alt_geometric_to_altimeter (
float           x,
float           y,           /* in Nautical Maile MM */
float           geometric_alt,      /* altitude is expected in feet and it will
                              * be returned in feet */
float          t,            /* time */
float          altimeter_setting /* airport altimeter setting */)
{
    float      pressure_alt   = 0.0;
    float      altimeter_alt  = 0.0;

    Wthr_data_st   *wthr_data;
    wthr_data = get_static_wthr_data_st_ptr ();
```

The conversion from geometric altitude to pressure altitude is commented out.

```
        /* MARK CHANGE */

        pressure_alt = geometric_alt;

       /*  pressure_alt =  INTERP_PRESS_ALT_XY_NM_G_ALT(wthr_data,
                      x, y, geometric_alt, t);

       */


       altimeter_alt = pressure_alt +
              (altimeter_setting - STD_ALTIMETER_SETTING) * 1000.0;

       if (altimeter_alt < FLIGHT_LEVEL_TRANSMISSION_ALTITUDE)
           return (altimeter_alt);
       else

           return (pressure_alt);
```

```
}

/***************  Functions to Interface with Old Weather ****************/
```

Equation of Motion Changes

The equations of motion are integrated using 4 2nd order Runge-Kutta algortihms. Each algorithm supports a different type of integration. The first two algorithms are the only ones which need modifications. A portion of the source code for these two algorithms is presented below with comments to explain the function of the code.

```
*-------------------------------------------------------------------------------
 * call_runge_kutta:
 *      executes RK functions based on the command table constraints
 *      .
 *
 * Arguments:
 *      Ruku            rktype
 *      Vtrue           vttype
 *      float           *gamtype
 * Returns:
 *      none
 *-------------------------------------------------------------------------------
 */
void
call_runge_kutta(Ruku rktype,Vtrue vttype, Gamma_t gamtype)
{
        switch(rktype)
        {
        case RUNGE_KUTTA_1:
           runge_kutta_1(vttype,gamtype);
           break;
        case RUNGE_KUTTA_2:
           runge_kutta_2(vttype,gamtype);
           break;
        case RUNGE_KUTTA_3:
           runge_kutta_3(vttype,gamtype);
           break;
        case RUNGE_KUTTA_4:
           runge_kutta_4(vttype,gamtype);
           break;
        default:
           break;
```

This section of the code calls the appropriate algorithm.

```
        }
        return;
}

/*
 *-------------------------------------------------------------------------------
 * runge_kutta_1:
 *      Second-order Runge-Kutta integration routine for solving two
 *      simultaneous first-order, non-linear, ordinary differential
 *      equations.  Used for all cases where: Mach or CAS held constant
 *
 *      First, find a first order estimate of rates at elapsed time.  Find the
 *      true airspeed in order to calculate the altitude rate.  Use these values
 *      to calculate the path distance and altitude at the end of the step.
 *      Once these are calculated, the state information at the end of the
 *      step can be found(second estimate).  The average of these two estimates
 *      is assumed to be the state at the end of the time step.
 *
 *
 * Arguments:
 *      Vtrue           vttype          subroutine to call to find TAS in ft/sec
 *      *gamtype        Gamma_t         subroutine to call to find true flight
 *                                      path angle in radians.
 * Returns:
 *      none
 *-------------------------------------------------------------------------------
 */
void
runge_kutta_1(Vtrue vttype, Gamma_t gamtype)
{
    /*
```

```
 * Second-order Runge-Kutta integration routine for solving two
 * simultaneous first-order, non-linear, ordinary differential
 * equations of the form: dx/dt = XDOT(x,z) dz/dt = ZDOT(x,z) DT is
 * the step size in t.  For forward integration, DT is positive; for
 * backward integration, DT is negative.
 *
 * Used for all cases where: 1) Mach or CAS held constant
 *
 * vtrue: Name of subroutine to call, in order to find the true-airspeed
 * in ft/sec. gammat: Name of subroutine to call, in order to find the
 * true flight path angle in radians.
 *
 * MACH, Da_airplane->cas, and Da_airplane->true_airspeed are initialized before
 * subroutine integrate_segment
 * is called, or for the case of starting in the middle of the
 * segment, they are initialized in integrate_segment; COURSE is
 * initialized here.
 *
 * NOTES: Originally, a fourth-order Runge-Kutta integration was used
 * here. A problem with this method occured for the following case:
 * With high wind, the mathematical finish of an integration segment
 * is in a turn, so that the final integration step of the segment
 * goes through the turn. This makes the ground-speed very non-linear;
 * so that if the capture parameter is a path-distance, or an altitude
 * when following a constant inertial-flight-path-angle descent, then
 * it is difficult for the capture algorithm to make a capture. The
 * second-order Runge-Kutta integration is more stable for this case.
 *
 * Also, the fourth-order Runge-Kutta requires twice as much computation
 * time as the second-order, and tests indicated that the difference
 * in accuracy did not justify the extra computation time:
 */

float           d1x,
                d1z;
float           d2x,
                d2z;
float           gamma,
                vtr2;
float           xp,
                zp,
                zp_50,
               dummy;
float          *dummy_gs;
```

Note the addition of new variables

```
/* MARK ADDED variables */

float Temp_Standard;
float Temp_Actual;
float dhp_dhg;

float Temp_Standard_50;
float Temp_Actual_50;
float dhp_dhg_50;


if ( !FLOAT_EQUAL(Da_airplane->integration_time_step, 0.0, 0.001) ) {
```

Note the `dhp_dhg` calculation for the instantaneous location

```
        /* MARK CHANGES */
        Temp_Actual= ts_air_temperature(Da_airplane->altitude);
        Temp_Standard = 518.69 - LAPSE_RATE*(Da_airplane->altitude);
        dhp_dhg = Temp_Standard / Temp_Actual;


/*        printf("%f %f %f \n", Temp_Actual,Temp_Standard,dhp_dhg); */



    d1x = Da_airplane->integration_time_step * Da_airplane->ground_speed;
    d1z = Da_airplane->integration_time_step * Da_airplane->altitude_rate*dhp_dhg;
/* MARK CHANGE */
```

Note the `dhp_dhg` addition to the `d1z` equation.

```
        /*
         * Use first order estimate of state at time = elapsed_time +
         * integration_time_step to compute rates of change:
         */

        xp = Da_airplane->path_distance + d1x;
        zp = Da_airplane->altitude + d1z;


        zp_50 = zp + ALT_DIFFERENCE;


          /*  MARK ADDED CHANGES */

          Temp_Actual_zp= ts_air_temperature(zp);
          Temp_Standard_zp = 518.69-(LAPSE_RATE*zp)  /* we need a macro for 518.69 */;
          dhp_dhg_zp = Temp_Standard_zp / Temp_Actual_zp;


        dummy = call_vtrue_gama_func(vttype, zp_50, &Da_airplane->mach, &Da_airplane-
>cas, &vtr2);
```

The following lines calculate a new true airspeed using zp. It should be noted here that `call_true_gama_func` calls functions which use a density function which in turn calls other weather functions.

```
        dummy = call_vtrue_gama_func(vttype, zp, &Da_airplane->mach, &Da_airplane->cas,
                 &Da_airplane->true_airspeed);
        Da_airplane->dtrue_airspeed_daltitude =
            (vtr2 - Da_airplane->true_airspeed) / ALT_DIFFERENCE;

        Da_airplane->course_angle = getcourse(xp);

        Da_airplane->ground_speed_at_z = vground(zp,
                                             Da_airplane->true_airspeed,
                                             Da_airplane->course_angle,
                                             Da_wind);
        gamma = call_gamma_func (gamtype, dummy_gs, &zp);


        d2x = Da_airplane->integration_time_step *
            Da_airplane->ground_speed_at_z;
        d2z = Da_airplane->integration_time_step *
            (Da_airplane->true_airspeed * gamma * dhp_dhg_zp);  /* MARK ADDED CHANGES
*/

        Da_airplane->elapsed_time += Da_airplane->integration_time_step;

        Da_airplane->path_distance += (d1x + d2x) / 2.0;
        Da_airplane->altitude += (d1z + d2z) / 2.0;


/*
 *-----------------------------------------------------------------------------
 * runge_kutta_2:
 *      Second-order Runge-Kutta integration routine for solving a
 *      simultaneous second-order, non-linear, differential
 *      equation.  Used for all cases where: 1) Neither Mach nor CAS held
 *      constant,inertial flight path angle held constant 2) Neither Mach
 *      nor CAS held constant, vertical speed held constant
 *
 *      Since runge_kutta_2 is used for cases where the true-airspeed is not
 *      commanded, runge_kutta_2 is used to determine the true-airspeed, by
 *      calling routine DVTRUEDT to find the rate of change of
 *      true-airspeed. runge_kutta_2 then integrates this rate.
 *
 *      Find a first order estimate of rates at elapsed time. Use these values
 *      to calculate the path distance and altitude at the end of the step.
 *      Once these are calculated, the state information at the end of the
 *      step can be found(second estimate)  The average of these two estimates
 *      is assumed to be the state at the end of the time step.  From this new
 *      location the mach and CAS are calculated, as well as the new rate of
 *      change of the true airspeed and the new true flight path angle.
 *
 * Arguments:
 *      Vtrue          vttype        subroutine to call to find true flight
 *                                   path angle in radians.
```

```
 *     *gamtype      Gamma_t        subroutine to call to find altitude
 *                                  rate in ft/sec
 * Returns:
 *     none
 *-----------------------------------------------------------------------------
 */
void
runge_kutta_2(Vtrue vttype, Gamma_t gamtype)
{
    /*
     * Second-order RungeKutta integration routine for solving two
     * simultaneous, non-linear, second-order differential equations of
     * the form: dx/dt = u,  du/dt = UDOT(x,z,u,w,t) dz/dt = w,  dw/dt =
     * WDOT(x,z,u,w,t).
     *
     * DT is the step size in t.  For forward integration, DT is positive;
     * for backward integration, DT is negative.
     *
     * Used for the cases where: 1) Neither Mach nor CAS held constant,
     * inertial flight path angle held constant 2) Neither Mach nor CAS
     * held constant, vertical speed held constant
     *
     * Since runge_kutta_2 is used for cases where the true-airspeed is not
     * commanded, runge_kutta_2 is used to determine the true-airspeed, by
     * calling routine DVTRUEDT to find the rate of change of
     * true-airspeed. runge_kutta_2 then integrates this rate.
     *
     *
     * vttype: Name of subroutine to call, in order to find the true flight
     * path angle in radians. This name is defined by the calling routine
     * as one of the following names: gama_1, if inertial flight path
     * angle held constant; gama_2, if vertical speed held constant.
     * gamtype: Name of subroutine to call, in order to find the altitude
     * rate in ft/sec. This name is defined by the calling routine as one
     * of the following names: dgamd_1, if inertial flight path angle held
     * constant; dgamd_2, if vertical speed held constant
     */
    float           d1vtr,
                    d1x,
                    d1z,
                    d2vtr,
                    d2x,
                    d2z,
                    gamma,
                    u1_estimate,
                    u2_estimate,
                    w1_estimate,
                    w2_estimate,
                    x1_estimate,
                    x2_estimate,
                    z1_estimate,
                    z2_estimate;
    float           *dummy1,
                    *dummy2,
                    *dummy3;
    static float    time_before = 0.0;
    static float    vtr_before = 0.0;
    static float    dvtr_dt_before = 0.0;
    static float    time_next_step = 0.0;
```

Note the addition of new variables

```
    /* MARK ADDED variables */

    float Temp_Standard;
    float Temp_Actual;
    float dhp_dhg;

    float Temp_Standard_z1_estimate;
    float Temp_Actual_z1_estimate;
    float dhp_dhg_z1_estimate;


if ( !FLOAT_EQUAL(Da_airplane->integration_time_step, 0.0, 0.001) ) {
        /*
         * If we are not iterating to a capture condition, then the values
         * of Da_airplane->true_airspeed, Da_airplane->dtrue_airspeed_dt are the
          * same as at the finish of the previous step,
         * so it would be redundant to re-calculate them. (The
```

```
         * integration must begin with an initial step where
         * integration_time_step=0.)
         */
        if (FLOAT_EQUAL(Da_airplane->elapsed_time, time_next_step, TOLERANCE)) {
            time_before = Da_airplane->elapsed_time;
            vtr_before = Da_airplane->true_airspeed;
            dvtr_dt_before = Da_airplane->dtrue_airspeed_dt;
        }
        else {
            if (Da_airplane->elapsed_time != time_before)
               error_handler(TS_SET_FATAL_ERROR_SINGLE_AIRCRAFT,
                   "RUNGE2 Software error", NO_DEBUG_DATA, 0, NULL);
            Da_airplane->true_airspeed = vtr_before;
            Da_airplane->dtrue_airspeed_dt = dvtr_dt_before;
        }
```

Note the `dhp_dhg` calculation for the instantaneous location

```
         /* MARK ADDED LINES */
         Temp_Actual= ts_air_temperature((Da_airplane->altitude));
         Temp_Standard = 518.69-(LAPSE_RATE*(Da_airplane->altitude));
         dhp_dhg = Temp_Standard / Temp_Actual;

        d1x = Da_airplane->integration_time_step * Da_airplane->ground_speed;

        d1z = Da_airplane->integration_time_step * (Da_airplane-
>altitude_rate)*dhp_dhg;
 /*   MARK CHANGE */

        d1vtr = Da_airplane->dtrue_airspeed_dt * Da_airplane->integration_time_step;

        Da_airplane->true_airspeed += d1vtr;

        x1_estimate = Da_airplane->path_distance + d1x;
        z1_estimate = Da_airplane->altitude + d1z;
        Da_airplane->course_angle = getcourse(x1_estimate);
        Da_airplane->ground_speed_at_z = vground(z1_estimate,
                                            Da_airplane->true_airspeed,
                                            Da_airplane->course_angle,
                                            Da_wind);

        u1_estimate = Da_airplane->ground_speed_at_z;

        w1_estimate = call_gamma_func (gamtype, &u1_estimate, dummy1);

        /*
         * Start calculations to get second-order estimate for
         * true-airspeed:
         */
        Da_airplane->mach = Da_airplane->true_airspeed / sonic_speed(z1_estimate);

        Da_airplane->cas = KTS_TO_FPS(convert_mach_to_cas(z1_estimate,
                                      Da_airplane->mach));

        Da_airplane->engine_control = Da_desired_values->engine_control;

        gamma = call_vtrue_gama_func( vttype, z1_estimate, dummy1, dummy2, dummy3);

              /* MARK ADDED LINES */
         Temp_Actual_z1_estimate= ts_air_temperature((z1_estimate));
         Temp_Standard_z1_estimate = 518.69-(LAPSE_RATE*(z1_estimate));
         dhp_dhg_z1_estimate = Temp_Standard_z1_estimate / Temp_Actual_z1_estimate;



        /*
         * Find the rate of change of the true-airspeed at approximately
         * the finish of the integration step
         * (and also Da_airplane->fuel_consumption)
         */
        Da_airplane->dtrue_airspeed_dt = dvtruedt(&z1_estimate, gamma);

        d2vtr = Da_airplane->dtrue_airspeed_dt *
            Da_airplane->integration_time_step;

        /*
         * Get second-order estimate for true-airspeed, by assuming that
         * the rate of change of the true-airspeed is the average of the
```

```
         * rates of change at the beginning and finish of the integration
         * step:
         */
        Da_airplane->true_airspeed = vtr_before + (d1vtr + d2vtr) / 2.0;

        d2x = Da_airplane->integration_time_step * u1_estimate;



/* SRD 1/23/97 Added dhp_dhg_z1_estimate factor */
        d2z = Da_airplane->integration_time_step * w1_estimate * dhp_dhg_z1_estimate;

        x2_estimate = Da_airplane->path_distance + (d1x + d2x) / 2.0;
        z2_estimate = Da_airplane->altitude + (d1z + d2z) / 2.0;

        Da_airplane->course_angle = getcourse(x2_estimate);

        Da_airplane->ground_speed_at_z = vground(z2_estimate,
                                        Da_airplane->true_airspeed,
                                        Da_airplane->course_angle,
                                        Da_wind);

        u2_estimate = Da_airplane->ground_speed_at_z;
        w2_estimate = call_gamma_func(gamtype, &u2_estimate, dummy1);

        d2x = Da_airplane->integration_time_step * u2_estimate;


        d2z = Da_airplane->integration_time_step * w2_estimate*dhp_dhg; /* Mark Added
Change */

        Da_airplane->elapsed_time += Da_airplane->integration_time_step;
        Da_airplane->path_distance += (d1x + d2x) / 2.0;

        Da_airplane->altitude += (d1z + d2z) / 2.0;
```

Weather Function Changes

There are several functions used in the Runge-Kutta integration routines which call weather in the process of calculating their output. These functions chiefly call three critical functions which are effected by the change from geometric altitude to pressure altitude. These functions are

- `ts_air_pressure`
- `ts_air_temperature`
- `ts_compute_wind_vector_and_components`

As shown in the source code, macros had to be created so that these functions could access pressure altitude.

```
/*
 *-----------------------------------------------------------------------
 * ts_air_pressure:
 *      Interpolate pressure with given X, Y, geometric altitude, time.
 *     "time" is a dummy variable now.
 *
 * arguments:
 *     double pres_altitude  pressure altitude in feet.
 *
 * return float:
 *     pressure in lb/ft**2.
 *
 * side effects:
 *     none.
 *-----------------------------------------------------------------------
 */
float
ts_air_pressure(double pres_altitude)  /* in Feet */
{
    double       pressure;   /* pressure is returned in "LB/FT**2" */
    Wthr_data_st   *wthr_data;
```

```
    switch (TS_weather->wthr_type)
    {
       case LANGLEY_WEATHER:
           pressure = PA_TO_PSF(get_std_atmosphere_value(STD_PRESSURE,
                                       FT_TO_M(pres_altitude)));
           break;

       case WEATHER_3D_BIN:
       default:
           wthr_data = get_static_wthr_data_st_ptr ();
```

The original macro was `INTERP_PRESSURE_XY_FT_G _ALT`. A new macro was written with pressure altitude as an input.

```
/* SRD 1/23/97 Change MACRO to pressure altitude dependence */
           pressure = INTERP_PRESSURE_XY_FT_P_ALT(wthr_data,
              Da_wind->x, Da_wind->y, pres_altitude,
              0.0 /* time */);
           break;
    }

    return ( (float)pressure );
}


/*
 *------------------------------------------------------------------------
 * ts_air_temperature:
 *      Interpolate temperature with given X, Y, geometric altitude, time.
 *     "time" is a dummy variable now.
 *
 * arguments:
 *     double pres_altitude  pressure altitude in feet.
 *
 * return float:
 *     temperature in Rankin.
 *
 * side effects:
 *     none.
 *------------------------------------------------------------------------
 */
float
ts_air_temperature(float pres_altitude) /* in Feet */
{
    float       temperature; /* temperature is returned in "Rankin" */
    Wthr_data_st   *wthr_data;

    switch (TS_weather->wthr_type)
    {
       case LANGLEY_WEATHER:
/* What kind altitude in langley data?  It treats the altitiude
 * in geometric altitude and not altimeter altitude for now.
 */
           interpolate_langley_temperature(pres_altitude, &temperature);
           temperature = C_TO_R (temperature);
           break;

       case WEATHER_3D_BIN:
       default:
           wthr_data = get_static_wthr_data_st_ptr ();
```

The original macro was `INTERP_TEMPERATURE_XY_FT_G _ALT`. A new macro was written with pressure altitude as an input.

```
/* SRD 1/23/97 Change MACRO to pressure altitude dependence */
           temperature = INTERP_TEMPERATURE_XY_FT_P_ALT(wthr_data,
              Da_wind->x, Da_wind->y, pres_altitude,
              0.0 /* time */);
           break;
    }

    return ( (float) temperature);
}
```

```
/*
 *------------------------------------------------------------------------
 * ts_compute_wind_vector_and_components:
 *      Interpolate wind component and compute wind vector with given X,
 *     Y, geometric altitude, time.  "time" is a dummy variable now.
 *
 * arguments:
 *     float  ac_x, ac_y    airplane's coordinates.
 *     double press_altitudepressure altitude in feet.
 *     float  *wind_speed    return wind speed in ft/sec.
 *     float  *wind_heading  return wind heading in radius.
 *     float  *wind_vector   return wind vector.
 *
 * return:
 *     return values (wind vector and wind components) all in passing
 *      arguments.
 *
 * side effects:
 *     none.
 *------------------------------------------------------------------------
 */
void
ts_compute_wind_vector_and_components(float ac_x, float ac_y, float press_altitude,
       float *wind_speed, float *wind_heading, WIND_COMPONENTS *wind_vector)
{
    Wthr_data_st      *wthr_data;
    wthr_data = get_static_wthr_data_st_ptr ();

    switch (TS_weather->wthr_type)
    {
       case LANGLEY_WEATHER:
           ts_interpolate_langley_wind(ac_x, ac_y, press_altitude,
              wind_speed, wind_heading,
              wind_vector);
           break;

       case WEATHER_3D_BIN:
       default:
```

The original macors were `INTERP_X_WIND_XY_FT_G _ALT` and `INTERP_Y_WIND_XY_FT_G _ALT`. New macros were written with pressure altitude as an input.

```
/* SRD 1/23/97 Change geo_altitude to press_altitude in MACROs */
           wind_vector->east  = INTERP_X_WIND_XY_FT_P_ALT(wthr_data,
              ac_x, ac_y, press_altitude,
              0.0 /* time */);
           wind_vector->north = INTERP_Y_WIND_XY_FT_P_ALT(wthr_data,
              ac_x, ac_y, press_altitude,
              0.0 /* time */);
           *wind_speed = wind_magnitude(wind_vector->east,
                                        wind_vector->north);
           *wind_heading = wind_direction(wind_vector->east, wind_vector->north);
           break;
    }

    wind_vector->upward  = 0.0;

    return;
}

/*
 *--------------------------------------------------------------------------
```

The macros written for the code are summarized below.

File: ts_weather.c
line: 419,465,513
Change: change macro to pressure altitude dependence
Reason: pressure altitude is the input instead of geometric altitude

line 419:

```
/* SRD 1/23/97 Change MACRO to pressure altitude dependence */
            pressure = INTERP_PRESSURE_XY_FT_P_ALT(wthr_data,
             Da_wind->x, Da_wind->y, pres_altitude,
```

```
            0.0 /* time */);
```

line 465:

```
/* SRD 1/23/97 Change MACRO to pressure altitude dependence */
            temperature = INTERP_TEMPERATURE_XY_FT_P_ALT(wthr_data,
                Da_wind->x, Da_wind->y, pres_altitude,
                0.0 /* time */);
```

line 513:

```
/* SRD 1/23/97 Change geo_altitude to press_altitude in MACROs */
            wind_vector->east  = INTERP_X_WIND_XY_FT_P_ALT(wthr_data,
                ac_x, ac_y, press_altitude,
                0.0 /* time */);
            wind_vector->north = INTERP_Y_WIND_XY_FT_P_ALT(wthr_data,
                ac_x, ac_y, press_altitude,
         0.0 /* time */);
         0.1
```

File: wthr_utils_defs.h
line: 232,268,297
Change: Add macros which take pressure altitude and x y measurements in ft.
Reason: These macros are needed for the changes made in ts_weather.c

line 232:

```
/* MARK ADDED MACRO */
#define INTERP_TEMPERATURE_XY_FT_P_ALT(wthr_data, X, Y, H, T)   \
        (INTERP_TEMPERATURE_XY_NM_G_ALT((wthr_data), FT_TO_NM((X)), FT_TO_NM((Y)),  \
           (INTERP_GEOM_ALT_XY_NM_P_ALT((wthr_data), FT_TO_NM((X)), FT_TO_NM((Y)),
(H), (T))),  \
           (T)))
```

line 268:

```
/* MARK ADDED MACROS */

#define INTERP_Y_WIND_XY_FT_P_ALT(wthr_data, X, Y, H, T)        \
        (INTERP_Y_WIND_XY_NM_P_ALT((wthr_data), (FT_TO_NM((X))),        \
                                                (FT_TO_NM((Y))),        \
                                                (H), (T)))

#define INTERP_X_WIND_XY_FT_P_ALT(wthr_data, X, Y, H, T)        \
        (INTERP_X_WIND_XY_NM_P_ALT((wthr_data), (FT_TO_NM((X))),        \
                                                (FT_TO_NM((Y))),        \
                                                (H), (T)))
```

line 297:

```
/* MARK ADDED MACRO */
#define INTERP_PRESSURE_XY_FT_P_ALT(wthr_data, X, Y, H, T)      \
        (INTERP_PRESSURE_XY_NM_G_ALT((wthr_data), FT_TO_NM((X)), FT_TO_NM((Y)),  \
           (INTERP_GEOM_ALT_XY_NM_P_ALT((wthr_data), FT_TO_NM((X)), FT_TO_NM((Y)),
(H), (T))),  \
           (T)))
```

Flight Path Angle Changes

As discussed in Section 4.2.4 there are three routines for determining the flight path angle. Each of these is called when the assumptions specific to the algorithm are valid. The assumptions are

- Assume or specify a constant throttle setting.
- Assume specified flight path angle.
- Assume a constant rate of descent.

The first two are not effected by the change from geometric altitude to pressure altitude, but the constant rate of descent will be effected. The function `gamma_3`, contained in `segment_integration`, is shown below.

```
/*
 *-----------------------------------------------------------------------------
 *  gamma_3:
 *      Find true flight path angle and the engine control as a function of
 *      altitude, altitude rate, the derivative of true airspeed w.r.t.
 *      altitude, and true airspeed.  This routine is used when Mach and
 *      vertical airspeed are held constant, or CAS and vertical airspeed
 *      are held constant.
 *
 *      The true flight path angle is found as a ratio of the desired altitude
 *      rate and the true airspeed. If the compute_fuel_consumption flag is not
 *      turned on, then return the value of gamma.  If not, then calculate the
 *      drag, and the change in wind vs. change in altitude. From these, find
 *      the desired thrust.  Use this to calculate engine control and  fuel
 *      flow.
 *
 * Arguments:
 *       *altitude       float       feet
 *
 * Returns:
 *      gamma          float   radians
 *
 * Modified:  by:           what:
 *-----------------------------------------------------------------------------
 */
float
gamma_3(float *altitude)
                             /* altitude in FT */
{
    /*
     * Find true flight path angle as a function of altitude, altitude
     * rate, derivative of true airspeed w.r.t. altitude, and true
     * airspeed.
     *
     * If the resulting thrust is less than the idle thrust, then find the
     * descent rate at idle thrust, as a function of the idle thrust and
     * the true airspeed.
     *
     * Also, the true flight path angle is found. Used for the cases where:
     * 1) Mach held constant, vertical speed held constant 2) CAS  held
     * constant, vertical speed held constant
     */

    float           gamma = 0.0;/* true flight path angle in radians */
    float           altitude_rate = 0.0;    /* vertical velocity in
                                             * ft/s */
    float           true_airspeed = 0.0;    /* local true airspeed in
                                             * ft/s */
    float           density = 0.0;  /* air density in slug/ft^3 */
    float           true_airspeed_squared = 0.0;   /* true_airspeed^2 */
    float           dynamic_pressure = 0.0;/* in lb/ft^2 */
    float           lift_coefficient = 0.0;/* lift coefficient (CL) */
    float           weight = 0.0;   /* aircraft weight in lb */
    float           wing_area = 0.0;        /* wing area (S) in ft^2 */
    float           number_of_engines = 0.0;
    float           drag_coefficient = 0.0;/* local drag  coefficient
                                             * (CD) */
    float           drag = 0.0;     /* local drag variable in lb */
    float           altitude_plus_50 = 0.0;/* local var. of current
                                             * alt + 50 FT */
    float           altitude_plus_1000 = 0.0;      /* local var. of current
                                             * alt + 1000 FT */
    float           ground_speed_at_alt_plus_50 = 0.0;    /* ground speed at
                                                    * alt+50 FT */
    float           ground_speed_at_altitude = 0.0;       /* ground speed at
                                                    * current alt. */
    float           wind_shear_limit = 0.0;/* max dwind/dh */
    float           dtrue_airspeed_daltitude = 0.0;       /* dvtrue/dh */
    ENGINE_CALLS_INPUT engine_input;
    ENGINE_CALLS_OUTPUT engine_output;
    ENGINE         *engine;  /* pointer to engine model */
    float           velocity_at_altitude_begin = 0.0;     /* for gradient */
    float           velocity_at_altitude_end = 0.0;       /* for gradient */
    float           save_wind_speed = 0.0; /* for gradient calcs */
    /* MARK CHANGES */
    float Temp_Actual;
    float Temp_Standard;
    float dhp_dhg;

    /* initialize engine_input and engine_output */
```

```
    memset((char *) (&engine_input), '\0', sizeof(ENGINE_CALLS_INPUT));
    memset((char *) (&engine_output), '\0', sizeof(ENGINE_CALLS_OUTPUT));

    true_airspeed = Da_airplane->true_airspeed;
    weight = Global_aircraft_model.descent_weight;
    wing_area = Global_aircraft_model.model_data->gross_wing_area;
    ground_speed_at_altitude = Da_airplane->ground_speed_at_z;
    wind_shear_limit = Da_limits->max_wind_shear;
    dtrue_airspeed_daltitude = Da_airplane->dtrue_airspeed_daltitude;
    number_of_engines =
                Global_aircraft_model.model_data->effective_number_of_engines;
    engine = Global_aircraft_model.model_data->engine;

    /* MARK CHANGES */
    Temp_Actual= ts_air_temperature(Da_airplane->altitude);
    Temp_Standard = 518.69 - LAPSE_RATE*(Da_airplane->altitude);
    dhp_dhg = Temp_Standard / Temp_Actual;
```

To insure that Da_desired_values->altitude rate is preserved in the integration routine, the value is divided by a factor of `dhp_dhg`.

```
    altitude_rate = Da_desired_values->altitude_rate / dhp_dhg; /* SRD 1/23/97 dhp_dhg
factor added. */

    gamma = altitude_rate / true_airspeed;

    if (!(Da_output_control->compute_engine_diagnostics)) {
       /* Skip the engine performance calculations, since they are */
       /* not needed in order to calculate the flight-path. */
       /* Zero the thrust, engine control, and fuel consumption so */
       /* that values left in memory will not appear in the */
       /* trajectory output data. */

       Da_airplane->total_thrust = 0.0;
       Da_airplane->engine_control = 0.0;
       Da_airplane->fuel_consumption = 0.0;

       return (gamma);
    }

    /* Find the drag of the aircraft assuming lift = weight */
    /* (level flight). */

    density = air_density(*altitude);
    true_airspeed_squared = true_airspeed * true_airspeed;
    dynamic_pressure = 0.5 * density * true_airspeed_squared;
    lift_coefficient = weight / (dynamic_pressure * wing_area);
    get_flap_and_gear_settings(Da_airplane->cas,
                         &(Global_aircraft_model.flap_angle),
                         &(Global_aircraft_model.landing_gear_down));
    drag_coefficient = get_drag_coefficient(&lift_coefficient);

/* The following qualifier will apply the drag scale factor only to the
 * descent phase.  This scaled drag will give a buffer to TOD position.
 * See PR 1415 for details.
 */
    if ( Da_desired_values->altitude_rate < 0.0)
       drag_coefficient *=
       Global_aircraft_model.model_data->descent_drag_scale_factor;

    drag = drag_coefficient * dynamic_pressure * wing_area;

    /* Find Da_wind->dwind_daltitude = change in winds (body fixed x dir) with alti-
tude. */

    save_wind_speed = Da_wind->wind_speed;

    if  (Da_wind->heading_norm_wind_gradient_flag) {
        altitude_plus_1000 = *altitude + 1000.0;
        velocity_at_altitude_begin = heading_normalized_wind_velocity(*altitude);
        velocity_at_altitude_end = head-
ing_normalized_wind_velocity(altitude_plus_1000);
       Da_wind->dwind_daltitude = (velocity_at_altitude_begin - veloc-
ity_at_altitude_end)
           / (*altitude - altitude_plus_1000);
    }
    else {

        altitude_plus_50 = *altitude + ALT_DIFFERENCE;
        ground_speed_at_alt_plus_50 = vground(altitude_plus_50,
```

```
                                        Da_airplane->true_airspeed,
                                        Da_airplane->course_angle,
                                        Da_wind);
        Da_wind->dwind_daltitude = (ground_speed_at_alt_plus_50 -
ground_speed_at_altitude) /
           ALT_DIFFERENCE;
    }

    Da_wind->wind_speed = save_wind_speed;

   /* Limit Da_wind->dwind_daltitude to defined limits. */

    if (Da_wind->dwind_daltitude > wind_shear_limit)
       Da_wind->dwind_daltitude = wind_shear_limit;
    if (Da_wind->dwind_daltitude < -wind_shear_limit)
       Da_wind->dwind_daltitude = -wind_shear_limit;

    Da_airplane->total_thrust = (weight / GRAVITY_ACCELERATION) *
       ((dtrue_airspeed_daltitude + Da_wind->dwind_daltitude) * true_airspeed +
        GRAVITY_ACCELERATION) * gamma + drag;

    engine_input.mach = Da_airplane->mach;
    engine_input.altitude = *altitude;
    engine_input.thrust = Da_airplane->total_thrust / number_of_engines;

    compute_engine_control_given_thrust(&engine_input, engine,
                                  &engine_output);

    Da_airplane->engine_control = engine_output.engine_control;
    Da_airplane->fuel_consumption = number_of_engines *
       engine_output.fuel_flow;

    return (gamma);
}
```

The variable, `Da_desired_values->altitude_rate`, is also contained in the function `call_vtrue_gama_func`, which is contained in the file segment_integration.c.

```
/*
 *---------------------------------------------------------------------------
 * call_vtrue_gama_func:
 *     finds true/calibrated airspeed or flight path angle based on
 *     the command table constraints
 *     .
 *
 * Arguments:
 *     Vtrue          vttype
 *     double         alt
 *     float          *mach (may not be used)
 *     float          *cas (may not be used)
 *     float          *tas (may not be used)
 * Returns:
 *     none
 *---------------------------------------------------------------------------
 */
float
call_vtrue_gama_func(Vtrue vttype, double alt, float *mach, float *cas, float *tas)
{
       float return_value;
       /* MARK ADDED variables */

       float Temp_Standard;
       float Temp_Actual;
       float dhp_dhg;

       switch(vttype)
       {
       case FIND_MACH_TAS:
          return_value = find_mach_tas(alt, mach, cas, tas);
          break;
       case FIND_CAS_TAS:
          return_value = find_cas_tas(alt, mach, cas, tas);
          break;
       case GAMA_1:
          return_value = (Da_desired_values->inertial_flight_path_angle *
           Da_airplane->ground_speed_at_z / Da_airplane->true_airspeed);
          break;
```

```
        case GAMA_2:
```

To insure that a constant altitude rate is preserved, the return value must be divided by a factor of `dhp_dhg`. When the altitude rate value is passed to the integration routine the proper altitude rate will be conserved.

```
            /* MARK CHANGES */
            Temp_Actual= ts_air_temperature(Da_airplane->altitude);
            Temp_Standard = 518.69 - LAPSE_RATE*(Da_airplane->altitude);
            dhp_dhg = Temp_Standard / Temp_Actual;
           return_value = (Da_desired_values->altitude_rate / Da_airplane-
>true_airspeed) / dhp_dhg;
           break;
        case DUMMY_VT:
        default:
           return_value = FLOAT_NOT_SET;
           break;
        }
        return(return_value);
}
```

Miscellaneous Changes

Most of the miscellaneous changes stem from one fundamental change which was made to the `traj_data` structure. This change distinguished between pressure altitude and geometric altitude to avoid confusion in the output sections of the code.

File: ts_struct.h
line: 841
Change: Distinguished between pressure altitude and geometric altitude
Reason: This was so they would both be tracked, saved and reported and distinguishable from each other in the output. The traj_data->pres_altitude replaces traj_data-> altitude.

```
 (NOT THE RIGHT PIECE OF CODE)

typedef struct Trajectory_data_st {
    float           time;
    float           path_distance;
    float           geo_altitude;
    float           pres_altitude;
    float           mach;
    float           true_airspeed;
    float           cas;
    float           dtrue_airspeed_dt;
    float           inertial_flight_path_angle;
    float           engine_control;
    float           total_thrust;
    float           fuel_consumption;
    float           fuel;
    float           altitude_rate;
    float           ground_speed;
    float           course_angle;
    float           turn_endpoint;
    float           wind_speed;
    float           wind_direction;
    float           air_temperature;
    float           air_pressure;
    float          x;
    float          y;
    struct Trajectory_data_st *next;
    struct Trajectory_data_st *previous;
}               TRAJECTORY_DATA;
```

File: file_output.c
line: 262
Change: Removal of /100 from altimeter_altitude
Reason: To see altimeter altitude instead of flight level

Code:

```
if (Da_output_control->print_secondary_parameters)
   {
       altimeter_altitude = TS_input_alt->control_altitude_alt;
       if ( altimeter_altitude < FLIGHT_LEVEL_TRANSMISSION_ALTITUDE )
           fprintf(Trajectory_file, "\t\t\t Alt = %5.0f (altimeter)\n",
               altimeter_altitude);
       else
           fprintf(Trajectory_file, "\t\t\t Alt = FL%5.0f (altimeter)\n",
               altimeter_altitude); /* MARK CHANGE */
   }
```

File: file_output.c
line: 1538
Change: Calculate geometric altitude from altimeter altitude
Reason: Calculate geometric altitude from altimeter altitude

```
/* SRD 1/22/97 Calculate geometric altitude here. */
                traj_data->geo_altitude =  alt_altimeter_to_geometric_MEP(
                            FT_TO_NM(traj_data->x),
                            FT_TO_NM(traj_data->y),
                            traj_data->pres_altitude,
                            0.0,  /* time */
                            TS_weather->altimeter_setting)
```

File: file_output.c
line: 1558
Change: Use traj_data->geo_altitude instead of previous traj_data->altitude
Reason: To distinguish between altitude types in traj_data structure

```
/* SRD 1/22/97 */
                fprintf(Trajectory_file, "%5.0f %7.1f %5.0f %5.3f %6.1f %6.1f ",
                   (traj_data->time +
                     Joint_trajectory_control->end_of_center_elapsed_time),
                   (-(FT_TO_NM(traj_data->path_distance))),
                   traj_data->geo_altitude,
                   traj_data->mach,
                   FPS_TO_KTS(traj_data->true_airspeed),
                   FPS_TO_KTS(traj_data->cas));
```

File: file_output.c
line: 1570
Change: Use traj_data->geo_altitude instead of previous traj_data->altitude
Reason: To distinguish between altitude types in traj_data structure

```
/* SRD 1/22/97 */
                fprintf(Trajectory_file, "%5.0f %6.1f %5.0f %5.3f %6.1f %6.1f ",
                   traj_data->time,
                   (-(FT_TO_NM(traj_data->path_distance))),
                   traj_data->geo_altitude,
                   traj_data->mach,
                   FPS_TO_KTS(traj_data->true_airspeed),
                   FPS_TO_KTS(traj_data->cas));
```

File: file_output.c
line: 1592
Change: Use traj_data->pres_altitude instead of previous traj_data->altitude
Reason: To distinguish between altitude types in traj_data structure and because alt_geometric_to_altimeter is now a conversion between pressure and altimeter altitudes.

```
/* SRD 1/22/97 */
                    altimeter_altitude = traj_data->pres_altitude;
                    if (TS_weather->wthr_type == WEATHER_3D_BIN)
                    {
                    /* SRD 1/2/97 Should change name of function to match action. */
                        altimeter_altitude = alt_geometric_to_altimeter(
                            FT_TO_NM(traj_data->x),
                            FT_TO_NM(traj_data->y),
                            traj_data->pres_altitude,
                            0.0, /* time */
                            TS_weather->altimeter_setting);
                    }
```

File: file_output.c
line: 1612
Change: Altimeter altitude format changed as well as removal of /100
Reason: So altimeter altitude is printed instead of flight levels

```
printf(Trajectory_file, "%4.0f %5.1f %4.0f %4.0f %5.1f %6.1f %5.0f",
                   FPS_TO_KTS(traj_data->ground_speed),
                   course_degrees,
                   FPS_TO_KTS(traj_data->wind_speed),
                   RAD_TO_DEG(traj_data->wind_direction),
                   traj_data->air_temperature,
                   traj_data->air_pressure,
                   altimeter_altitude); /* MARK CHANGE /100 out! */
```

File: file_output.c
line: 1636
Change: Calculate geometric alt from pressure alt using new function alt_altimeter_to_geometric_MEP. Note use of traj_data->pres_altitude
Reason: Geo_altitude must now be calculated from pressure for the output file instead of visa versa

```
/* SRD 1/22/97 Calculate geometric altitude here. */
                traj_data->geo_altitude =  alt_altimeter_to_geometric_MEP(
                            FT_TO_NM(traj_data->x),
                            FT_TO_NM(traj_data->y),
                            traj_data->pres_altitude,
                            0.0,  /* time */
                            TS_weather->altimeter_setting)
```

File: file_output.c
line: 1655
Change: Use traj_data->geo_altitude instead of previous traj_data->altitude
Reason: To distinguish between altitude types in traj_data structure

```
/* SRD 1/22/97 */
                fprintf(Trajectory_file, "%5.0f %6.1f %5.0f %5.3f %6.1f %6.1f ",
                   traj_data->time,
                   (-(FT_TO_NM(traj_data->path_distance))),
                   traj_data->geo_altitude,
                   traj_data->mach,
                   FPS_TO_KTS(traj_data->true_airspeed),
                   FPS_TO_KTS(traj_data->cas));
```

File: file_output.c
line: 1696
Change: Altimeter altitude format changed as well as removal of /100
Reason: So altimeter altitude is printed instead of flight levels

```
 fprintf(Trajectory_file, "%4.0f %5.1f %4.0f %4.0f %5.1f %6.1f %5.0f",
                   FPS_TO_KTS(traj_data->ground_speed),
                   course_degrees,
                   FPS_TO_KTS(traj_data->wind_speed),
                   RAD_TO_DEG(traj_data->wind_direction),
```

```
traj_data->air_temperature,
traj_data->air_pressure,
altimeter_altitude);  /* MARK CHANGE TO GET RID OF /100 */
```

File: file_output.c
line: 1741
Change: Use traj_data->pres_altitude instead of previous traj_data->altitude
Reason: To distinguish between altitude types in traj_data structure

```
/* SRD 1/22/97 */
if ((index_speed == VARIABLE_MACH_CAS) ||
    (traj_data->pres_altitude != traj_data->next->pres_altitude) ||
     (dt < 1.0)) {

    if ((index_speed == VARIABLE_MACH_CAS) ||
        (traj_data->pres_altitude == traj_data->next->pres_altitude) ||
        (dt < 1.0)) {
```

File: descent_profile.c
line: 1110
Change: Refer to pres_altitude Instead of altitude in traj_data
Reason: To distinguish between altitude types in traj_data structure

```
/*
     * Find the altitudes of integration steps in the constant Mach
     * segment which bracket altitude. First, find the offsets in the
     * trajectory_data array for the two integration steps which bracket
     * ALT2
     */

    traj_data =
           Da_trajectory->segment[segment_number_capture_mach].trajectory_data;
    while (traj_data->next != NULL)
    {
        /* SRD 1/22/97 */
        if (traj_data->pres_altitude > Da_segment->altitude)
            break;
        traj_data = traj_data->next;
    }
    if (traj_data->previous)
    {
        /* SRD 1/22/97 */
        altitude_1 = traj_data->previous->pres_altitude;
        distance_1 = traj_data->previous->path_distance;
    }
        else
    {
        altitude_1 = 0;
        distance_1 = 0;
    }
    /* SRD 1/22/97 */
    altitude_3 = traj_data->pres_altitude;
    distance_3 = traj_data->path_distance;
```

File: descent_profile.c
line: 1205
Change: Refer to pres_altitude Instead of altitude in traj_data
Reason: To distinguish between altitude types in traj_data structure

```
while (traj_data != NULL)
    {
        /* SRD 1/22/97 */
        if (traj_data->pres_altitude > Da_segment->altitude)
            break;
        traj_data = traj_data->next;
        index_j++;
    }
```

File: descent_profile.c
line: 1370
Change: Refer to pres_altitude Instead of altitude in traj_data
Reason: To distinguish between altitude types in traj_data structure

```
while (traj_data != NULL)
        {
            /* SRD 1/22/97 */
            if (traj_data->pres_altitude > input->altitude)
                break;
            traj_data = traj_data->next;
            index_j++;
        }
        Da_trajectory->segment[Da_segment->segment_number-1].highest_integration_step
                 = index_j;

        if (traj_data->previous)
        {
            /* SRD 1/22/97 */
            altitude_1 = traj_data->previous->pres_altitude;
            distance_1 = traj_data->previous->path_distance;
        }
        else
        {
            altitude_1 = 0;
            distance_1 = 0;
        }
        /* SRD 1/22/97 */
        altitude_3 = traj_data->pres_altitude;
        distance_3 = traj_data->path_distance;
```

File: descent_profile.c
lines: 1416, 1498,1531,1611
Change: Refer to pres_altitude Instead of altitude in traj_data
Reason: To distinguish between altitude types in traj_data structure

line 1416

```
       /* SRD 1/22/97 */
        altitude_1 = traj_data->pres_altitude;
```

line 1498

```
/* SRD 1/22/97 */
        if (traj_data->pres_altitude > Da_segment->altitude)
        {

            /* SRD 1/22/97 */
            altitude_1 = traj_data->previous->pres_altitude;
            altitude_3 = traj_data->pres_altitude;
```

line 1531

```
/* SRD 1/22/97 */
        if (traj_data->pres_altitude > Da_segment->altitude)
        {

            /* SRD 1/22/97 */
            altitude_1 = traj_data->previous->pres_altitude;
            altitude_3 = traj_data->pres_altitude;
```

line 1611

```
while (traj_data != NULL)
    {
            /* SRD 1/22/97 */
            if (traj_data->pres_altitude > Da_segment->altitude)
                break;
            traj_data = traj_data->next;
            index_j++;
    }
```

File: specific_profile.c
lines: 6111,6122,6446
Change: Refer to pres_altitude Instead of altitude in traj_data
Reason: To distinguish between altitude types in traj_data structure

line 6111:

```
/* SRD 1/22/97 */
Da_time_control->da_input.altitude = traj_data->pres_altitude;
```

line 6122:

```
/* SRD 1/22/97 */
aircraft->track.altitude    = traj_data->pres_altitude;
```

line 6446:

```
    /* SRD 1/22/97 */
    center_dat_to->pres_altitude = center_dat_from->pres_altitude;
```

File: segment_integration.c
lines: 236,1026
Change: Refer to pres_altitude Instead of altitude in traj_data
Reason: To distinguish between altitude types in traj_data structure

line 236:

```
    /* SRD 1/22/97 */
    Da_airplane->altitude = traj_data->pres_altitude;
```

line 1026:

```
/* SRD 1/22/97 */
traj_data->pres_altitude = Da_airplane->altitude;
```

File: shared_memory.c
lines: 930,962,1002,1195,1323
Change: Refer to pres_altitude Instead of altitude in traj_data
Reason: To distinguish between altitude types in traj_data structure

line 930:

```
 /* SRD 1/22/97 */
 One_aircraft_trajectory->trajectory_altitude[j] =
     traj_data->pres_altitude;
```

line 962:

```
/* SRD 1/22/97 */
 One_aircraft_trajectory->trajectory_altitude[j-1] =
```

line 1002:

```
/* SRD 1/22/97 */
 One_aircraft_trajectory->trajectory_altitude[j-1] =
     traj_data->pres_altitude;
```

line 1195:

```
/* SRD 1/22/97 */
One_aircraft_trajectory->trajectory_altitude[j-1] =
traj_data->pres_altitude;
```

line 1323:

```
/* SRD 1/22/97 */
One_aircraft_trajectory>trajectory_altitude[j-1] =
 traj_data->pres_altitude;
```