DAA/Ames

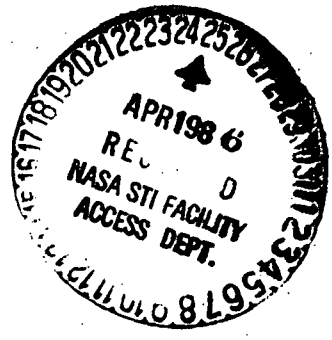
Center for Reliable Computing

EXECUTABLE ASSERTIONS AND FLIGHT SOFTWARE

Aamer Mahmood, Dorothy M. Andrews, and Edward J. McCluskey

CRC Technical Report No. 84-16
(CSL TR No. 84-258)

November 1984



CENTER FOR RELIABLE COMPUTING
Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

Imprimatur: Mario L. Cortes and Aydin Ersoz

This work was supported in part by the NASA-AMES under contract No. NAG 2-246.

EXECUTABLE ASSERTIONS AND FLIGHT SOFTWARE

Aamer Mahmood, Dorothy M. Andrews, and E. J. McCluskey

CRC Technical Report No. 84-16
(CSL TR No. 84-258)

November 1984

Center for Reliable Computing
Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305 USA

ABSTRACT

Executable assertions can be used to test flight control software. However, the techniques used for testing flight software are different from the techniques used to test other kinds of software. This is because of the redundant nature of flight software. An experimental setup for testing flight software using executable assertions is described. Techniques for writing and using executable assertions to test flight software are presented. The error detection capability of assertions is studied and many examples of assertions are given. The issues of placement and complexity of assertions as well as the language features to support efficient use of assertions are also discussed.

KEYWORDS: Executable assertions, software testing, flight software, digital flight control system.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

The complete software testing process involves generation of test data, determination of expected behaviour, program execution, observation of behaviour, and comparison of observed behaviour with the expected behaviour. The expected behaviour is usually determined by hand calculations, simulation, or by alternate solutions to the same problem. The test data can be generated either randomly, exhaustively, or by using some kind of functional or structural analysis. Software testing techniques can either be static (peer review, walkthrough, flow analysis, symbolic execution) or dynamic (including the use of monitors or counters). [Adrion 82] and [Ramamoorthy 75] contain very good surveys of software testing and automated testing tools, respectively.

Executable assertions can be used for dynamic testing of software. An executable assertion is a logical statement about the program variables or a block of code, such that, if there is no error during execution, the assertion statement results in a true value. Assertions not only serve as a good medium for documentation, but they are also useful for testing purposes throughout the lifecycle of software. They can be used for validation during the design phase and for exception handling and error detection during the operation phase.

Assertions can be written by making use of either the specifications or some property of the problem or algorithm. Assertions are usually based either on the inverse of the problem, the range of variables, or

the relationship between variables. Some examples of assertions from [Hecht 76] [Mahmood 83] are as follows:

(1) If the problem is to find the discrete Fourier transform of an N point input sequence x(j), then Parseval's relationship can be used as an assertion

$$\sum |x(j)|^2 = \frac{1}{N} \sum |X(k)|^2 \qquad j \text{ , } k = 0 \text{ to } N-1$$

where X(k) is the discrete Fourier transform.

(2) If the problem is to find eigenvalues of a NxN matrix then the following must be true

$$\sum A_{ii} = \sum L_i \qquad i = 1 \text{ to } N$$

where A_{ii} are the diagonal elements and L_i are the eigenvalues.

(3) The longitude calculation by a routine in flight control software can be checked by

$$\text{New\_Long} \geqslant \text{Prev\_Long} + (\text{Prev\_Long} - \text{Next\_Prev\_Long}) - K$$

and

$$\text{New\_Long} \leqslant \text{Prev\_Long} + (\text{Prev\_Long} - \text{Next\_Prev\_Long}) + K$$

where K represents the threshold for the test.

Assertions have been used in program verification [Floyd 67] [Hoare 69] [Manna 69] [Luckham 75] [King 76], in program testing [Stucki 75] [Andrews 81], and for reasonableness checks in the recovery block scheme of software fault tolerance [Horning 74] [Randell 75] [Carter 79]. The use of executable assertions for detecting hardware and software faults has also been suggested in [Saib 77] [Andrews 78] [Andrews 79]. [Leveson 83] describes the use of assertions for increasing the safety of systems. The objective of this paper is to study the use of executable assertions for testing flight software. The error detection capability of assertions has also been studied in [Glass 80] [Andrews 81]. However, the software used in those studies was different. Also, this study of assertions has a different emphasis, covering all aspects from writing of assertions to use of assertions. The paper is organized as follows: (a) the digital flight control system used in the experiments is discussed in Section 2, (b) Section 3 describes the experimental setup used to write assertions and test flight software, (c) writing of assertions and testing of flight software is explained in Section 4, and (d) Section 5 discusses some of the language features which would make understanding and writing assertions easier.

2 DIGITAL FLIGHT CONTROL SYSTEM

The software analyzed in this experimental study is a part of a digital flight control system, which is an integrated system that

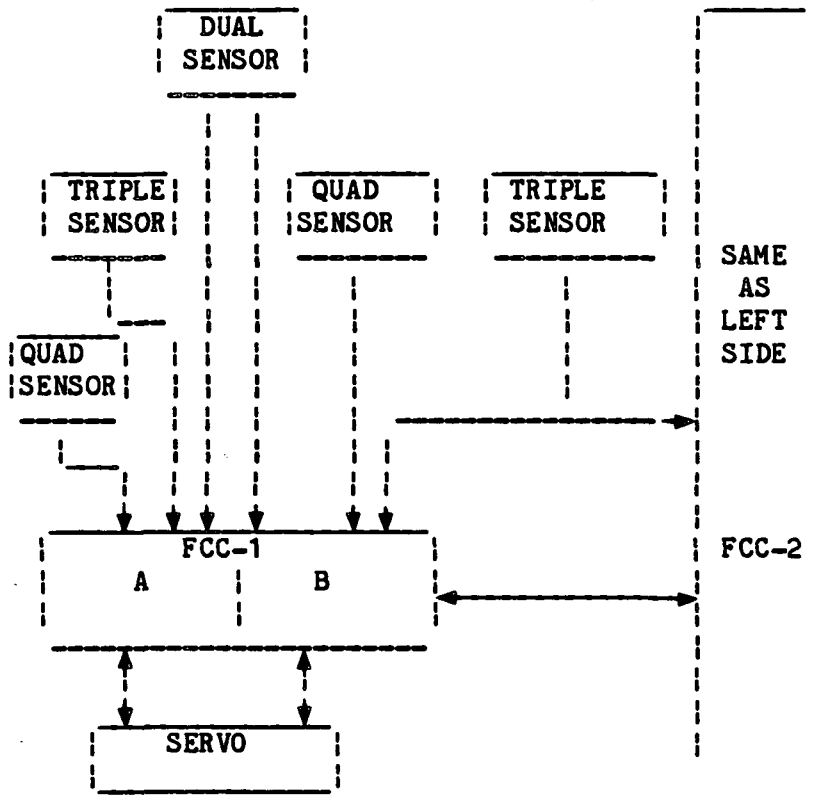


Fig. 1 Dual-Dual Architecture

provides autopilot and flight director modes of operation for automatic and manual control of a commercial airplane during all phases of flight [DFCR-96 80] [Bendixen 83]. It includes two identical flight control computers known as FCC-201; each FCC-201 includes two CAPS-6 (Collins Adaptive Processing System) processors, referred to as Channels A and B. Figure 1 shows the architecture of the dual-dual redundant system containing two FCC-201 computers, and Fig. 2 gives the organization of each FCC-201 computer.

The flight control software is written in AED (Automated Engineer Design), an ALGOL like language. From a functional point of view it consists of five major parts: (a) control and navigation, (b) logic, (c) testing and voting, (d) input/output, and (e) executive. The executive software can be divided into two major groups, foreground and background. The foreground tasks consist of time critical functions such as command generation and executive monitoring. The background programs perform non-time-critical operations like processor self-test and memory checksum. Figure 3 describes the foreground software structure and the timing relationship. The software consists of one segment performing pitch rate inner loop calculations at a rate of 60 per second. After every third execution of the 60 per second segment, the multipath software segment is restarted. This means that the multipath segment is executed 20 times per second. The multipath software segment contains segments which are executed at three different rates: 20, 10, and 5 times per second. At the end of each foreground

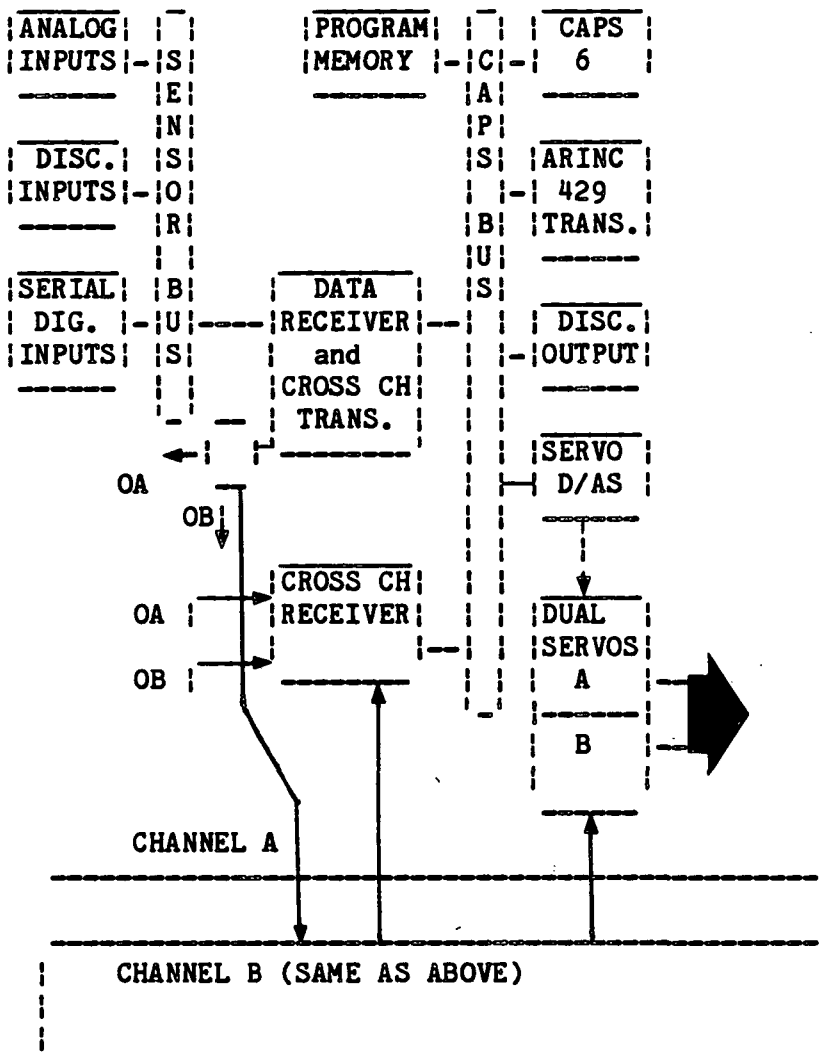


Fig. 2 FCC-201 Architecture

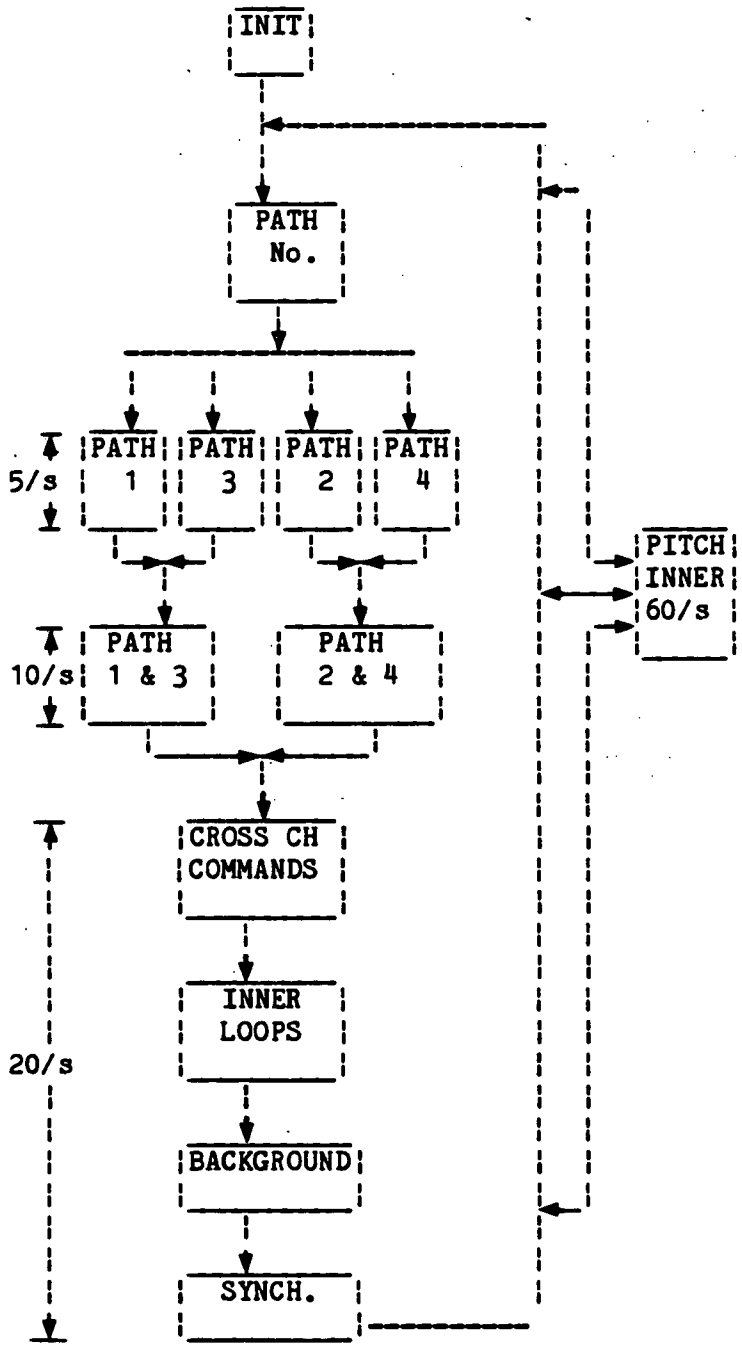


Fig. 3 Software Timing

execution, the executive schedules the background process. Synchronization between the two channels is performed 20 times per second. The software programs of the two channels are not identical, but there is some overlap. Functions performed by each of the two channels are shown in Table 1.

3 EXPERIMENTAL SETUP

The experimental setup of the flight simulator at NASA-AMES Research Center is shown in Fig. 4. More details can be found in [Defeo 82]. A PDP-11/60 is used to modify the flight software (insert assertions and errors), under the UNIX operating system. The flight software is compiled at a different location and the compiled code is transferred to the PDP-11/60 via a modem link. The executable code is then transferred to the flight computers. The PDP-11/60 is then used to simulate the airplane in real-time under the RSX operating system. Some important parts of the experimental setup are as follows:

(a) CAPS TEST ADAPTER (CTA): Each CTA is dedicated to one processor and allows the operator access to the associated CAPS transfer bus directly from its front panel control or from the HP terminal. Some of the capabilities provided by the CTAs are: (1) Display of transfer bus address and data, (2) examine and modify any bus-addressable location, (3) monitor the contents of a selected address, etc.

Table 1 Flight Software Functions

CHANNELS A and B	
1 PITCH AUTOLAND	
2 ROLL AUTOLAND	
3 YAW AUTOLAND	
4 TOGA	
5 ENGAGE LOGIC	
6 SERVO MONITORING	
7 SYNCHRONIZATION	
8 INSTRUMENTATION	
9 ANUNCIATION	
10 YAW SAS	
11 INNER LOOPS	
CH. A	**CH. B**
1 ROLL OUTER	1 PITCH OUTER
2 ALT ALERT	2 AUTOTHROTTLE
3 MODE LOGIC	
4 GLARESHIELD INTERFACE	
5 SENSOR COMPARISON	

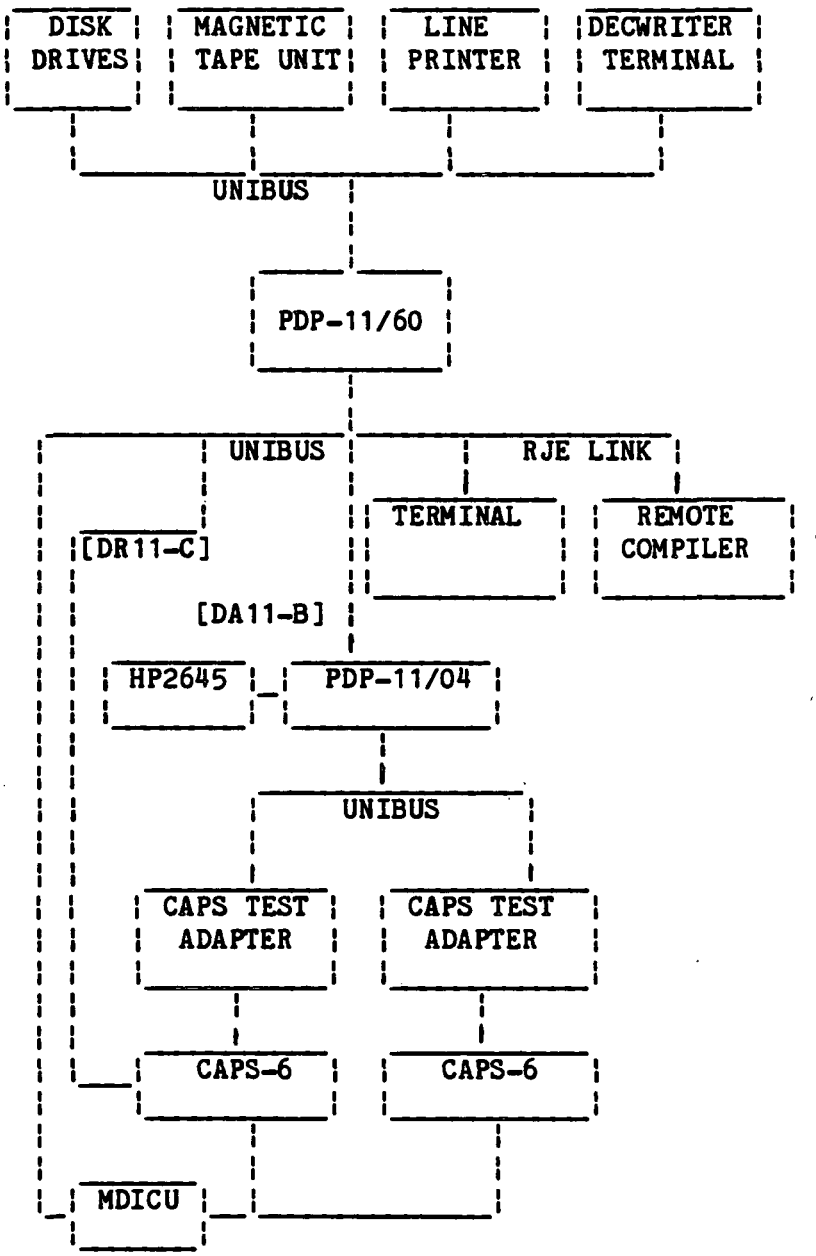


Fig. 4 Experimental Setup

(b) MODULAR DIGITAL INTERFACE CONTROL UNIT (MDICU): It is a CAPS-6 based data distributor whose primary function is the control of the flow and format of the simulated aircraft parameters generated by the PDP-11/60 and of the control commands generated from the flight computers. This function enables the closed loop operation. The HP-2645 terminal provides direct operator control over the operation of the MDICU.

(c) PDP-11/04: The PDP-11/04 is used as an interface between the PDP-11/60 or the HP-2645 and the FCC. The PDP-11/04 combined with the HP-2645 can duplicate all the functions of the CTA. It can also be used for uploading and downloading blocks of FCC memory into internal devices and the PDP-11/60.

(d) PDP-11/60: It is the central element of the experimental setup. It supports two distinct environments: A code-developing (static) environment and a dynamic environment where the flight software can be exercised in closed loop real-time. In the dynamic environment the PDP-11/60 holds the aircraft model. The flight data is transmitted from the PDP-11/60 to the MDICU, which converts the data so that the flight computers can use them. The flight computers compute control surface commands which are fed back to the flight equations.

4 TESTING FLIGHT SOFTWARE

The flight software was tested in the heading select mode (change of direction) at constant speed and constant altitude. Initial testing was done using the setup shown in Fig. 4 at NASA-AMES Flight Software Verification Laboratory [DeFeo 82]. The simulations for the second phase of testing, as described in Sec. 4.2, were performed on Stanford's DECSYSTEM-20.

Ideally the assertions should be written from the specifications. However, since no specifications were available, extensive simulations were performed to understand the software. The purpose was not only to study what the programmers have written but also to find out why they have written it. In order to limit the complexity of the problem, only the portion of flight software which is responsible for changing the heading (direction) of the plane was studied.

The heading is changed by rolling the plane. As long as the bank (roll) angle is greater than zero, the plane continues to turn. The banking (roll) of the plane itself is controlled by the ailerons on the wings. The ailerons must be opened for the specified amount of time to achieve the required bank angle. The longer the ailerons are kept open, the larger the bank angle will become. The larger the bank angle, the faster the plane turns. For correct and safe turning of the plane, the plane must be banked to the correct angle by opening the ailerons for a specified amount of time. When the heading error (difference of where

the plane is and where it should go) falls below a fixed value, the straightening of the plane should begin by again opening the ailerons for a specified amount of time.

The timing relationship between the relevant procedures and the data flow from the input (selected heading) to the output (commands to the ailerons) is shown in Fig. 5. A brief description of each of the modules is as follows:

(1) A_LAT_COM: This module computes heading and airspeed gain (KTAS) for use by the HDG_SEL module.

(2) HDG_SEL: This module performs the heading select computations using selected heading, true heading and yaw rate. It generates a roll-attitude command (LAT_LIM_CMD) which is passed to the LAT_LIMITER module. As long as the heading error is greater than a fixed value, the LAT_LIM_CMD remains constant at 0.5. When the heading error becomes less than the fixed value, the LAT_LIM_CMD becomes proportional to it.

(3) LAT_LIMITER: This module performs magnitude and rate limiting (where the limits depend on the airspeed) of the roll-attitude command from the HDG_SEL module and generates LAT_CPL_CMD which is passed to the A_LAT_COUPL module. The LAT_CPL_CMD increases at a fixed rate to a fixed value. The rate of change and the maximum value is determined by the airspeed. Consider the following two lines of code taken from this module:

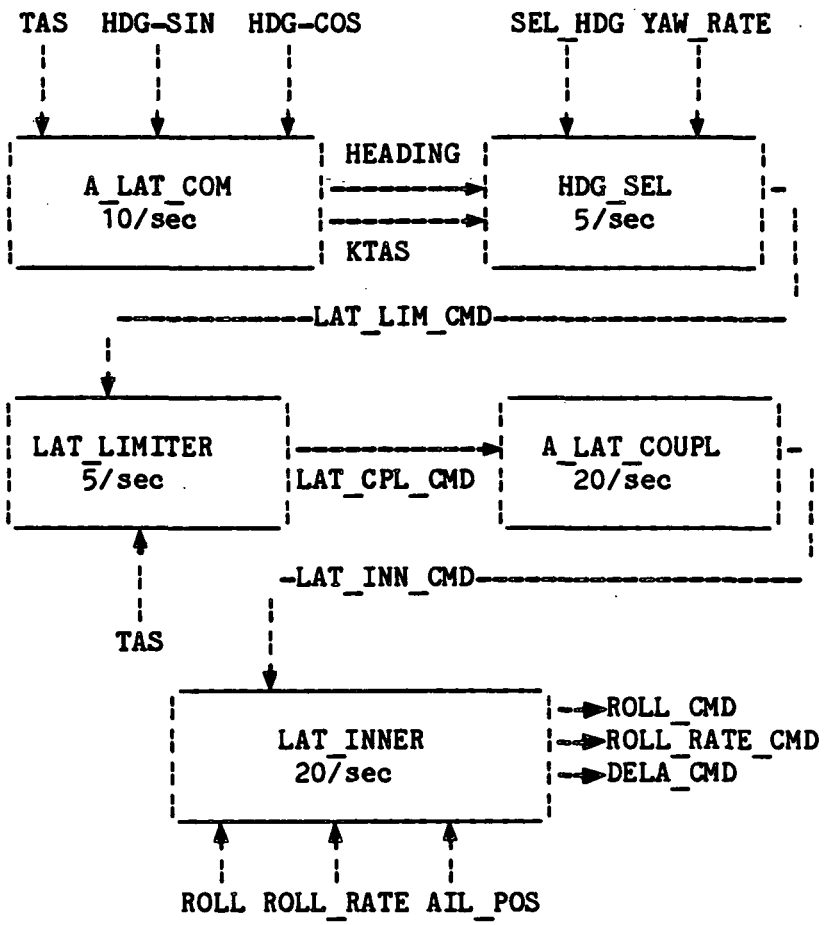


Fig. 5 Data Flow

RATE_LIMIT = 0.006667 * KRTAS;

MAG_LIMIT = 0.203067 * KRTAS;

Then MAG_LIMIT/RATE_LIMIT = 30.45. As the module is executed 5 times a second, this means that the LAT_CPL_CMD will reach its maximum value in about 6 seconds irrespective of the airspeed. (This is an example of the case where it is important to know the intent of the programmer and not just the code). When LAT_LIM_CMD decreases below a fixed value, the LAT_CPL_CMD becomes proportional to it.

(4) A_LAT_COUPL: This module performs coupling between the outer loop modules and the LAT_INNER module. It generates LAT_INN_CMD which is passed to the LAT_INNER module. The LAT_INN_CMD is just the filtered version of the LAT_CPL_CMD.

(5) LAT_INNER: This module performs the inner loop computations for the lateral axis. It includes roll attitude and rate feedback, lead-lag compensation, command limiting, aileron limit override logic, etc. The output generated by this module includes ROLL_CMD, ROLL_RATE_CMD, and DELA_CMD (command to the ailerons). For correct and safe turning of the plane, the DELA_CMD should achieve its maximum value between 3-6 seconds and should return to a mean value of about zero after 9 seconds.

Waveforms of some of the important variables are shown in Fig. 6.

4.1 TESTING - PHASE ONE

Initially the assertions were inserted only in the LAT_INNER module. Table 2 contains examples of some of those assertions. Table 3 shows a

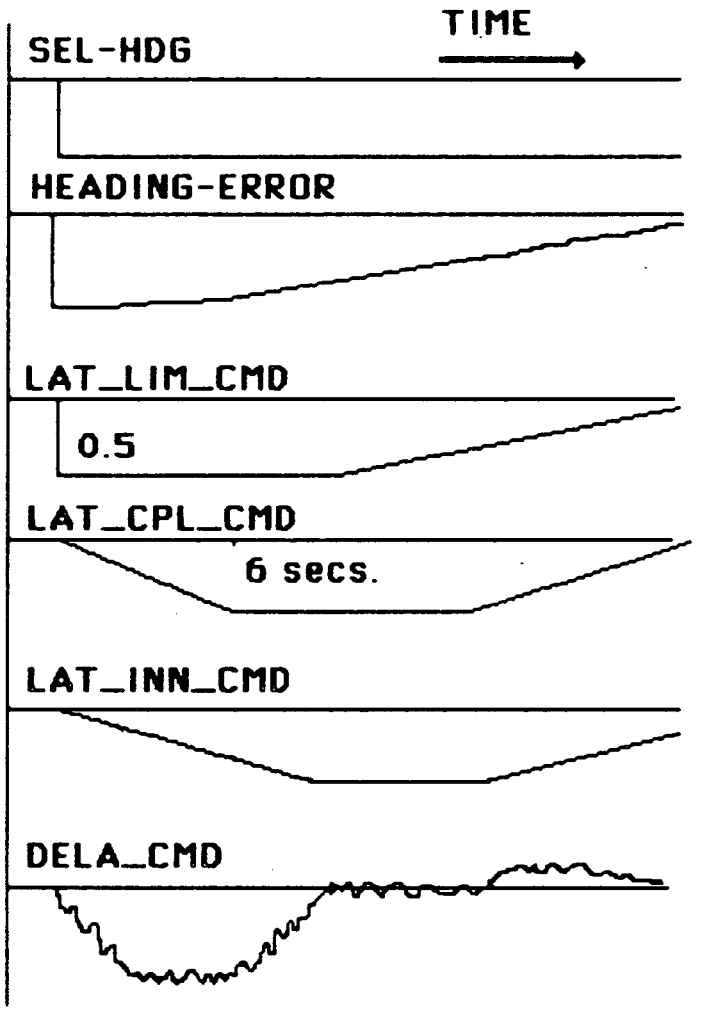


Fig. 6 Waveforms of Some Variables

Table 2 Initial Assertions

(a) ABS(LAT_LIM_CMD) $\leqslant$ 0.5
(b) ABS(LAT_CPL_CMD) $\leqslant$ 0.11
(c) ABS(LAT_CPL_CHG) $\leqslant$ 0.0034
(d) ABS(LAT_INN_CMD) $\leqslant$ 0.18333
(e) ABS(ROLL) $\leqslant$ 0.165
(f) ABS(HDG_CHG) $\leqslant$ 0.0046
(g) TIME TO CHANGE HEADING $\leqslant$ MAXIMUM TIME
(h) HEADING ERROR DECREASES MONOTONICALLY

Table 3 A Program Segment with an Assertion

```
Define Procedure LAT.INNER to be

   begin
   .
   .
   if R.TEST.COMPL
   then  begin
      RL8 = RL8.D = DLIMIT(RL8.D + RL11.D + RL11.D.S, 0.258);
      RL11.D.S = RL11.D;
      RL13 = LIMIT(RL7 + RL8, 0.171429)/ 0.203333;
      end
   else  RL13 = TEST.CMD (RAM.PTR (R.TEST.PTR));
   .
   .
   DELA.CMD = RL13;

COMMENT  ASSERT  ABS(DELA.CMD) < 0.13;
   .
   .
   end;
```

segment of the LAT_INNER module with an assertion inserted in it. The program is processed by a preprocessor which converts all the inserted assertions in compiler recognizable code. It was found that only 25 % of the errors inserted in the software were detected by these initial assertions. The two main reasons for the low detection rate were the inadequacy of the first set of assertions used and the nature of flight control software.

The flight control system is very redundant in hardware and software. Examples of the hardware redundancy are replication and hardware limiters. The software redundancy comes from the software limiters and from voting on the input and output. This redundancy tends to mask errors. As an example, consider the variable LAT_LIM_CMD (output of HDG_SEL module). Its value is limited to 0.5, as long as the heading error is greater than a fixed value. This makes the output independent of the input conditions. Similarily, LAT_CPL_CMD (generated by the LAT_LIMITER module) increases at a fixed rate to a fixed value. This makes the LAT_CPL_CMD independent of the input changes. Another aspect of flight software which makes it different from the other software is that it contains a great number of boolean variables and decision points. This makes it difficult to write the same kind of assertions as were written for the other kind of more computational intensive software. For such a computational intensive code it is easy to use range assertions. This is not true for the flight software.

The inadequacy of the assertions used was the other reason for low error detection. Ideally the assertions should be written during the design phase from specifications. The lack of any specification document made it very difficult to write good and meaningful assertions. Some of the main flaws in the assertions used were as follows:

(a) The assertions were only placed in the last module (LAT_INNER). It is very difficult to write such global assertions which can take every possible condition into consideration. The complexity of assertions starts to approach the complexity of the program itself. One solution is to use many simple assertions at various points in the program. Placement of assertions is very important for good error coverage. This has also been discussed in [Milli 81] and is confirmed by the present study.

(b) Most assertions were based on worst case conditions. However, many errors did not cause the worst case conditions to be exceeded.

(c) Some of the assertions only checked the maximum value. However, in the case of some errors, the maximum value achieved by the variables during a certain time frame was much less than the correct value. It was not possible to check for the minimum value because the correct minimum value of variables is zero most of the time. One solution is to make time a parameter of assertions. Then the values of a variable can be sampled at particular times and checked to be within a maximum and minimum range.

4.2 TESTING - PHASE TWO

In order to improve the turnaround time, the relevant portions of the software were rewritten in PASCAL and the simulation was done on Stanford's DECSYSTEM-20. The assertions were written by using the information about the range or the state of variables at different points in the program and by making use of the inverse relationships. Most of the variables used in the assertions were either the output of modules or the input from sensors. Assertions were placed at the output of modules and before limiters and filters implemented in the software. Some examples of assertions inserted in each of the modules are as follows:

(1) HDG_SEL:

(a) IF ABS(hdg_error * tas) $\geqslant$ 0.02442 then ABS(lat_lim_cmd) = 0.5.

(2) LAT_LIMITER:

(a) Time for lat_cpl_cmd to reach maximum lies between 5.5 and 6.5 seconds.

(b) IF ABS(lat_cpl_cmd) is decreasing then

(i) ABS(hdg_error) $\leqslant$ Constant.

(ii) ABS(0.055556 * lat_cpl_cmd) = (0.155556 - 0.2222 * krtas) * (sel_hdg - 0.736667*yaw_rate - hdg).

(3) A_LAT_COUPL:

(a) Maximum value of lat_inn_cmd $\leqslant$ maximum possible value of lat_cpl_cmd.

(b) Time for lat_inn_cmd to reach maximum lies between 6 and 9 seconds.

(c) lat_inn_cmd, lat_cpl_cmd, and lat_lim_cmd must all be reset to zero.

(4) LAT_INNER:

(a) lat_inn_cmd = 0.5 * (r15+0.764*roll+0.1525*roll_rate).

(b) ABS(r17) $\leqslant$ 0.032.

(c) Time for DELA_CMD to reach maximum lies between 2.5 and 6 seconds.

(d) ABS(dela_cmd) $\leqslant$ 0.13.

16

The above assertions can be divided into three main classes, range assertions, inverse assertions and state assertions. Examples of assertions based on the range of variables are as follows:

(a) ABS(LAT_CPL_CMD) $\leqslant$ 0.11.

(b) MAX. VALUE OF LAT_INN_CMD $\leqslant$ MAX. POSSIBLE VALUE OF LAT_CPL_CMD.

(c) ABS(ROLL) $\leqslant$ 0.165.

(d) ABS(HEADING CHANGE) $\leqslant$ 0.0046/sec.

(e) ABS(DELA_CMD) $\leqslant$ 0.13.

Examples of assertions based on the inverse relationships are as follows:

IF ABS(LAT_CPL_CMD) IS DECREASING THEN

(i) ABS(HEADING-ERROR) $\leqslant$ CONSTANT.

(ii) ABS(0.055555 * LAT_CPL_CMD) = (0.155556 - 0.2222 * KRTAS) * (SEL_HDG-0.736667*YAW_RATE-HDG).

These assertions are based on the fact that the LAT_CPL_CMD decreases only when the LAT_LIM_CMD (and hence the heading error) has decreased to a specified value. At that time the LAT_CPL_CMD becomes proportional to the LAT_LIM_CMD which is itself proportional to the heading error given by sel_hdg-0.736667*yaw_rate-hdg.

Assertions in flight software which check the state of variables are based on the observation that the values of variables can be divided into three distinct regions. The first region is where the value of a variable is increasing, the second is where the value becomes constant, and the third is where a variable returns to its initial value. This can also be seen in Fig. 6. For such variables the following conditions can be checked: (a) rate of increase of variables, (b) maximum value attained by the variable, (c) time to reach maximum value, (d) time when the variable starts returning to its initial value, and (e) rate of change when the variable is returning to its initial value. Examples of assertions which check the state of variables are as follows:

(1) TIME FOR LAT_CPL_CMD TO REACH MAXIMUM LIES BETWEEN 5.5 AND 6.5 SECONDS.

(2) TIME FOR LAT_INN_CMD TO REACH MAXIMUM LIES BETWEEN 6 AND 9 SECONDS.

(3) LAT_INN_CMD, LAT_CPL_CMD, AND LAT_LIM_CMD MUST ALL BE RESET TO THEIR INITIAL VALUE.

(4) TIME FOR DELA_CMD TO REACH MAXIMUM LIES BETWEEN 2.5 AND 6 SECONDS.

The software was seeded with errors, one at a time, and executed to see how many of the seeded errors cause assertion violations. Error types and frequencies were similar to those in the NASA-AMES data base of errors. The insertion of errors was done independently from the writing of assertions. The results of the experiment are given in Table 4. Currently, the software is only partially asserted, that is, the current assertions only check for the errors in the software which is executed during the heading select mode. It can be seen that 66 % of all the errors inserted in the partially asserted software were detected. Some of the reasons for undetected errors are as follows:

(a) The default value assigned to the variables by the compiler was the same as the initial value of the variables. So the error caused by deleting the initialization statement was not detected.

(b) In the case of some boolean statements, the final result was independent of the value of some variables. Any error in the value of those variables could not have been detected.

(c) Some of the errors were in a section of code which was not executed during this phase of testing.

(d) Some errors changed the name of one boolean variable into another. However, since the value of both variables was the same, the error was not detected.

(e) Some errors simulated the condition of a multiple sensor failure. Such errors could not have been detected.

Table 4 Preliminary Experimental Results

ERROR TYPE	ERRORS INSERTED	% ERRORS DETECTED	
		PARTIALLY ASSERTED	FULLY ASSERTED
DATA HANDLING	22	63.6	90.9
LOGIC	19	47.3	84.2
DATABASE	19	78.9	94.7
COMPUTATIONAL	21	76.1	80.9
TOTAL	81	66.6	87.6

The error coverage can be increased to more than 87 % by fully asserting the software, that is, by writing assertions for all of the flight modes. The current assertions only check for the errors in the software which is executed during the heading select mode. Errors in the software which have no effect on the results are redundant. However, these errors would be caught by a different set of assertions, written specifically to check that particular flight mode. Currently, assertions are being written for two other modes: altitude select mode and autoland mode. It is believed that the use of these assertions would increase the error coverage to more than 87 %. More extensive assertion testing of flight software will provide more definitive results.

5 LANGUAGE FEATURES

Currently assertions can only be a single logical statement. This is very restrictive. Consider the following assertion:

```
IF ABS(HEADING-ERROR*TAS) > 0.024 THEN ABS(LAT_LIM_CMD) = 0.5
```

Using the current format the above assertion would be written as

```
ASSERT ((ABS(HEADING-ERROR*TAS) > 0.024) AND (ABS(LAT_LIM_CMD)=0.5)) OR
       (ABS(HEADING-ERROR*TAS) ≤ 0.024)
```

This restriction makes it difficult to write and understand assertions. Usually assertions require extra code to be inserted. It must be possible to write assertions which consist of procedures,

functions, and a sequence of statements. The presence of the following features in programming languages greatly facilitate the use of executable assertions:

(1) Provisions to conditionally execute an assertion, that is the assertion is only executed if a certain condition holds.

(2) Being able to use functions, procedures, or sequence of statements in assertions.

(3) Being able to refer to previous values of variables.

(4) Provisions for specifying the range (max., min.) of variables.

(5) Being able to check the initial and final value of variables.

(6) Provisions for conditionally compiling assertions.

The use of executable assertions has been supported in the past by either developing new languages like EUCLID [Popek 77] or by using a preprocessor for recognizing the assertions and converting them into compiler recognizable code [Stucki 75]. Many languages have been extended to support the use of executable assertions. Some of the above mentioned features have been included in these languages. [Chow 76] and [Taylor 80] discuss in detail some of the features needed to facilitate the use of executable assertions. [Krieg-Bruckner 80] [Luckham 84] describe the extension of ADA to support specifications and assertions. Their ANNA (Annotated ADA) is the most recent language which supports assertions. It contains many of the above mentioned features, which make the use of assertions very easy.

6 SUMMARY

Executable assertions can be used for detecting errors throughout the lifecycle of software. They can be written using the information provided in the specifications. Sometimes the writing of assertions is not easy, but it can help increase the reliability of software. The use of assertions forces programmers to explicitly write their assumptions and goals, thereby not only providing good documentation but also increasing their own understanding of the problem. Techniques for writing and using assertions to test flight software were presented. Language features to support efficient use of assertions were also discussed. Many examples of assertions that check the inverse relationships, range of variables, rate of change of variables, and time spent by variables in different states were given. The experimental setup for testing flight software was described. Preliminary experimental results show that assertions can detect more than 66 % of the errors. The error coverage can be increased to more than 87 % by using a different set of assertions for different flight modes. This also reduces the complexity of individual assertions. In order to get high error coverage it is important to place assertions intelligently. Instead of using a few complex assertions many simple assertions must be used. It must be pointed out that the use of assertions by itself does not solve the problem of test data generation. It provides the means for checking the output, once appropriate inputs are applied. However,

the use of excutable assertions combined with other testing techniques results in a very good and efficient testing methodology.

ACKNOWLEDGEMENTS

The authors would like to thank Mr. Roger Tapper of Rockwell International (Collins Avionics), without whose help this experiment would not have been possible. Thanks are also due to Jim Saito and Doug Doane (NASA-AMES) and to the members of Center for Reliable Computing (Stanford University), especially Ms. Lydia Christopher, for their comments and suggestions.

Partial support for this work was provided by the NASA-AMES under Contract No. NAG 2-246.

REFERENCES

[Adrion 82] Adrion, W. R., M. A. Branstad, and J. C. Cherniavsky, "Validation, Verification, and Testing of Computer Software," ACM Computing Surveys, Vol. 14, No. 2, pp. 159-192, June 1982.

[Andrews 78] Andrews, D. M., "Software Fault Tolerance through Executable Assertions," Conference Record, 12th Asilomar Conference on Circuits, Systems and Computers, pp. 641-645, Pacific Grove, California, November 6-8, 1978.

[Andrews 79] Andrews, D. M., "Using Executable Assertions for Testing and Fault Tolerance," Digest, 9th Annual International Symposium on Fault Tolerant Computing (FTCS-9), pp. 102-105, Madison, Wisconsin, June 20-22, 1979.

[Andrews 81] Andrews, D. M., and J. P. Benson, "An Automated Program Testing Methodology and its Implementation," Proceedings of 5th International Conference on Software Engineering, pp. 254-261, San Diego, California, March 9-12, 1981.

[Bendixen 83] Bendixen, G. E., "The Digital Flight Control and Active Control Systems on the L-1011," Proceedings, IEEE/AIAA, 5th Digital Avionics Systems Conference, pp. 11.2.1-11.2.11, Seattle, Washington, October 31-November 3, 1983.

[Carter 79] Carter, W. C., "Fault detection and Recovery Algorithm for Fault Tolerant Systems," EURO IFIP 79, pp. 725-739, North Holland Publishing Company, 1979.

[Chow 76] Chow, T. S., "A Generalized Assertion Language," Proceedings, 2nd International Conference on Software Engineering, pp. 392-399, San Francisco, CA, October 13-15, 1976.

[DeFeo 82] DeFeo, P., D. Doane, and J. Saito, "An Integrated User - Oriented Laboratory for Verification of Digital Flight Control Systems - Features and Capability," NASA Technical Memorandum 84276, Ames Research Center, Moffett Field, California, 94035, August 1982.

[DFCR-96 80] L-1011 DAFCS Software Description, DFCR-96R1, L-1011 Digital Flight Control System Report, Collins Avionics Division, Rockwell International, 1980.

[Floyd 67] Floyd, R. W., "Assigning Meaning to Programs," Proceedings Symposia in Applied Mathematics, Vol. 19, pp. 19-32, American Mathematics Society, Providence, Rhode Island, 1967.

[Glass 80] Glass, R. L., "A Benefit Analysis of Some Software Reliability Methodologies," ACM SIGSOFT, Software Engineering Notes, Vol. 5, No. 2, pp. 26-33, April 1980.

[Hecht 76] Hecht, H., "Fault-Tolerant Software for Real-Time Applications," ACM Computing Surveys, Vol. 8, No. 4, pp. 391-407, December 1976.

[Hoare 69] Hoare, C. A. R., "An Axiomatic Basis of Computer Programming," Comm. ACM, Vol. 12, No. 10, pp. 576-580, October 1969.

[Horning 74] Horning, J. J., "A Program Structure for Error Detection and Recovery," Lecture Notes in Computer Science, Vol. 16, pp. 171-187, Springer Verlag, New York, New York, 1974.

[King 76] King, J. C., and S. L. Hantler, "An Introduction to Proving the Correctness of Programs," ACM Computing Surveys, Vol. 8, No. 3, pp. 331-353, September 1976.

[Krieg-Bruckner 80] Krieg-Bruckner, B., and D. C. Luckham, "ANNA: Towards a Language for Annotating ADA Programs," ACM SIGPLAN Notices, Vol. 15, No. 11, pp. 128-138, November 1980.

[Leveson 83] Leveson, N. G., and P. R. Harvey, "Analyzing Software Safety," IEEE Transactions on Software Engineering, Vol. SE-9, No. 5, pp. 569-579, September 1983.

[Luckham 75] Luckham, D. C., and F. W. von Henke, " A Methodology for Verifying Programs," Proceedings International Conference on Reliable Software, pp. 156-164, Los Angeles, California, April 21-23, 1975.

[Luckham 84] Luckham, D. C., et al., "ANNA: A Language for Annotating ADA Programs," Preliminary Reference Manual, Technical Report No. 84-261, Computer Systems Laboratory, Stanford University, July 1984.

[Mahmood 83] Mahmood, A., D. J. Lu, and E. J. McCluskey, "Concurrent Fault Detection using a Watchdog Processor and Assertions," Proceedings, 1983 International Test Conference, pp. 622-628, Philadelphia, Pennsylvania, October 18-20, 1983.

[Manna 69] Manna, Z., "The Correctness of Programs," Journal of Computer and System Sciences, Vol. 3, No. 2, pp. 119-127, May 1969.

[Milli 81] Milli, A., G. Metze, "Self-Checking Programs: An Axiomatization of Program Validation by Executable Assertions," Digest, 11th Annual International Symposium on Fault Tolerant Computing (FTCS-11), pp. 118-120, Portland, Maine, June 24-26, 1981.

[Popek 77] Popek, G. J., et al., "Notes on the Design of EUCLID," ACM SIGPLAN Notices, Vol. 12, No. 3, pp. 11-18, March 1977.

[Ramamoorthy 75] Ramamoorthy, C. V., and S. F. Ho, "Testing Large Software with Automated Software Evaluation System," Proceedings of International Conference on Reliable Software, pp. 382-394, Los Angeles, California, April 21-23, 1975.

[Randell 75] Randell, B., "System Structure for Software Fault Tolerance," IEEE Trans. on Software Engineering, Vol. SE-1, No. 2, pp. 220-232, June 1975.

[Saib 77] Saib, S. H., "Executable Assertions - An Aid to Reliable Software," Conference Record, 11th Asilomar Conference on Circuits, Systems and Computers, pp. 277-281, Pacific Grove, California, November 7-9, 1977.

[Stucki 75] Stucki, L. G., G. L. Foshee, "New Assertion Concepts for Self Metric Software Validation," Proceedings of International Conference on Reliable Software, pp. 59-71, Los Angeles, California, April 21-23, 1975.

[Taylor 80] Taylor, R. N., "Assertions in Programming Languages," ACM SIGPLAN Notices, Vol. 15, No. 1, pp. 105-114, January 1980.