DOT/FAA/CT- 87/15

FAA TECHNICAL CENTER
Atlantic City International Airport
N.J. 08405

Navigation Recovery Block Design Description

E.F. Hitt
S.A. Prater

Battelle
Columbus Divison
505 King Avenue
Columbus, Ohio 43201

March 1987

U.S. Department of Transportation
Federal Aviation Administration

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
DOT/FAA/CT-87/15		

4. Title and Subtitle	5. Report Date
Navigation Recovery Block Design Description	March 1987
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
E. F. Hitt, S. A. Prater	DOT/FAA/CT-87/15
	10. Work Unit No. NAS2-11853
9. Performing Organization Name and Address Lockheed-Georgia Company Marietta, Georgia 30063	11. Contract or Grant No.
	13. Type of Report and Period Covered Contractor Report
12. Sponsoring Agency Name and Address U.S. Department of Transportation Federal Aviation Administration Technical Center Atlantic City Airport, New Jersey 08405	14. Sponsoring Agency Code

15. Supplementary Notes

Point of Contact: W. E. Larsen/MS 210-2
Ames Research Center
Moffett Field, CA 94035

16. Abstract

This report describes the recovery block approach to fault-tolerant software and illustrates its application to a flight navigation problem. The navigation problem using VOR (Visual Omnirange) and DME (Distance Measuring Equipment) is developed, especially with regard to its software specification and implementation needs. Related aspects of the recovery block approach are reviewed, and Ada implemented source code is presented.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Acceptance Test, Avionics, Digital Flight Systems, Error Detection, Fault-Tolerant Software, Navigation Aids, Recovery Block, Software Structure, System Architecture	Unlimited Subject Category 38

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price*
Unclassified	Unclassified		

*For sale by the National Technical Information Service, Springfield, Virginia 22161

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

EXECUTIVE SUMMARY

The detailed design of the software for a navigation recovery block is specified. The navigation software models the Very High Frequency (VHF) Omnidirectional Range (VOR)/Distance Measuring Equipment (DME). A description of the program structure is provided. Also, an example problem and the program source listing are included.

INTRODUCTION

PURPOSE.

The contents of this document establish the software design for the Very High Frequency (VHF) Omnidirectional Range (VOR)/Distance Measuring Equipment (DME) model software. The VOR/DME model is implemented with the fault tolerant software technique of a recovery block.

OBJECTIVE.

This document is intended to demonstrate the type of software specification and implementation that the certification specialist may be expected to encounter in the next generation of digital or all-electric aircraft.

SYMBOLOGY

The flow diagrams in this document have been developed according to top-down methods. Table 1 summarizes the symbols used in the flowcharts in this document.

REQUIREMENTS

The recovery block method is a fault tolerant software technique which provides alternate components which may be switched in (usually serially) to take the place of a faulty component that has been rejected by the acceptance test. These alternate components are designed independently from the main software component (the primary alternate) and generally only provide partial functionality of the software component, thus reducing it to a degraded, simpler mode. Prior to entering an alternate, the state of the process is restored to that current just before entry to the primary alternate [1]. Software fault tolerance is described in further detail in Appendix A [2].

The hierarchical diagram for the executive functions for running a recovery block is shown in Figure 1. The recovery block time line is shown in Figure 2 [3]. The following report (included in Appendix B) provides detailed information concerning the sequential processing and forms part of this specification:

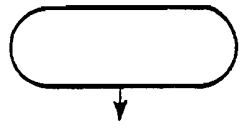
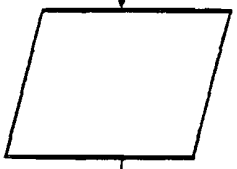
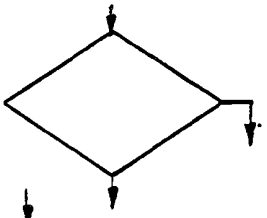
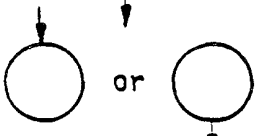
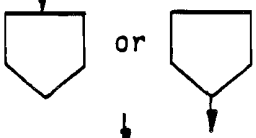
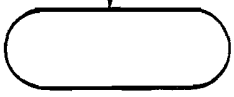
> Hitt, Ellis, "Software System Error Detection and Correction Techniques", Battelle Columbus Laboratories, Columbus, Ohio, March 5, 1986.

The navigation recovery block consists of two components: a primary alternate and a secondary alternate. Figure 3 gives a simple block diagram representation of the recovery block structure.

VOR/DME.

The hierarchical diagram of the primary alternate for the VOR/DME model is shown in Figure 4. The hierarchical diagram for the secondary alternate of the recovery block for the VOR/DME model is given in Figure 5. The

TABLE 1. FLOWCHART SYMBOLS

Symbol	Definition
	Beginning of subroutine.
	Processing function; defined operation(s) causing a change in value, form, or location of information.
	Input/Output function; information available for processing (input) or recording of processed information (output).
	A decision block that determines which of a number of alternative paths to follow.
	On-page connector.
	Off-page connector.
	Termination of subroutine.

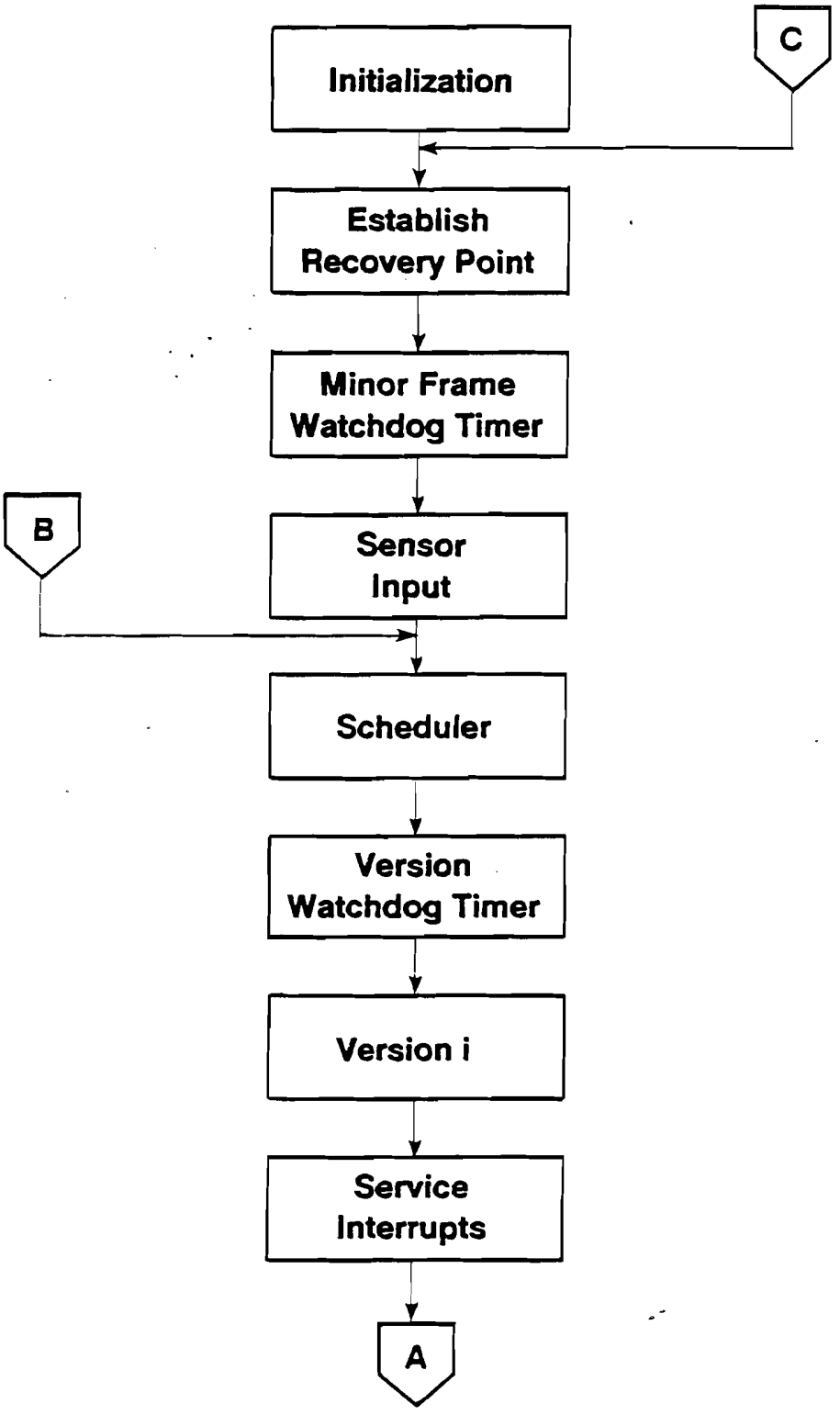


FIGURE 1. EXECUTIVE SOFTWARE FOR NAVIGATION SYSTEM RECOVERY BLOCK

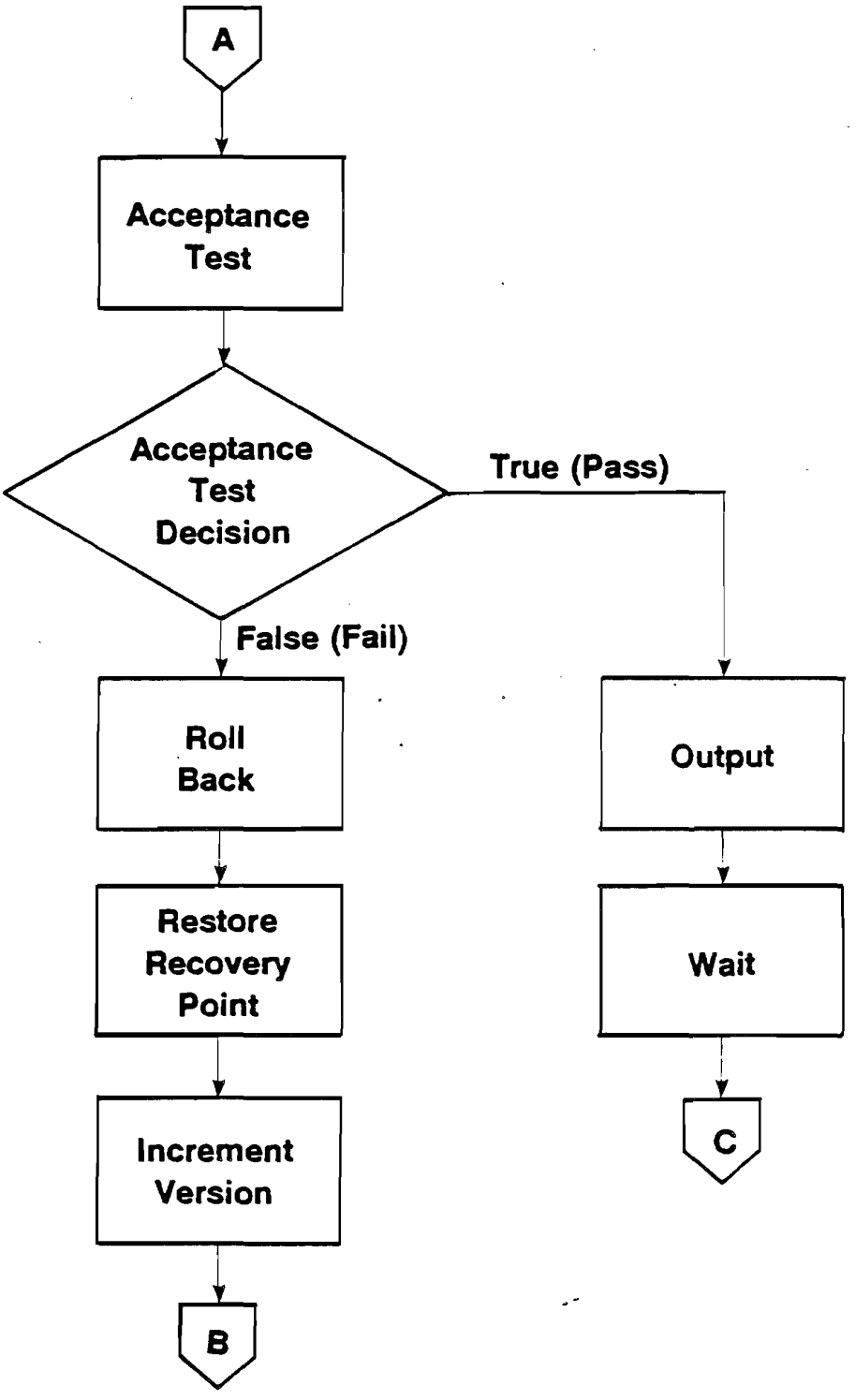


FIGURE 1 (Continued). EXECUTIVE SOFTWARE FOR NAVIGATION SYSTEM RECOVERY BLOCK

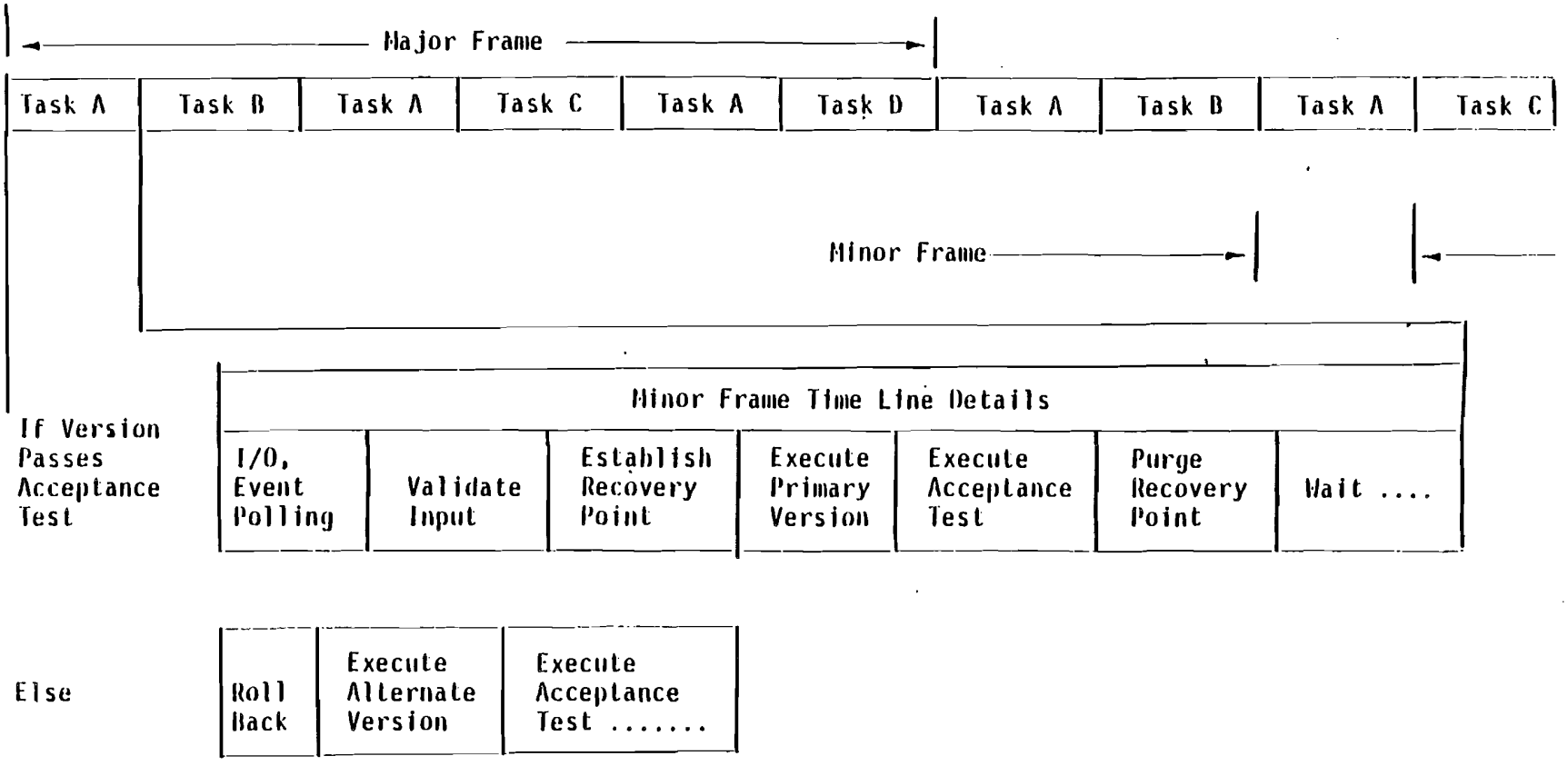


FIGURE 2. RECOVERY BLOCK TIME LINE [3]

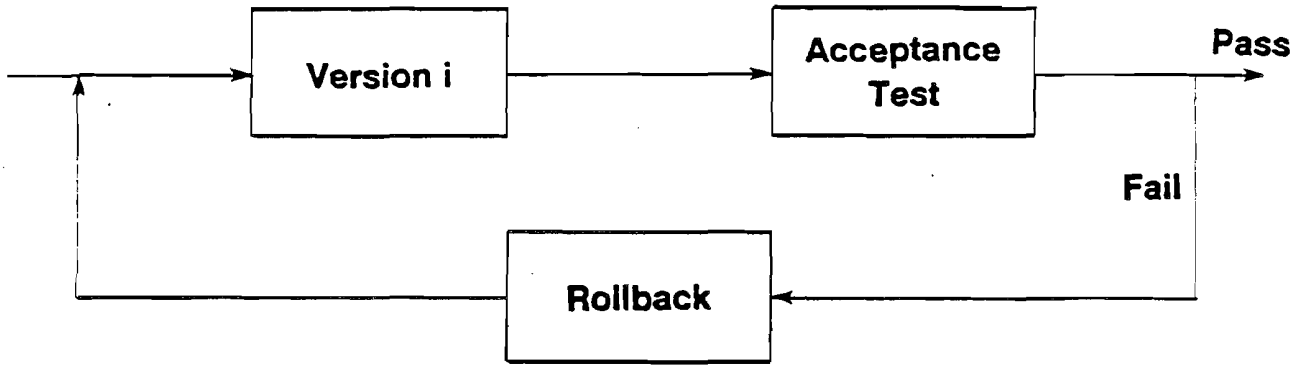


FIGURE 3. RECOVERY BLOCK STRUCTURE

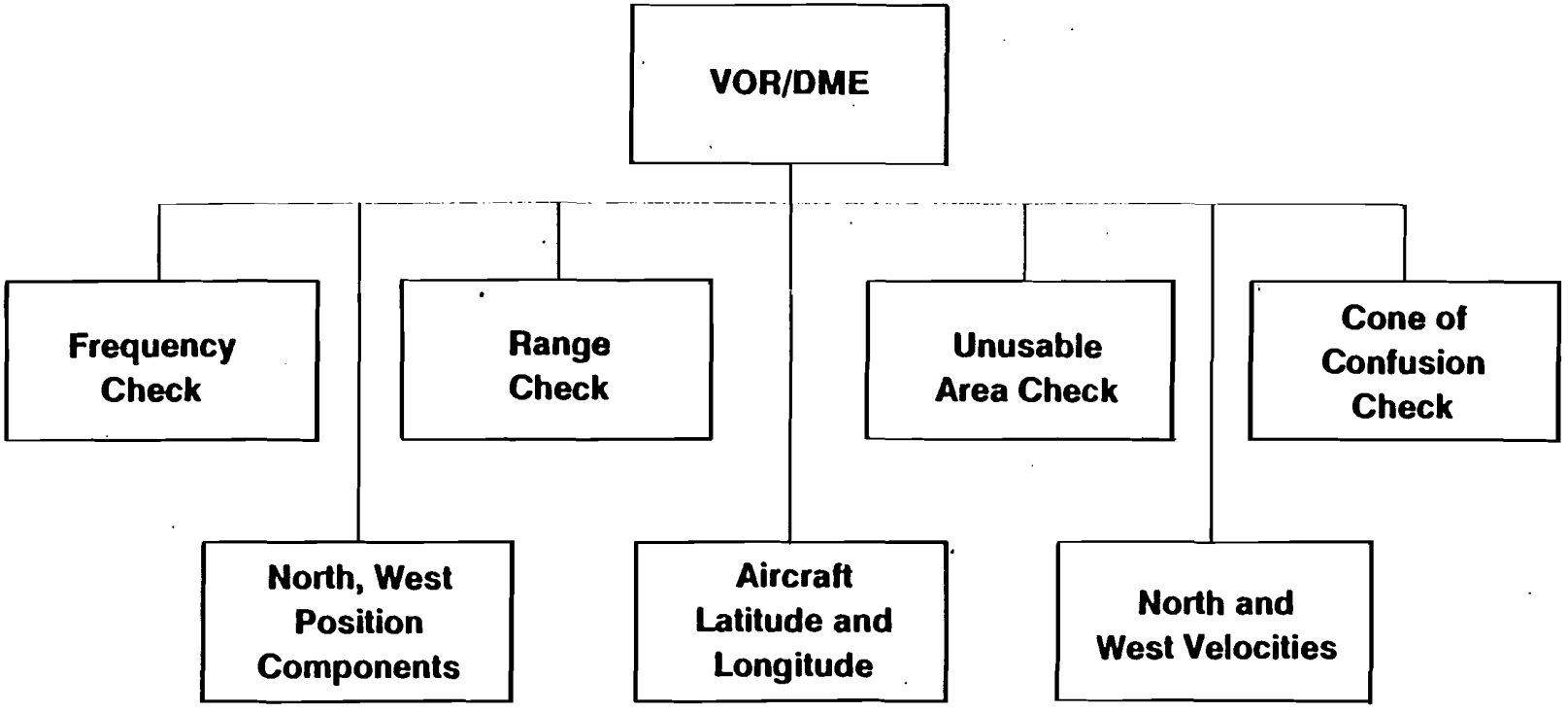


FIGURE 4. VOR/DME PRIMARY ALTERNATE HIERARCHICAL DESIGN

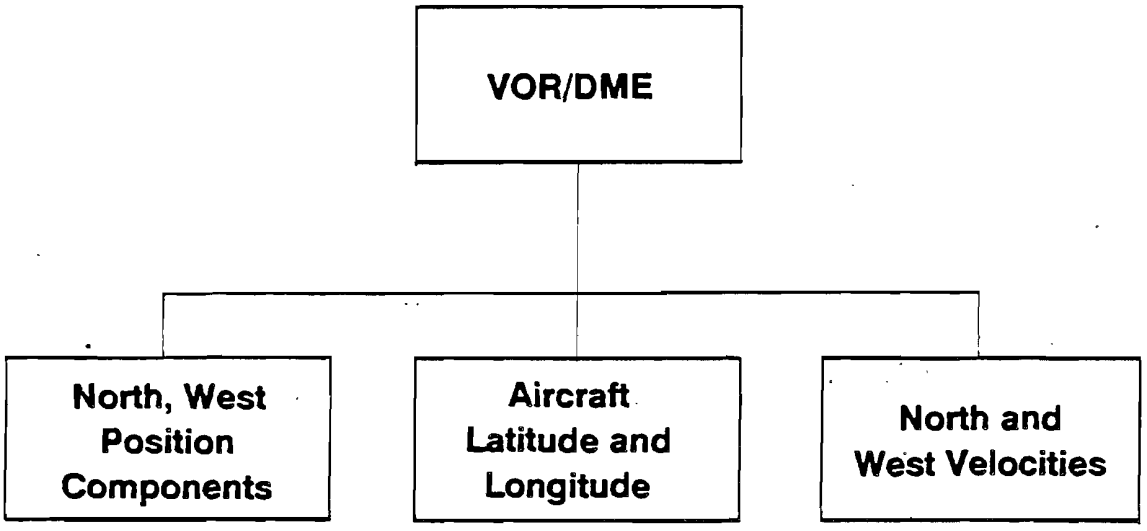


FIGURE 5. VOR/DME SECONDARY ALTERNATE HIERARCHICAL DESIGN

VOR/DME module calculates the aircraft north and west position and velocity components and the aircraft latitude and longitude from the aircraft altitude and the range and bearing of the aircraft from the tuned navigation station.

PRIMARY ALTERNATE. The primary alternate performs the following checks prior to performing the actual calculations:

a. frequncy check
b. range check
c. unusable area check
d. cone of confusion check

Frequency Check. In the frequency check module, the software loops through the stored navigation station data to determine if a station frequency matches the tuned frequency. If the tuned navigation frequency does not match a station in the navigation area, then the probability of a signal, P_s, is zero. Therefore, the total probability of the signal, P_t, which is calculated after each check is performed, is zero since

$$P_t = P_t * P_s.$$

If the tuned navigation frequency does match a station in the navigation area, then $P_s = 1$ [4].

The VOR receiver should operate on the 50 KHz spaced channels (a total of 160) from 108.00 MHz through 117.95 MHz [5]. When a DME transponder is intended to operate in association with a single VHF navigational facility, it also operates in the 108.00 MHz to 117.95 MHz frequency band [6]. Appendix C provides a table of stations in the United States and their frequencies.

Range Check. This software module checks whether or not the aircraft is within the effective range of the tuned navigation station. The DME equipment should provide at least a 300 nautical mile range [6]. If the aircraft is within the effective station range, then $P_s = 1$. If not, then $P_s = 0$. The effective station ranges are given in Table 2.

To combine the result of this check with the result of the frequency check,

$$P_t = P_t * P_s.$$

Unusable Area Check. This module determines if the aircraft is within an unusable area of the tuned navigation station. This check is performed by looping through all of the specified unusable areas for the tuned station and determining the location of the aircraft with respect to the unusable area [4]. The various unusable areas for each station will be stored in a data base for comparison. If it is determined that the aircraft is within an unusable area, then $P_s = 0$. Otherwise, $P_s = 1$. Again, to update the total probability of the signal,

$$P_t = P_t * P_s.$$

TABLE 2. EFFECTIVE STATION RANGES BY CLASS [4]

Class	Altitude	Effective Range (in miles)
T	≤ 12,000 feet	25
L	< 18,000 feet	40
H	< 18,000 feet	40
H	Within the 48 conterminous states between 14,500 feet and 17,999 feet	100
H	≥ 18,000 feet and ≤ flight level (FL) 450 (45,000 feet)	130
H	> FL 450	100

Legend: T = Terminal; L = Low; H = High

Cone of Confusion Check. This module determines if the aircraft is within the cone of confusion of the tuned navigation station. This check calculates the elevation angle from the tuned station to the aircraft. If the elevation angle is too large, then the aircraft is in the area above the VOR/DME station and it will experience a loss of signal. A simplified diagram representing this is shown in Figure 6.

VOR/DME ALGORITHMS. The primary and secondary alternates have the following modules in common and utilize the same equations:

a. Aircraft Latitude and Longitude
b. North and West Position Components
c. North and West Velocities

Aircraft Latitude and Longitude. Figure 7 shows the aircraft and VOR/DME station positions. The VOR/DME station position is represented with fixed geocentric spherical coordinates (R, θ, ϕ: $\hat{i}_1$, $\hat{j}_1$, $\hat{k}_1$) with θ representing the station longitude measured positive westward from the Greenwich meridian, and ϕ representing the station latitude measured positive northward from the equator. The aircraft's position is represented by tangent plane coordinates fixed at the site of the VOR/DME station with the $\hat{i}_2$ unit vector denoting true north and the $\hat{j}_2$ unit vector denoting true west. (See Figure 8.) We define

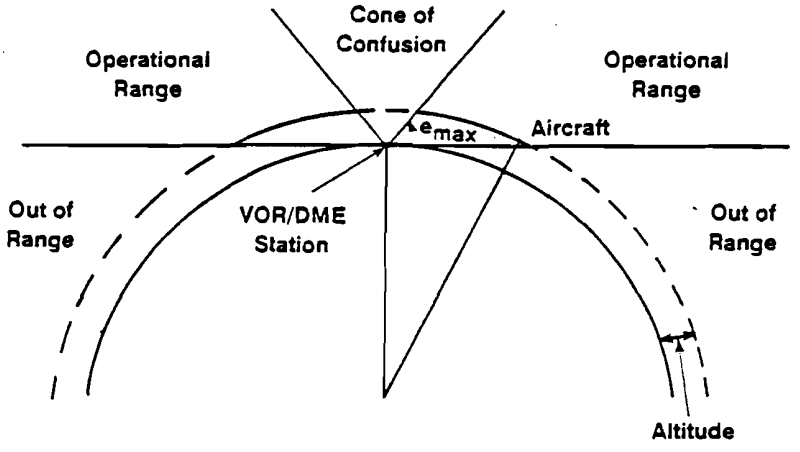


FIGURE 6. CONE OF CONFUSION [4]

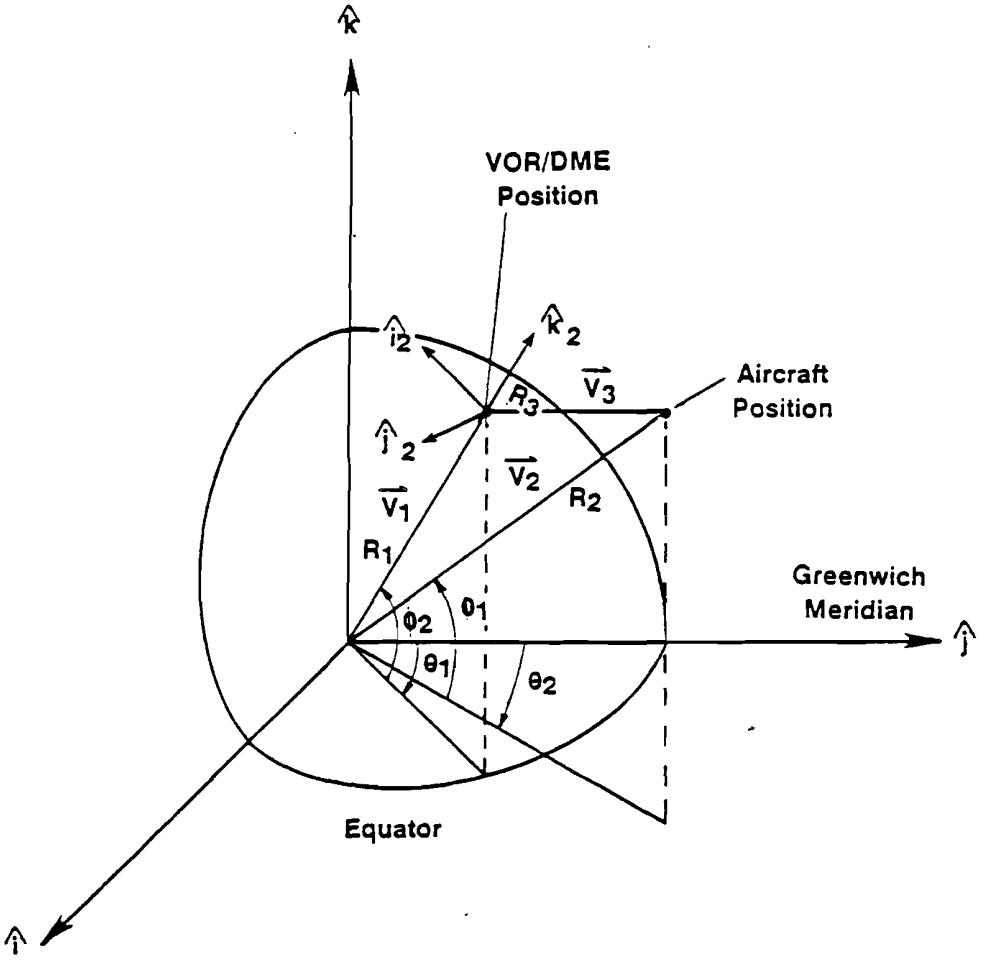


FIGURE 7. AIRCRAFT AND VOR/DME STATION POSITIONS

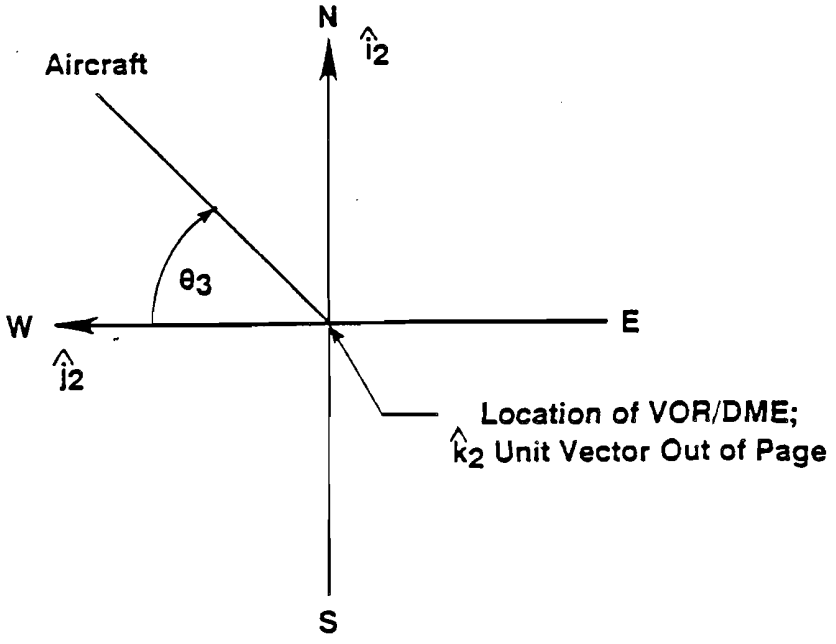


FIGURE 8. AIRCRAFT POSITION ON TANGENT PLANE COORDINATES

$\vec{V}_1 \triangleq$ vector representing the location of the VOR/DME station

$\vec{V}_2 \triangleq$ vector representing the location of the aircraft

$\vec{V}_3 \triangleq$ vector representing the position of the aircraft relative to the VOR/DME station

$R_1 \triangleq$ (the sea level radius of the earth) + (the VOR/DME station elevation above sea level)

$R_2 \triangleq$ (the radius of the earth) + (the aircraft's altitude)

$R \triangleq$ the sea level radius of the earth

Consequently, we can define

$$\vec{V}_1 = (R_1\cos\phi_1\sin\theta_1\hat{i}_1,\ R_1\cos\phi_1\cos\theta_1\hat{j}_1,\ R_1\sin\phi_1\hat{k}_1) = (a_1\hat{i}_1,\ b_1\hat{j}_1,\ c_1\hat{k}_1)$$

$$\vec{V}_2 = (R_2\cos\phi_2\sin\theta_2\hat{i}_1,\ R_2\cos\theta_2\cos\theta_2\hat{j}_1,\ R_2\sin\phi_2\hat{k}_1) = (a_2\hat{i}_1,\ b_2\hat{j}_1,\ c_2\hat{k}_1)$$

To calculate ϕ_3 (reference Figure 9),

$R_2 = R + h$, where $h \triangleq$ the aircraft altitude

$R_3 \triangleq$ the distance of the aircraft from the VOR/DME station

$\phi_3 \triangleq$ the angle that the aircraft makes with the $\hat{i}_2, \hat{j}_2$ plane

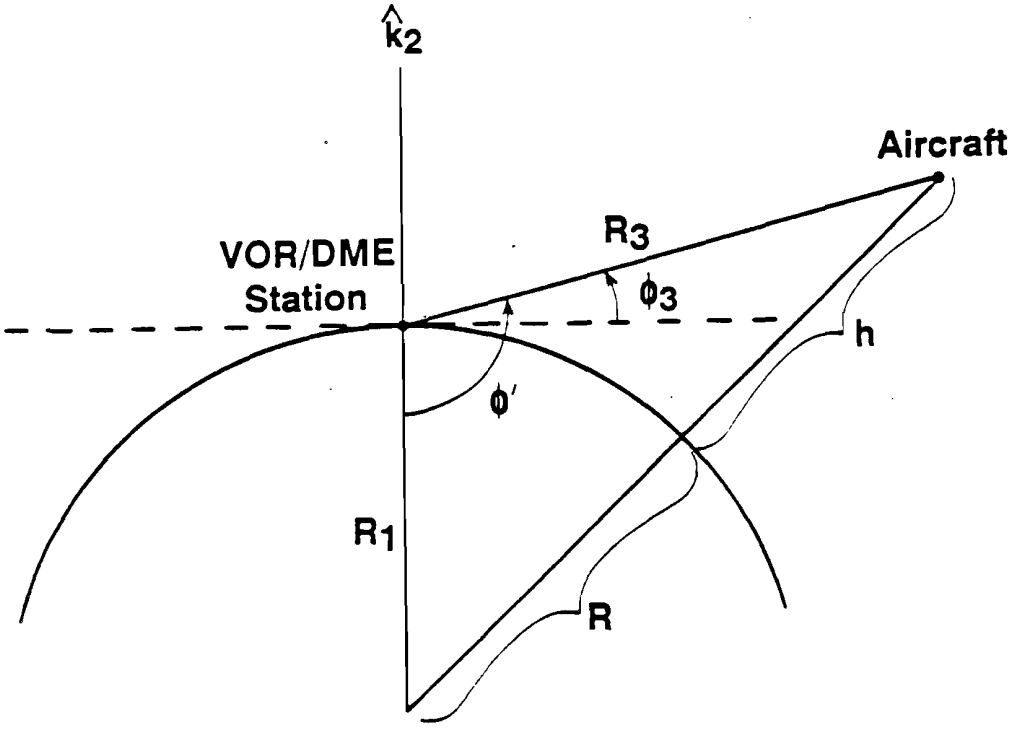


FIGURE 9. EDGE VIEW OF $\hat{i}_2, \hat{j}_2$ PLANE

By the law of cosines,

$$(R+h)^2 = R_1^2 + R_3^2 - 2R_1R_3\cos\phi'$$

$$\phi' = \cos^{-1}\left[\frac{R_1^2 + R_3^2 - (R+h)^2}{2R_1R_3}\right]$$

$$\phi_3 = \phi' - \pi/2$$

Recalling,

$\theta_3 \triangleq$ the aircraft bearing measured positive from west (i.e., $\pi/2$ radians plus the bearing from true north).

$\vec{V_3}$ can be written as

$$\vec{V_3} = (R_3 \cos\phi_3 \sin\theta_3\, \hat{i}_2,\ R_3 \cos\phi_3 \cos\theta_3\, \hat{j}_2,\ R_3 \sin\phi_3\, \hat{k}_2) =$$

$$= (a_3\hat{i}_2,\ b_3\hat{j}_2,\ c_3\hat{k}_2).$$

We want to express $\vec{V_3}$ in terms of the geocentric coordinates (i.e., $\vec{V_3} = (a_3\hat{i}_1,\ b_3\hat{j}_1,\ c_3\hat{k}_1)$.

The proper coordinate transformations are

$$a_3 = (-\sin\phi_1 \sin\theta_1 a_2 + \cos\theta_1 b_2 + \cos\phi_1 \sin\theta_1 c_2)$$

$$b_3 = (-\sin\phi_1 \cos\theta_1 a_2 - \sin\theta_1 b_2 + \cos\phi_1 \cos\theta_1 c_2)$$

$$c_3 = (\cos\phi_1 a_2 + \sin\phi_1 c_2)$$

By substitution,

$$\vec{V_3} = R_3[(-\sin\phi_1 \sin\theta_1 \cos\phi_3 \sin\theta_3 + \cos\theta_1 \cos\phi_3 \cos\theta_3 + \cos\phi_1 \sin\theta_1 \sin\phi_3)\,\hat{i}_1 + (-\sin\phi_1 \cos\theta_1 \cos\phi_3 \sin\theta_3 - \sin\theta_1 \cos\phi_3 \cos\theta_3 + \cos\phi_1 \cos\theta_1 \sin\phi_3)\hat{j}_1 + (\cos\phi_1 \cos\phi_3 \sin\theta_3 + \sin\phi_1 \sin\phi_3)\,\hat{k}_1]$$

$\vec{V_2}$ is related to $\vec{V_1}$ and $\vec{V_3}$ by the vector equation

$$\vec{V_2} = \vec{V_1} + \vec{V_3}.$$

Equating components gives

$\hat{k}_1$: $c_2 = c_1 + c_3$

$$R_2 \sin\phi_2 = c_1 + c_3$$

$$\phi_2 = \sin^{-1}\left[\frac{1}{R_2}(c_1 + c_3)\right].$$

The i and j components can now be used to compute the sine and cosine of θ_2.

$\hat{i}_1$: $a_2 = a_1 + a_3$

$$\sin \theta_2 = \left[\frac{1}{R_2 \cos\phi_2} (a_1 + a_3)\right]$$

$\hat{j}_1$: $b_2 = b_1 + b_3$

$$\cos \theta_2 = \left[\frac{1}{R_2 \cos\phi_2} (b_1 + b_3)\right]$$

The longitude, θ_2, is then given by

$$\theta_2 = \tan^{-1} \left(\frac{\sin \theta_2}{\cos \theta_2}\right) = \tan^{-1} \left(\frac{a_1 + a_3}{b_1 + b_3}\right).$$

When performing this computation, care must be exercised to assure that θ_2 falls in the correct quadrant of the equatorial plane. The function is defined so that it yields longitude values which range between $-\pi$ and $+\pi$.

$$\theta_2 = \tan^{-1}\left(\frac{a_1 + a_3}{b_1 + b_3}\right) \quad \text{for } (b_1 + b_3) > 0,$$

$$\theta_2 = \tan^{-1}\left(\frac{a_1 + a_3}{b_1 + b_3}\right) + \pi \quad \text{for } (b_1 + b_3) < 0 \text{ and } (a_1 + a_3) > 0,$$

$$\theta_2 = \tan^{-1}\left(\frac{a_1 + a_3}{b_1 + b_3}\right) - \pi \quad \text{for } (b_1 + b_3) < 0 \text{ and } (a_1 + a_3) < 0,$$

$$\theta_2 = \pi/2 \quad \text{for } (b_1 + b_3) = 0 \text{ and } (a_1 + a_3) > 0,$$

$$\theta_2 = -\pi/2 \quad \text{for } (b_1 + b_3) = 0 \text{ and } (a_1 + a_3) < 0.$$

North and West Position Components. Neglecting the curvature of the earth, the $\hat{i}_2, \hat{j}_2$ components of the aircraft position are

$$\text{Position}_{\text{west}} = X_1 = R_3 \cos \phi_3 \sin \theta_3 \, \hat{i}_2$$

$$\text{Position}_{\text{north}} = Y_1 = R_3 \cos \phi_3 \cos \theta_3 \, \hat{j}_2$$

with $R_3 \triangleq$ the distance of the aircraft from the VOR/DME station

$\phi_3 \triangleq$ the angle that the aircraft makes with the $\hat{i}_2, \hat{j}_2$ plane

$\theta_3 \triangleq$ the aircraft bearing measured positive from west (i.e., 90 degrees plus the bearing from true north.)

North and West Velocities. The velocity components are calculated with the true airspeed as a function of the aircraft's heading.

We define

$\psi \triangleq$ the pitch angle (the angle that the longitudinal axis of the aircraft makes with the $\hat{i}_2, \hat{j}_2$ plane)

$\theta_4 \triangleq$ the heading of the aircraft relative to the north (positive clockwise)

$U_A \triangleq$ true airspeed

The northern component of the velocity is given by

$$U_N = U_A \cos \psi \cos \theta_4 \hat{i}_2.$$

The western component of the velocity is given by

$$U_W = -U_A \cos \psi \sin \theta_4 \hat{j}_2.$$

Therefore, the ground speed is

$$|U| = (U_N^2 + U_W^2)^{\frac{1}{2}} = U_A \cos \psi.$$

RECOVERY BLOCK INDEPENDENCE.

To "force" the independence of the two alternates in the recovery block, the equations which determine the north and west position components, the aircraft latitude and longitude, and the north and west velocities are "forced" to perform their calculations differently. The primary alternate calculates the sine and cosine values with the use of the Ada math library. However, the sine and cosine terms in the secondary alternate are determined with the use of a power series representation for these functions. The Maclaurin series for the sine function is

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} + \ldots.$$

The Maclaurin series for the cosine function is

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \ldots + (-1)^n \frac{x^{2n}}{(2n)!} + \ldots.$$

The Maclaurin series may be used to approximate values of the sine and cosine functions. When the polynomial approximation for the sine function involves using the sum of the first two terms, the error is less than $|x^5|/5!$. The error involved in using the sum of the first two terms in the cosine function is less than $x^4/4!$.

In the determination of the aircraft latitude and longitude, the inverse of the tangent function is used. To make the two alternates explicitly different, the primary alternate uses the Ada math library to determine these values and the secondary alternate uses the interval halving method of numerical analysis. This method is sometimes called the bisection method and enables the determination of a root of $f(x) = 0$, accurate within a specified tolerance value, given values of x_1, and x_2 such that $f(x_1)$ and $f(x_2)$ are of the opposite sign [7].

ACCEPTANCE TEST.

An acceptance test is a logical expression or algorithm which checks the acceptability of the results that are generated by a software component [1]. For the navigation software, the acceptance test will be an independent calculation of the aircraft's velocity components. These velocity components are compared to those determined earlier.

For the acceptance test, the north and west velocity components are determined by the change in the north and west position components with respect to the change in time. For an aircraft in flight, with points P_1 and P_2 indicating the aircraft's positions at times t_1 and t_2, respectively, the corresponding north and west position components (X_1, Y_1 and X_2, Y_2) are calculated. The velocity components are then determined using the following equations:

$$\text{Velocity}_{west} = (X_2 - X_1)/(t_2 - t_1)$$

$$\text{Velocity}_{north} = (Y_2 - Y_1)/(t_2 - t_1)$$

If the velocity component calculations are found to be correct with the acceptance test, then it will be assumed that all of the navigation recovery block software is correct. If the primary alternate is in use and the acceptance test fails, the software will recover the input state to its condition prior to when the incorrect or faulty version was run (known as "rollback") and restart the computation using the secondary alternate. If the acceptance test fails while the secondary alternate is in use, then the entire navigation software will fail.

EXAMPLE.

Consider the navigation station located in Albuquerque, New Mexico. Assume that the station elevation is at sea level. Assume that an aircraft is flying at an altitude of 15,000 feet, a distance of 50 nautical miles from the station, with a northeast heading. The aircraft's tuned frequency is 113.2 MHz. Its true airspeed is 200 miles per hour.

From Table C-1, we find the station frequency for Albuquerque, New Mexico is 113.2 MHz. and its class is high (H). Since the tuned frequency matches the station frequency, the frequency check passes.

With Table 2, since the station's class is high, it has an effective range of 100 nautical miles for aircraft flying at an altitude betwen 14,500 feet and 17,999 feet. We have assumed that our aircraft is flying at an altitude of 15,000 feet, so its distance must be less than or equal to the station's effective range of 100 nautical miles. In this case, 50 nautical miles is less than 100 nautical miles, so the range check passes.

The unusable area check passes since we do not have any unusable areas declared for this navigation station. Similarly, the cone of confusion check passes since the aircraft's elevation angle with respect to the station is not too large.

The aircraft latitude and longitude are determined using the following values:

$$\phi_1 = 35: 2: 37.4 \text{ N} = 35.04372°$$

$$\theta_1 = 106: 48: 56.6 \text{ W} = 106.81572°$$

(The station latitude and longitude are obtained from Table D-1.)

$\theta_3 = 135°$ = the bearing measured positive from west

= 90° + the bearing from true north

$$R_1 = 3443.93$$

(The earth's radius is assumed to be 6378.163 km = 3443.93 n mi = 20925732 feet [8].)

$$R_3 = 50 \text{ n mi}$$

$$R_2 = R + h = 3446.77$$

[The aircraft altitude, h, is 15,000 feet = 2.4687 nautical miles (1 nautical mile = 6076.115 feet [9]).]

$$\phi_3 = \cos^{-1}\left[\frac{R_1^2 + R_3^2 - R_2^2}{2R_1R_3}\right] - 90°$$

$$= \cos^{-1}\left[\frac{(3443.93)^2 + (50)^2 - (3446.77)^2}{2(3443.93)(50)}\right] - 90°$$

$$\phi_3 = 92.84° - 90° = 2.84°$$

The aircraft latitude is computed by

$$c_1 = R_1 \sin \phi_1 = (3443.93) \sin (35.04372) = 1977.5092$$

$$c_3 = R_3 (\cos \phi_1 \cos \phi_3 \sin \theta_3 + \sin \phi_1 \sin \phi_3) =$$

(50) [cos (35.04372) cos (2.84) sin (135) + sin (35.04372) sin (2.84)

= 30.3329

$\phi_2 = \sin^{-1} [\frac{1}{R_2} (c_1 + c_3)] = \sin^{-1} [\frac{1}{3446.77} (1977.5092 + 30.3329)]$

$= \sin^{-1} (0.5825286) = 35.6286°$

Therefore, the aircraft latitude = 35.6286° = 35: 37: 42.92 N.

The aircraft longitude is computed by

$b_1 = R_1 \cos \phi_1 \cos \theta_1 = (3443.93) \cos (35.04372) \cos (106.81572) =$

$= -815.69292$

$b_3 = R_3 (-\sin \phi_1 \cos \theta_1 \cos \phi_3 \sin \theta_3 - \sin \theta_1 \cos \phi_3 \cos \theta_3 +$

$\cos \phi_1 \cos \theta_1 \sin \phi_3) = (50) [-\sin (35.04372) \cos (106.81572)$

cos (2.84) sin (135) - sin (106.81572) cos (2.84) cos (135) +

cos (35.04372) cos (106.81572) sin (2.84)]

= 39.081

$a_1 = R_1 \cos \phi_1 \sin \theta_1 = (3443.93) \cos (35.04372) \sin (106.81572)$

= 2699.0288

$a_3 = R_3(-\sin \phi_1 \sin \theta_1 \cos \phi_3 \sin \theta_3 + \cos \theta_1 \cos \phi_3 \cos \theta_3 +$

$\cos \phi_1 \sin \theta_1 \sin \phi_3)$

= (50)[-sin (35.04372) sin (106.81572) cos (2.84) sin (135) +

cos (106.81572) cos (2.84) cos (135) + cos (35.04372)

sin (106.81572) sin (2.84)]

= -7.2520886

$\theta_2 = \tan^{-1} (\frac{a_1 + a_3}{b_1 + b_3}) = \tan^{-1} (\frac{2699.0288 - 7.2520886}{-815.69292 + 39.081})$

$\theta_2 = \tan^{-1} (-3.4660512) + 180°$ since $(b_1 + b_3) < 0$ and $(a_1 + a_3) > 0$.

$\theta_2 = 106.09353°$

Hence, the aircraft longitude is 106.09353° = 106: 5: 36.71 W.

The north and west position components are determined as:

West position component

$$X_1 = R_3 \cos \phi_3 \sin \theta_3$$

$$X_1 = 35.3119.$$

North position component

$$Y_1 = R_3 \cos \phi_3 \cos \theta_3$$

$$Y_1 = -35.3119.$$

Here, we assume that the aircraft's pitch angle is 10° and the heading is 45°.

The north and west velocity components are computed as

$$\text{velocity}_{\text{west}} = 200 \cos 10° \sin 45° = 139.27 \text{ miles per hour}$$

$$\text{velocity}_{\text{north}} = 200 \cos 10° \cos 45° = 139.27 \text{ miles per hour}$$

NOMENCLATURE.

The modules for these calculations incorporate the recommended nomenclature of the Integrated Control Mathematical Routines for MIL-STD-1750 Built-In Functions (BIFs). This nomenclature is shown in Table 3.

TABLE 3. RECOMMENDED NOMENCLATURE [10]

Variable	Nomenclature	Ada Identification
Latitude	LAT	LAT:FLOAT;
Longitude	LONG	LONG:FLOAT;
Relative Bearing	BEAR_REL	BEAR_REL:FLOAT;
True Airspeed	AIRSPEED_TRUE	AIRSPEED_TRUE:FLOAT;
True Heading	HDG_TRUE	HDG_TRUE:FLOAT;

APPENDIX A

SOFTWARE FAULT-TOLERANCE

SOFTWARE FAULT-TOLERANCE

by

Ellis F. Hitt

INTRODUCTION

Real-time systems employing software to implement application functions which may be critical to the safety of the vehicle and its occupants must be assured of the continuous correct operation of software (and the hardware on which the software executes). Since it is virtually impossible through exhaustive testing to prove that software is free of all design and implementation faults, the use of redundant software modules (to achieve robustness to external input faults and to tolerate design and implementation faults) has been the subject of sponsored research and academic studies. The primary methods of using redundant software modules involve either sequential or parallel execution. The method followed in implementation and processing of the redundant software modules distinguishes the software fault-tolerance techniques. Software fault-tolerance techniques can be broadly classified into two categories: self-checking software, limited to detection of software failures; and fault-tolerant software which allows the system to recover after a software failure has occured [MAKAM]. This distinction is important since, up to the point of recovery, the two categories could be identical.

A software fault is any defect within a software component (e.g. a module, a procedure, a process, a collection of processes). Software faults may be due to mistakes in

translating specification into a design or in implementation of the software design. A failure occurs whenever the external behavior of a system does not conform to that described by the system specification [ANDER81]. Figure A-1 represents the relationship between these terms.

Software faults are caused by design mistakes as contrasted with hardware faults, which are caused by both physical wearout and design mistakes. The case where the software accepts external-to-the-system faulted input as good input is a software fault. Although all software faults are due to human mistakes and as such, might seem to cause only unanticipated faults, there can be anticipated software faults. These faults are exemplified by divide-by-zero and overflow faults.

Fault tolerant principles can be discussed in terms of four phases:

(1) Fault detection
(2) Damage assessment
(3) Recovery
(4) Fault treatment

Fault Detection

One way of viewing fault detection in a unified way is that all erroneous-state detection is accomplished by run-time assertions. A run-time assertion is "a logical expression specifying a program state that must exist or a set of conditions that program variables must satisfy at a particular point during program execution" [IEEE 729-83].

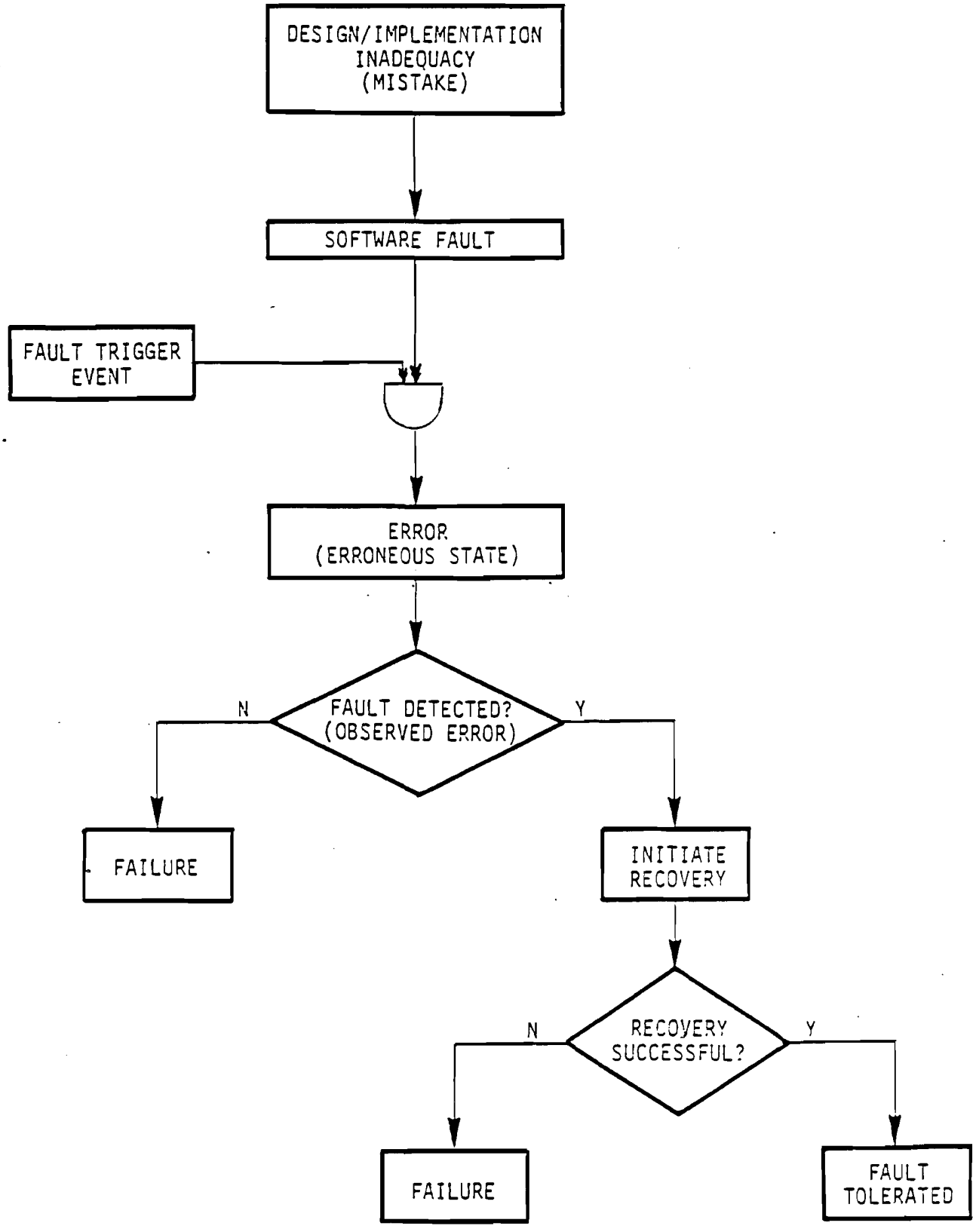


FIGURE A-1. FAULT TOLERANT SOFTWARE EVENT RELATIONSHIPS

In real-time applications, fault detection occurs at the following levels:

(1) Correct operation of each processing unit
(2) Valid transmission of data between digital subsystems
(3) Data validity, prior to use in subsequent computation
    (a) Input of sensor data prior to execution of the algorithm implemented in each version of the control software
        (i) Input range limit test
        (ii) Input rate of change test
        (iii) Parameter correlation check
        (iv) Parameter majority logic check
        (v) Output wraparound test
        (vi) Known input test
        (vii) Known output test
        (viii) Loop dynamic check
        (ix) End of conversion not detected
    (b) Dynamic fault detection consisting of a comparison signal generation followed by a decision mechanism based on the comparison signals
        (i) Comparison signals
            -Redundant measurements
            -Identical sensors
            -Non-identical sensors which have one or more common components in the state vector
            -Two sensors, having no common component, but having a component in common with a third dedicated observer
            -Analytic redundancy
        (ii) Decision mechanism
            Generalized likelihood ratio
            Sequential probability ratio tests
            Modified sequential probability ratio tests

(4) Run-time assertion after execution of the software module implementing the application function prior to output of the result to a display used by the crew or a control actuator.

The run-time assertion takes the form of an Acceptance Test for recovery blocks and the form of a voter for Multi-Version Software (MVS) (also referred to as N-Version Software (NVS)).

Damage Assessment

When a fault in the software state information has been detected, it is necessary to determine the extent of the damage done by the fault before recovery can be accomplished. Damage assessment may be derived from constraints on the flow of information in a system. Encapsulation is the concept of containing the effects of an action to only the objects to which that action in a software component has legal access. Through the proper enforcement of encapsulation, it is possible to assess the extent of the damage to the state information due to a fault.

Recovery

After the extent of the damage has been determined, the system must be restored to a consistent state in order to resume processing. This entire process (detection, damage assessment, and recovery) must take place fast enough to satisfy real-time requirements. There are two main techniques to accomplish recovery, forward recovery and backward recovery.

Forward recovery restores the system to a consistent state by compensating for inconsistencies found in the current state. Forward fault recovery in a single process implies detailed knowledge of the extent of the damage done and a strategy for fixing the inconsistencies. While this may be possible in certain cases where the recovery process was designed to handle the detected fault, it is difficult to conceive appropriate strategies in the case of unanticipated faults such as transient faults.

Backward recovery, often referred to as roll-back, involves restoring the system to some previous known correct state and restarting the computation from that point. "The basic problem is to keep copies of past states of processes, being sure that: a) copies of process states are consistent with one another, so that the state reached after a recovery is really correct; an important issue is to minimize the amount of information needed for roll-back; b) copies of process states are protected against failure of system components." [BARIGAZZI]

Backward recovery must obey the "roll-back consistency rule: If a process Pi is rolled back to a state of its past history, in which the last message exchanged with process Pj was M, then Pj must also be rolled back to a state in which the last message exchanged with Pi was M." [BARIGAZZI]

Fault Treatment

Once a fault has been detected, the damage assessed, and fault recovery accomplished, fault treatment attempts to remedy the fault condition. The damage assessment phase should

identify which software component caused the fault such as through the use of the concept of encapsulation. If the level of encapsulation is at the software component, then possible courses of action are to ignore the fault either temporarily (i.e., skip frame) or forever (i.e., delete the function performed by the software component and continue to operate the system in a degraded mode), retry the function with existing components, or reconfigure with alternate components.

Response Time And Synchronization

No matter which fault tolerant software method is used, real-time systems must arrive at a consistently correct solution within the time frame determined by the control system dynamics. A rule of thumb often given for selecting sampling rates is that a rate of at least five times per system time constant is a good choice. This necessitates that the response time, defined as the delay between the triggering of a job by the arrival of the relevant sensor input to the actuator/display output that finally results, be no more than 1/5 of the system time constant. The response time is the sum of the time for reaching agreement on the sensor data, the time for execution of the application function, synchronization and reaching agreement that its output is correct, and either moving the actuator, or displaying the result. Obviously, time spent in fault detection, damage assessment, recovery, and fault treatment at each level must be accounted for in determining the response time. Failure can occur due to excessively long response times, e.g. the system goes unstable since the hard deadlines for code execution are missed.[KRISHNA]

Synchronization, such as that required at each point of fault detection, can be performed using software or hardware. Software synchronization has significant overhead and the trend is to hardware synchronization [KRISHNA p.6, ALLAN], although some commercial systems utilize software voting on inputs and hardware voting on outputs.

DESCRIPTION OF PRIMARY FAULT TOLERANT SOFTWARE TECHNIQUES

There are two primary methods for providing software fault tolerance: multi-version programming (N-version programming) and recovery blocks. Investigations into combinations of these methods have been conducted at many universities [HITT]. This report will concentrate on the two main methods and the reader is referred to the foregoing reference for description of the combinations.

N-Version Programming (NVP)

N-version programming is the process of independently specifying, designing, generating, and maintaining multiple versions (N>2) of a program (module) for the same application function. This is done by separate, independent, non-communicating programmers who may use different algorithms and even different languages [CHA78]. It is important to realize that N-version programming is defined to be a general methodology for producing highly independent, redundant program modules. It does not restrict the way these multiple versions are operated in a computer. Thus, NVP could very well be used to produce several alternate blocks for use in the recovery

block method [MAKAM p3]. Recent experimental results reject the hypothesis of different versions having independent faults [KNIGHT/LEVESON]. This does not mean that NVP does not provide fault tolerant software, but does indicate that software reliability models based on the assumption of independence may give overly optimistic predictions.

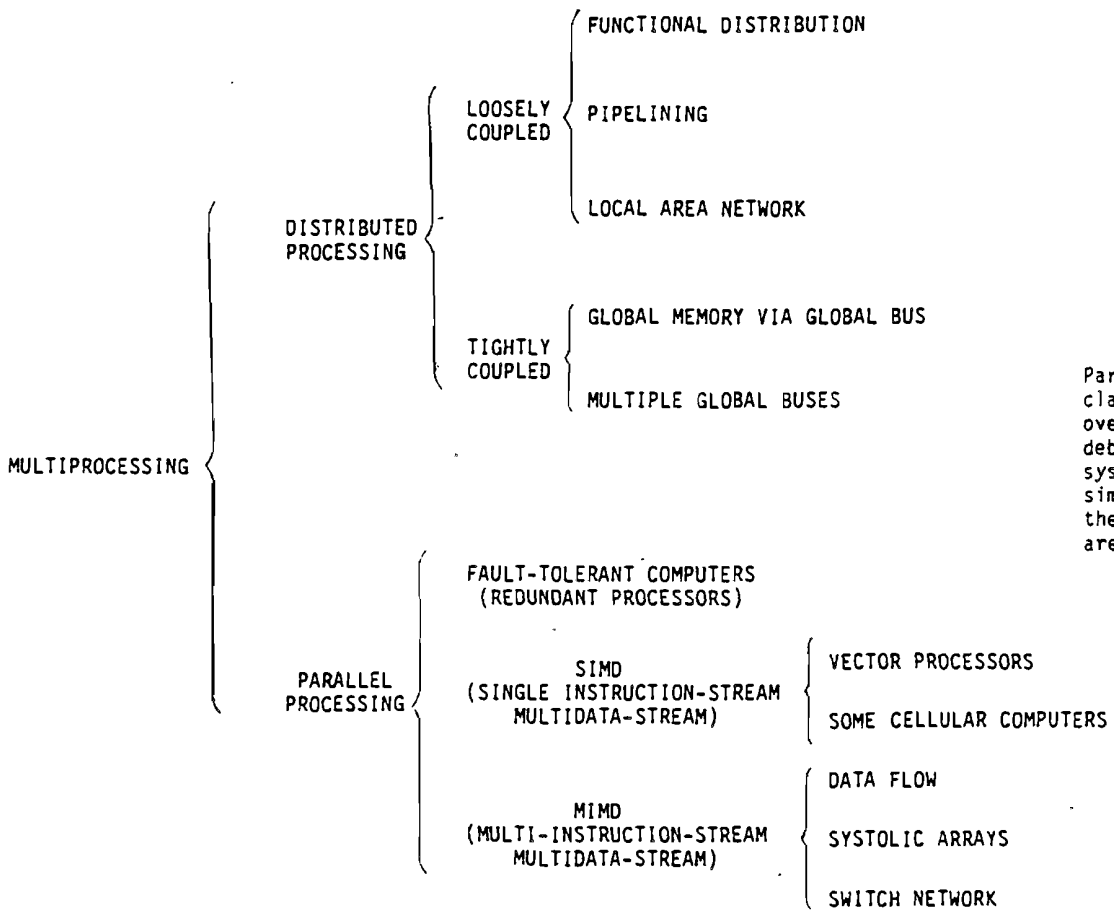
Multi-Version Software (MVS)

The software obtained by NVP, controlled with a system level driver and executed in parallel on a computer system as a unit is called multi-version software (MVS) [MAKAM, ANDER81]. If there are N active versions executing in parallel, this has been called N-version software (NVS) [MAKAM].

"The driver is responsible for:

(i) invoking each of the versions;
(ii) waiting for the versions to complete their execution;
(iii) comparing and acting upon the N sets of results." [ANDER81]

The organization and operation of the computer system on which NVS is implemented must be taken into consideration. If multiple processors are used which is the normal case for NVS, the architecture of the processors impacts the performance of the system. Figure A-2 depicts the classifications of various parallel-processing architectures. Relevant factors include whether the system architecture is parallel or distributed. If parallel, whether it is single instruction/multiple data (SIMD), where all processing elements



Parallel-processing classifications are loose, overlapping, and subject to debate. Because many of the system differences and similarities are not included, the classifications shown here are relatively crude.

SOURCE: ROBIN CHANG, INTERNATIONAL PARALLEL MACHINES

FIGURE A-2. PARALLEL-PROCESSING CLASSIFICATION

simultaneously/synchronously execute the same instructions on different data, or multiple instruction/multiple data (MIMD), where the processing elements are not necessarily synchronous and execute different instructions on different data. At the initiation of a major frame, each of the N-versions must have access to an identical set of input values. Intermediate results would not necessarily have the same data, but the final result output to the voter must represent the output data value and be in a consistent format.

If a distributed processing architecture is used, the consideration of tightly coupled versus loosely coupled architectures also must be taken into account. In a tightly coupled architecture, the computers are not autonomous and may be interconnected by a hardware clock. Tightly coupled multiprocessors share access to common memory and share one copy of the operating system. Loosely coupled architectures supply each processor with its own memory and its own copy of the operating system, allowing each to operate relatively autonomously [HINDIN]. Practically speaking, if the processors exchange results over an external data bus, as contrasted with an internal backplane bus, the architecture will not be tightly coupled since the bus protocol of existing buses, along with bus transport delays, effectively makes control by a master clock impractical. Hence, the driver software must implement the synchronization.

Synchronization is very important in MVS. There are three primary software methods:

(i) Clock synchronous
(ii) Frame synchronous
(iii) Event synchronous.

The synchronization method has to allow for different execution times for each version, including the case where a version might get stuck in an infinite loop due to a design or implementation fault. This case is often handled using a watchdog timer (a timeout mechanism) coupled with the synchronization method. Clock and frame synchronous systems have been designed to achieve tolerance to hardware faults, the best examples being Fault Tolerant Multiple Processors (FTMP) and Software Implemented Fault Tolerance (SIFT), respectively.

Clock synchronous performs the voting check (usually bit-by-bit) at predetermined fixed times rather than waiting for completion signals from the versions. This is very difficult to implement for an NVS as compared to the case of N modular redundant (NMR) processors executing identical versions in parallel [ANDER81].

Frame synchronous mechanizations partition the control task a priori into a number of subtasks, each of which needs to be iteratively executed in some time relationship to the other subtasks and each of which can be completed in a single frame in a block of frames. These subtasks can then be assigned to frames in specific patterns such that the sequencing of the subtasks is correct and such that the iteration frequency of each subtask is correct. The resulting pattern normally repeats every n frames and this block of n frames is called a major frame as shown in Figure A-3. Exchanges between channels of input data and output data are scheduled a priori. This approach requires multiple frames to complete the process of sampling an input, transfer of this input to all versions, processing of the input to produce an output for each version, and voting on the outputs. This results in a transport delay of three or four frames.

Event synchronous systems are loosely coupled and evolved from the frame synchronous systems in order to achieve

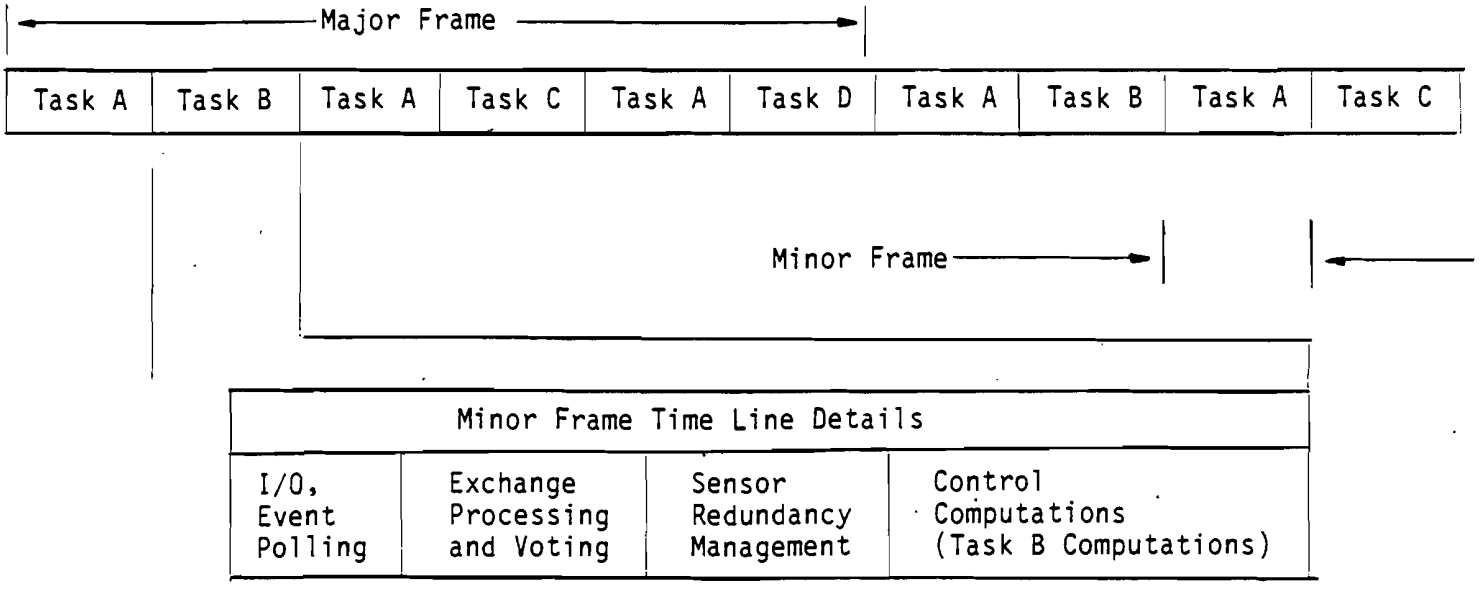


FIGURE A-3. FRAME SYNCHRONOUS TIME LINE

tolerance to design faults in the software. The major events at which synchronization takes place between redundant computing elements are: inputs, outputs, interrupts, programmed exceptions, and the intermodule and interprocess communication in the multi-version software [MAKAM].

Initially a version is in an inactive state. When invoked by the driver, it enters into a waiting state where it waits for a synchronization signal representing a request for service from the driver. When this signal is received, it transfers into a running state as shown in Figure A-4. If any terminating condition is signaled by the status in the comparison vectors, then the execution of this version is terminated and it returns to the inactive state. Otherwise, it generates a comparison-vector when a cross check (cc)-point is satisfied. It then uses a synchronization signal to notify the driver that a comparison-vector is ready, and finally returns to the wait state.

Synchronization between the N processors is accomplished under control of the driver using a synchronization algorithm. Each processor enters a synchronization phase when it generates a synchronizing event and transmits a synchronization message or when it receives the first synchronization message from another processor. A time-out clock is activated when a processor enters a synchronization phase. A processor enters a wait-phase within a synchronization phase only if it generates a synchronization event but not when it receives a synchronization message from another processor. The wait phase is terminated by an internally generated time-out interrupt at the end of a preset time-out limit, or earlier if N-1 synchronization messages are received. The synchronization phase is terminated when all the synchronization messages are processed by the processor and its normal execution is resumed [MAKAM].

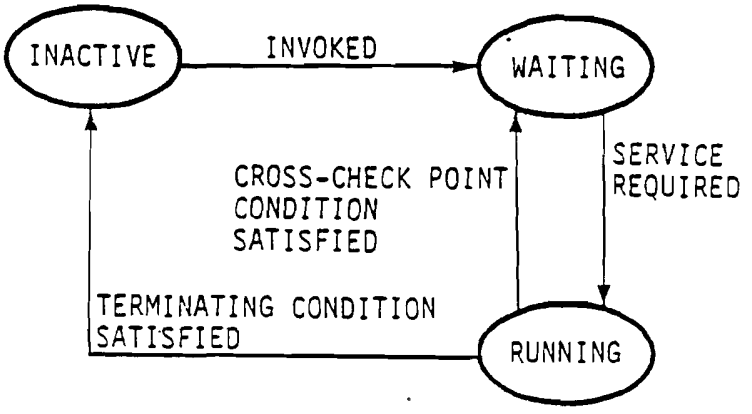


FIGURE A-4. STATE TRANSITIONS OF A VERSION

Error Detection

It is desirable to be able to distinguish between errors due to software design faults and hardware physical failures or malfunctions. Software faults can be in the driver operating system (OS) or in the application software (AS). Errors are detected by two basic mechanisms: the synchronization procedure and the decision function (voter). Synchronization, voting, and error recovery are provided by the underlying operating system and hardware in NVS.

Synchronization errors can be grouped into incompatibility errors and timing violations [MAKAM]. The actual processing of synchronization messages involves checking for compatibility of event-types (and label fields), running the decision function on the comparison-vector data to get the most acceptable results, and finally updating the local comparison-vector.

Since physical clocks do not keep perfect time, but can drift with respect to one another, the clocks must be periodically resynchronized. This can be done in software by exchanging clock values. A global functional resynchronization of processes should be performed periodically by the OS to recover from faults of any type and origin which cause a processor to temporarily lose its context and hence its coordination with the other processors. Dynamic resynchronization in an event synchronous design requires the restart points be defined a priori and implemented as part of the global scheduling algorithm in the driver operating system.

Applications Software. Consider the forms of logical structure available in Ada as shown in Figure A-5. These are: blocks, subprograms, packages, and tasks. A block is a section

of program code located in the executable part of some logical unit (subprogram, main program, package, or task) optionally preceded by a declarative part and optionally followed by exception handlers. A task is a subunit whose parent task may be a subprogram, a declare block, a package, or another task. The execution of the parent task determines the start and finish of the execution of the task.

Application Software

```
Package              Subprogram           Task
  Subprogram           Procedure            Task
    Procedure            Block                Block
      Block                Statement              Statement
        Statement        Task
      Task                 Block
        Block            Function
    Function               Expression
      Expression
```

FIGURE A-5. SOFTWARE STRUCTURE HIERARCHY

Versions Error Detection. The decision function (algorithms used for comparison of output variables from each version) for an event synchronous system depends on the type of comparison-vector, but must permit an inexact match at most crosscheck (cc) points due, for example, to inexact sampling of inputs, and different computational precision in each version. Characterization of a cc-point includes the specification of the data formats of all the comparison-variables in the comparison vector, the error tolerance limits (if non-zero), and the time constraints to be satisfied (in real-time

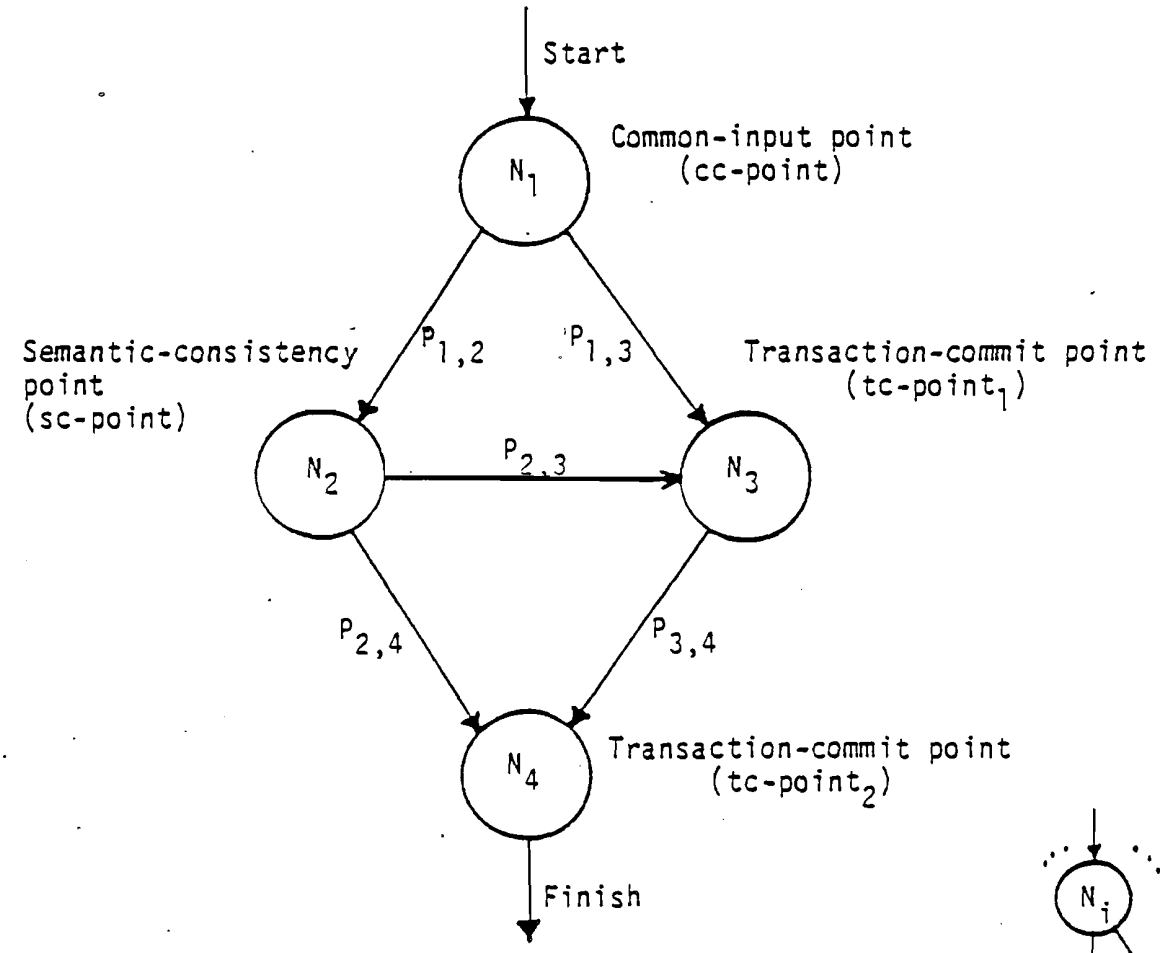
applications) by the segments of software between any two adjacent nodes as shown in the cc-point graph of Figure A-6a & A-6b. [MAKAM p.40]. All N versions have the same number of built-in cross-check points which serve as both synchronization points as well as communication ports between the different versions. The driver operating system executes the decision function, updates the comparison variables (or performs output), controls the exchange of data, and finally returns control to the application software under control of the operating system scheduler which is event synchronized.

Damage Assessment

Damage assessment is not required in an NVS if the activities of the versions are atomic [ANDER81, p.279]. Error location becomes important if one wants to distinguish between errors due to software design faults (operating system or application software) and hardware physical faults. The strategy for discovering the origin of a fault depends on the design approach. Heuristics have been suggested for isolating a hardware failure and a software fault [MAKAM p.75-76].

Recovery

Recovery in NVS normally involves ignoring the values identified as erroneous by the check [ANDER81 p.279] and continuing to the next major frame as long as a minority of the processors' synchronization messages disagree. The minority processors should attempt resynchronization with the majority good processors. If a clear majority of the processors do not produce compatible synchronization messages, then a global resychronization becomes necessary [MAKAM p. 70].



i - a cc-point
(i,j) - a directed branch
$t_{i,j}$ - specified time constraint
R_i - a priori reliability of cc-point

FIGURE A-6a. CC-POINT GRAPH OF A SIMPLE PROGRAM VERSION

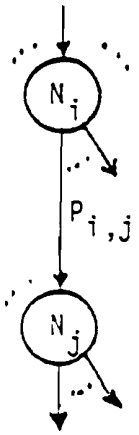


FIGURE A-6b. PARTIAL GRAPH OF A COMPLEX PROGRAM

Fault Treatment

Fault treatment in NVS simply results in the versions determined to have produced erroneous results being ignored [ANDER81 p.279]. The operating system should provide a default failure exception handler task to determine the type of such exception and save the status in a special buffer to be used later by the error manager. At the next cross-check point, the decision function will indicate the particular version in error. The reconfiguration task can then either reinitialize the failed task such that there are no states which are inconsistent, or replace the version by unlinking the module completely encapsulating the failed program segment and replacing it with a compatible but more reliable unit from a different version.

Alternate NVS System Architectures

Figure A-7 depicts various alternate architectures for an NVS system.

The first architecture is illustrative of a system with redundant analog sensors whose output must be sampled and converted to a digital value for input to a version that implements a function such as computing the present state vector of the vehicle. The driver operating system must synchronize the start of the conversion by each analog-to-digital (A/D) converter. If a voter is part of each version as indicated in Figure 7a, each voter module must

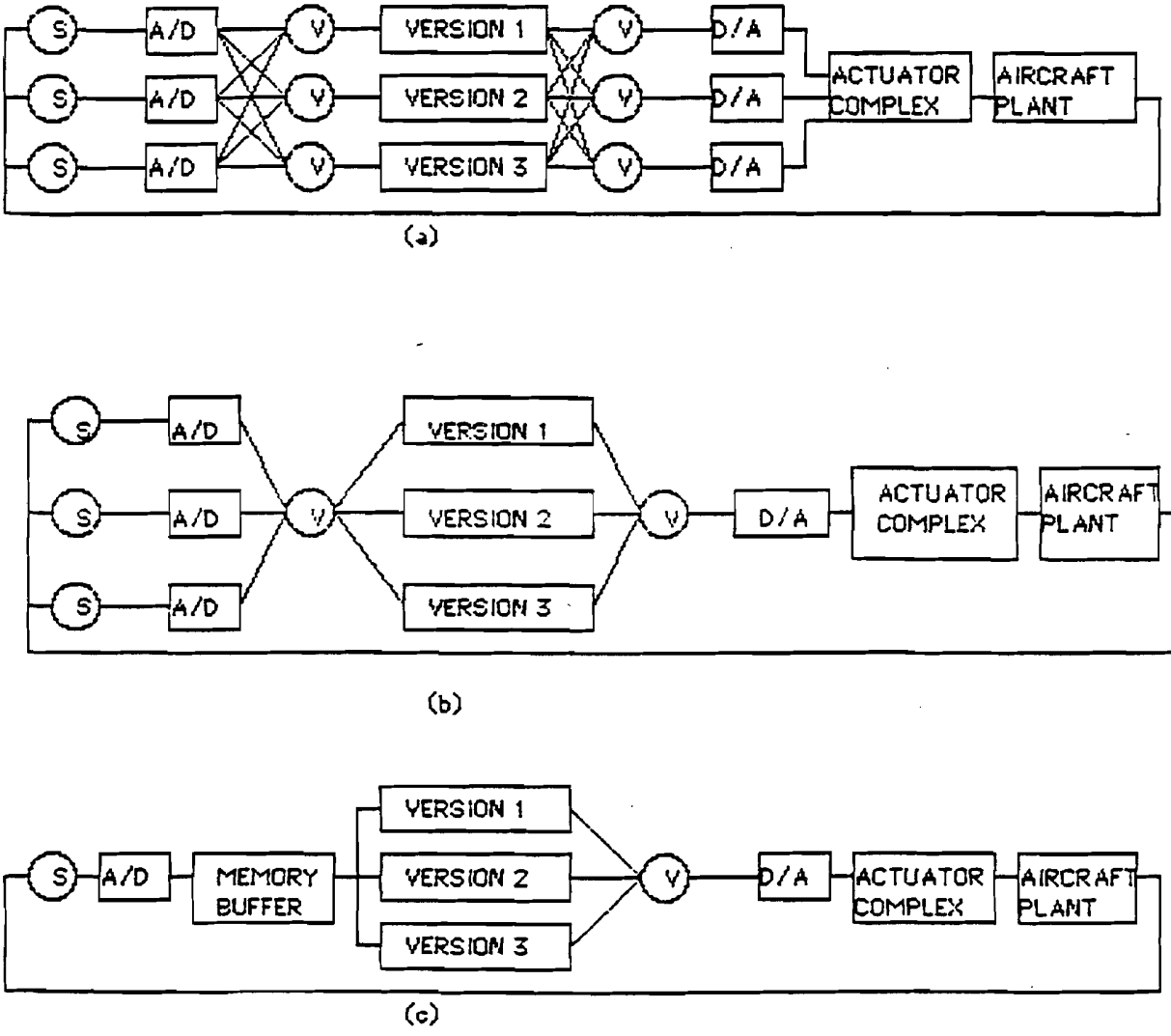


FIGURE A-7. ALTERNATE N-VERSION SOFTWARE SYSTEM ARCHITECTURES

(whether hardware or software) start a timer and await the DATA READY indication from each converter, at which time the data can be transmitted over the processor data bus for use by the input voter. When all three A/D outputs are received, the voter decision function can be invoked either by the local operating system or the driver operating system to arrive at the value to be input to the version. The operating system must monitor the completion of all voters prior to synchronizing the start of each version's processing. Since the versions will have different execution times, the voter on the output of each version must wait until all versions' results are available for the vote using the applicable decision function, or the version execution time watchdog timer times out. The operating system must again monitor the status of each versions completion, and vote completion. The outputs from each version to the digital-to-analog (D/A) converter should be synchronized by the director operating system. The input to a D/A is parallel. The final step of the voting process is to prepare the output for transmission under command of the director operating system to the D/A. If the D/A is connected to the processer's parallel data bus, no further conversion would be required once the data is received by the D/A. If a serial data bus is used to transmit the output from the processor containing each version, the D/A must be preceded by a serial to parallel conversion to utilize the data. This serial to parallel conversion should be synchronized by the director operating system so that the actuator complex receives the analog values computed by each version at the same time instant for each channel. Obviously, there are tradeoffs of software complexity as to whether the local operating system for each version of the director operating system controls the cross check points of the input voter, start of each version, completion of each version, and output voter. To summarize, the director operating system must control the synchronization of the start of A/D and D/A conversion at a minimum.

Figure 7b represents a somewhat simpler, but potentially less fault tolerant architecture due to the single point failure possibility for either voter. In this architecture, the input and output voters are not part of the version but are implemented in the driver operating system. In this architecture, the director operating system:

(1) controls start of A/D conversion,
(2) implements and controls the input voter,
(3) controls start of each version,
(4) implements and controls the output voting, and
(5) controls the transmission to the single D/A.

The architecture in Figure 7c is typical of a system which has a single sensor for measuring some data parameter required by a function. In this case, the driver operating system controls:

(1) the start of A/D conversion and the writing to a memory buffer accessible to each version,
(2) the synchronous start of each version,
(3) implements and controls the output voting, and
(4) controls the transmission to the single D/A.

As can be seen from these examples, implementing an NVS system requires much more than simply developing independent versions of a module, and voting on the output. The driver operating system contains some different procedures for each architecture. The processes of damage assessment, recovery, and fault treatment for each of these architectures will have some similarities but also differences dependent upon the approach taken by the designers.

Recovery Block

The recovery block technique for providing tolerance to software faults in sequential programs is based on the concept of developing multiple independent versions of a software module with a single version processed at a time, followed by subjecting the output of that version to a run time assertion, called an acceptance test. Should the output of the version fail the acceptance test, the system is restored to the state which existed prior to the execution of the version which failed the acceptance test, and the operating system invokes the first alternate version for the function whose primary version failed. The output of the first alternate version is subjected to the same acceptance test used for the primary version. If it passes, the operating system invokes the primary version for the next function to be executed and the process proceeds in a similar manner as depicted in Figure A-8.

```
<recovery block>::=ensure <acceptance test> by
                   <primary alternate>
                   <other alternates> else error

<primary alternate>::=<alternate>
<other alternates>::<empty>|<other alternates>
                   else by <alternate>
<alternate>::=<statement list>
<acceptance test>::=<logical expression>
```

FIGURE A-8. SYNTAX FOR RECOVERY BLOCK [HELLIAR-SMITH]

If the first alternate fails, the second alternate version is scheduled if the remaining time permits it to execute and be subjected to the acceptance test while still maintaining real-time operation. In a real-time environment, missing a hard time deadline for executing a process can lead to system failure, or at least a degradation in performance. In the case of the recovery block, this can happen due to a number of reasons including:

(1) a faulty acceptance test,
(2) exhausting the spare modules with none passing a valid acceptance test, and
(3) run time of the versions and acceptance test exceeding the time frame dictated by the real-time requirement.

If the recovery block fails, recovery is attempted at the next higher level which could be another recovery block in the case of nested recovery blocks.

Error Detection

"The first stage in providing fault tolerance is to detect errors arising from the execution of the primary module" [ANDER81, p. 251]. Assertions can be included in the module itself. This by itself is not sufficient and must be followed by the completion of an acceptance test which executes after the primary module has run. The acceptance test shall raise an exception if the module output does not pass the test.

"The function of the acceptance test is to ensure that the operation performed by the recovery block is to the satisfaction of the program which invoked the block. The

acceptance test is therefore performed by reference to the variables accessible to that program, rather than variables local to the recovery block, since these can have no effect or significance after exit from the block. Indeed the different alternates will probably have different sets of local variables." [RAND75]

"When an acceptance test is being evaluated, any non-local variables that have been modified must be available in their original as well as their modified form because of the possible need to reset the system state. For convenience and increased rigor, the acceptance test is enabled to access such variables either for their modified value or for their original (prior) value." [RAND75]

Damage Assessment

Damage assessment is not currently used in the recovery block technique. In the case of a fault being detected by the acceptance test, one would ideally like to return to the state nearest in time to the present time that allows full regeneration of lost results. To perform such a feat in real-time is very difficult, so no attempt is made to locate that state in current recovery block implementations.

Recovery

Recovery blocks primarily rely on backward recovery. For single sequential processes, the system saves the state at the a priori determined recovery point prior to beginning the execution of the primary version. No damage assessment attempt

is made. The inherent danger in this approach for a system with concurrent processes is the rollback propagation in which rollback of a process may cause other processes to rollback because of process interaction. This rollback propagation can continue until a globally consistent state is reached and could, in the worst case, necessitate a restart [ANDER81, GOLDBERG, SHIN, VELARDI]. Two approaches to dealing with this phenomena are the techniques of "conversations" between processes [RAND75, GOLDBERG] and the recovery cache [LEE, MELLIAR-SMITH].

Fault Treatment

Fault treatment in the recovery block technique relys on one of the alternate versions being correct and passing the acceptance test. In real-time applications which repetitively perform the same functions througout a mission, a version which persistently fails the acceptance test would be removed from the execution schedule and replaced with a spare if one were available. Whether a spare is available or not, the fault log would be retained and the faulty version removed from the execution schedule and subjected to analysis to determine the specific faults that caused it to consistently fail the acceptance test.

REFERENCES

[ALLAN] Allan, Roger, "For Fault-Tolerant Computing, Software is Finding a Powerful Ally in Hardware", Electronic Design, Oct. 31, 1985, pp. 110-117.

[ANDER81] Anderson, T. and Lee, P. A., Fault Tolerance: Principles and Practice, Prentice/Hall International, London, 1981.

[BARIGAZZI] Barigazzi, G. and Strigini, L., Application-Transparent Setting of Recovery Points, IEEE, 1983.

[CHA78] Chen, L. and Avizienis, A., "N-Version programming: a fault-tolerance approach to reliability of software operation", Digest FTSC-8, Toulouse, France, June 1978, pp. 3-9.

[GOLDBERG] Goldberg, Jack, and Levitt, Karl, Architectural and Hardware Implications of Fault Tolerant Software, NAS1-17412, SRI International, 1984.

[HINDIN] Hindin, Harvey J., "Parallel Processing Promises Faster Program Execution", Computer Design, August 15, 1985., pp. 57-66.

[HITT] Hitt, Ellis F., Webb, Jeffrey J., and Bridgman, Michael S., Comparative Analysis of Fault-Tolerant Software Design Techniques, NAS1-17412, Battelle Memorial Institute, Feb. 15, 1984.

[IEEE 729-83] IEEE Standard Glossary of Software Engineering Terminology, IEEE 729, 1983.

[KNIGHT/LEVESON] Knight, John C. and Leveson, Nancy, and St. Jean, Lois D., "A Large Scale Experiment in N-Version Programming", Digest FTSC 15, The University of Michigan, June 1985, pp. 135-139.

[KRISHNA] Krishna, C. M., Shin, Kang G., and Butler, Ricky W., Synchronization and Fault-Masking in Redundant Real-Time Systems, NASA TM-87478, Nov. 1983.

[LEE] Lee, P. A., Ghani, N., and Heron, K., "A Recovery Cache for the PDP-11", Digest FTSC-9, June 1979, pp. 3-8.

[MAKAM] Makam, Srinivas V., Design Study of a Fault-Tolerant Computer System to Execute N-Version Software, Report No. CSD-821222, UCLA Computer Science Department, December 1982.

[MELLIAR-SMITH] Melliar-Smith, Peter Michael, Development of Software Fault-Tolerance Techniques, NASA CR 172122, SRI International, March 1983.

[RAND75] Randell, B., "System Structure for Software Fault Tolerance", IEEE Trans. on Software Engineering", Vol. SE-1, No. 2, June 1975.

[SHIN] Shin, Kang G. and Lee, Yann-Hang, Analysis of Backward Error Recovery for Concurrent Processes with Recovery Blocks, The University of Michigan, 1983.

[VELARDI] Velardi, Paola, and Ciciani, Bruno, "Recovery Blocks for Communicating Systems", Microprocessing and Microprogramming 11 (1983), North Holland, pp. 287-294.

APPENDIX B

SOFTWARE SYSTEM ERROR DETECTION AND CORRECTION TECHNIQUES

SOFTWARE SYSTEM ERROR DETECTION AND CORRECTION TECHNIQUES

by

Ellis F. Hitt

1. INTRODUCTION

1.1 Requirements Definition

The software requirements for the experiments which will provide data from real-time implementation of one or more of the existing fault tolerant software techniques include the functions to be implemented, the test drivers, and the data to be acquired. This document accompanied by the formal software specification to be developed under Subtask 2.2 meet the requirements of Document No. 2 referenced in RTCA/DO-178A [RTCA].

2. Recovery Blocks

2.1 Functions to be Implemented

The components of the system state vector to be estimated are: latitude, longitude, altitude, north and east velocities. The north and east velocities will be used to compute the desired velocity output, the true velocity. The recovery block will be evaluated for implementing these navigation functions which provide the components of the state vector:

(1) Altitude from air data
(2) VOR/DME/air data [BRYSON] yields estimates of latitude, longitude, and north and east velocities.

The coordinate systems to be used are the:
(1) inertial frame
(2) earth fixed
(3) local level or geographic
(4) body
(5) navigation (local horizontal) [HITT85].

2.2 Recovery Block Implementation of Navigation

2.2.1 Sequential Processing

A major frame will be composed of a sequence of minor frames as shown in Figure B-1. At the start of each minor frame, the frame timer is reset to zero. In the minor frame the navigation process is to execute, the inputs will be read from the respective addresses. The recovery point will be established. The inputs will be tested for validity and then be processed by the primary version. The output will be subjected to the acceptance test. If the output passes the acceptance test, it is written to the output addresses and no further processing of the navigation function is required until the next minor frame in which navigation is scheduled. If the output fails the

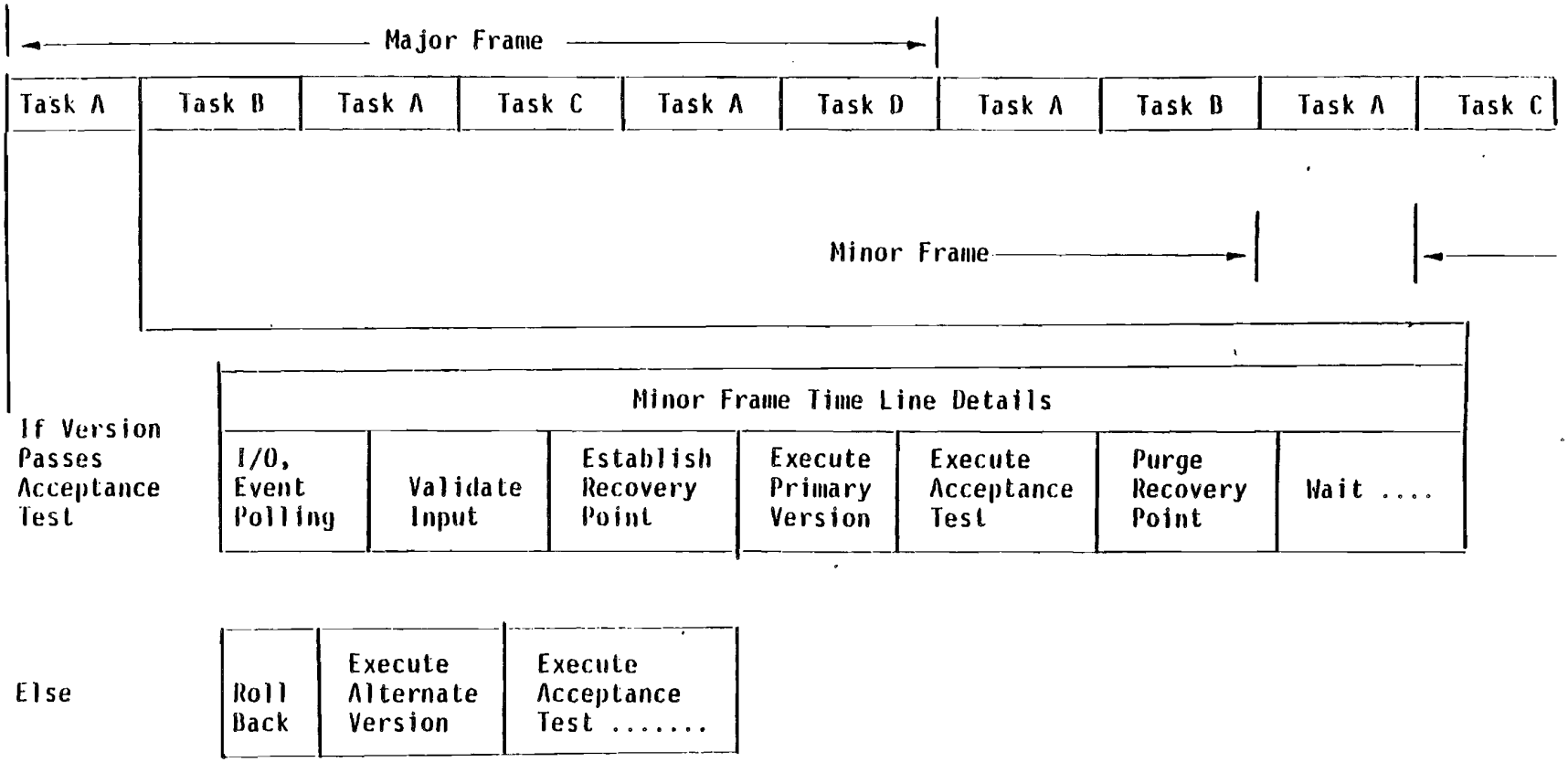


FIGURE B-1. RECOVERY BLOCK TIME LINE

acceptance test, the system rolls back to the recovery point and invokes the first alternate version. This process continues until the output passes the acceptance test or the minor frame timer times out.

2.2.2 Response Time

The minor frame time shall be 25 milliseconds. Forty minor frames shall comprise the major frame. Within each minor frame the recovery block is scheduled to execute, the total time must be allocated to the input processing, establishing the recovery point, executing the primary version, executing the acceptance test, and either purging the recovery point if the output passes the acceptance test, or rolling back to the recovery point and repeating the process using an alternate version. The primary version as well as each alternate version must execute in a given time increment. For the purpose of this project, this time shall be less than 2 milliseconds. The entire process from minor frame timer start to completion of the execution of the first acceptance test shall be less than 5 milliseconds.

2.3 Requirements

The recovery block model of the system shall make use of the states and system model given in BRYSON. A primary and at least one alternate version shall be provided. The acceptance test shall be separately specified and developed.

2.3.1 Sensor Inputs

Sensor inputs shall be assumed to be transmitted over a simulated MIL-STD-1553B data bus. This establishes a 16 bit data word with parity as the last bit. The MSB is transmitted first and parity last.

2.3.1.1 Air Data Inputs

2.3.1.1.1 Data Type

The input variable data type shall be integer as specified in [ASD].

2.3.1.1.2 Input Data Word Formats

The data word format is the structure, order, and value represented by the bits in a signal data transmission. The data word format is as follows for each input variable:

Barometric Altitude
Units: Feet

MSB: 16,384
LSB: 1
Coding: 2's complement, Integer
Max: +32,768
Min: - 1,000

BIT	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TIMES	SYNC		SIGN	MSB	*	*	*	*	*	*	*	*	*	*	*	*	*	*	LSB	PARITY

Transmission Rate: 20 Hz

Indicated Airspeed
Units: Knots

MSB: 4,096
LSB: 2E-3
Coding: BNR
Max: 8,192
Min: 0

```
BIT   1  2  3  4   5  6  7  8  9 10 11 12 13 14 15 16 17 18 19  20
TIMES   SYNC  MSB  *  *  *  *  *  *  *  *  *  *  *  *  *  * LSB PARITY
```

Transmission Rate: 20 Hz

2.3.1.2 VOR/DME Inputs

2.3.1.2.1 Data Type

The input data shall be real.

2.3.1.2.2 Input Data Word Formats

The data word format follows for each input variable:

Magnetic Bearing to VOR MSB: 90
Units: Degrees LSB: 0.04394531
Coding: BNR
Max: +180
Min: -180

```
BIT   1  2  3  4   5  6  7  8  9 10 11 12 13 14 15 16 17 18 19  20
TIMES   SYNC SIGN MSB  *  *  *  *  *  *  *  *  *  * LSB 0  0  0 PARITY
```
Transmission Rate: 20 Hz

Distance to VORTAC MSB: 327.68
Units: Nautical Miles LSB: 0.01
Coding: BNR
Max: 655.36
Min: 0

```
BIT   1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19  20
TIMES   SYNC  MSB  *  *  *  *  *  *  *  *  *  *  *  *  *  * LSB PARITY
```
Transmission Rate: 20 Hz

2.3.2 Processing

The VOR/DME measurement model shall be based on that given in BRYSON.

The recovery block [HITT86] mechanization shall sequentially:

1. Validate input data
2. Establish recovery point
3. Execute subprogram for primary version
4. Execute acceptance test
5. If subprogram passes acceptance test, then purge recovery point and wait until next scheduled minor frame, else rollback to recovery point
6. Execute subprogram for alternate version
7. If subprogram passes acceptance test, then purge recovery point and wait until next scheduled minor frame, else rollback to recovery point
8. If another alternative available, then execute subprogram for alternate version, else fail.

2.3.3 Output

The output of the version which passes the acceptance test shall be converted into the output format which follows for each component of

the state vector.

2.3.3.1 Data Type

The output variable data type shall be as given in the data word coding format.

2.3.3.2 Output Data Word Formats

The data word format follows for each output variable:

Latitude, Present
Units: Degrees
MSB: 45
LSB: 5.36E-6
Coding: BNR
Max: +90 (+ is North)
Min: -90

Word 1

BIT	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TIMES	SYNC		SIGN	MSB	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	PARITY

Word 2

BIT	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TIMES	SYNC			*	*	*	*	*	*	*	*	LSB	0	0	0	0	0	0	0	PARITY

Computation Rate: 20 Hz

Longitude, Present
Units: Degrees
MSB: 90
LSB: 5.36E-6
Coding: BNR
Max: +180 (+ is East)
Min: -180

Word 1

BIT	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TIMES	SYNC		SIGN	MSB	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	PARITY

Word 2

BIT	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TIMES	SYNC			*	*	*	*	*	*	*	*	LSB	0	0	0	0	0	0	0	PARITY

Computation Rate: 20 Hz

Barometric Altitude
Units: Feet
MSB: 16,384
LSB: 1
Coding: 2's complement, Integer
Max: +32,768
Min: - 1,000

BIT	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TIMES	SYNC		SIGN	MSB	*	*	*	*	*	*	*	*	*	*	*	*	*	*	LSB	PARITY

Computation Rate: 20 Hz

True Velocity
Units: Knots
MSB: 4,096
LSB: 2E-3
Coding: BNR
Max: 8,192
Min: 0

BIT	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

TIMES SYNC MSB * * * * * * * * * * * * * * LSB PARITY
Computation Rate: 20 Hz

3. Data to be Acquired

The individual subprograms should be designed to automatically acquire certain data. Software probes will be embedded in the code. Data to be acquired from each subprogram includes execution time, input variables' values, and output variables' values for each version on a stand-alone basis. When the complete recovery block is integrated, the outputs shall include the execution time, input variables' values, and output variables' values for each version which fails the acceptance test.

4. Test Drivers

The test drivers for each subprogram shall be derived from the input specifications for each subprogram. The test drivers for the integrated set of subprograms required to implement the real-time recovery block shall simulate the input data transmission rates as well as the full range of dynamics derived from a simulated aircraft trajectory.

REFERENCES

[ASD] MIL-STD-1553 Multiplex Data Bus Word Formats, ADA 121934, Boeing Military Airplane Company, December 1981.

[BRYSON] Bryson, A.E. Jr., and Bobick, J.C., "Improved Navigation by Combining VOR/DME Information and Air Data", Journal of Aircraft, Vol. 9, No. 6, pp. 420-426, June 1972.

[HITT85] Hitt, Ellis F., McCown, Robert B., Schweikert, Katherine M., and Gallivan, Richard, Integrated Control Mathematical Routines ;for MIL-STD-1750 Built-In-Functions (BIFS), AFWAL-TR-85-3056, Battelle Columbus Laboratories, August 1985.

[HITT86] Hitt, Ellis F., Software Fault Tolerance, Technical Report, Battelle Columbus Division, Jan. 9, 1986.

[RTCA] Software Considerations in Airborne Systems and Equipment Certification, Document No. RTCA/DO-178A, Radio Technical Commission for Aeronautics, March, 1985.

APPENDIX C

NAVIGATION STATION FREQUENCIES

NAVIGATION STATION FREQUENCIES

Table C-1 lists some of the navigation stations in the United States and their corresponding station frequencies. This data is from a VORTAC listing that was compiled around 1970, so there may be old VORTACs that are listed that should be deleted. However, for use in our navigation recovery block, this table satisfactorily illustrates our intentions.

TABLE C-1. NAVIGATION STATION FREQUENCIES [11]

Location	Frequency	Location	Frequency
Nabb, IN.	113.5	Attica, OH.	112.8
Allentown, PA.	117.5	Waterloo, DE.	112.6
Abilene, TX.	113.7	Watertown, SD.	116.6
Albuquerque, NM.	113.2	Augusta, ME.	111.4
Aberdeen, SD.	113.0	Austin, MN.	108.8
Albany, GA.	116.1	Austin, TX.	112.5
Anton Chico, NM.	110.0	Wausau, WI.	111.6
Nantucket, MA.	117.7	Akron, OH.	114.4
Waco, TX.	115.3	Atlantic City, NJ.	108.6
Ardmore, OK.	116.7	Addison, TX.	111.4
Camp Springs, MD.	113.1	Albert Lea, MN.	109.8
Alexandria, LA.	116.1	Pittsburgh, PA.	110.0
Augusta, GA.	113.9	Athens, GA.	109.6
Alliance, NE.	111.8	Bellaire, OH.	117.1
King Salmon, AK.	112.8	Akron, CO.	114.4
Albany, NY.	117.8	Allendale, SC.	116.7
Alice, TX.	114.5	Waterloo, IA.	108.2
Alamosa, CO.	113.9	Walla Walla, WA.	111.8
Amarillo, TX.	117.2	Alma, GA.	115.1
Anniston, AL.	108.8	Anchorage, AK.	114.3
Anderson, SC.	108.6	Annette Island, AK.	117.1
Ainsworth, NE.	112.7	Anthony, KS.	112.9
Lima, OH.	108.4	Altoona, PA.	108.8
Napa, CA.	112.1	Appleton, OH.	116.7
Naperville, IL.	116.0	Alpena, MI.	108.8
Acton, TX.	110.6	Yardley, PA.	108.2
Walnut Ridge, AR.	114.5	Watertown, NY.	109.8
Astoria, OR.	114.0	Atlanta, GA.	115.6

APPENDIX D

NAVIGATION STATION LOCATIONS

NAVIGATION STATION LOCATIONS

Table D-1 gives some of the navigation stations (listed by city and state) and their corresponding latitude and longitude. The table is incomplete, but suffices to illustrate its use in the calculations.

TABLE D-1. NAVIGATION STATION LOCATIONS [11]

City, State	Latitude	Longitude
Nabb, IN.	38: 35: 19.5 N	85: 38: 9.7 W
Allentown, PA.	40: 43: 35.7 N	75: 27: 18.4 W
Abilene, TX.	32: 28: 52.5 N	99: 51: 47.1 W
Albuquerque, NM.	35: 2: 37.4 N	106: 48: 56.6 W
Aberdeen, SD.	45: 25: 2.7 N	98: 22: 6.1 W
Albany, GA.	31: 39: 18.2 N	84: 17: 35.5 W
Anton Chico, NM	35: 6: 41.9 N	105: 2: 21.7 W
Nantucket, MA.	41: 16: 54.2 N	70: 1: 38.0 W
Akron, OH.	41: 6: 28.2 N	81: 12: 6.2 W
Waco, TX.	31: 39: 43.7 N	97: 16: 7.4 W
Atlantic City, NJ.	39: 27: 20.7 N	74: 34: 36.2 W
Ardmore, OK.	34: 12: 41.3 N	97: 10: 4.9 W
Addison, TX.	32: 58: 24.6 N	96: 50: 8.3 W
Camp Springs, MD.	38: 48: 25.6 N	76: 51: 59.4 W
Albert Lea, MN	43: 40: 60.0 N	93: 22: 8.0 W
Alexandria, LA.	31: 15: 23.2 N	92: 30: 2.0 W
Pittsburgh, PA.	40: 16: 42.9 N	80: 2: 27.9 W
Augusta, GA.	33: 32: 39.8 N	82: 7: 59.6 W
Athens, GA.	33: 56: 50.9 N	83: 19: 29.6 W
Alliance, NE.	42: 3: 18.4 N	102: 48: 14.9 W
Bellaire, OH.	40: 1: 1.0 N	80: 49: 2.8 W
King Salmon, AK.	58: 43: 31.3 N	156: 44: 59.9 W
Akron, CO.	40: 9: 20.1 N	103: 10: 45.2 W
Albany, NY.	42: 44: 49.9 N	73: 48: 13.0 W
Allendale, SC.	33: 0: 44.4 N	81: 17: 32.6 W
Alice, TX.	27: 44: 22.2 N	98: 1: 15.5 W
Waterloo, IA.	42: 33: 23.4 N	92: 23: 55.5 W
Alamosa, CO.	37: 20: 56.9 N	105: 48: 48.7 W
Walla Walla, WA.	46: 5: 13.5 N	118: 17: 29.1 W

APPENDIX E

NAVIGATION RECOVERY BLOCK ADA CODE

NAVIGATION RECOVERY BLOCK ADA CODE

The following pages are the source code listing for the navigation recovery block described in this report. The source code is written in Ada.

The decision for the use of a maximum of seven significant figures in all of the constants in this program was governed by the use of the predefined standard Ada real numeric type FLOAT (which is implemented on the VAX using F-floating representation). The F-floating representation has a size of 32 bits and provides six digits of precision. Although the use of the Ada numeric type LONG_FLOAT (implemented using D-floating or G-floating representation) would provide additional digits of precision (D-floating has a size of 64 bits and provides nine digits of precision; G-floating has a size of 64 bits and provides fifteen digits of precision), its use would result in source code that is less portable [12,13].

```
with TEXT_IO, FLOAT_MATH_LIB;

procedure NAVIGATION is

     use TEXT_IO, FLOAT_MATH_LIB;
     package INT_I_O is new INTEGER_IO(INTEGER);
     package FLOAT_I_O is new FLOAT_IO(FLOAT);
     use INT_I_O, FLOAT_I_O;

type TUNED_FREQUENCY is delta 0.1 range 108.0 .. 118.0;
type FREQUENCY is (113.5, 117.5, 113.7, 113.2, 113.0, 116.1);

-- Note that the frequencies listed are given just for this example
-- and would need to be changed for your specific use.

type LOCATION is (STATES, NOTSTATES);

-- The general location of the aircraft is necessary for the range
-- check.  LOCATION is defined as within the 48 conterminous states,
-- STATES, or not, NOTSTATES.

NUMBER_ITERATIONS, NUMBER_TRIALS: INTEGER;

ACCEPTANCE, FREQ_CHECK, NEW_PROB_SIGNAL, PROB_SIGNAL,
TOTAL_PROB_SIGNAL: BOOLEAN;

A, AIRCRAFT_ALTITUDE, AIRCRAFT_LAT, AIRCRAFT_LONG, AIRSPEED_TRUE,
A_ONE, A_THREE, AT_NORTH, AT_WEST, BEAR_TRUE, B_ONE, B_THREE,
C_ONE, C_THREE, DME_RANGE, EFFECTIVE_RANGE, ELEVATION,
ELEVATION_ANGLE, F_ONE, F_TWO, F_THREE, HDG_TRUE, M, N,
NORTH_POSITION, OLD_NORTH_POSITION, OLD_TIME, OLD_WEST_POSITION,
P, PHI_THREE, PITCH, Q, QUOTIENT, RADIUS_EARTH, R_ONE, R_TWO,
R_THREE, S, STATION_ELEVATION, STATION_LAT, STATION_LONG,
THETA_THREE, THETA_FOUR, TIME, TOLERANCE, UNUSABLE_AREA,
VELOCITY_NORTH, VELOCITY_WEST, WEST_POSITION, X, Y, Y_ONE, Y_TWO,
Y_THREE, Y_MAX: FLOAT;

type STATION_CLASS is (TERMINAL, LOW, HIGH);

     function ARCTANGENT(NUMBER_ITERATIONS: in INTEGER;
              X, TOLERANCE, Y_MAX: in FLOAT) return FLOAT is

     begin

          Y_ONE := -Y_MAX;
          Y_TWO := Y_MAX;
```

```
        FIND:

             for J in 1 .. NUMBER_ITERATIONS loop

                  Y_THREE := (Y_ONE + Y_THREE)/2;
                  F_ONE := TAN(Y_ONE) - X;
                  F_TWO := TAN(Y_TWO) - X;
                  F_THREE := TAN(Y_THREE) - X;

                  if (F_ONE < TOLERANCE) then
                       Y := Y_ONE;
                       exit FIND;
                  elsif (F_TWO < TOLERANCE) then
                       Y := Y_TWO;
                       exit FIND;
                  elsif (F_THREE < TOLERANCE) then
                       Y := Y_THREE;
                       exit FIND;
                  end if;

                  if ((F_THREE/F_ONE) < 0.0) then
                       Y_TWO := Y_THREE;
                  else
                       Y_ONE := Y_THREE;
                  end if;

             end loop FIND;

     end;


-- The above function, ARCTANGENT, is used to determine the arc
-- tangent in the secondary alternate.  To "force" explicit
-- differences to exist between the primary and secondary.
-- alternates, the primary alternate uses the Ada math library to
-- determine these values, while the secondary alternate uses the
-- interval halving method.  This enables the determination of a
-- root of f(x) = 0, accurate within a specified tolerance value.


-- The following "begin" starts the main program.


begin

     GET (NUMBER_TRIALS);
```

```
VORDME:

     for I .. NUMBER_TRIALS loop

          GET (TUNED_FREQUENCY);

          -- The frequency check is performed by comparing the
          -- tuned navigation frequency, TUNED_FREQUENCY, to the
          -- stored navigation station data, STATION_FREQUENCY.
          -- If the tuned navigation frequency does not match a
          -- station in the navigation area, then the probability
          -- of a signal, PROB_SIGNAL, is zero.  If the frequency
          -- does match a station, then the probability of a
          -- signal is one.  The total probability of the signal,
          -- TOTAL_PROB_SIGNAL, is calculated after each check is
          -- performed and is dependent upon the individual
          -- probability of signal results.

          FREQ_CHECK := FALSE;
          PROB_SIGNAL := FALSE;
          TOTAL_PROB_SIGNAL := FALSE;
          NEW_PROB_SIGNAL := FALSE;

          for STATION_FREQUENCY in FREQUENCY'FIRST .. FREQUENCY'LAST
              loop

               if TUNED_FREQUENCY = STATION_FREQUENCY then
                    FREQ_CHECK := TRUE;
               end if;

               if FREQ_CHECK = TRUE then
                    PROB_SIGNAL := TRUE;
               end if;

          end loop;

          TOTAL_PROB_SIGNAL := PROB_SIGNAL;

          -- The range check determines whether or not the aircraft
          -- is within the effective range of the tuned navigation
          -- station.  If the aircraft is within the effective
          -- range, then the probability of signal, PROB_SIGNAL, is
          -- one.  Otherwise, the probability of a signal is zero.

          PROB_SIGNAL := FALSE;
```

```
-- The range is obtained from the distance measuring
-- equipment (DME).  The aircraft's altimeter gives the
-- AIRCRAFT_ALTITUDE.

DME_RANGE : = 0;

GET (STATION_CLASS);
GET (LOCATION);
GET (DME_RANGE);
GET (AIRCRAFT_ALTITUDE);

EFFECTIVE_RANGE : = 0;

case STATION_CLASS is
     when TERMINAL = >
          if AIRCRAFT_ALTITUDE < = 12_000 then
               EFFECTIVE_RANGE : = 25;
          end if;
     when LOW = >
          if AIRCRAFT_ALTITUDE < 18_000 then
               EFFECTIVE_RANGE : = 40;
          end if;
     when HIGH
          if (AIRCRAFT_ALTITUDE < 18_000) and (LOCATION
             /= STATES) then
               EFFECTIVE_RANGE : = 40;
          end if;

          if (AIRCRAFT_ALTITUDE > = 14_500 and
             AIRCRAFT_ALTITUDE < = 17_999) and
             (LOCATION = STATES) then
               EFFECTIVE_RANGE : = 100;
          end if;

          if (AIRCRAFT_ALTITUDE > = 18_000) and
             (AIRCRAFT_ALTITUDE < = 45_000) then
               EFFECTIVE_RANGE : = 130;
          end if;

          if (AIRCRAFT_ALTITUDE > 45_000) then
               EFFECTIVE_RANGE : = 100;
          end if;
end case;
```

```
if DME_RANGE < =EFFECTIVE_RANGE then
     PROB_SIGNAL : = TRUE;
else
     PROB_SIGNAL : = FALSE;
end if;

NEW_PROB_SIGNAL : = (TOTAL_PROB_SIGNAL and PROB_SIGNAL);
TOTAL_PROB_SIGNAL := NEW_PROB_SIGNAL;

-- The unusable area check determines if the aircraft is
-- within an unusable area of the tuned navigation
-- station.  This check is performed by looping through
-- all of the specified unusable areas for the tuned
-- station and determining the location of the aircraft
-- with respect to the unusable area.  These values would
-- need to be changed to suit your use.

PROB_SIGNAL : = FALSE;

UNUSABLE_AREA := -1;

case TUNED_FREQUENCY is
     when 113.5 = > UNUSABLE_AREA : = 10;
     when 117.5 = > UNUSABLE_AREA : = 40;
     when 113.7 = > UNUSABLE_AREA : = 100;
     when 113.2 = > UNUSABLE_AREA : = -1;
     when 113.0 = > UNUSABLE_AREA : = -1;
     when 116.1 = > UNUSABLE_AREA : = -1;
end case;

if DME_RANGE < = UNUSABLE_AREA then
     PROB_SIGNAL : = FALSE;
else
     PROB_SIGNAL : = TRUE;
end if;

NEW_PROB_SIGNAL := TOTAL_PROB_SIGNAL and PROB_SIGNAL;
TOTAL_PROB_SIGNAL : = NEW_PROB_SIGNAL;

-- The cone of confusion check determines if the aircraft
-- is in the area above the VOR/DME station and might
-- experience a loss of signal.  As with the unusable
-- area check, sample data has been inserted into the
-- cases for this program.  This data would have to be
```

```
-- changed to suit your individual needs.

PROB_SIGNAL := FALSE;

case TUNED_FREQUENCY is
     when 113.5 => ELEVATION := 89;
     when 117.5 => ELEVATION := 88;
     when 113.7 => ELEVATION := 89;
     when 113.2 => ELEVATION := 87;
     when 113.0 => ELEVATION := 86;
     when 116.1 => ELEVATION := 89;
end case;

GET (ELEVATION_ANGLE);

if ELEVATION_ANGLE > ELEVATION then
     PROB_SIGNAL := FALSE;
else
     PROB_SIGNAL := TRUE;
end if;

NEW_PROB_SIGNAL := TOTAL_PROB_SIGNAL and PROB_SIGNAL;
TOTAL_PROB_SIGNAL := NEW_PROB_SIGNAL;

if TOTAL_PROB_SIGNAL = TRUE then

     -- This part of the primary alternate determines the
     -- aircraft latitude and longitude.

     GET (STATION_LAT);              -- in degrees
     GET (STATION_LONG);             -- in degrees
     GET (BEAR_TRUE);
     GET (STATION_ELEVATION);

     -- The earth's radius is assumed to be 6378.163 km =
     -- 3443.93 n mi = 20_925_732 feet.  (Reference 8.)

     RADIUS_EARTH := 3443.93;

     THETA_THREE := BEAR_TRUE + 90;

     R_ONE := RADIUS_EARTH + STATION_ELEVATION;
     R_TWO := RADIUS_EARTH + BEAR_TRUE;
```

```
M := (R_ONE**2) + (DME_RANGE**2) - (R_TWO**2);
N := 2*R_ONE*DME_RANGE;

-- PHI_THREE is defined as the angle in degrees that
-- the aircraft makes with the i2, j2 plane.

PHI_THREE := ACOSD(M/N) -90;

-- To compute the aircraft latitude.

C_ONE := R_ONE * SIND(STATION_LAT);
C_THREE := DME_RANGE * ((COSD(STATION_LAT) *
     COSD(PHI_THREE) * SIND(THETA_THREE)) +
     (SIND(STATION_LAT) * SIND(PHI_THREE)));

P := (C_ONE + C_THREE)/R_TWO;

AIRCRAFT_LAT := ASIND(P);

-- To compute the aircraft longitude.

B_ONE := R_ONE * COSD(STATION_LAT) * COSD(STATION_LONG);
B_THREE := DME_RANGE * (((-1)*SIND(STATION_LAT) *
     COSD(STATION_LONG) * COSD(PHI_THREE) *
     SIND(THETA_THREE)) - (SIND(STATION_LONG) *
     COSD(PHI_THREE) * COSD(THETA_THREE) +
     (COSD(STATION_LAT) * COSD(STATION_LONG) *
     SIND(PHI_THREE)));

A_ONE := R_ONE * COSD(STATION_LAT) * SIND(STATION_LONG);
A_THREE := DME_RANGE * (((-1)*SIND(STATION_LAT) *
     SIND(STATION_LONG) * COSD(PHI_THREE) *
     SIND(THETA_THREE)) + (COSD(STATION_LAT) *
     COSD(PHI_THREE) * COSD(THETA_THREE)) -
     (COSD(STATION_LAT) * SIND(STATION_LONG) *
     SIND(PHI_THREE)));

Q := A_ONE + A_THREE;
S := B_ONE + B_THREE;

if S>0 then
     AIRCRAFT_LONG := ATAND(Q/S);
elsif S<0 and Q>0 then
     AIRCRAFT_LONG := ATAND(Q/S) + 180;
elsif S<0 and Q<0 then
     AIRCRAFT_LONG := ATAND(Q/S) - 180;
elsif S=0 and Q>0 then
     AIRCRAFT_LONG := 90;
elsif S=0 and Q<0 then
     AIRCRAFT_LONG := -90;
end if;
```

```
-- This part of procedure NAVIGATION for the primary
-- alternate determines the north and west position
-- components of the aircraft.  The range has already
-- been obtained from the range check.  The bearing of
-- the aircraft was obtained in the aircraft latitude
-- and longitude calculations.

WEST_POSITION := DME_RANGE * COSD(PHI_THREE) *
     SIND(THETA_THREE);
NORTH_POSITION := DME_RANGE * COSD(PHI_THREE) *
     COSD(THETA_THREE);

-- This part of the primary alternate determines the
-- aircraft's north and west velocity components.

GET (PITCH);
GET (HDG_TRUE);
GET (AIRSPEED_TRUE);

VELOCITY_NORTH := AIRSPEED_TRUE * COSD(PITCH) *
     COSD(HDG_TRUE);
VELOCITY_WEST := AIRSPEED_TRUE * COSD(PITCH) *
     SIND(HDG_TRUE);

-- This part sets up the conditions initially for the
-- acceptance test to be run.

GET (TIME);

if I = 1 then
     OLD_WEST_POSITION := WEST_POSITION;
     OLD_NORTH_POSITION := NORTH_POSITION;
     OLD_TIME := TIME;
end if;

-- This is the acceptance test.  If the acceptance
-- test fails, then the secondary alternate is run.
-- Otherwise, calculations will continue to be made
-- with the primary alternate, and the secondary
-- alternate will not be used.

if I > 1 then
     AT_WEST := (WEST_POSITION - OLD_WEST_POSITION)/
          (TIME - OLD_TIME);
     AT_NORTH :=(NORTH_POSITION - OLD_NORTH_POSITION)/
          (TIME - OLD_TIME);
```

```
      if (AT_WEST = VELOCITY_WEST) and
         (AT_NORTH = VELOCITY_NORTH) then
           ACCEPTANCE := TRUE;
      else
           ACCEPTANCE := FALSE;
      end if;
end if;

-- If the acceptance test passes, then the results of the
-- calculations for the north and west position and
-- velocity components, along with the aircraft latitude
-- and longitude, will be printed out.  Otherwise, the
-- program will enter the secondary alternate to
-- re-calculate these desired values.

if ACCEPTANCE   TRUE then
      PUT ("THE FOLLOWING VALUES ARE FROM THE PRIMARY
           ALTERNATE");
      NEW_LINE;
      PUT ("AIRCRAFT LATITUDE IS "); PUT (AIRCRAFT_LAT);
      NEW_LINE;
      PUT ("AIRCRAFT LONGITUDE IS "); PUT (AIRCRAFT_LONG);
      NEW_LINE;
      PUT ("WEST POSITION COMPONENT IS ");
      PUT (WEST_POSITION);
      NEW_LINE;
      PUT ("NORTH POSITION COMPONENT IS ");
      PUT (NORTH_POSITION);
      NEW_LINE;
      PUT ("WEST VELOCITY COMPONENT IS ");
      PUT (VELOCITY_WEST);
      NEW_LINE;
      PUT ("NORTH VELOCITY COMPONENT IS ");
      PUT (VELOCITY_NORTH);
      NEW_LINE;

      OLD_WEST_POSITION := WEST_POSITION;
      OLD_NORTH_POSITION := NORTH_POSITION;
      OLD_TIME := TIME;
end if;

-- This is the entrance into the secondary alternate.

if ACCEPTANCE = FALSE then
      -
      -- This part of the secondary alternate determines
      -- the aircraft latitude and longitude.

      RADIUS_EARTH := 3443.93;

      THETA_THREE := BEAR_TRUE + 90;
```

```
R_ONE := RADIUS_EARTH + STATION_ELEVATION;
R_TWO := RADIUS_EARTH + BEAR_TRUE;

M := (R_ONE * R_ONE) + (DME_RANGE * DME_RANGE) -
     (R_TWO * R_TWO);
N := 2 * R_ONE * DME_RANGE;

PHI_THREE := ACOSD(M/N) - 90;

-- In the following calculations, the sine and
-- cosine functions are represented by a power
-- series.

C_ONE := R_ONE * ((STATION_LAT * 0.0174533) -
   ((STATION_LAT * 0.0174533)**3)/6);
C_THREE := DME_RANGE *   (1 - (STATION_LAT *
   0.0174533)**2/2) * (1 - (PHI_THREE *
   0.0174533)**2/2) *  ((THETA_THREE *
   0.0174533) - ((THETA_THREE * 0.0174533)**3)/6) +
   (( STATION_LAT * 0.0174533) -((STATION_LAT *
   0.0174533)**3)/6) *  ((PHI_THREE * 0.0174533) -
   ((PHI_THREE * 0.0174533)**3)/6);

P := (C_ONE + C_THREE)/R_TWO;

AIRCRAFT_LAT := ASIND(P);

B_ONE:= R_ONE * (1 - (STATION_LAT * 0.0174533)**2/2)
   * (1 - (STATION_LONG * 0.0174533)**2/2);
B_THREE := DME_RANGE * (((-1)*(STATION_LAT *
   0.0174533) - ((STATION_LAT * 0.0174533)**3)/6) *
   (1 - (STATION_LONG * 0.0174533)**2/2) *
   (1 - (PHI_THREE * 0.0174533)**2/2) * ((THETA_THREE
   * 0.0174533) - ((THETA_THREE * 0.0174533)**3)/6) -
   ((STATION_LONG * 0.0174533) - ((STATION_LONG *
   0.0174533)**3)/6) * (1 - (PHI_THREE *
   0.0174533)**2/2) * (1 - (THETA_THREE *
   0.0174533)**2/2) + (1 - (STATION_LAT *
   0.0174533)**2/2) * (1 - (STATION_LONG *
   0.0174533)**2/2) * ((PHI_THREE * 0.0174533) -
   ((PHI_THREE * 0.0174533)**3)/6);

A_ONE := R_ONE * (1 - (STATION_LAT *
   0.0174533)**2/2) * ((STATION_LONG * 0.0174533) -
   ((STATION_LONG * 0.0174533)**3)/6);
A_THREE := DME_RANGE * ((-1) * (STATION_LAT *
   0.0174533) - ((STATION_LAT * 0.0174533)**3)/6)
   * ((STATION_LONG * 0.0174533) - ((STATION_LONG
   * 0.0174533)**3)/6) * (1 - (PHI_THREE *
   0.0174533)**2/2) * ((THETA_THREE * 0.0174533) -
   ((THETA_THREE * 0.0174533)**3)/6) +
   (1 - (STATION_LAT * 0.0174533)**2/2) *
   (1 - (PHI_THREE * 0.0174533)**2/2) *
```

```
   (1 - (THETA_THREE * 0.0174533)**2/2) -
   (1 - (STATION_LAT * 0.0174533)**2/2) *
   ((STATION_LONG * 0.0174533) - ((STATION_LONG
   * 0.0174533)**3)/6) * ((PHI_THREE *
   0.0174533) - ((PHI_THREE * 0.0174533)**3)/6);

Q := A_ONE + A_THREE;
S := B_ONE + B_THREE;

QUOTIENT := Q/S;

GET (Y_MAX);
GET (TOLERANCE);
GET (NUMBER_ITERATIONS);

if S > 0 then
     AIRCRAFT_LONG := ARCTANGENT(QUOTIENT) * 57.29578;
elsif S < 0 and Q > 0 then
     AIRCRAFT_LONG := ARCTANGENT(QUOTIENT) *
          57.29578 - 180;
elsif S < 0 and Q > 0 then
     AIRCRAFT_LONG := ARCTANGENT(QUOTIENT) *
          57.29578 - 180;
elsif S = 0 and Q > 0 then
     AIRCRAFT_LONG := 90;
elsif S = 0 and Q < 0 then
     AIRCRAFT_LONG := -90;
end if;

-- This part of the secondary alternate determines
-- the north and west position components of the
-- aircraft.

WEST_POSITION := DME_RANGE * (1 - (PHI_THREE**2)/2) *
     (THETA_THREE - (THETA_THREE**3)/6);

NORTH_POSITION := DME_RANGE * (1 - (PHI_THREE**2)/2) *
     (1 - (THETA_THREE**2)/2);

-- This part of the secondary alternate determines
-- the north and west velocity components of the
-- aircraft.

VELOCITY_NORTH := AIRSPEED_TRUE * (1 - (PITCH**2)/2) *
     (1 - (HDG_TRUE**2)/2);

VELOCITY_WEST := AIRSPEED_TRUE * (1 - (PITCH**2)/2) *
     HDG_TRUE - (HDG_TRUE**3)/6);
```

```
                PUT ("THE.FOLLOWING VALUES ARE FROM THE SECONDARY
                     ALTERNATE");
                NEW_LINE;
                PUT ("AIRCRAFT LATITUDE IS "); PUT (AIRCRAFT_LAT);
                NEW_LINE;
                PUT ("AIRCRAFT LONGITUDE IS ");
                PUT (AIRCRAFT_LONG);
                NEW_LINE;
                PUT ("WEST POSITION COMPONENT IS ");
                PUT (WEST_POSITION);
                NEW_LINE;
                PUT ("NORTH POSITION COMPONENT IS ");
                PUT (NORTH_POSITION);
                NEW_LINE;
                PUT ("WEST VELOCITY COMPONENT IS ");
                PUT (VELOCITY_WEST);
                NEW_LINE;
                PUT ("NORTH VELOCITY COMPONENT IS ");
                PUT (VELOCITY_NORTH);
                NEW_LINE;

                OLD_WEST_POSITION := WEST_POSITION;
                OLD_NORTH_POSITION := NORTH_POSITION;
                OLD_TIME := TIME;

            end if;

        end if;

    end loop VORDME;
```

APPENDIX F

REFERENCES

REFERENCES

1. Randell, B., "System Structure for Software Fault Tolerance", IEEE Transactions on Software Engineering, Volume SE-1, Number 2, June 1975.

2. Hitt, Ellis F., "Software Fault-Tolerance", Battelle Columbus Division, Columbus, Ohio, January 9, 1986.

3. Hitt, Ellis, "Software System Error Detection and Correction Techniques", Battelle Columbus Laboratories, Columbus, Ohio, March 5, 1986.

4. Kluse, Michael and Rea, Fred G., Dynamic Navigation Signal Simulation (DNSS) Program Development Specifications, DCSF-SD-2060, Battelle Columbus Laboratories, Columbus, Ohio, May 20, 1980.

5. AIRBORNE VOR RECEIVER, Aeronautical Radio, Inc., Annapolis, Maryland, ARINC Characteristic No. 579-1, February 5, 1971, p. 6.

6. MARK 5 AIRBORNE DISTANCE MEASURING EQUIPMENT, Aeronautical Radio, Inc., Annapolis, Maryland, ARINC Characteristic No. 709-3, January 19, 1981, pp. 6 and 40.

7. Gerald, Curtis F., Applied Numerical Analysis, Addison-Wesley Printing Company, Reading, Massachusetts, 1978, pp. 8-14.

8. Kayton, Myron and Fried, Walter R., Avionics Navigation Systems, John Wiley & Sons, Inc., New York, 1969, pp. 17 and 18.

9. Webster's Ninth New Collegiate Dictionary, Merriam-Webster, Inc., Springfield, Massachusetts, 1985, p. 789.

10. Hitt, Ellis F., McCown, Robert B., Schweikert, Katherine M., and Gallivan, Richard, Integrated Control Mathematical Routines for MIL-STD-1750 Built-In Functions (BIFs), AFWAL-TR-85-3056, Battelle Columbus Laboratories, Columbus, Ohio, August 20, 1985.

11. "Radio and Navigation Aid Data", Battelle Columbus Laboratories, November 7, 1972.

12. VAX™ Ada® Language Reference Manual, AA-EG29A-TE, Digital Equipment Corporation, Maynard, Massachusetts, February 1985, p. 3-27.

13. Booch, Grady, Software Engineering with Ada®, The Benjamin/Cummings Publishing Company, Inc., Reading, Massachusetts, 1983, p. 89.

™ VAX is a trademark of Digital Equipment corporation.

® Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.