WL-TR-92-3101

URV FLIGHT TEST OF AN ADA IMPLEMENTED SELF-REPAIRING FLIGHT CONTROL SYSTEM

M. Mears, S. Pruett, J. Houtz

Control System Development Application Branch
Flight Control Division

AUG 1992

FINAL REPORT FOR 01/01/85 - 08/31/92

FLIGHT DYNAMICS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT PATTERSON AFB OH 45433-6553

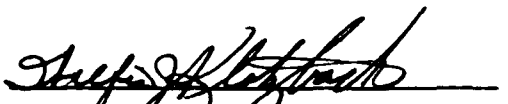
NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

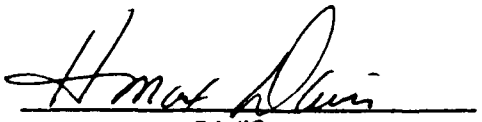
This report is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

MARK J. MEARS, Electronic Engr
Control Systems Development
and Application Branch
Flight Control Division



JAMES K. RAMAGE, Chief
Control Systems Development
and Application Branch
Flight Control Division



H. MAX DAVIS
Assistant For R&T
Flight Control Division

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify WL/FIGS , WPAFB, OH 45433-6553 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	AUG 1992	FINAL 01/01/85--08/31/92

4. TITLE AND SUBTITLE URV FLIGHT TEST OF AN ADA IMPLEMENTED SELF-REPAIRING FLIGHT CONTROL SYSTEM

5. FUNDING NUMBERS

C
PE 62201F
PR 2403
TA 07
WU 37

6. AUTHOR(S) D. Mears, S. Pruett, J. Houtz

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Control System Development Application Branch
Flight Control Division

8. PERFORMING ORGANIZATION REPORT NUMBER

WL-TR-92-3101

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

WRIGHT LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT PATTERSON AFB OH 45433-6553
WL/FIGS, Attn: MEARS 513-2558291

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT RELEASE; DISTRIBUTION IS UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Self-Repairing Flight Control System (SRFCS) technology is an extension of traditional redundancy management. It uses existing redundancy in aircraft control surfaces to compensate for control surface failures and battle damage effects. This report describes the results of flight tests of a SRFCS, coded in Ada for an Unmanned Research Vehicle (URV). This includes a description of the design of the control reconfiguration method, the Failure Detection (FDI) method, the Ada code, the URV model, data analysis of the flight test time histories, and the computational aspects of the algorithms. Time history plots of selected flight test results are included in this report and acceptable performance was achieved for all the failure cases which were flown.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Flight Control, Self-Repairing, Ada, FDI	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Foreword

The URV flight test and this final report are the result of the efforts of several individuals. In particular, Dave Dawson (WL/FIGL), Bill Adams, Douglas Roy and Jim Miller (Lear Siegler) were responsible for the operation of the URV during the project. It is due to their efforts that the designs were implemented and flight tests were completed without incident.

Two groups of flight tests were performed during this study. The first flights were performed in August and September 1987. However all time histories shown in this report were a product of the later flights in August 1989.

The URV flight test work was begun to support the WL/FIGL Self-Repairing Flight Control System (SRFCS) Program. Mark J. Mears is the principle author of this technical report, however, verbal and written inputs have also been provided by other project participants.

The author wishes to express special thanks to Stanley H. Pruett and John Houtz (WL/FIGL), who participated in the flight test activities and who wrote and edited portions of this report.

The chief engineer of the SRFCS program, Philip R. Chandler (WL/FIGL), supported and participated in the flight test project on the URV. Much of the flight test work was managed and performed by Dave Dawson, Capt. Robert Kelly and William Lindsay. Many individuals participated in designing the SRFCS for the URV; these include Dr. Kuldip Rattan (Wright State University), Tom Molnar, Greg Carter, Capt. Harry Gross, Capt. Barry Migyanko (WL/FIGL), and Capt. Brian Ray (WL/FIGC). Ada code development and implementation for the SRFCS was done by John Houtz, Stan Pruett (WL/FIGL) and Douglas Roy (Lear Siegler). During all tests the URV was piloted by Jim Miller (Lear Siegler).

Table of Contents

List of Figures

Nomenclature

A	Small perturbation linear model stability matrix
AUTH	weighting matrix
KLINKS	Linkage ratio matrix
A_{ij}	the (i,j) element of the matrix A
A_{nxn}	nxn matrix A
Bi	impaired B matrix (in general)
Bia	impaired B matrix for failed aileron
Bie	impaired B matrix for failed elevator
Bir	impaired B matrix for failed rudder
Bo	nominal B matrix
inv()	denotes matrix inverse
Ki	impairment gain matrix (in general)
Kia	impairment gain matrix for failed aileron case
Kie	impairment gain matrix for failed elevator case
Kir	impairment gain matrix for failed rudder case
Ko	unimpaired gain matrix
[L]	lower diagonal matrix
p	roll rate
p_f	Probability of having a surface fail
P_{fa}	Probability of false alarm
p_h	Probability of being hit (battle damaged)
p_k	Probability of being killed
p_{klf}	Probability of being killed given that a failure has occurred
p_{klh}	Probability of being killed after having been hit
P_{md}	Probability of missed detection
q	pitch rate
r	yaw rate
$\underline{u}$	system inputs vector
[U]	upper diagonal matrix
$\underline{x}$	aircraft state vector (in radians)
α	angle of attack
β	side slip angle
$\underline{\delta}$	surface command vector (in degrees)

δ_{al}	left aileron deflection angle
δ_{ar}	right aileron deflection angle
δ_{com}	deflection command
δ_{el}	left elevator deflection angle
δ_{er}	right elevator deflection angle
δ_{fl}	left flap deflection angle
δ_{fr}	right flap deflection angle
δ_{r}	rudder deflection angle
Δt	time increment
ϕ	roll attitude angle
θ	pitch attitude angle
$()^{+}$	matrix pseudoinverse of ()
$()^{T}$	matrix transpose of ()

Abbreviations:

AFTI	Advanced Fighter Technology Integration
FDI	failure detection/isolation
RPV	Remotely Piloted Vehicle
SRFCS	Self Repairing Flight Control System
SVD	Singular Value Decomposition
URV	Unmanned Research Vehicle

1.0 Background

1.1 Self-Repairing Flight Control System Design

In aircraft control system design, one of the primary concerns is safety of flight. Stringent requirements regarding the number of failures allowed in a given period of time are defined for mission completion and system failure[1]. Another driving concern is aircraft survivability in battle scenarios. These concerns are driven by factors such as pilot safety, weapon system effectiveness and cost. The objective of the Self-Repairing Flight Control System (SRFCS) methodology is to meet these requirements by allowing the designer to use aircraft resources more wisely.

Operational reliability requirements have traditionally been met in one of two ways. The first of these is the use of highly reliable components to meet high reliability requirements. However, these components are typically expensive. Another way to satisfy stringent reliability requirements is to use hardware redundancy. But hardware redundancy implies additional hardware which adds weight and decreases the system performance.

SRCFS is a means of achieving high flight control system reliability by intelligently using available aerodynamic redundancy in aircraft control surfaces. However, to use this redundancy requires knowledge of the aircraft dynamics, and computing resources to mechanize the algorithm.

The Flight Control Division's Control System Development Branch (WL/FIGL), began investigating the viability of SRFCS methods in the 1980's and decided to use the Unmanned Research Vehicle (URV) as a means for test and evaluation of the concept. The primary benefit of using the URV was cost; the URV was an FIGL resource which had been used to demonstrate flight control concepts. Also, flight safety for the URV is less of a concern than that associated with manned flight test.

Information from the SRFCS program provided a starting point for the URV flight test study, and the final SRFCS program reports provide a baseline for future SRFCS studies[4,5,6,7].

1.2 Use of the Ada Programming Language

The Ada computer programming language is the official mandated DoD High Order Language (HOL) to be used for mission critical weapon systems software. One objective of the effort was to assess the impact of Ada on a real-time flight control related application.

The purpose for mandating use of one software language is to reduce software life cycle costs. Such features as information hiding and strong typing may increase readability and understandability of the code, and result in less time and effort during modifications.

1.3 History of the XBQM-106

The XBQM-106 is an in-house designed Mini-RPV (referred to as 'URV' - Unmanned Research Vehicle) with a pusher propeller configuration[2]. The URV has a wing span of twelve feet, a fuselage length of ten feet and weigh approximately two hundred pounds. Fig. 1 is a line drawing of the URV.

The purpose of the URVs is to test and demonstrate emerging technology. To support their use in flight control applications, wind tunnel work was performed in 1978 to provide aerodynamic model information[2]. This data has been modified as the RPV characteristics have changed due to changes in weight and c.g. location[3], and with the inclusion of flaps as control surfaces.

Some of the early work done on the URV resulted in development of an inner loop control algorithm to improve the URV dynamic characteristics[9]. This algorithm runs on a flight control computer which uses a micro-controller that was mass produced for automobile engine control. This same flight control system was used during the SRFCS flight tests.

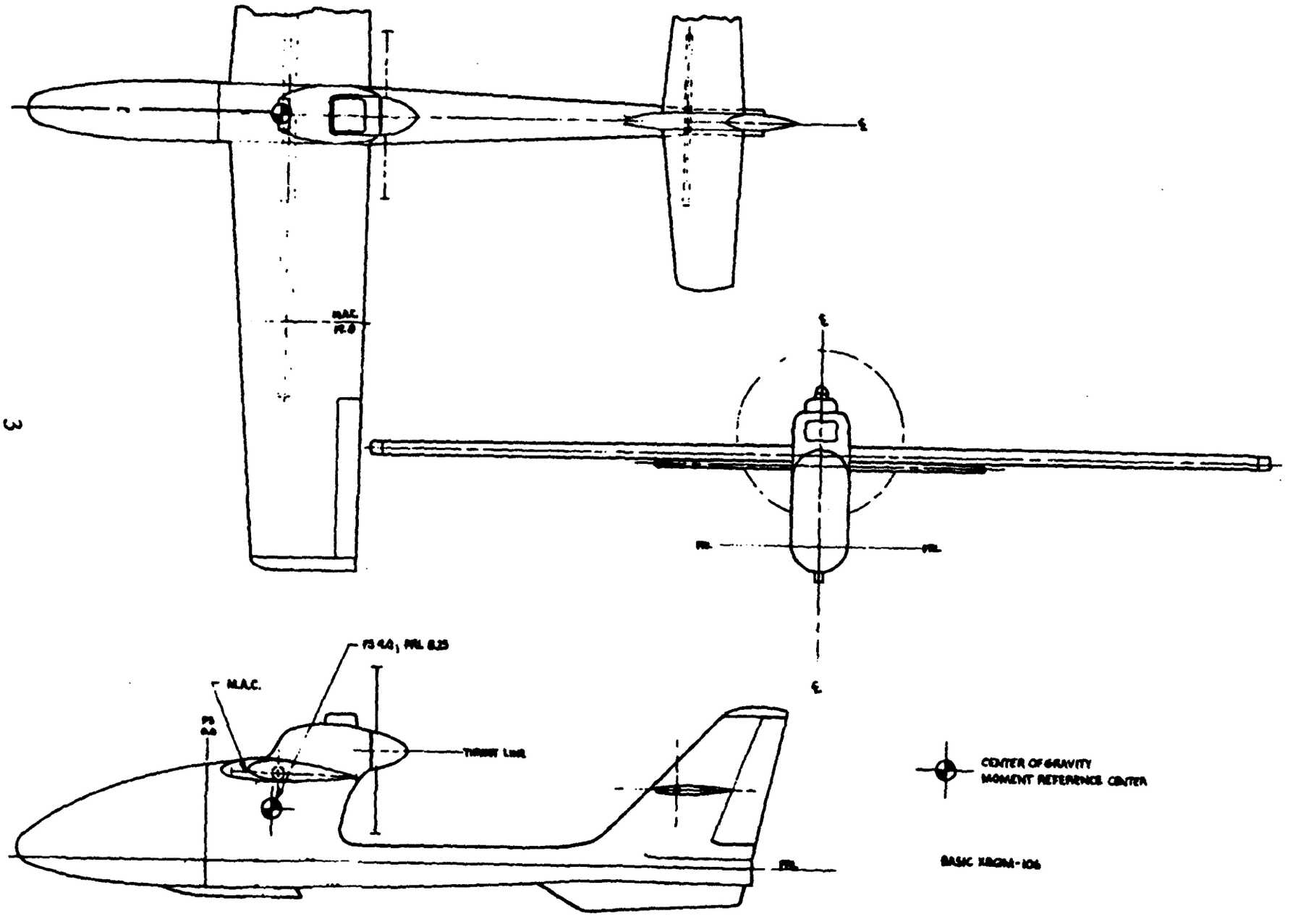


Figure 1. URV Draw

2.0 Self-Repairing Flight Control System Introduction

Self-Repairing Flight Control Systems (SRFCS) are an extension of inner loop control, motivated by the desire to reduce the probability of aircraft[16] loss (p_k). The p_k is decreased by reducing the conditional probability of aircraft loss from battle damage ($p_{k|h}$). This goal is achieved by allowing the control system to adapt to battle damage or surface failures by altering the signals sent to the remaining healthy control surfaces. With this technology, the aircraft does not rely as heavily on each individual surface since other surfaces can combine to perform the same (or nearly the same) function.

Reduction in vulnerability is a direct result of the control system adaptation, which then permits a trade off between improved system reliability (afforded by the SRFCS) and system hardware redundancy. That is, the additional reliability allows the designer to reduce the number of actuators, hydraulic lines, etc. and thus, the aircraft weight. These trades make for an aircraft which performs better and is less likely to be damaged in battle.

These system benefits do not come without some cost. One aspect of this cost is the computational burden that must be absorbed by the flight control computer. The computations are required to model dynamics, predict states, estimate surface effectiveness and perform various matrix, and vector calculations.

Another cost associated with any adaptive control technique is that incorrect action may be taken by the controller. This may occur because of increased algorithm complexity, sensor noise, or modelling uncertainty/inaccuracy.

Adaptive control (including SRFCS) can be mechanized in a direct or indirect manner. A direct mechanization implies that explicit FDI is not performed. The controller simply reacts to changes in sensed variables and makes alterations in control as a matter of course[17]. Indirect adaptation uses separate diagnosis and action algorithms. The methods used on the URV were confined to indirect adaptation.

Fault diagnosis is performed by an FDI algorithm which compares actual outputs with modeled values to determine if faults exist, establishes the source of the fault and then updates the internal model. The reconfiguration strategy acts on the information provided by the FDI by redistributing control authority to "best" cope with the changes.

2.1 Control System Reconfiguration Strategy

The purpose of the control strategy is to distribute remaining aircraft control authority after an anomaly has been isolated. This can be done in a number of ways. The method implemented in the URV was the "Control Mixer" which uses a pseudoinverse to determine a new gain matrix.

The pseudoinverse works by operating on the basic linear state space equation.

$$\dot{\underline{x}} = A\underline{x} + B\,\underline{\delta} \tag{2.1}$$

where

$$\underline{\delta} = Ko * \underline{u} \tag{2.2}$$

The pseudoinverse effects only the contribution of the command. The nomenclature is defined below.

$\underline{\delta}$ — surface command vector
$\underline{u}$ -- system input vector
Bo -- nominal B matrix
Bi -- impaired B matrix
Ko -- unimpaired gain matrix
Ki -- impaired gain matrix
m -- number of non-zero rows of the Bi matrix
n -- number of non-zero columns of the Bi matrix

The Bi matrix represents control effectiveness after surface failures. The Bi matrices representing locked surfaces are created by zeroing the column of the Bo matrix corresponding to the failed surface. The effects of surfaces locked at non-zero deflection angles could be created by adding a bias term in the state equati :. The Bi matrices for partially missing surfaces are generated by replacing the column of the Bo matrix associated with the failed surface, with the effective percentage of that column. Both partially missing and failed at non-

zero deflection angles failure modes can be reconfigured by the mixer, however only the locked at zero deflection modes were investigated during flight tests.

The objective of the Control Mixer is to modify the calculations involving the system inputs ($\underline{u}$) to the impaired system so that they have the same effect as they would have had on the unimpaired system. This can be done by equating the control effects of the state space equation before and after impairment.

$$Bo * Ko * \underline{u} = Bi * Ki * \underline{u} \qquad (2.3)$$

This implies that

$$Bo*Ko = Bi*Ki \qquad (2.4)$$

To find the Ki matrix, an inverse is used. The characteristics of the inverse depend on the Bi matrix and can fall into one of the three general categories listed below[10].

1. Rows (m) and columns (n) of the Bi matrix equal.

This case is the simplest and implies that after the failure the number of states being controlled is the same as the number surfaces available. If the rank of the Bi matrix is full (rank of Bi = m = n) then a standard matrix inverse can be used to find the Ki gain matrix. By multiplying the above equation by inv(Bi)

$$Ki = inv(Bi)*Bo*Ko \qquad (2.5)$$

2. Rows (m) of the Bi matrix greater than the columns (n).

This is the overdetermined case where there are more states to be controlled than surfaces to control them. In this case, one cannot hope to reproduce the exact time response of the unimpaired aircraft with the impaired aircraft. Instead, one is attempting to reduce the mean squared error between the nominal (unimpaired) response and the response of the impaired aircraft.

$$Bi*Ki = Bo*Ko \qquad (2.6)$$

$$Bi^T*Bi*Ki = Bi^T*Bo*Ko \quad (2.7)$$

Taking the inverse of Bi^T*Bi, and pre-multiply each side of the above equation by this value results in the following equation.

$$Ki = inv(Bi^T*Bi)*Bi^T*Bo*Ko \quad (2.8)$$

The above inverse will exist if the columns of Bi are linearly independent.

3. Rows (m) of the Bi matrix less than the columns (n).

This is the underdetermined case which is of concern in the URV demonstration because the URV has more surfaces available than states to be controlled. This case allows a time history flown by the unimpaired aircraft model to be duplicated exactly by the impaired aircraft model. The mixer minimizes the Euclidean norm of the deflections needed to achieve the desired response. The mixer gain matrix for this case is derived in the following manner:

$$Bi*Ki = Bo*Ko \quad (2.9)$$

Inserting an identity into the right side of the equation does not change it.

$$Bi*Ki = I*Bo*Ko \quad (2.10)$$

Now let

$$I = V*inv(V) \quad (2.11)$$

where

$$V = Bi*Bi^T \quad (2.12)$$

then

$$Bi*Ki = (Bi*Bi^T)*inv(Bi*Bi^T)*Bo*Ko \quad (2.13)$$

The inverse of V will exist if the rows of Bi are linearly independent. Thus

$$Bi*Ki = Bi*(Bi^T*inv(Bi*Bi^T))*Bo*Ko \qquad (2.14)$$

or

$$Ki = Bi^T*inv(Bi*Bi^T)*Bo*Ko \qquad (2.15)$$

It is worth noting that the zero rows of the B matrices represent states that are integrals of other states (e.g., θ is the integral of q) and these B matrix zero rows have no effect on the states. Since zero rows cause rank deficiency in the pseudoinversion calculations, they are removed before performing the mixer gain calculations.

One problem with the above derived control mixer implementation, is that it does not recognize surface saturation. In fact there may be considerable authority for a particular command remaining in unsaturated surfaces which the mixer would not use. One way around this is to provide the mixer with information about each surface's deflection angle. This is done by way of an "authority" matrix, AUTH. The AUTH matrix is a diagonal matrix with elements equal to the absolute value of the difference between each surface limit and the corresponding surface deflection. With the AUTH matrix the gain derivation is similar to that above, with

$$V = (Bi*AUTH)*(Bi*AUTH)^T \qquad (2.16)$$

The new mixer gain becomes

$$Bi*Ki = (Bi*AUTH)*(Bi*AUTH)^T * inv[(Bi*AUTH)*(Bi*AUTH)^T] * Bo * Ko \qquad (2.17)$$

rearranging terms

$$Ki = (AUTH*AUTH^T) *Bi^T * inv[Bi*(AUTH*AUTH^T) * Bi^T] * Bo * Ko \qquad (2.18)$$

This gain matrix has the effect of reducing the weighted mean squared deflections.

Addressing the saturation problem with this effector weighting method requires that the gain matrix be updated as the surfaces move (after the FDI has isolated a failure). This can be a significant computational burden requiring a pseudoinverse calculation each time a new gain matrix is required.

The control mixer uses a model of the control effects to calculate the gain matrix. Thus, the gains are only as good as the representation of the controls by the B matrix in equation 2.1. If the model changes due to aircraft state and effector position, the gains can cause results different from those desired.

The mixer produces good results when the aircraft has a surface failure for which there exists another surface with similar characteristics. However, this can make the FDI problem more difficult. In general, surface configurations which are good for FDI, are bad for reconfiguration and vice versa.

Obviously no two surfaces have the exact same effects in all axes, at all flight conditions, because no two surfaces can occupy the same location on an aircraft. However it is possible that finite dimensional, linear models for the aircraft at some flight conditions will have linearly dependent (or nearly linearly dependent) B matrix columns. Thus, it may be helpful to form the controllability Grammian of each impaired system to determine which directions in the state space are most difficult to control[12].

2.2 Failure Detection and Isolation

The objective of FDI is to detect system failures when they occur and isolate the cause of the failure. Other SRFCS programs have been concerned with both aerodynamic ("global") and actuator ("local") FDI. Global FDI presents a much greater problem than local FDI, however only local FDI was investigated in the URV study. Since only local FDI was performed, isolation of the failure is not an issue. Once the failure is detected, the cause of the failure is known.

The local FDI methodology implemented on the URV was a modified version of contractor developed code for the Advanced Fighter Technology Integration (AFTI) F-16 that was adapted for the URV by FIGL. Actuators are flagged as failed if their filtered residual, which are formed by low pass filtering the difference between an actuator model and the real actuator, exceeds an analytically determined threshold value. The logic flow can be seen in Fig. 3.

The FDI algorithm is constrained to work quickly to prevent the failure transient from having an adverse effect on the aircraft. This time to detect is therefore, driven by the speed of the aircraft dynamics.

The time to detect failure is also based, in part, on surface activity. If the surface has relatively small inputs, there may not be a sufficient signal to noise ratio to detect a failure. However, failures which are not immediately detected because of low information content, are by definition, not critical since small surface activity causes small transients. These failures are detected when it is important that the surface become active.

FDI is driven by two opposing parameters; probability of false alarm (P_{fa}) and probability of missed detection (P_{md}). The amount of noise present in the system determines what threshold will work in light of the required P_{fa} and P_{md}. The question of isolation is not of concern with local FDI since each actuator has its own failure detection flag. Since failures were modeled as locked at zero deflection, the issue of estimation of the remaining authority was not addressed in this study.

3.0 Model Description

The URV Control System (see Fig. 2) shows the components of the simulation. This includes the control mixer, aircraft, sensors and control system. The control mixer is a gain matrix and implements the control reconfiguration. The "Aircraft Model" part of the simulation diagram contains five blocks. The dark shaded region denotes components of the actuators. The "Servos" block represents the dynamics of the electric motors which drive the surfaces. The "KLINKS" block models the linkage ratios associated with each surface. The "Limits" block is the position limits of each surface. The "Equations of Motion" block is the state space formulation of the aircraft equations of motion relating surface deflections to aircraft body attitude and rate. The remaining block in the "Aircraft Model" section is a conversion to bring out the states in degrees.

The URV stability and control derivatives were derived using DIGITAL DATCOM[8] and have been modified using flight data[3]. The resulting model is a constant coefficient, linear, small perturbation, decoupled lateral-directional and longitudinal representation with second order actuator models. The linear model takes the form of equation 3.1 where

$$A = \begin{bmatrix} -2.4776 & 0.0000 & 0.9748 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \\ 0.0000 & 0.0000 & 1.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \\ -42.512 & 0.0000 & -3.3361 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & -0.745 & 0.2451 & 0.0010 & -0.9848 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 1.0000 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & -25.804 & 0.0000 & -8.7554 & 2.7068 \\ 0.0000 & 0.0000 & 0.0000 & 23.2230 & 0.0000 & 0.0647 & -2.0286 \end{bmatrix} \quad (3.1)$$

and the unimpaired B matrix is

$$B = \begin{bmatrix} -0.0034 & -0.0034 & -0.0022 & -0.0022 & -0.0040 & -0.0040 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \\ -0.5812 & -0.5812 & -0.0481 & -0.0481 & -0.0660 & -0.0660 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0016 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \\ 0.2455 & -0.2455 & 0.6697 & -0.6697 & 0.6221 & -0.6221 & 0.0554 \\ -0.0130 & 0.0130 & -0.0428 & 0.0428 & -0.0403 & 0.0403 & -0.1789 \end{bmatrix} \quad (3.2)$$

The states of the model are

$$\underline{x} = [\alpha, \theta, q, \beta, \phi, p, r]^T \qquad \text{(radians)} \qquad (3.3)$$

(The model uses the states in radians (or rps), however they are converted to degrees (or deg/sec) before they are output as simulation time histories.) The control inputs are

$$\underline{\delta} = [\delta_{el}, \delta_{er}, \delta_{al}, \delta_{ar}, \delta_{fl}, \delta_{fr}, \delta_r]^T \text{ (degrees)} \qquad (3.4)$$

where the following variable definitions are given:

α	angle of attack	δ_{el}	left elevator deflection angle
θ	pitch attitude angle	δ_{er}	right elevator deflection angle
q	pitch rate	δ_{al}	left aileron deflection angle
β	side slip angle	δ_{ar}	right aileron deflection angle
ϕ	roll attitude angle	δ_{fl}	left flap deflection angle
p	roll rate	δ_{fr}	right flap deflection angle
r	yaw rate	δ_r	rudder deflection angle

The actuator model is described by the following transfer function:

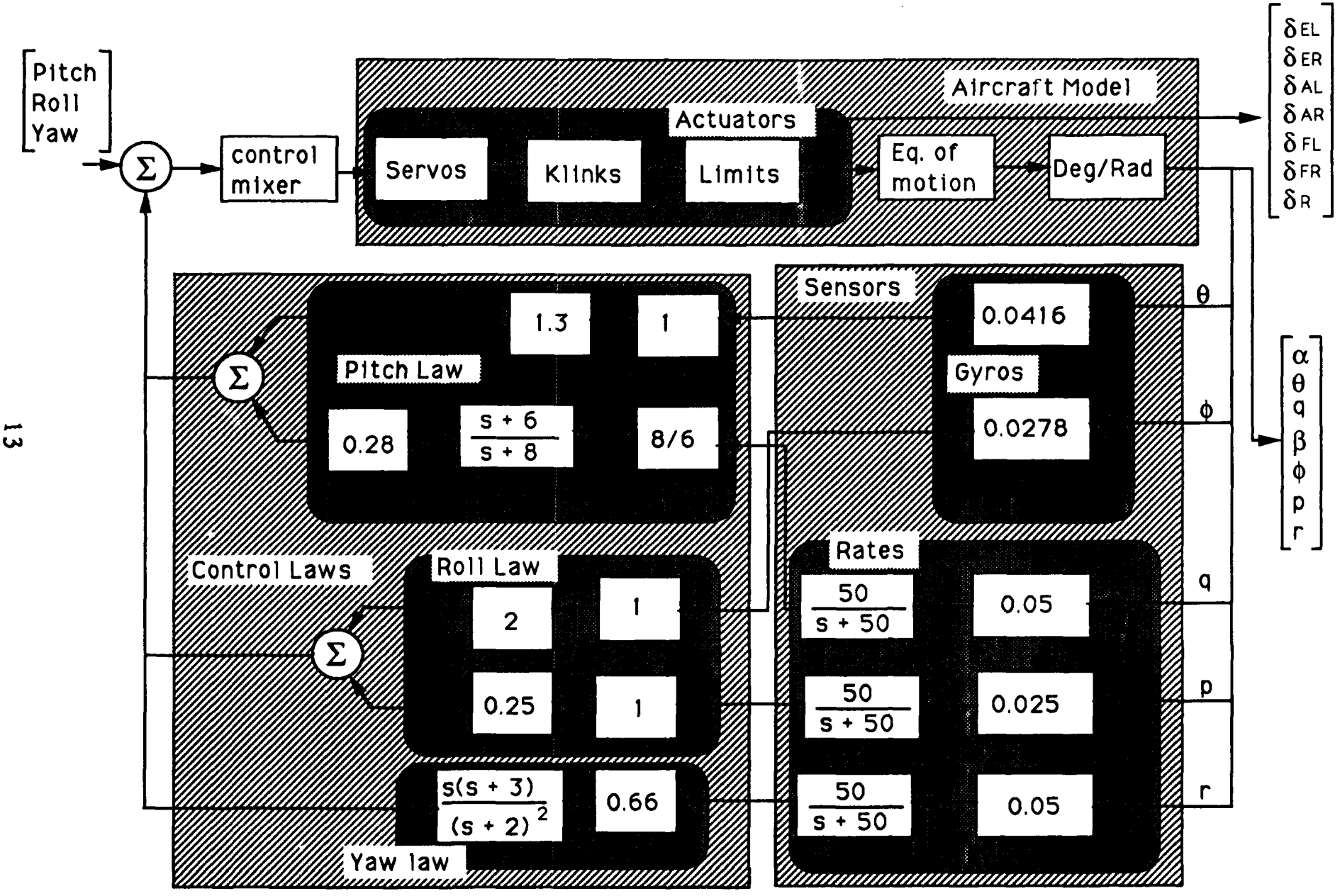


Figure 2. URV Control System

$$\frac{324}{s^2 + 25.4s + 324} \tag{3.5}$$

The output equation of the model is

$$\underline{y} = C\underline{x} \tag{3.6}$$

where the C matrix is a 7-by-7 identity matrix. Thus, the outputs are equivalent to the states of the model.

Another part of the model (Fig. 2) is the lightly shaded region representing the sensors. One darkly shaded region is the Gyros; one for pitch and one for roll. These measure the pitch and roll angle and have dynamics of significantly higher bandwidth than the rest of the system and are thus modeled as gains only[9]. These gains represent the sensitivities of each attitude gyro. The sensitivities are:

pitch -- 2.5 volts/60 deg = .0416
roll -- 2.5 volts/90 deg = .0278

The body rates are measured by fluidic rate sensors with dynamics approximated as a first order lag with corner frequency at 50 rps and sensitivities of 2.5 volts per 50 deg/sec in pitch and yaw, and 2.5 volts per 100 degrees in roll[9]. These quantities are shown in the darkly shaded region named "Rates".

The "Control Laws" part of the diagram represent the control equations that are in the URV.

Simulation time histories are shown in Appendix B, Fig. B.1 to B.8. These responses are for the unfailed URV in response to unit pitch, roll and yaw commands. The longitudinal and lateral axes are decoupled. Responses not shown for states or surfaces deflections that have not been excited and remain at zero.

4.0 Application of SRFCS to the URV

One of the first steps to implementing control system reconfiguration on the URV was an analytical investigation determining the controllability of the URV. This study was based on a linear model developed during past efforts.

The URV originally had differential ailerons, collective elevators, one rudder and no flaps; the first linear model reflecting this three input configuration. It was clear, however, that both differential and collective deflections of the ailerons and elevators were needed. Therefore, the model was altered by attributing the effects of individual left and right elevators and ailerons to separate B matrix columns; each with half the combined (left and right) authority. This resulted in a five input model.

This change to a five input system was still not adequate. With five inputs, locked surface failures still result in an overdetermined system for which the Control Mixer cannot restore nominal performance. The best it can do is minimize the norm of deviations from the desired states.

Another problem with use of the control mixer in dealing with the five input URV model, was that the pseudoinverse calculation for rudder failure resulted in large values in the mixer gain matrix. The reason for these high gains is explained as follows: The ailerons have much more effect in yaw than do the elevators and the mixer favors their use over that of the elevators. Also, the ailerons have much larger effect in roll than in yaw and small yaw commands induce large rolling moments from the ailerons which must countered by elevator deflections. To generate the commanded yaw effect, the ailerons and elevator end up fighting each other in roll. This can be seen in the values of the matrices without flap inputs for the case of rudder failure shown later in this section.

The columns of the Ko and Ki matrices indicate pitch, roll and yaw commands respectively. The entries in the Kir matrices are large because the roll, which is induced while the aileron attempts to create yaw, must be balanced. The ratio of roll authority to yaw authority for the aileron is 15.647, while this ratio for the elevators is 18.885. Since these ratios are close, the deflections must be large to generate movements in one axis while balancing the other axis. In

other words, this is attributable to nearly linearly dependent effects of ailerons and elevators in roll and yaw.

Methods for reducing these matrix gain values for the failed rudder case were considered. One method was to zero certain elements of the impaired B matrix. These zero elements had the effect of reducing the use of the surface, corresponding to the zeroed column entry, for effect in the state corresponding to the zeroed row. However, without flaps, adjustments to the model (zeroing of B matrix model elements) could not be found to reduce the gains to reasonable magnitudes.

When flaps are added to the model, the system is underdetermined because it has more surfaces than states to be controlled. In these cases the control mixer minimizes the norm of the surface deflections and restores the model performance to exactly what it was before the failure.

Also, flaps add about 22.5% of the yaw control power of the rudder[3]. Various methods for increasing the yaw control power were investigated and these are described in a Technical Memorandum[3].

The values for the mixer gain matrices, for an assortment of failures and configurations, is contained in Appendix A. However, to illustrate the difficulties associated with rudder failure, the mixer gain matrix for rudder failure (Kir) is shown below for the "no-flap" and "flap" configurations.

"No-flap" configuration with rows corresponding to δ_{el}, δ_{er}, δ_{al}, and δ_{ar} respectively:

$$Kir = \begin{bmatrix} 1.0000 & 0.0000 & -32.5982 \\ 1.0000 & 0.0000 & 32.5982 \\ 0.0000 & 1.0000 & 11.9913 \\ 0.0000 & -1.0000 & -11.9913 \end{bmatrix} \qquad (4.1)$$

"Flap" configuration with rows corresponding to δ_{el}, δ_{er}, δ_{al}, δ_{ar}, δ_{fl} and δ_{fr} respectively:

$$Kir = \begin{bmatrix} 0.9996 & 0.0988 & -30.7668 \\ 0.9996 & -0.0988 & 30.7668 \\ 0.0175 & 0.5086 & 2.8785 \\ 0.0175 & -0.5086 & -2.8785 \\ -0.0093 & 0.4901 & 9.0873 \\ -0.0093 & -0.4901 & -9.0873 \end{bmatrix} \quad (4.2)$$

When the rudder column of the Bo matrix is zeroed to become the Bir matrix, it becomes a rank four matrix. By eliminating the zero row, a normal inverse replaces the pseudoinverse, but makes the matrix multiplications used in the Kir matrix solution non-conformal. Also, given the model, the impaired aircraft has no way to recreate the β authority. Therefore, the sideslip was ignored.

Thus, addition of flaps had three purposes: 1) To make the system underdetermined which allowed the mixer to find gains that would completely regain performance. 2) To add yaw authority, in an attempt to reduce the gain values calculated by the mixer for the failed rudder case. 3) To help alleviate the yaw command, aileron induced roll for the case of failed rudder. (The flaps were also desired to reduce landing speed, but this was not related to reconfiguration.)

The addition of flaps, in itself, did not significantly reduce the values in the impaired gain matrices. In the case of elevator failures, the flap configured model resulted in higher gains. The reason the addition of flaps had little effect on the mixer gain matrix values was that the B matrix columns associated with ailerons and flaps are vectors that are nearly linearly dependent.

To reduce mixer gain matrix values to a useable level, the designer was forced to fool the mixer (gain calculation) by zeroing B matrix elements. By doing this the designer tells the system which surface to use to produce a state change; the mixer discourages use of surfaces in axes where a zero B matrix entry is placed.

By zeroing elements of the B matrix, the designer changes physics to meet the problem. In the physical system, use of the surfaces for commands or moment

balancing may well generate changes in states corresponding to zero B matrix entries in the model. Therefore the aircraft does not perform exactly as expected.

The linkage ratios were not considered during the calculation of the gain matrices. This resulted in the values in the gain matrix being a different magnitude than they should have been. It was fortunate that these values in the gain matrices were not drastically different in magnitude since this would have caused unexpected flight test performance. The driving reason that the differences caused by neglect of the linkage ratios did not make drastic differences between flight test and simulation, is that all surfaces on the URV have a very large portion of their total authority in one axis. For aircraft with axes coupled surfaces, the linkage ratios would make a much greater difference in the gains and thus surface deflections after failure.

The FDI design used for the URV had originally been done for the AFTI F-16. The local FDI section was removed from the original code and modified for use with the URV. The logic flow of the local FDI for the URV was as shown in Fig. 3.

The first task of the FDI algorithm was to transform the data into floating point format. The next task was filtering of the position information to reduce the noise. This used a 20 radian per second first order filter. This frequency is slightly beyond the actuator cutoff to minimize the impact of the filter on the signal information. The basis of the FDI was comparison of the surface rate measurements with expected surface rates. This required data measurement and a model. Thus, the position was measured and the "measured" rate calculated as in Eqn. 4.3 where Δt is the time step, (k) indicates the present time and (k-1) is one time step in the past.

$$\text{rate}(k) = [\text{position}(k) - \text{position}(k-1)]/(\Delta t) \qquad (4.3)$$

The model rate was based upon the transfer function of the actuator. The digitized state space representation (1/60 second) of each of the seven actuators is:

$$\dot{\underline{x}} = \begin{bmatrix} .9610 & .0134 \\ -4.34 & .623 \end{bmatrix} \underline{x} + \begin{bmatrix} .0390 \\ 4.34 \end{bmatrix} \delta_{com} \qquad (4.4)$$

where

$$\underline{x}^T = [\text{position} \quad \text{rate}] \qquad (4.4a)$$

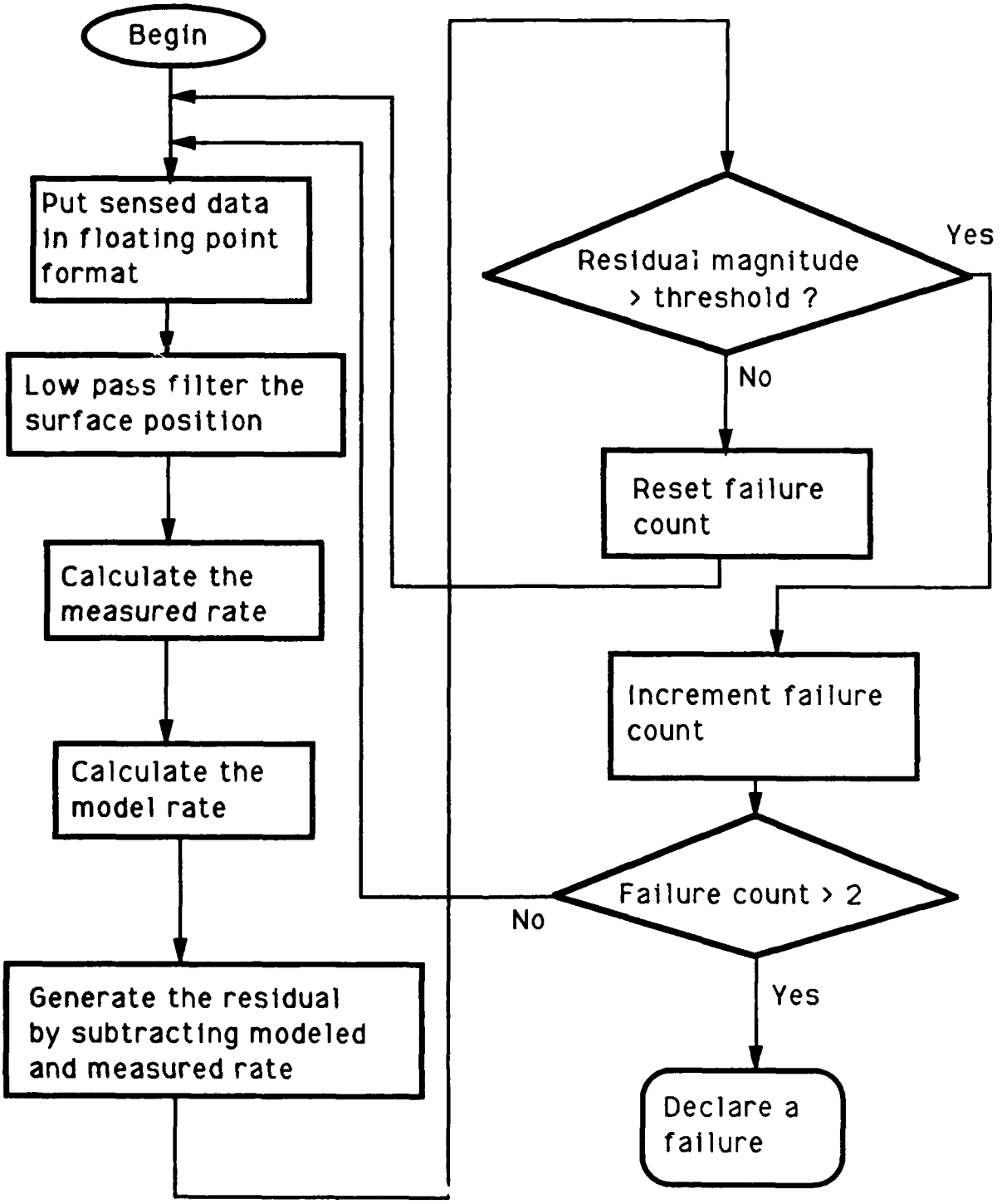


Figure 3. FDI Flow Chart

and

$$\delta_{com} = \text{actuator command} \qquad (4.4b)$$

To improve the accuracy of the model, model position was not used. Instead the measured position information was used to generate each model rate. This can be seen in the code which is included in Appendix D.

For each of the seven actuators, the modeled rate and measured rate were compared. If the difference was greater than a predefined threshold, a failure counter for that actuator was incremented. If the difference was less than the threshold, the counter was cleared. If a counter reached a count of three, that actuator was flagged as failed and the reconfiguration algorithm was activated.

Given accurate noise data and an actuator model, the threshold value for FDI could be set analytically based upon the probability of false alarm (P_{fa}) and probability of missed detection (P_{md}). However, the threshold values used in flight test were based upon trial and error. Due to inaccuracies in the actuator model dynamics, the threshold values had to be set very large.

5.0 Computational Problem Overview

The computational hardware used in the SRFCS experiment consisted of two separate computers connected by a telemetry link. The control mixer and FDI, which make up the SRFCS, ran on the ground based computer, while the on-board computer performed the feedback control functions. By partitioning the software in this way, the previously existing flight control system hardware and software could remain essentially the same, and the SRFCS could be coded in Ada and targeted for the 68020 based ground computer. The telemetry system relayed information from the on-board computer/sensors to the FDI algorithm on the ground computer, and telemetered control system gain information from the control mixer algorithm, running on the ground computer, to the on-board computer.

The SRFCS ran on the ground based computer. However since it provided, inner loop gain information to the on-board flight control computer, the structure and implementation of the FDI and control mixer software was a critical part of the entire system.

The role of FDI is conceptually simple, but its execution should be synchronized with the SRFCS and the control system. The period of execution of the FDI should be an integer multiple of the control system period to insure that all actuator data is measured during the same sample period. However, due to the characteristics of the telemetry system linking the ground based FDI computations with the flying URV, the FDI and the URV control system were not synchronized.

The URV telemetry system was used in previous experiments to download flight data for analysis. No need for synchronous interaction between a ground based system and the on-board flight computer had ever been established. The rate at which the telemetry system ran was determined by the number of variables transmitted and the telemetry bandwidth limits. Although hardware and software to allow synchronous execution of the ground and flight systems was developed, it was ignored during implementation. Since the maximum possible data latency due to the asynchronous running SRFCS was very small in comparison to the actuator time constant, the fact that the telemetry and control system were asynchronous, by itself, turned out to be a relatively small part of the problems that affected the performance of the FDI. Therefore, during the SRFCS

flight tests, the flight control computer and the ground based SRFCS were run asynchronously; the control system rate remained 60 Hz and the ground computer sample rate, driven by the telemetry system, was about 28.5 Hz.

FDI problems were compounded by inaccuracies in the actuator model due to mismodelled dynamics and the fact that the model had been digitized at the control system rate, while it ran at the telemetry system rate on the ground computer. This mistake could have been easily fixed, but was overlooked during implementation.

The discrete dynamic equations of the FDI requires the use of a hardware interrupt or code to continually poll the system clock. In the case of the URV, polling the system clock would be not only computationally wasteful, but result in timing delays as a result of the 1/128 second quantization of the Ada system clock (a 128 Hz clock is defined as a part of the Ada language).

By using a hardware interrupt, problems associated with Ada system clock were avoided. However, in keeping with the ideals of the Ada programming language, the interrupts were serviced using Ada tasking constructs instead of assembly language routines. This meant that the exact times of execution of the user routines (FDI and control mixer) were affected by the efficiency of the real-time operating system, the number interrupts and the order in which interrupts were handled. However, for the URV SRFCS code, interrupts were serviced almost immediately and interrupt handling delays had no noticeable impact upon system performance.

5.1 Ada Implementation

Ada was the programming language chosen for the SRFCS algorithms on the 68020 based ground computer. This choice was made to become familiar with, and to evaluate some of the features of Ada as used in real-time systems.

In an effort to reduce life cycle costs, the Ada language was designed to allow readable structured program development, and to include many features that would be useful to software developers. The scope of the language definition of Ada makes the impact of the compiler and real-time library routines of great importance.

Ada tasking permits the development of independent processing entities that can execute simultaneously on separate processors, or can share a single processor. Ada tasks are structured to communicate with each other by copying

within a common memory or by using input/output statements between processors that do not share memory. Entry calls and accept statements are the primary method for synchronizing tasks and communicating values between tasks. The interaction between the calling and accept statements is referred to as the Ada rendezvous.

Ada also has interrupt-handling constructs available. These interrupt handling constructs are described in section 13.5.1 of the Ada Language Reference Manual[14].

Ada representation clauses are used to specify how the type declarations are to be mapped onto the target machine. They can also be used to associate data objects, subprograms task units, and task entries with a specific address.

5.2 FDI Computation

The purpose of the FDI task is to detect a failure in any of the actuators. This is done by comparing a model of each actuator with measurements of the state of the actuator. This task by itself is not a large computational burden. However, with the use of Ada tasking constructs, this task poses a nontrivial computational problem. In fact, during the URV flight test, the processing overhead associated with the Ada tasking was greater than the actual numeric computation.

In Ada, even with a hardware interrupt, a rendezvous must take place. Variables must be passed to the control mixer task using the Ada rendezvous. By using a hardware interrupt with assembly language, this Ada overhead can be avoided. However, for this program, the Ada overhead was accepted to avoid the use of an assembly language serviced hardware interrupt.

5.3 Control Mixer Gain Computation

After the FDI detects a failure, the control strategy (control mixer) code must be initiated. The FDI initiates the control strategy by setting a flag. (See Appendix D for the Ada code.)

The control mixer task (SLOGI routine) computes the twenty one control mixer gains (7 by 3 matrix) used to redistribute the commands to the surface actuators. This allows the unfailed effector surfaces to take over the activity of the failed surface.

The gains are computed using Eqn. 5.1 (Bi^+ denotes the pseudoinverse of Bi). This equation dictates that the rates of change of the states before and after a failure will be equal, or at least as close as can be achieved.

$$Ki = (Bi)^+ * Bo * Ko \qquad (5.1)$$

From the implementation viewpoint, this matrix equation presents two problems: 1) matrix computations require some overhead (for this reason they are not generally used in real-time applications), 2) the equation requires the computational expense and potential numeric difficulty of a pseudoinverse operation. Both of these problems combine to make the equation difficult to implement in real-time on an embedded processor.

Embedded processors tend to be smaller and more limited than larger mainframe or minicomputers and often lack hardware for floating point and matrix calculations. The hardware used for the FDI and control strategy calculations for the URV flight tests was a VME133 board[18] which has the capability to perform floating point operations directly in hardware. However, the VME133 board had no special purpose hardware for matrix operations.

5.4 Pseudoinverse Algorithms

The implementation of the pseudoinverse operation can be achieved by two distinct categories of algorithms. The first category does not use an explicit matrix inverse operation, but instead uses a more direct approach (e.g. singular value decomposition algorithm for computing the pseudoinverse). The second category of algorithms that is commonly referenced in the literature[15] makes use of a standard matrix inverse operation. This second class of algorithms essentially recast the pseudoinverse operation so that a standard matrix inverse operation can be used to achieve a pseudoinverse operation.

A singular value decomposition (SVD) algorithm was considered, but found to be unsuitable for real-time implementation using the available processing hardware. A vendor supplied direct pseudoinverse method was also investigated, however the performance of the was algorithm never acceptable. It was never determined whether the cause was ill-conditioned numerical computations or possibly a software "bug" that caused poor numerical performance.

The pseudoinverse method for the URV flight tests uses the explicit matrix inverse calculation. The method utilizes the equation for the underdetermined pseudoinverse (see Eqn. 2.15) which is shown below.

$$Bi^{+} = Bi^{T} * inv(Bi * Bi^{T}) \tag{5.2}$$

From Eqn. 5.2, it can be observed that the following operations must be performed to complete the pseudoinverse operation: matrix transpose, matrix multiplication, and matrix inversion. There is also a need to assure that the product of the matrix multiplication ($Bi*Bi^{T}$) is invertible.

FDI algorithms that estimate surface effectiveness for a general set of failure conditions require a check for singularity/conditioning of ($Bi*Bi^{T}$). However, the failure scenarios considered in this program resulted in adequately conditioned ($Bi*Bi^{T}$) in all cases except the case of failed rudder (which was not flight tested).

Various methods for performing the matrix inverse of Eqn. 5.2 were investigated. The particular method chosen for this project relies on the following equation.

$$\underline{y}_{nx1} = A_{nxn} * \underline{x}_{nx1} \tag{5.3}$$

where $\underline{y}$ and $\underline{x}$ are vectors of length n and A_{nxn} is a matrix of dimension, n by n. Note that the A matrix in Eqn. 5.3 is not related to the A matrix in Eqns. 2.1 or 3.1 (the URV dynamics).

This method first employs a variation of LU Decomposition using the Crout[11] method, and then uses backsubstitution and forward substitution to find the vector, $\underline{x}$. This is done for each column vector, $\underline{y}$, of the identity matrix. The resulting $\underline{x}$ vectors are then concatenated to form the inverse of [A].

6.0 Data Analysis

The Self-Repairing Flight Control System (SRFCS) flight test was performed in two phases. The first phase was the reconfiguration strategy algorithm (control mixer) flight test with the FDI stubbed. During these tests, various surface failures were emulated and the new (pre-computed) gain matrix values were immediately used (perfect FDI). In the second phase the FDI was tested in conjunction with the control mixer. The control mixer was not initiated until the FDI detected a failure. Failures were emulated, the FDI acted on the changed system, sent information to the control mixer, and the control mixer reconfigured the gains to recover nominal flight performance.

Fig. 4 shows the coordinate axes and sign conventions that were used for the project. The sign convention for the surface deflections is given below.

Surface	Sign convention
δ_{el}	positive - trailing edge up
δ_{er}	positive - trailing edge up
δ_{al}	positive - trailing edge down
δ_{ar}	positive - trailing edge up
δ_{fl}	positive - trailing edge down
δ_{fr}	positive - trailing edge down
δ_{r}	positive - trailing edge right

Nominal performance of the URV is shown in Appendix C Figs. C.1 - C.6 for cases of pitch, roll and yaw doublets. This data is provided as a basis of comparison for the failed and reconfigured URV. Maneuvers for both nominal and reconfiguration/failure cases were doublets of pitch, roll or yaw. Doublets were used to test the response to both positive and negative commands. The reconfiguration strategy was tested for cases of a single failure of aileron, elevator, or flap failed to locked at zero degrees deflection angle, with each of the pitch, roll and yaw doublet commands.

The amount of data gathered from the flight tests is very large. The discussion of flight data and inclusion of strip chart recordings is limited to a small fraction of the total data gathered and analyzed. This report includes some

Notes:

1. Positive directions of force cofficients, moment coefficients, and angles are indicated by arrow

2. For clarity, origins of wind and stability axes have been displaced from the center of gravity

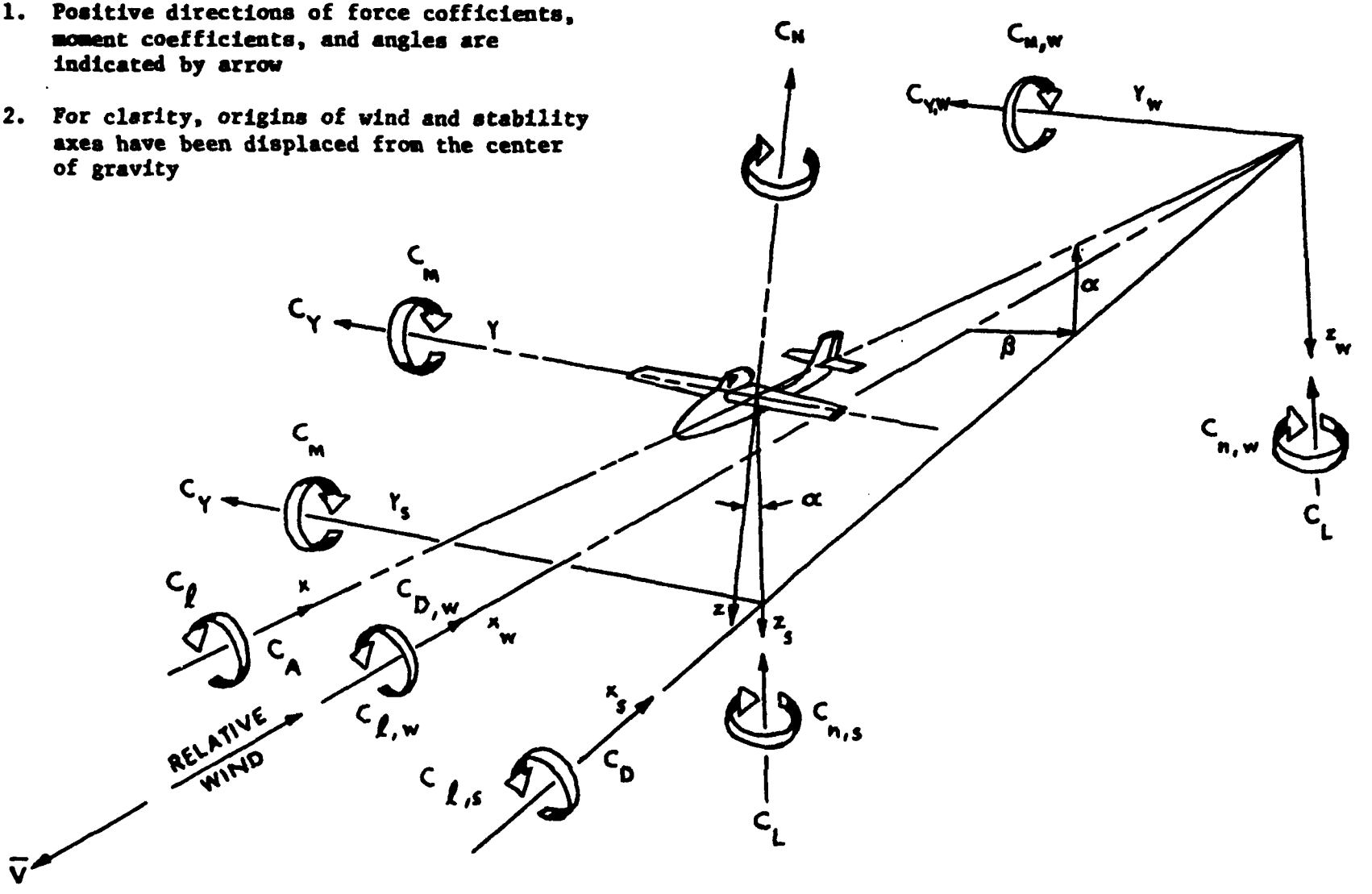


Figure 4. URV Sign Convention

interesting cases of reconfiguration which worked properly, cases involving problems with the control mixer and its implementation, and FDI related strip chart recordings.

6.1 FDI Stubbed

Failed Right Aileron, Roll Command:

Reconfiguration worked well for a failed left aileron with a roll command. This time history is shown in the strip chart of Appendix C, Figs. C.10 - C.13. The data shows that the primary surface for the maneuver is the remaining right aileron and the flaps. The flaps and remaining aileron very nearly restore performance to nominal. The elevators have little effect in the roll axis compared with their effect on the pitch axis and are thus not significant in this maneuver.

The neglect of the KLINKS matrix during the mixer gain matrix calculation caused the sign of some deflections to be wrong for initial flight tests. For this case (failed right aileron and roll command), this resulted in the elevators responding with deflections of the wrong sign during the first flight tests for this maneuver. After this was noted, the sign of the gain values were changed. The time histories shown here depict the results after the sign of elevator deflections had been corrected.

The notable differences between the unfailed roll response and the failed aileron case, are the more sluggish roll rate and exaggerated adverse yaw after the failure.

Left Elevator Failed, Pitch Command:

The case of failed left elevator (Appendix C, Figs. C.7 - C.9) with a pitch command should reveal problems associated with the decision to alter the B matrix before performing the mixer gain calculations. However, it is difficult to determine the origin of various effects from the traces. For this case, the B matrix has zeroed the contribution of the flaps to the roll and ailerons to pitch. This change in the B matrix is done to prevent force fights between the flaps and ailerons (which result in high surface deflections which are not desired). This change is not a manifestation of the physics of the problem, however it does keep the surface deflections small without adverse results.

During initial flight tests (not shown in this report) of this condition, the sign of the right flap was incorrect. Because the flap has minimal affect for this

case, it went largely unnoticed and caused no large problems. The sign was changed after the first set of flight tests and data shown in this report is for flight tests after this correction had been made. With this sign change the right flap then deflects upward for positive pitch commands improving the pitch performance.

The gain matrix sign changes after the first flights were based on heuristics and do not reflect the accompanying changes in ratios that the mixer algorithm would have generated by inclusion of KLINKS. However, the performance of the URV shown by the strip charts is rather good. Nearly unfailed performance is attained in pitch without significant degradation in other system states.

If the URV had control surfaces which were more evenly multi-axis effective, the sign variations would have been noted during ground test. The decoupled nature of the control surfaces meant that the non-primary control surface deflections were comparatively small and unnoticed. This same decoupled control surface characteristic also meant that these sign errors had little effect on the system performance during flight.

The end result of zeroing the flap and aileron B matrix entries, for roll and pitch respectively, is to greatly reduce the effect of the flaps on the URV performance. This can be seen by observing the values of the Kie matrix for flaps in Eqn. A.21.

6.2 FDI Evaluations

The procedure for testing the FDI was to emulate failure by commanding the "failed" surface to zero deflection angle during flight and observing the performance of the FDI in terms of time to detect the failure. However, the maneuvers for testing the FDI were not regimented to include pitch, roll and yaw doublets as was the case in the test of the mixer. Because the FDI is local (looking for failures in each actuator independently), the states of the URV are not relevant. The only requirement of the maneuver, is that it attempt to excite the failed surface.

During the FDI tests, the flaps are not commanded until reconfiguration is initiated. Therefore, flap activity signifies that the FDI has found a failed surface, calculated the new gain matrix and initiated the reconfiguration. Also, the sample rate of the FDI is 28.5 Hz. The FDI requires a minimum of three sample

periods (about 100ms) to detect a failure, while the mixer gain calculation takes about 70ms. Therefore, the shortest amount of time possible for the FDI and reconfiguration is about 170ms.

Failed Left aileron:

A representative set of time histories for the case of failed aileron are shown in Appendix C, Figs. C.14 - C.17. In these traces, the introduction of the failure of the left aileron is marked by the initiation of the command (from the failure emulator) of left aileron to zero degrees. The commands during these traces are in response to a pitch and roll command. The time for detection is equal to the difference between the beginning of the activity of the flaps and the "failure" command to the aileron, minus the gain matrix calculation time (about 70ms).

The total SRFCS time is difficult to accurately determine from the strip chart recordings. However, this time is relatively small and appears to be about 200ms. This would indicate about 130ms for the FDI time. No appreciable transients appear since the URV dynamics are much too slow to be affected by this small detection time. For the most part, the flaps are commanded to fill the roll of the failed aileron. The somewhat oscillatory roll and pitch rate can be attributed to the fact that most of the recording was taken during a banking turn. The elevators are called upon to generate pitch and appear to be unaffected by the failure. The near nominal performance of the elevators is to be expected since their gains change very little as a result the reconfiguration.

Failed Right Elevator:

Appendix C, Figs. C.18 - C.22, show a typical right elevator failure time history plots. Like the previous FDI case, this failure is inserted during a rather large maneuver. The URV is being driven by large pitch and yaw commands when this failure is inserted. The relatively large deflection angles needed to respond to these commands cause the FDI system to detect quickly. From the strip charts, the detection time is measured as the difference between the time of initiation of the simulated failure (the zero command to the right elevator) and the time that the flaps begin to respond, less the gain calculation time. Again, the time to detect the failure appears to be between 100ms and 200ms.

The elevator recordings show that both elevators were beginning to decrease in magnitude, since the desired maneuver was nearly complete when the failure occurred. The failure forced the left elevator to respond to the feedback error immediately. As soon as the detection occurs, the flaps begin to aid the ailerons in reducing the roll rate (since the roll axis was not commanded).

The importance of reconfigurable flight controls depends on severity of surface failures/battle damage, the dynamics of aircraft, and the flight condition of the aircraft. For some aircraft, impairments require immediate corrective action. However, the dynamics of the URV are relatively slow and the pilot is able to fly the aircraft with failed surfaces. Therefore FDI response time is not critical.

There is a trade-off between the detection threshold levels and the P_{fa}. To reduce the P_{fa}, the thresholds must be large. This also means the surfaces must have large deflections to trip the FDI.

During steady flight, a failure (on any aircraft) may not be immediately detected if the surface is lightly loaded. There must be activity for detection to occur. Thus, those failures which are not immediately detected are not necessarily detrimental to system performance.

7.0 Conclusions

The URV flight test of a Self-Repairing Flight Control System (SRFCS) coded in Ada, was one of the first tests of an air vehicle with part of its inner loop control system coded in Ada, and one of the first flight tests of a Control Reconfiguration System. This project provided a basis for understanding the pros and cons of both SRFCS and Ada coded flight control.

The FDI worked well for large maneuvers due to the large signal to noise ratio. However, the thresholds had to be set large to prevent false alarms. Because the thresholds were large, the maneuvers used to test the FDI had to be large also. Therefore, to show the FDI working, it was started while the URV was aggressively maneuvering. The need for high thresholds can be attributed to a variety of difficulties including; 1) inadequacy of the actuator model dynamics, 2) improper sampling rate used to digitize the actuator model, and 3) asynchronous operation of the ground based SRFCS functions and the on-board flight computer.

One difficulty with test of any FDI method concerns the adequacy of the failure scenarios considered, as well as the fidelity of their mechanization. For the URV flight test, the failure modes and their mechanization were chosen only because they simplified SRFCS mechanization. This failure scenario topic deserves considerably more effort in any future work.

The benign nature of the URV provided a good platform for test and demonstration of Self-Repairing Controls and the control mixer worked well for most maneuvers which were considered. However, the gains found by the mixer were not always the most logical for recovering system performance. There was a tendency for the mixer to provide gains that would cause surfaces to oppose each other and eventually saturate. However, this was primarily due to inadequate control effector modelling in the state equation over the range of states and surface deflections encountered.

The B matrix of Eqn. 3.2 is derived assuming small perturbations and linearity. This assumption is, for the most part, valid. However, for the yaw rate equation, these assumptions are largely violated. Although the constant B matrix of Eqn. 3.2 seems adequate for most of the states for the entire flight envelope, the entries in the yaw rate row change magnitude and sign during maneuvers. This was the reason for the large gains generated for the failed rudder case and resulted in the decision not to flight test the SRFCS for the failed rudder case.

The lack of a redundant yaw control surface was a problem for the URV and is typical of modern fighter aircraft. However, alternative yaw authority is often available. The real difficulty is plant identification, to determine the source of the authority, and its intelligent use by the SRFCS algorithm. The URV with flaps may have had sufficient yaw control power (throughout its envelope) from ailerons and flaps, to deal with rudder failures. The problem was identification of the yaw rate control effects of the surfaces for more than a narrow operating region, and then best using this plant information to fulfill the command without surface saturation. The identification and intelligent use of drag induced yaw authority is one of the most difficult problems involving SRFCS.

Because the control mixer used the B matrix to generate new gain matrices (after failures), the gains reflect the errors in the model representation. If the model were updated during flight, the results would be much better. However, this would require the addition of an on-line model identification routine. Also, the need to update the mixer gain matrix based upon distance to maximum deflection of each surface (to prevent saturation of surfaces) would require the pseudoinverse calculations whenever the deflections of the surfaces change enough to affect performance. In all, these requirements create a larger computational burden and increase the problem complexity beyond the budget and schedule of this first SRFCS flight test on the URV.

The task of SRFCS implementation on the URV was aided by three factors: 1) availability of "off-the-shelf" hardware, 2) the use of Ada for development, and 3) test on an unmanned vehicle.

The processing hardware available at the time of the flight test was sufficiently advanced to easily perform all the SRFCS functions without affecting the URV dynamics. The control mixer (taking about 70ms) and the FDI (taking about 100ms) were not able to work within one time step (FDI requires more than one time step in general), however the tasks were able to run fast enough to prevent noticeable effects in the time histories of the flight test.

Ada contributed during algorithm implementation primarily by assisting code complexity management. Even for a small flight test project, the number of people and software involved made hierarchical procedure support, strong typing and HOL interrupt handling very useful language characteristics.

Because this project used an unmanned aircraft, most safety issues associated with manned flight test were not applicable. This simplified all aspects

of the flight tests including hardware selection and design, software required and flight test planning.

The use of unmanned vehicles and Ada, for studying complex, safety critical algorithms, have proven useful during the Ada implemented SRFCS study. Many issues associated with SRFCS, including stability changes, pilot notification of status, failure/damage scenarios, model identification, nonlinear aircraft characteristics and computational requirements need further investigation.

8.0 References

1. "General Specification for Flight Control System", Mil-F-87242, March 1986.

2. Pigford, J. A., "Static Aerodynamic Characteristics of a Full Scale Powered Model of the AFFDL XBQM-106 Mini-RPV with and without Side Force Surfaces," AFFDL-TM-78-60-FXS, June 1978.

3. Ray, B. S., "Aerodynamic Characteristics of the XBQM-106 Unmanned Research Vehicle Modified for Flight Control Reconfiguration", AFWAL-TM-86-183-FIGC, April 1986.

4. Stifel, J. M., Dittmar, C. J. and Zampi, M. J., "Self-Repairing Digital Flight Control System Study", AFWAL-TR-88-3007, May 1988.

5. Weinstein, W. and Merchandante, R., "Control Reconfigurable Combat Aircraft", AFWAL-TR-88-3118, July 1989.

6. Caglayan, A. K., Allen, S. M. and Rahnamai K., "Failure Detection, Isolation, and Estimation for Reconfiguration of Aircraft Flight Control Systems Subjected to Actuator Failure and Surface Damage", WRDC-TR-89-3058, June 1989.

7. Wehmuller, K. and Nguyen B., "Reconfigurable Control Laws for the Control Reconfigurable Combat Aircraft Subjected to Actuator Failures and Surface Damage", WRDC-TR-89-3052, June 1989.

8. Finck, R. D., "USAF Stability and Control DATCOM",AFWAL-TR-83-3048, October 1960, Revised 1978.

9. Koogler, O. D. and Tietz, D. E., "Microprocessor-Based Digital Autopilot Development for the XBQM-106 Mini-RPV", MS Thesis, AFIT/GE/EE/78D-31, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, February 1979, AD-A064315.

10. Rattan, K. S., Study of Control Mixer Concept for Reconfigurable Flight Control System", Proceeding NAECON, May 1985, pp. 560-569.

11. Press, W. H., Flannery, B., Teulosky, S. and Vetterling, W., *"Numerical Recipes: The Art of Scientific Computing"*, Cambridge University Press, 1986.

12. Reid, J. G., *"Linear System Fundamentals"*, McGraw-Hill, 1983.

13. Matrix_X User's Guide, Version 7, 1988.

14. Ada Programming Reference Manual, ANSI/MIL-STD-1815 A, January 1983.

15. Strang, G., "*Linear Algebra and Its Applications*", Academic Press, 1980.

16. Chandler, P. R., Issues in Flight Control Design for Robustness to Failures and Damage", *International Control Conference*, 1989.

17. Gross, H. G. and Migyanko B. "Application of Supercontroller to Fighter Aircraft Reconfiguration", *American Control Conference*, June 1988.

18. "MVME133 VMEmodule 32-bit Monoboard Microcomputer User's Manual", Motorola Inc., 1986.

Appendix A
Control Mixer Gain Calculations

This appendix includes matrices and some of the $Matrix_X$ calculations used to calculate the control mixer gain matrices for left side failure scenarios. All matrices refer to left side failures. (The gain matrices for the right side failures are very similar.) The nomenclature is defined below.

Ko == the nominal mixing gain matrix
Boo == nominal control authority matrix
Bor == nominal control authority for failed rudder
Bie == failed elevator control authority matrix
Bia == failed aileron control authority matrix
Bir == failed rudder control authority matrix
Kie == failed elevator gain matrix
Kia == failed aileron gain matrix
Kir == failed rudder gain matrix

The following section of data is that which would result with no flaps and no bogus (zeroing of elements) modifications to the B matrix.

$$Ko = \begin{bmatrix} 1. & 0. & 0. \\ 1. & 0. & 0. \\ 0. & 1. & 0. \\ 0. & -1. & 0. \\ 0. & 0. & 1. \end{bmatrix} \qquad (A.1)$$

$$Boo = \begin{bmatrix} -0.0034 & -0.0034 & -0.0022 & -0.0022 & 0.0000 \\ -0.5812 & -0.5812 & -0.0481 & -0.0481 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0016 \\ 0.2455 & -0.2455 & 0.6697 & -0.6697 & 0.0554 \\ -0.0130 & 0.0130 & -0.0428 & 0.0428 & -0.1789 \end{bmatrix}$$

(A.2)

$$Bie = \begin{bmatrix} 0.0000 & -0.0034 & -0.0022 & -0.0022 & 0.0000 \\ 0.0000 & -0.5812 & -0.0481 & -0.0481 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0016 \\ 0.0000 & -0.2455 & 0.6697 & -0.6697 & 0.0554 \\ 0.0000 & 0.0130 & -0.0428 & 0.0428 & -0.1789 \end{bmatrix}$$

(A.3)

$$Bia = \begin{bmatrix} -0.0034 & -0.0034 & 0.0000 & -0.0022 & 0.0000 \\ -0.5812 & -0.5812 & 0.0000 & -0.0481 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0016 \\ 0.2455 & -0.2455 & 0.0000 & -0.6697 & 0.0554 \\ -0.0130 & 0.0130 & 0.0000 & 0.0428 & -0.1789 \end{bmatrix}$$

(A.4)

$$Bir = \begin{bmatrix} -0.0034 & -0.0034 & -0.0022 & -0.0022 \\ -0.5812 & -0.5812 & -0.0481 & -0.0481 \\ 0.2455 & -0.2455 & 0.6697 & -0.6697 \\ -0.0130 & 0.0130 & -0.0428 & 0.0428 \end{bmatrix}$$

(A.5)

Now to avoid rank deficiency, eliminate the zero column of Bie.

<> bie=bie(1:5,2:5)

$$Bie = \begin{bmatrix} -0.0034 & -0.0022 & -0.0022 & 0.0000 \\ -0.5812 & -0.0481 & -0.0481 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.0016 \\ -0.2455 & 0.6697 & -0.6697 & 0.0554 \\ 0.0130 & -0.0428 & 0.0428 & -0.1789 \end{bmatrix}$$

(A.6)

<> kie=inv(bie'*bie)*bie'*boo*ko

$$Kie = \begin{bmatrix} 2.0000 & 0.0000 & 0.0000 \\ 0.3679 & 1.0000 & 0.0000 \\ -0.3678 & -1.0000 & 0.0000 \\ -0.0307 & 0.0000 & 1.0000 \end{bmatrix}$$

(A.7)

The above Kie matrix has rows corresponding to right elevator, left aileron, right aileron and rudder.

Now, as above with the Bie matrix, the Bia matrix is modified.

<> bia=[bia(1:5,1:2),bia(1:5,4:5)]

$$Bia = \begin{bmatrix} -0.0034 & -0.0034 & -0.0022 & 0.0000 \\ -0.5812 & -0.5812 & -0.0481 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.0016 \\ 0.2455 & -0.2455 & -0.6697 & 0.0554 \\ -0.0130 & 0.0130 & 0.0428 & -0.1789 \end{bmatrix}$$

(A.8)

<> kia=inv(bia'*bia)*bia'*boo*ko

$$Kia = \begin{bmatrix} 1.0000 & 2.7153 & 0.0000 \\ 1.0000 & -2.7151 & 0.0000 \\ 0.0000 & -0.0024 & 0.0000 \\ 0.0000 & 0.0833 & 1.0000 \end{bmatrix}$$

(A.9)

The above K matrix has rows representing left elevator, right elevator, right aileron, and rudder.

Now Bir is modified. In the case of the B matrix for rudder calculations, the β row of the matrix is removed to remove singularity and maintain conformal matrices.

$$Bir = \begin{bmatrix} -0.0034 & -0.0034 & -0.0022 & -0.0022 \\ -0.5812 & -0.5812 & -0.0481 & -0.0481 \\ 0.2455 & -0.2455 & 0.6697 & -0.6697 \\ -0.0130 & 0.0130 & -0.0428 & 0.0428 \end{bmatrix}$$

(A.10)

<> bor=[bor(1:4,1:4),bor(1:4,7)]

$$Bor = \begin{bmatrix} -0.0034 & -0.0034 & -0.0022 & -0.0022 & 0.0000 \\ -0.5812 & -0.5812 & -0.0481 & -0.0481 & 0.0000 \\ 0.2455 & -0.2455 & 0.6697 & -0.6697 & 0.0554 \\ -0.0130 & 0.0130 & -0.0428 & 0.0428 & -0.1789 \end{bmatrix}$$

(A.11)

<> kir=inv(bir)*bor*ko

$$Kir = \begin{bmatrix} 1.0000 & 0.0000 & -32.5982 \\ 1.0000 & 0.0000 & 32.5982 \\ 0.0000 & 1.0000 & 11.9913 \\ 0.0000 & -1.0000 & -11.9913 \end{bmatrix}$$

(A.12)

The Kir matrix shown above has rows of left elevator, right elevator, left aileron and right aileron.

The next set of matrices that follow are those that result from adding flap effects to the B matrices and performing the calculations as above. The no-fail gain matrix does not use the flaps. Therefore the Ko matrix is

$$Ko = \begin{bmatrix} 1. & 0. & 0. \\ 1. & 0. & 0. \\ 0. & 1. & 0. \\ 0. & -1. & 0. \\ 0. & 0. & 0. \\ 0. & 0. & 0. \\ 0. & 0. & 1. \end{bmatrix} \qquad (A.13)$$

$$Bie = \begin{bmatrix} 0.0000 & -0.0034 & -0.0022 & -0.0022 & -0.0040 & -0.0040 & 0.0000 \\ 0.0000 & -0.5812 & -0.0481 & -0.0481 & -0.0660 & -0.0660 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0016 \\ 0.0000 & -0.2455 & 0.6697 & -0.6697 & 0.6221 & -0.6221 & 0.0554 \\ 0.0000 & 0.0130 & -0.0428 & 0.0428 & -0.0403 & 0.0403 & -0.1789 \end{bmatrix} \qquad (A.14)$$

<> kie=bie'*inv(bie*bie')*boo*ko

$$Kie = \begin{bmatrix} 0.0000 & 0.0000 & 0.0000 \\ 1.9603 & -0.0038 & 0.0000 \\ 5.7599 & 1.0759 & 0.0000 \\ -3.9942 & -0.9049 & 0.0000 \\ -5.3321 & -0.0359 & 0.0000 \\ 4.3947 & -0.0549 & 0.0000 \\ 0.0000 & 0.0000 & 1.0000 \end{bmatrix}$$

(A.15)

$$Bia = \begin{bmatrix} -0.0034 & -0.0034 & 0.0000 & -0.0022 & -0.0040 & -0.0040 & 0.0000 \\ -0.5812 & -0.5812 & 0.0000 & -0.0481 & -0.0660 & -0.0660 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0016 \\ 0.2455 & -0.2455 & 0.0000 & -0.6697 & 0.6221 & -0.6221 & 0.0554 \\ -0.0130 & 0.0130 & 0.0000 & 0.0428 & -0.0403 & 0.0403 & -0.1789 \end{bmatrix}$$

(A.16)

<> kia=bia'*inv(bia*bia')*boo*ko

$$Kia = \begin{bmatrix} 1.0012 & 0.1453 & 0.0000 \\ 0.9985 & -0.1313 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 \\ 0.0135 & -0.6239 & 0.0000 \\ 0.0032 & 0.8517 & 0.0000 \\ -0.0104 & -0.5205 & 0.0000 \\ 0.0000 & 0.0000 & 1.0000 \end{bmatrix}$$

(A.17)

$$Bir = \begin{bmatrix} -0.0034 & -0.0034 & -0.0022 & -0.0022 & -0.0040 & -0.0040 \\ -0.5812 & -0.5812 & -0.0481 & -0.0481 & -0.0660 & -0.0660 \\ 0.2455 & -0.2455 & 0.6697 & -0.6697 & 0.6221 & -0.6221 \\ -0.0130 & 0.0130 & -0.0428 & 0.0428 & -0.0403 & 0.0403 \end{bmatrix}$$

(A.18)

<> bor=[boo(1:2,1:7);boo(4:5,1:7)]

$$Bor = \begin{bmatrix} -0.0034 & -0.0034 & -0.0022 & -0.0022 & -0.0040 & -0.0040 & 0.0000 \\ -0.5812 & -0.5812 & -0.0481 & -0.0481 & -0.0660 & -0.0660 & 0.0000 \\ 0.2455 & -0.2455 & 0.6697 & -0.6697 & 0.6221 & -0.6221 & 0.0554 \\ -0.0130 & 0.0130 & -0.0428 & 0.0428 & -0.0403 & 0.0403 & -0.1789 \end{bmatrix}$$

(A.19)

<> kir=bir'*inv(bir*bir')*bor*ko

$$Kir = \begin{bmatrix} 0.9996 & 0.0988 & -30.7668 \\ 0.9996 & -0.0988 & 30.7668 \\ 0.0175 & 0.5086 & 2.8785 \\ 0.0175 & -0.5086 & -2.8785 \\ -0.0093 & 0.4901 & 9.0873 \\ -0.0093 & -0.4901 & -9.0873 \end{bmatrix}$$

(A.20)

The above Kir matrix has rows corresponding to left elevator, right elevator, left aileron, right aileron, left flap and right flap.

Note that in the above Kie matrix which includes the flaps, that the values in the first column are rather large in comparison to the Kie matrix which does not include flaps. This is due to the force fight between the flaps and ailerons.

The cases of failed ailerons did not zero any elements in the B matrix, therefore the values flown are the values shown above (except for some sign changes). However, the gain matrices that were flown for the elevator failure cases were derived with B matrices that had some elements zeroed. Therefore the gain matrix for the left elevator failure which flew (shown below) is different from that shown above.

$$Kie = \begin{bmatrix} 0.0000 & 0.0000 & 0.0000 \\ 1.9852 & -0.0025 & 0.0000 \\ 0.2568 & 0.9812 & 0.0000 \\ -0.4709 & -1.0179 & 0.0000 \\ -0.0011 & 0.0113 & 0.0000 \\ 0.1314 & 0.0111 & 0.0000 \\ 0.0000 & 0.0000 & 1.0000 \end{bmatrix}$$

(A.21)

The values in the above gain matrix are reduced due to the zeroing of the B matrix elements.

The gains that result from if linkage ratios are considered as part of the control matrix are the gain matrices that follow. (These matrices have zeroed B matrix values as was done for the flight tested matrices.)

$$Kia = \begin{bmatrix} 1.0029 & -0.2237 & 0.0000 \\ 0.9964 & 0.2010 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 \\ -0.0208 & -0.6475 & 0.0000 \\ -0.0048 & 0.8462 & 0.0000 \\ 0.0159 & -0.5025 & 0.0000 \\ 0.0000 & 0.0000 & 1.0000 \end{bmatrix}$$

(A.22)

$$Kie = \begin{bmatrix} 0.0000 & 0.0000 & 0.0000 \\ 1.9656 & 0.0039 & 0.0000 \\ -0.0713 & 0.9814 & 0.0000 \\ 0.3899 & -1.0177 & 0.0000 \\ -0.0550 & 0.0111 & 0.0000 \\ -0.1390 & 0.0110 & 0.0000 \\ 0.0000 & 0.0000 & 1.0000 \end{bmatrix}$$

(A.23)

Appendix B
Simulation Time Histories

This appendix shows time histories for the nominal (without surface failures) URV for pitch, roll, and yaw commands using the linear small perturbation model given in Eqn. 2.1 and matrices defined in Eqns. 3.1 and 3.2. The inputs to the model are unit steps and the non-unity magnitudes of the output variables are due to the system gain. Each set of traces indicates response to a single step input while the other two commands are zero and only those output variables that displayed activity are included in these figures.

Nominal Pitch Response

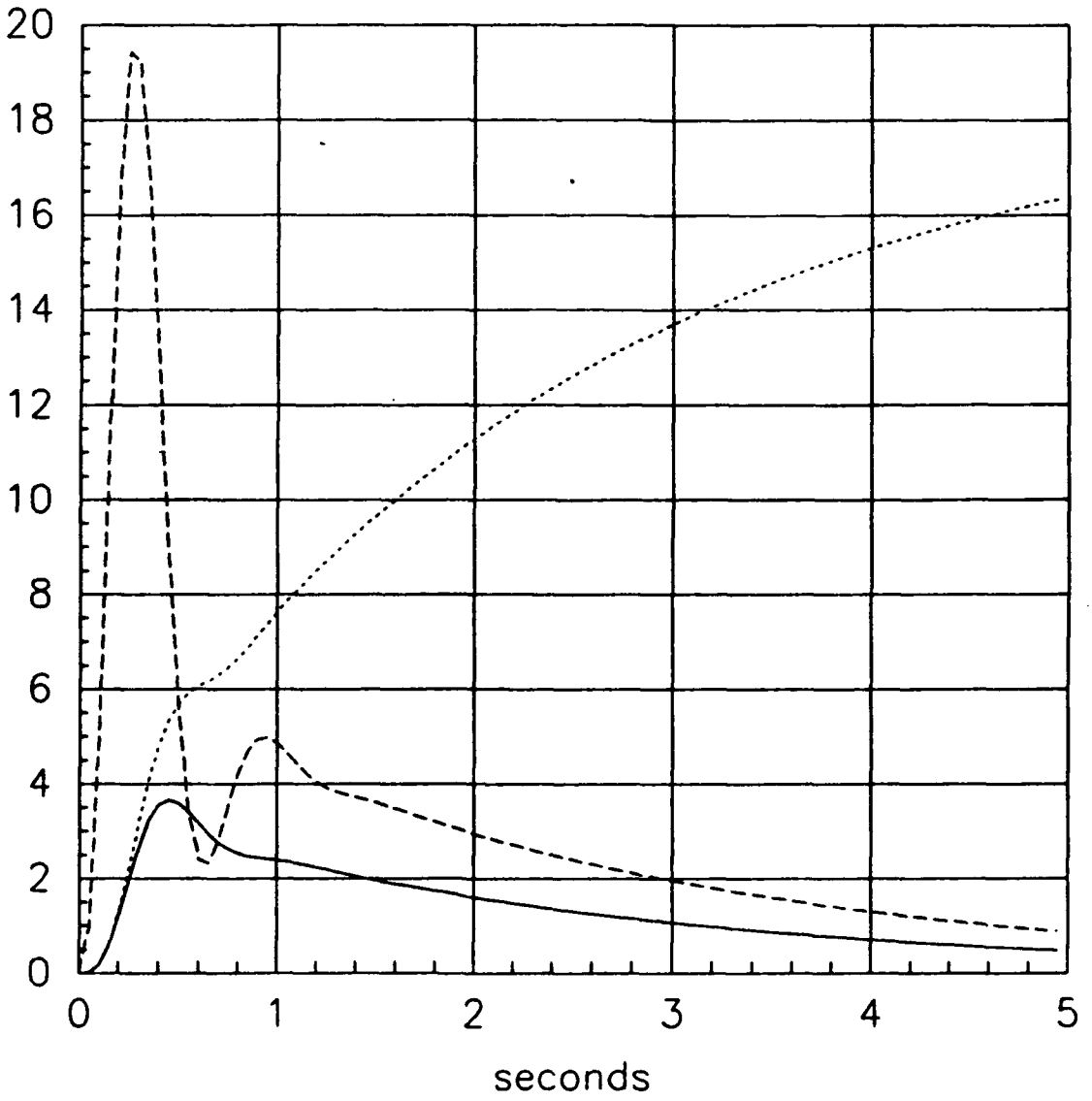
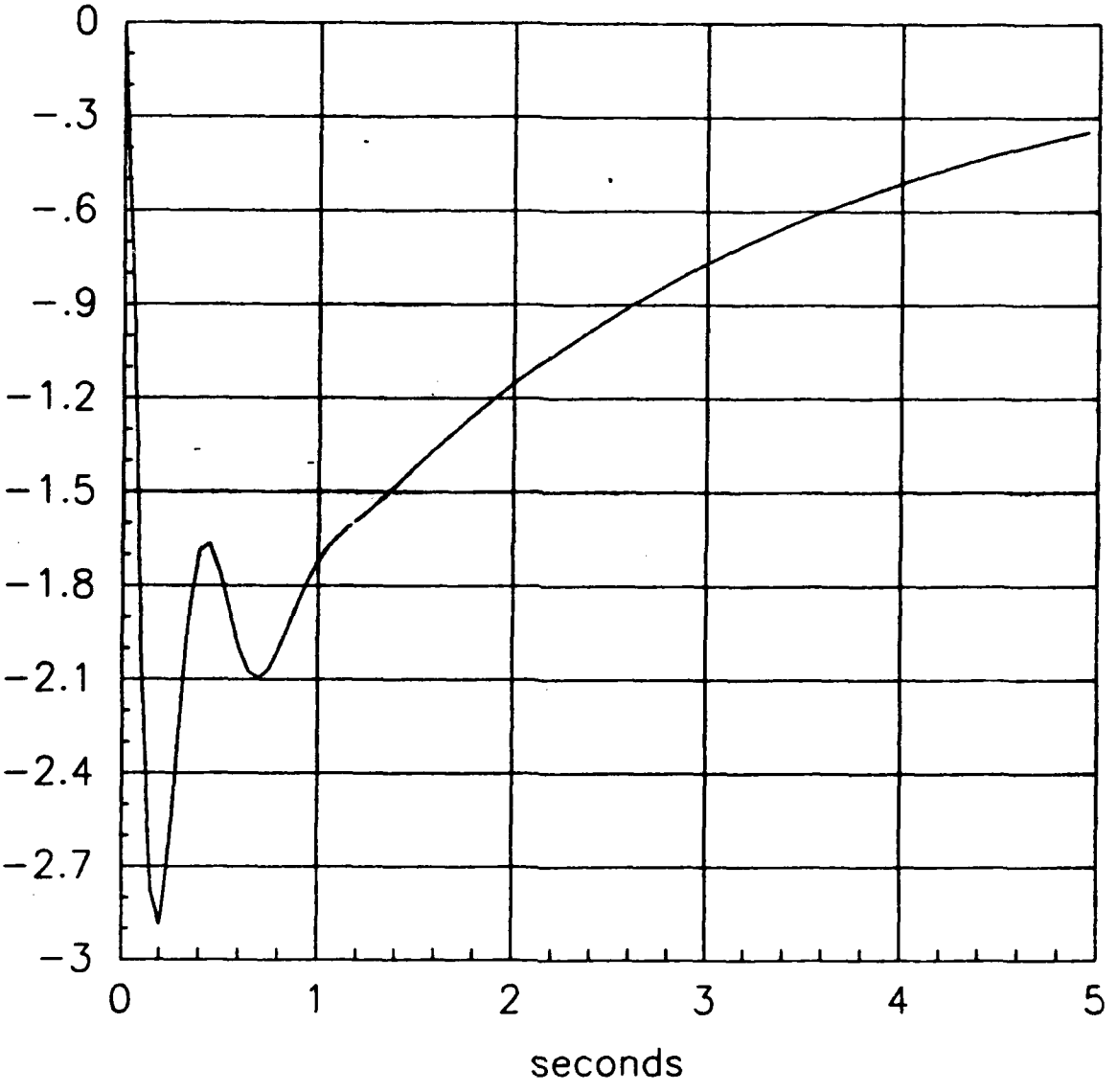


Fig. B.1

Nominal Pitch Response



Left/Right Elevator

(degrees)

Fig. B.2

Nominal Roll Response

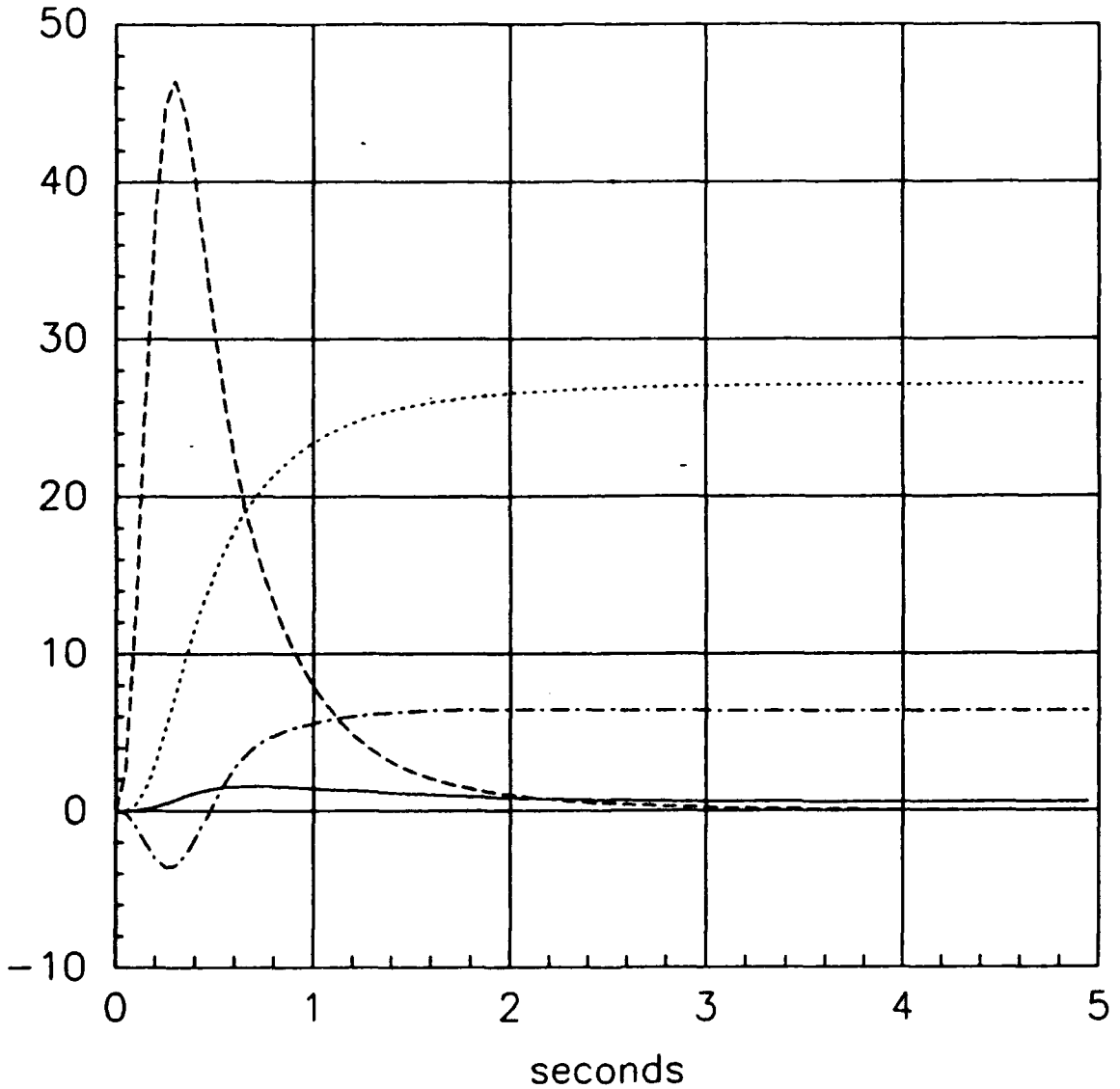
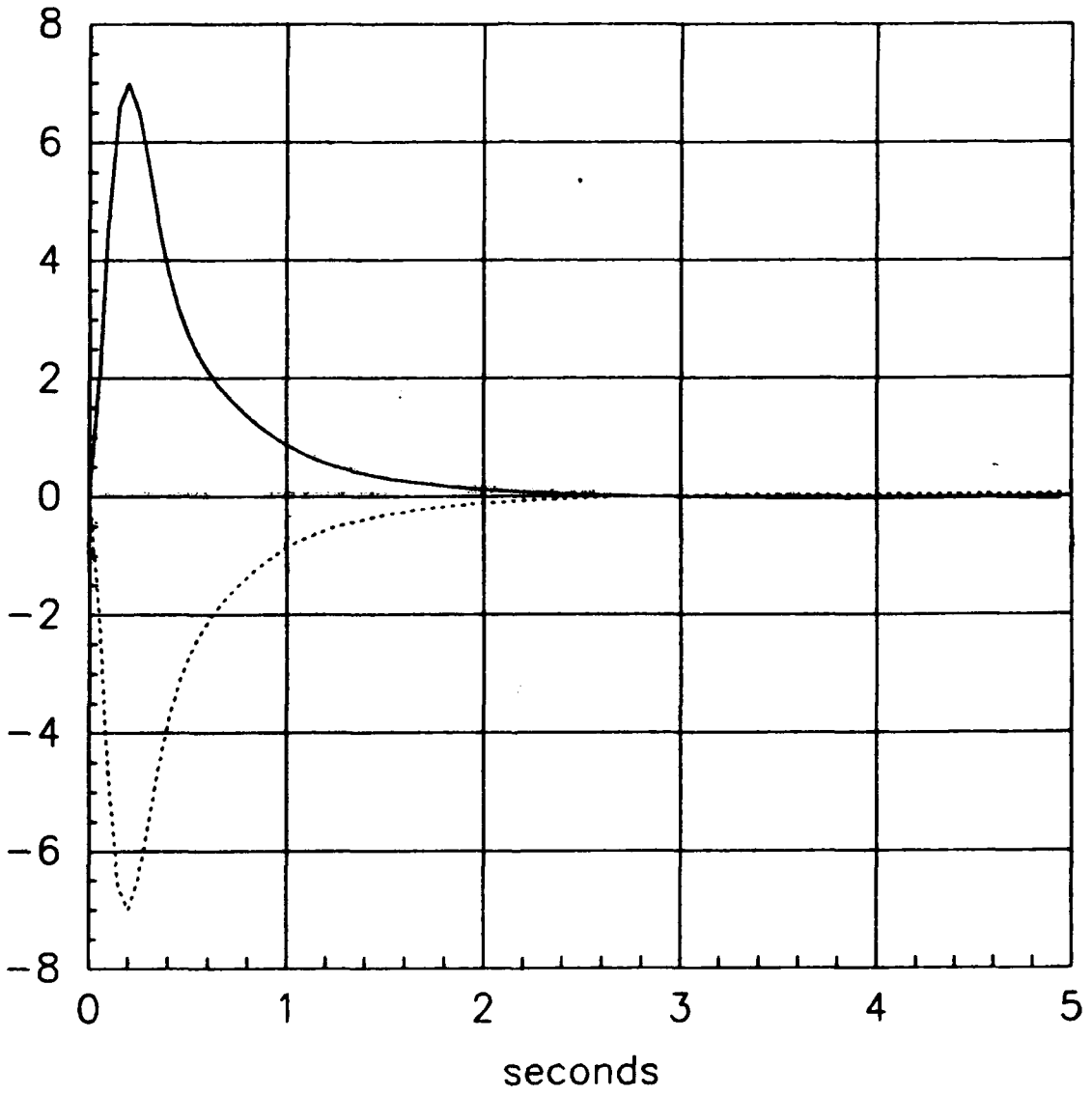


Fig. B.3

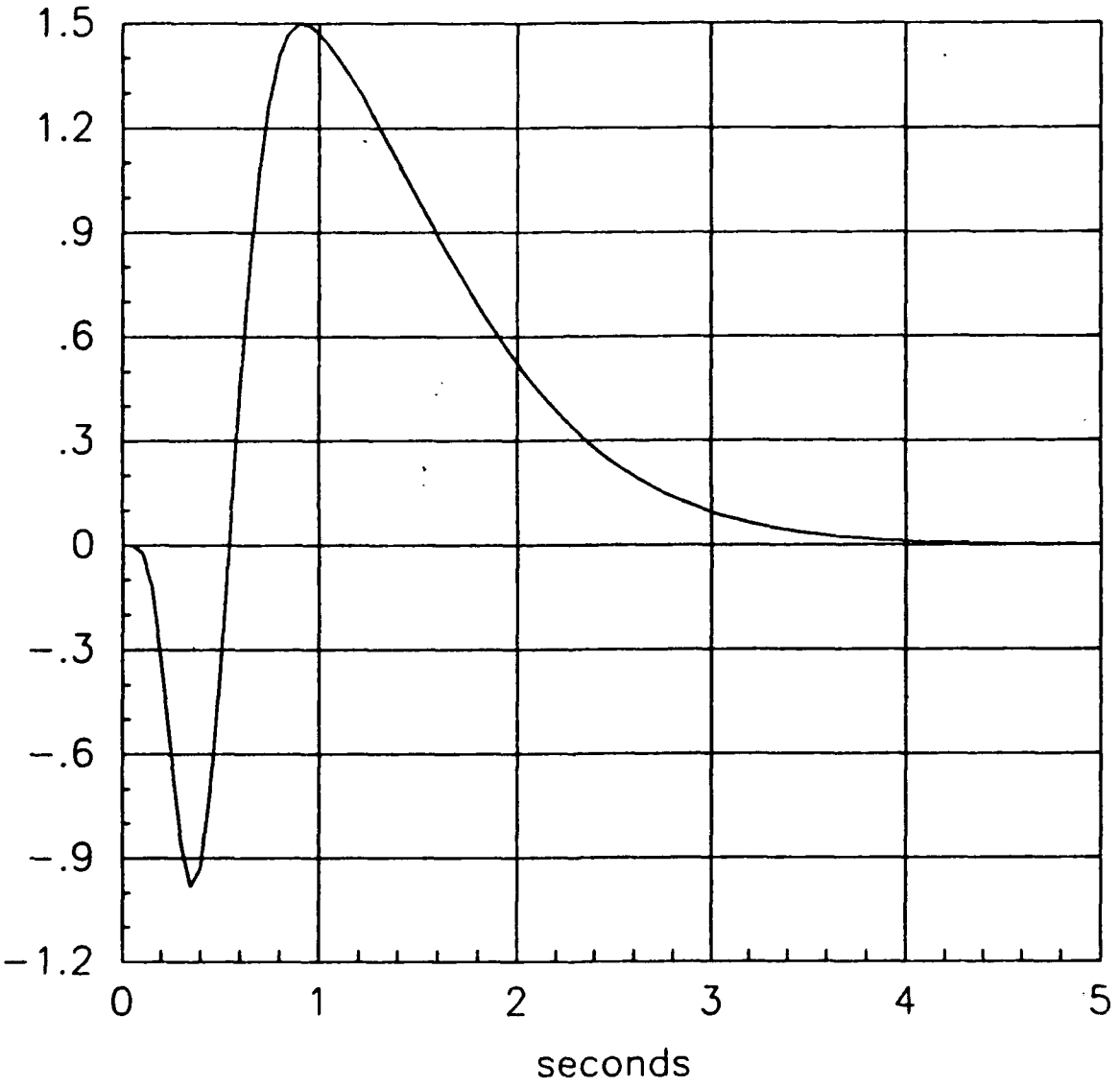
Nominal Roll Response



Left/Right Aileron
(degrees)

Fig. B.4

Nominal Roll Response



Rudder
(degrees)

Fig. B.5

Nominal Yaw Response

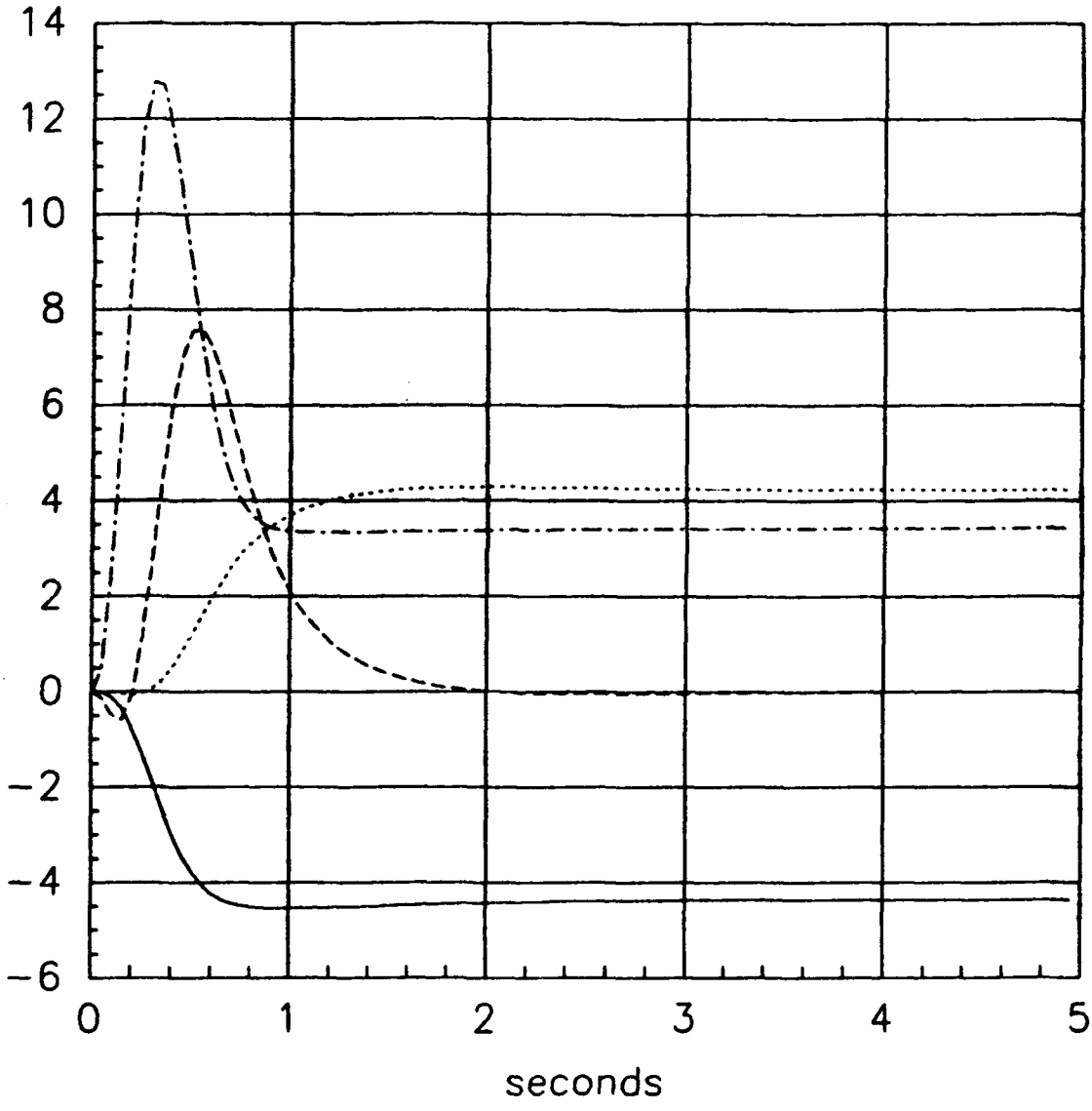


Fig. B.6

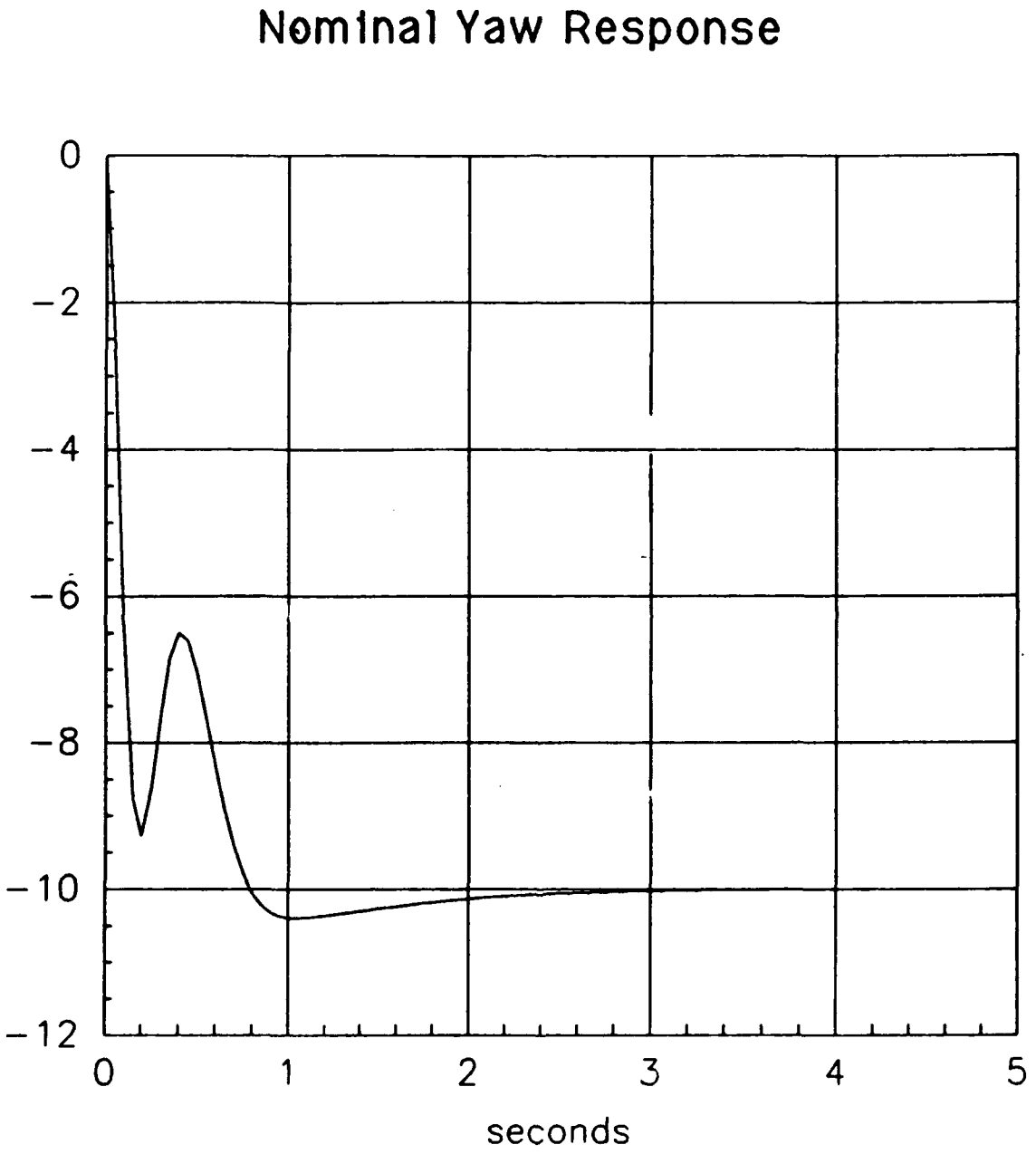
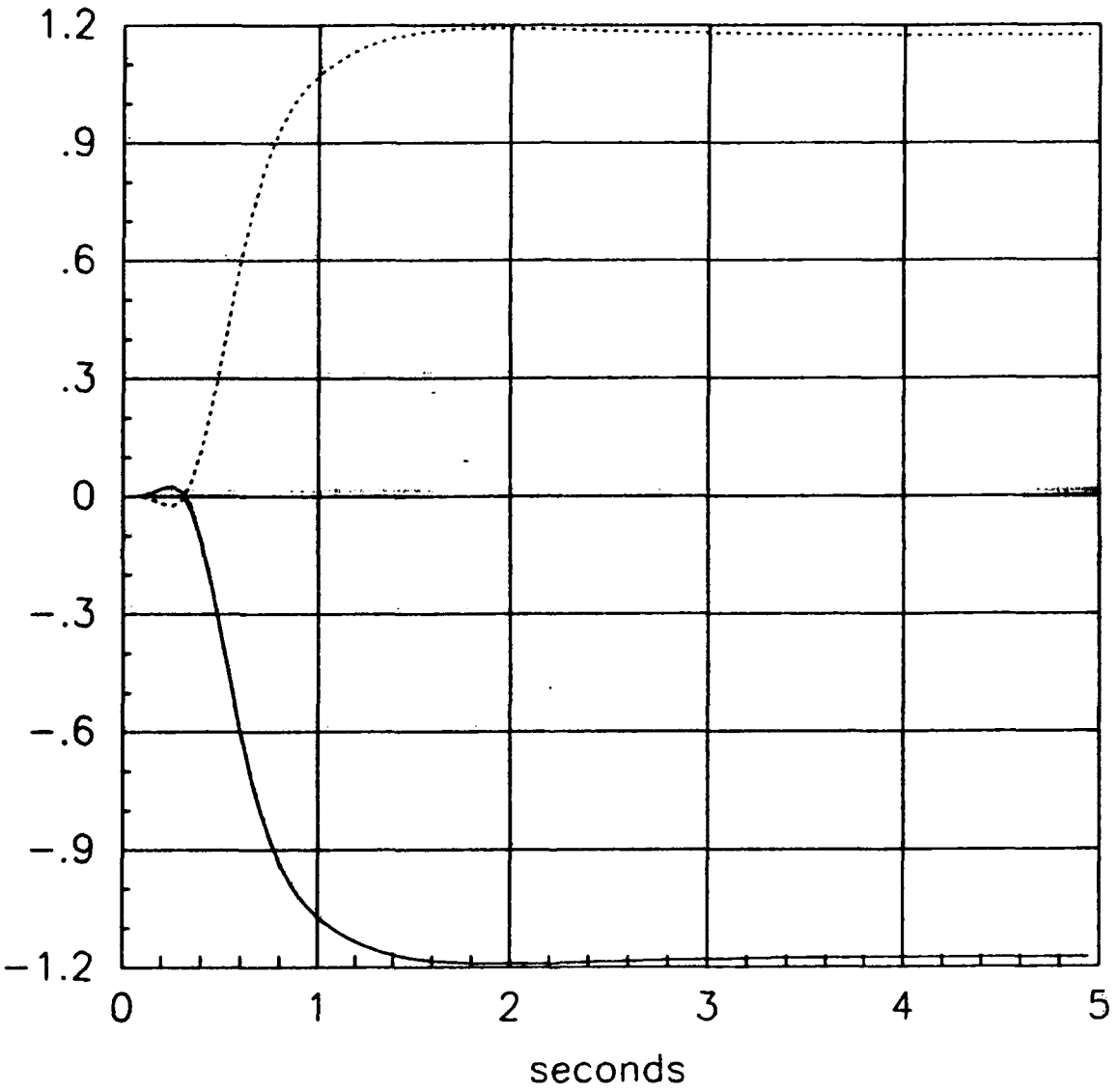

Nominal Yaw Response
0
-2
-4
-6
-8
-10
-12
0
1
2
3
4
5
seconds
Rudder
(degrees)

Fig. B.7

Nominal Yaw Response



Left/Right Aileron
(degrees)

Fig. B.8

Appendix C
Flight Data Time Histories

This appendix shows a few of the time history strip charts taken during the flight test to record the activity of the states and surfaces of the URV. Only traces for states and surfaces which are significant (in terms of SRFCS) are shown. The charts are labeled to note the state or surface represented, the direction of increasing time, the time scale, the units, and denote a point in time which is common for each plot of a particular maneuver (t_0).

No Fail, Pitch

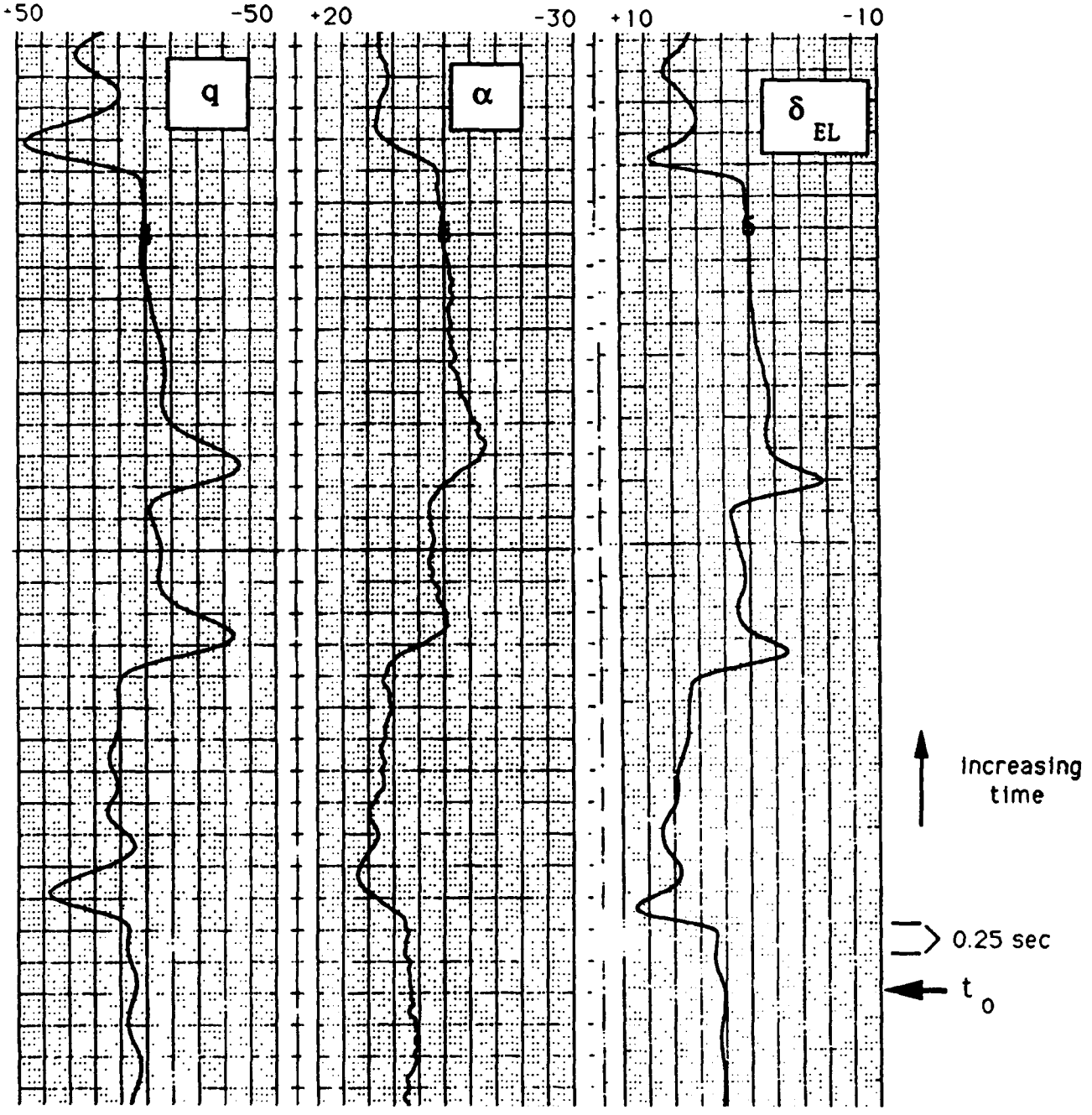


Fig. C.1

No Fail Pitch

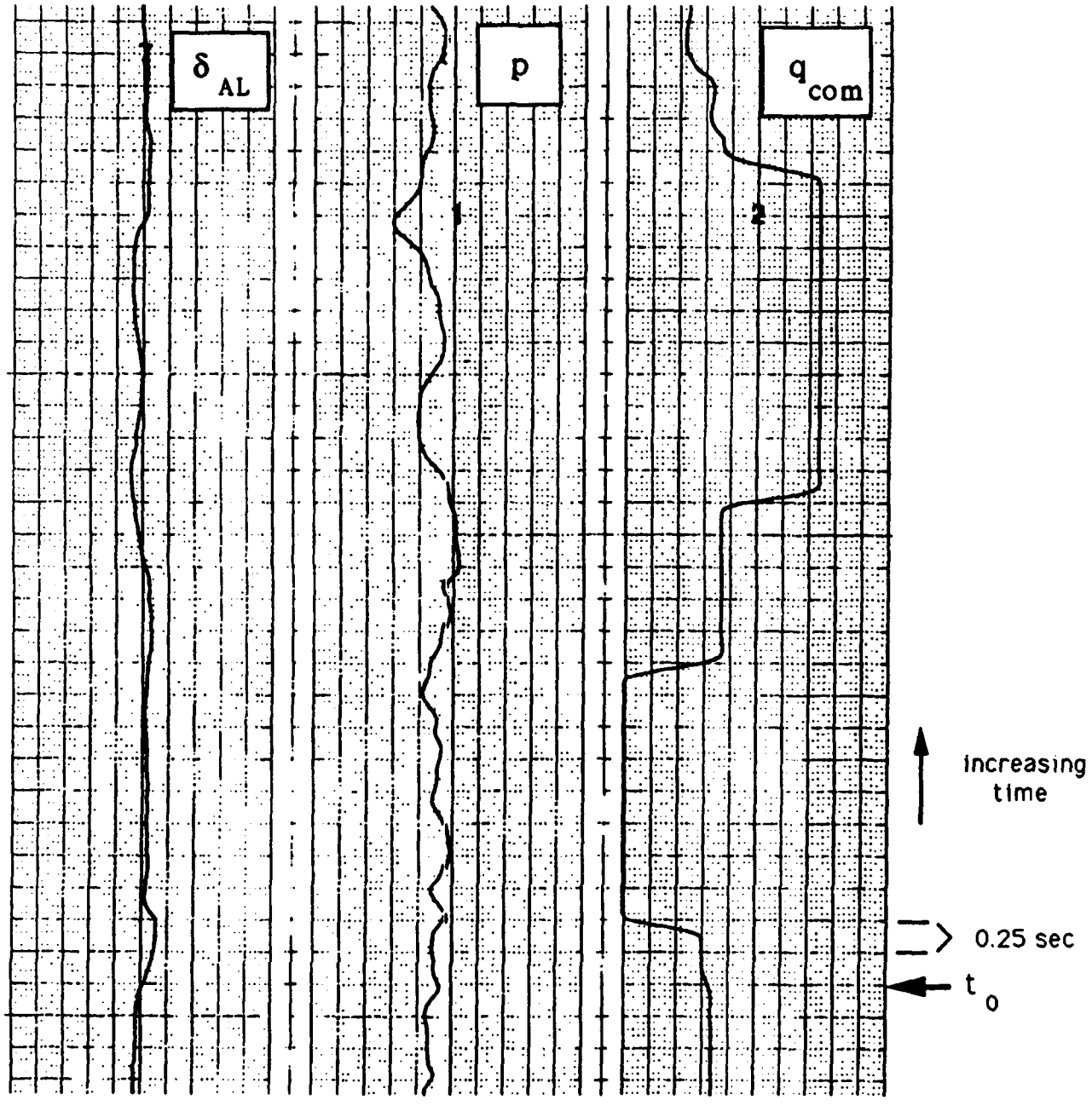


Fig. C.2

No Fail, Roll

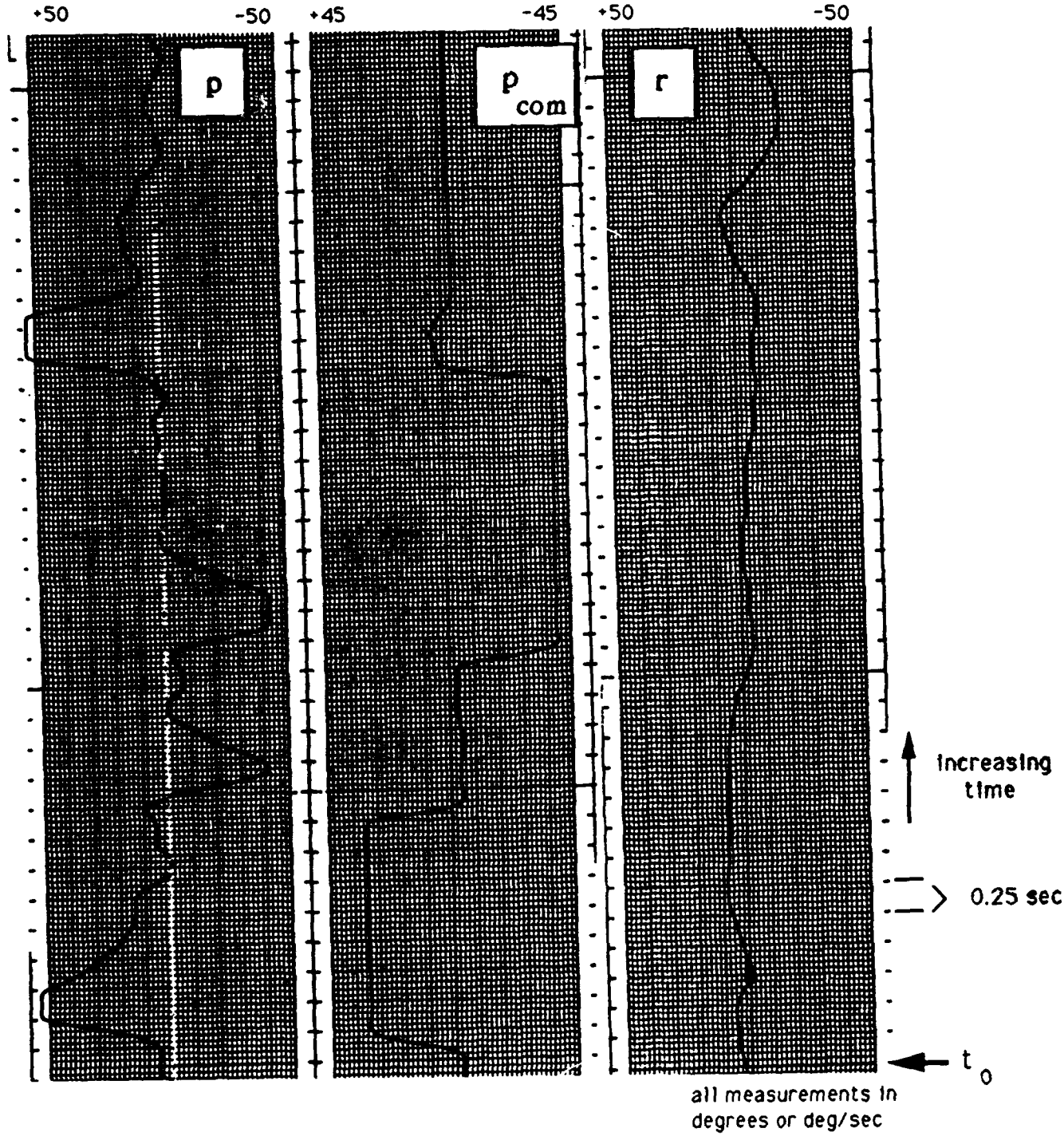


Fig. C.3

No Fail, Roll

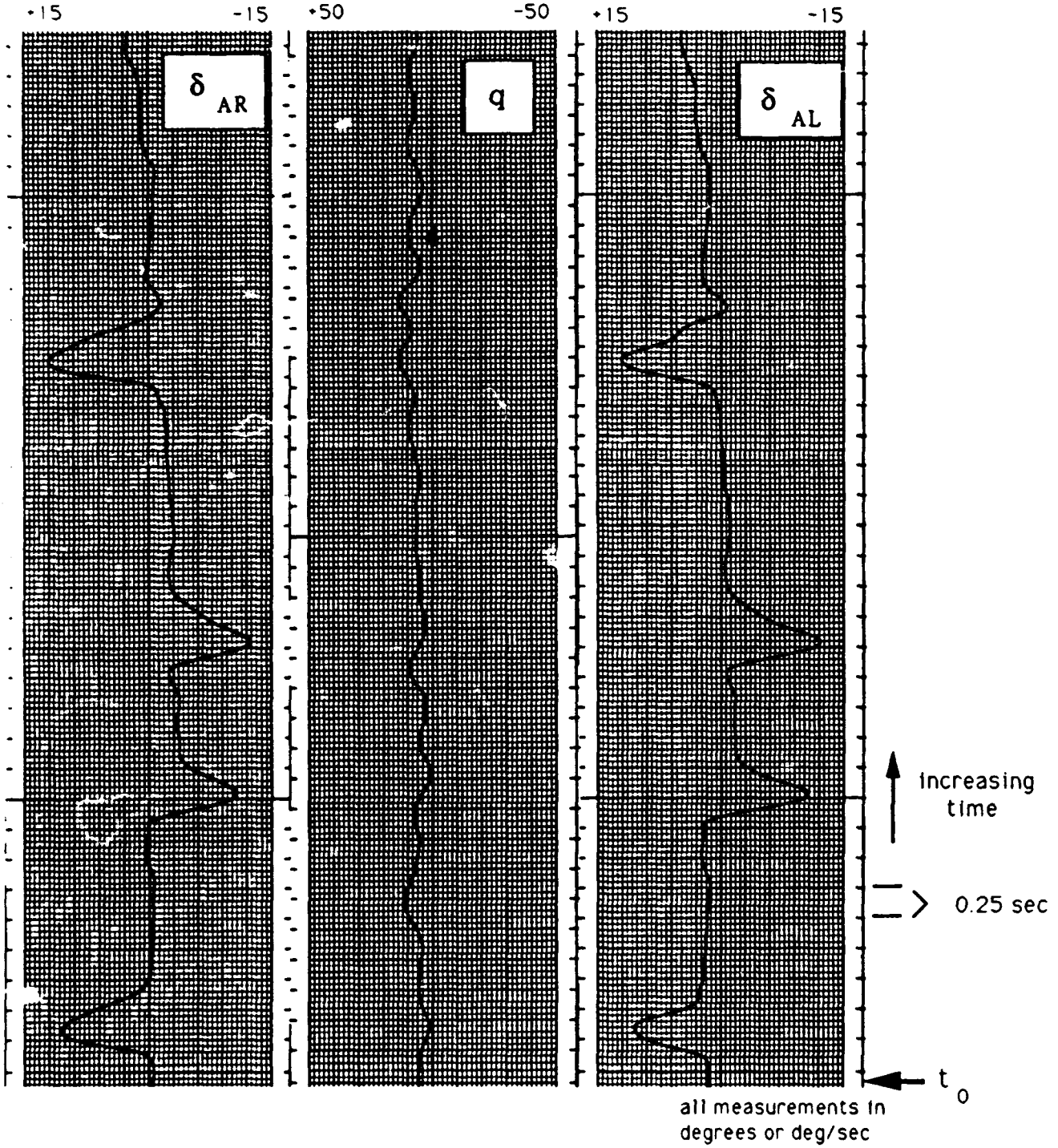


Fig. C.4

No Fail, Yaw

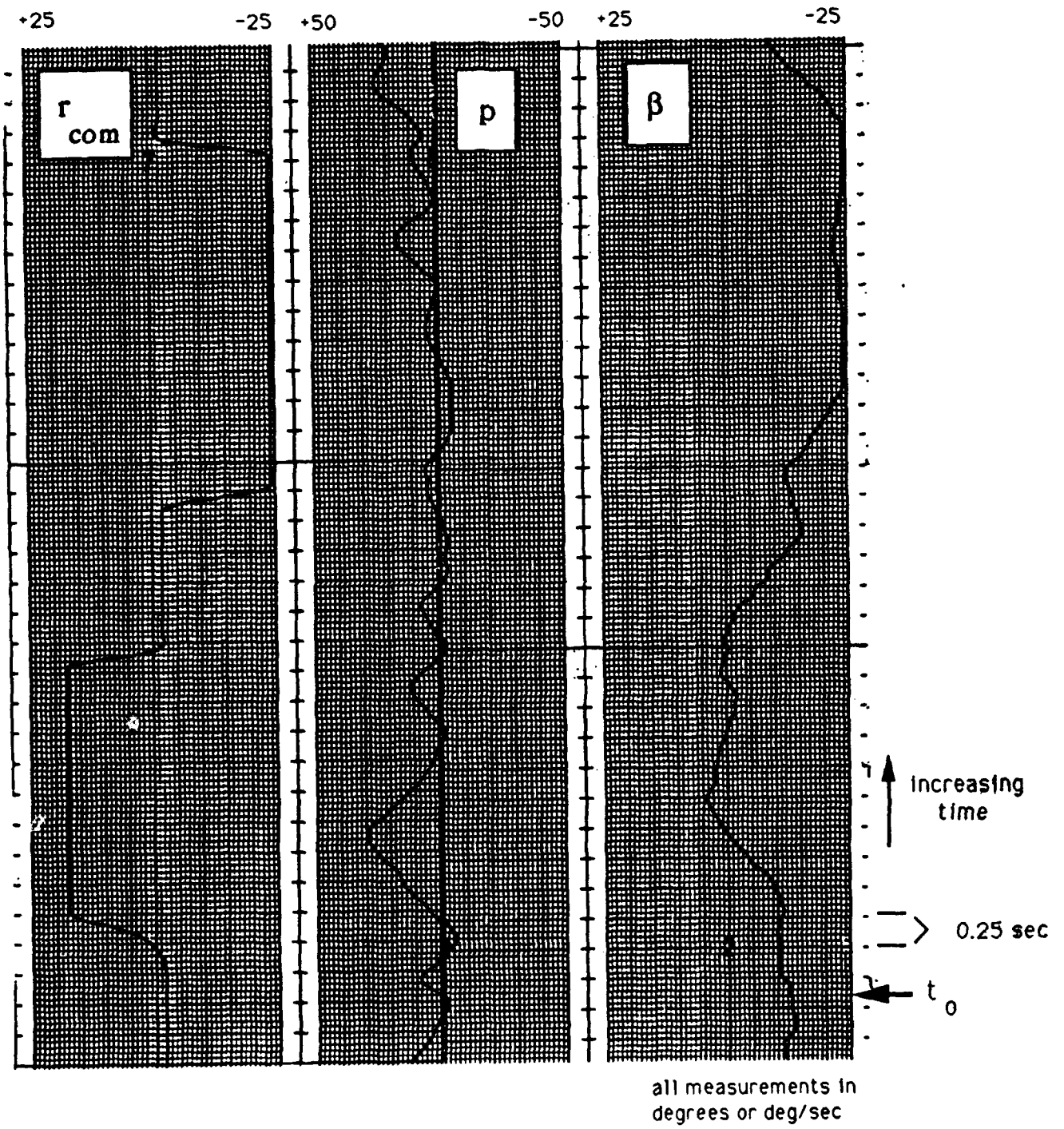


Fig. C.5

No fail, Yaw

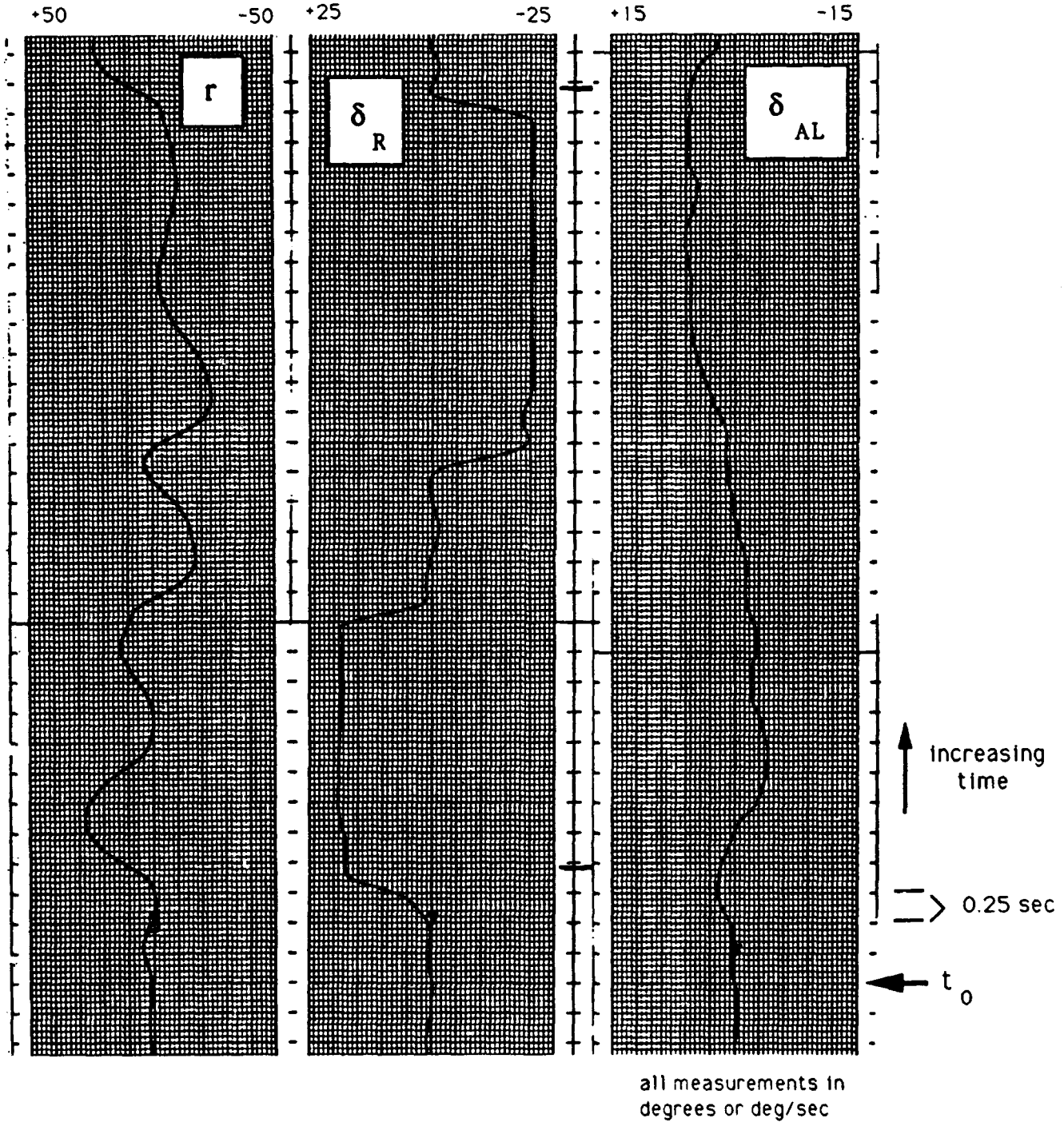


Fig. C.6

Left Elevator Failed, Pitch

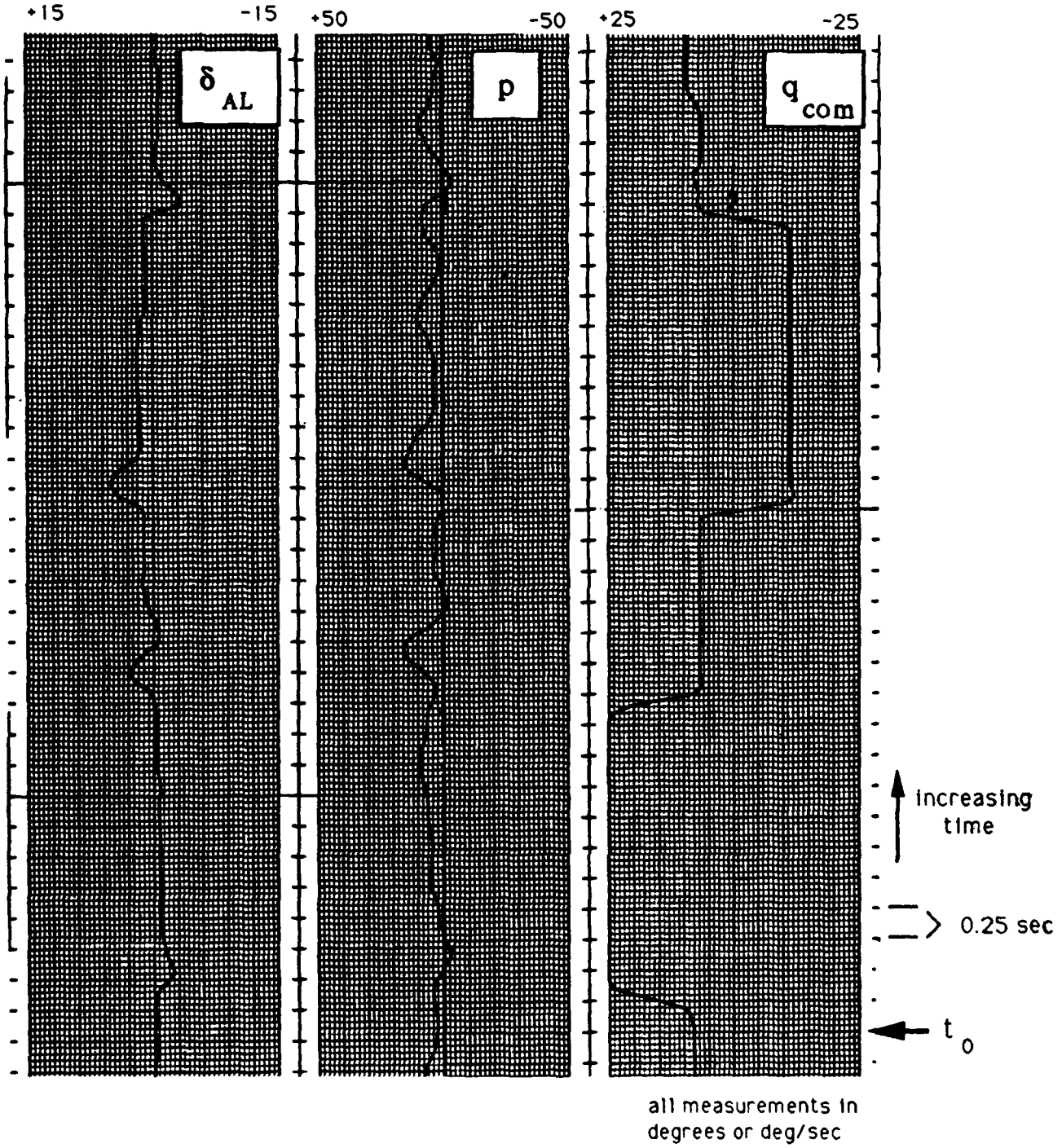


Fig. C.7

Left Elevator Failed, Pitch

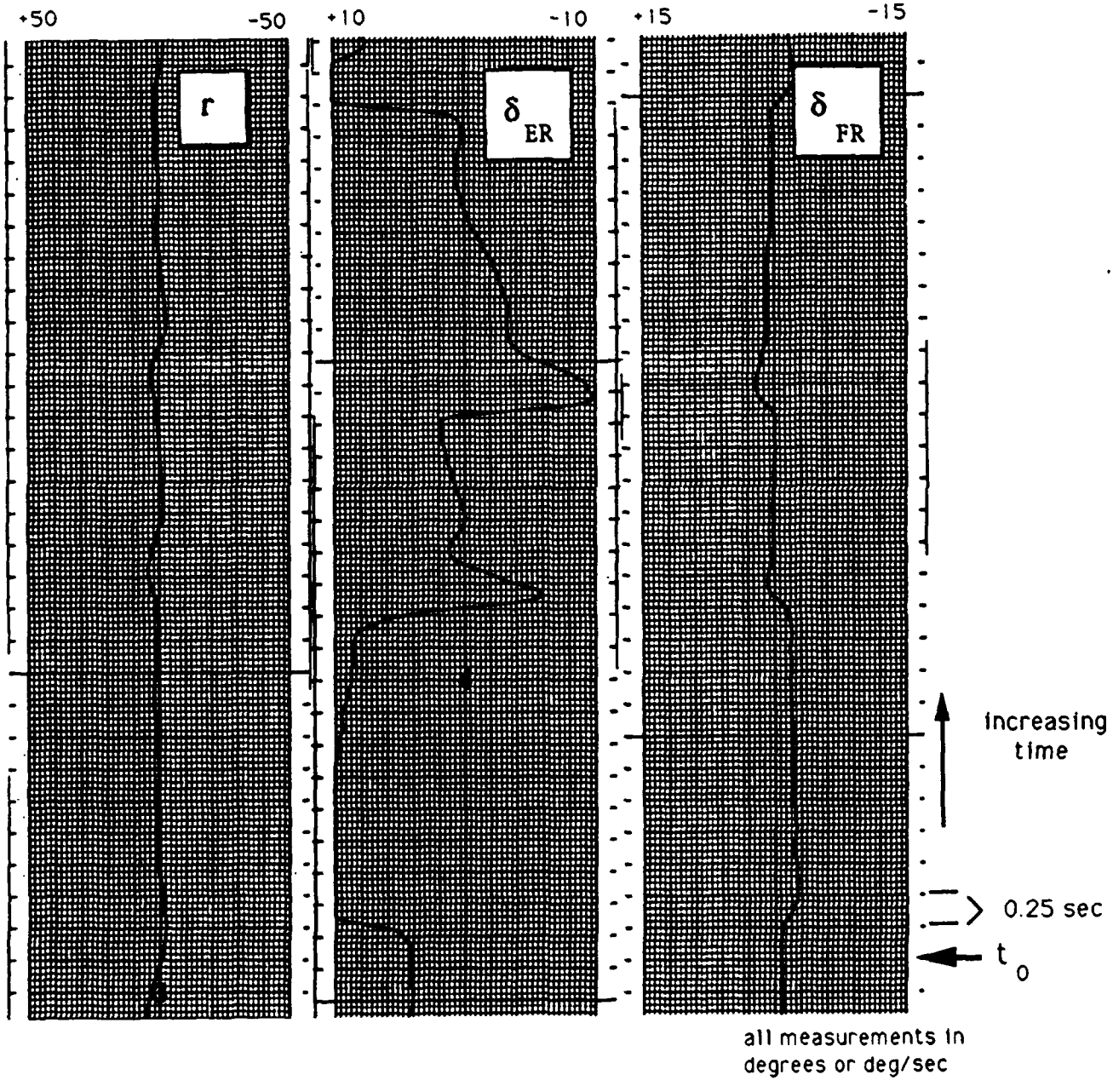


Fig. C.8

Left Elevator Failed, Pitch

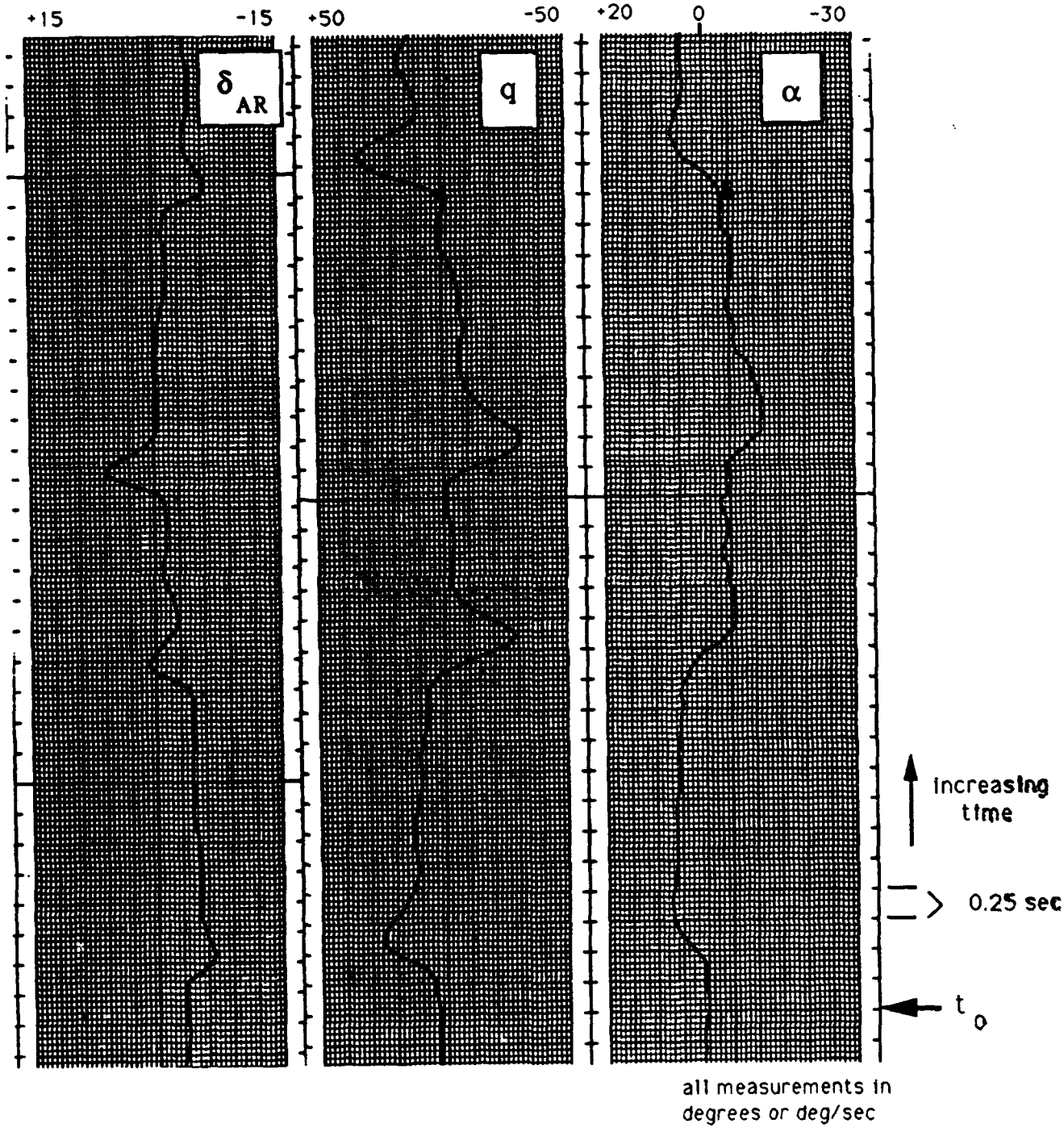


Fig. C.9

Right Aileron Failed, Roll

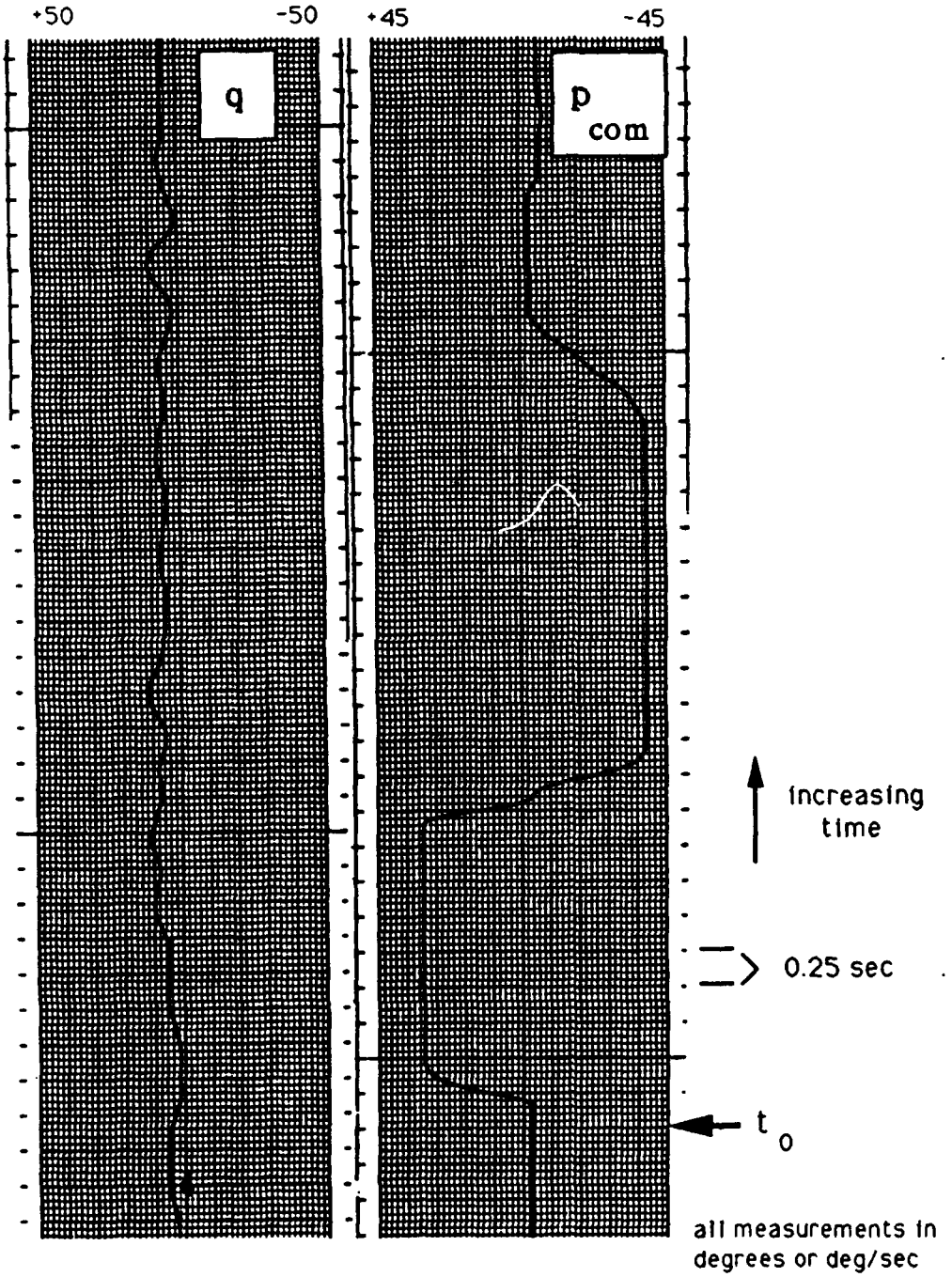


Fig. C.10

Right Aileron Failed, Roll

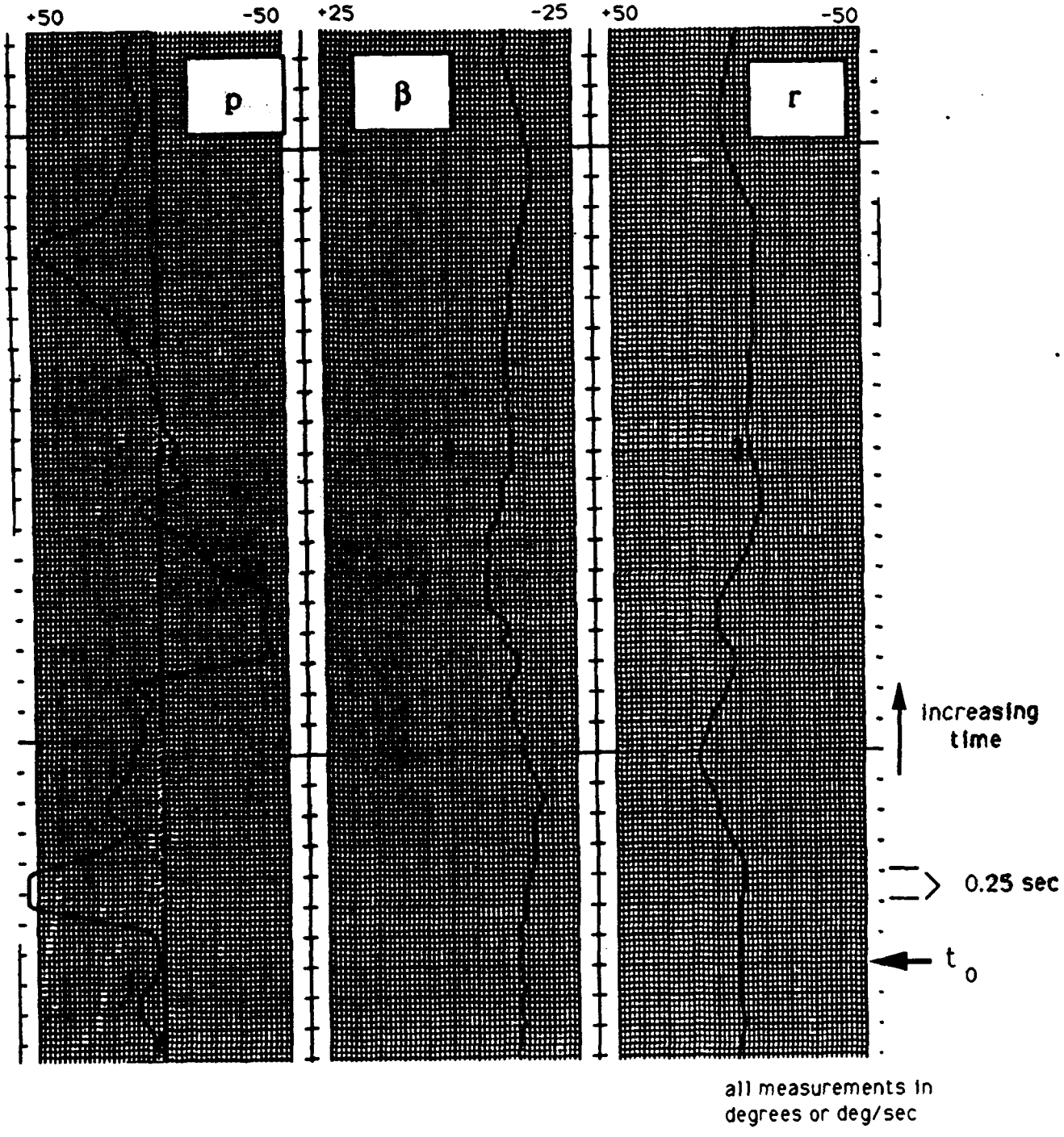


Fig. C.11

Right Aileron Failed, Roll

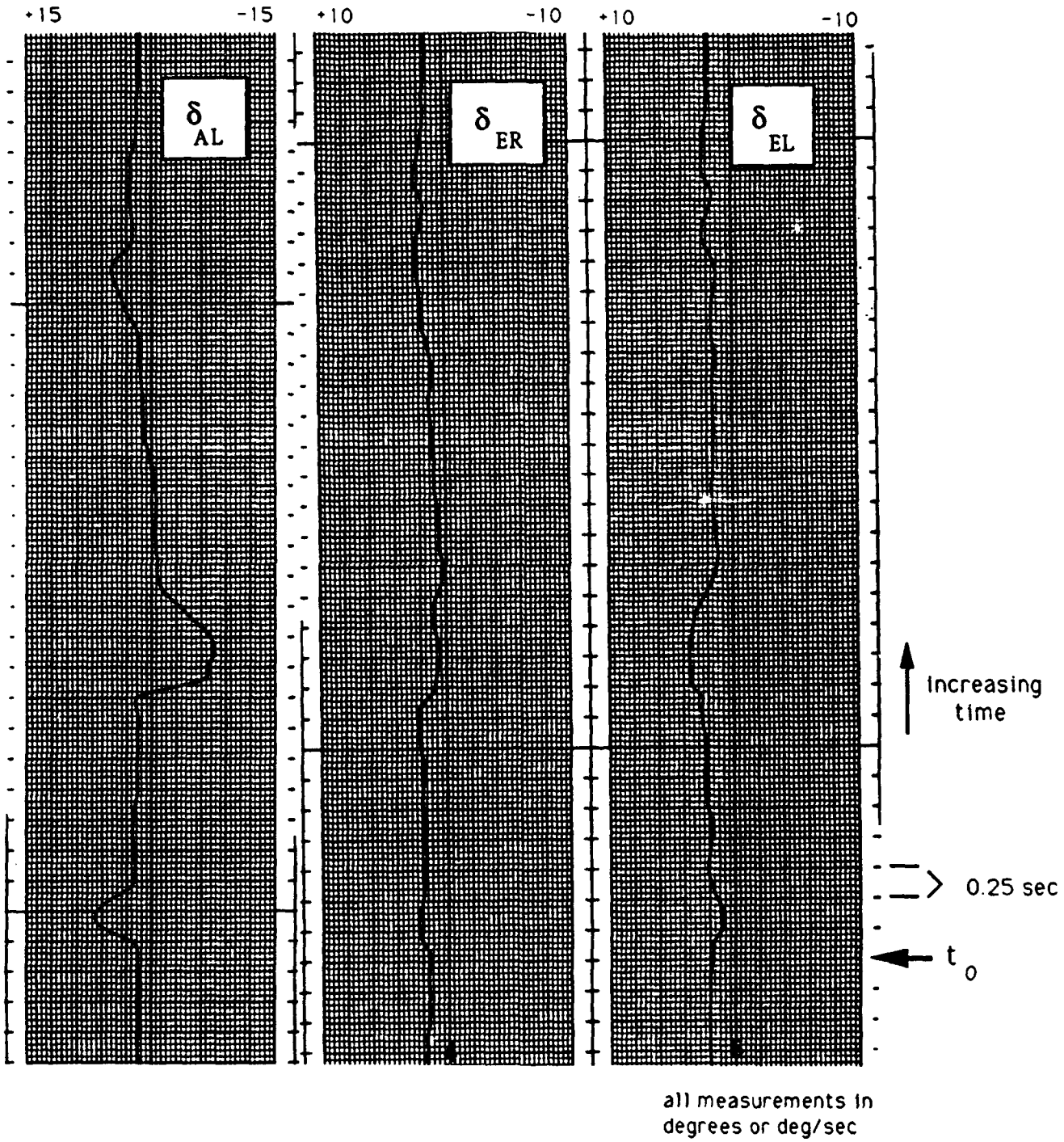


Fig. C.12

Right Aileron Failed, Roll

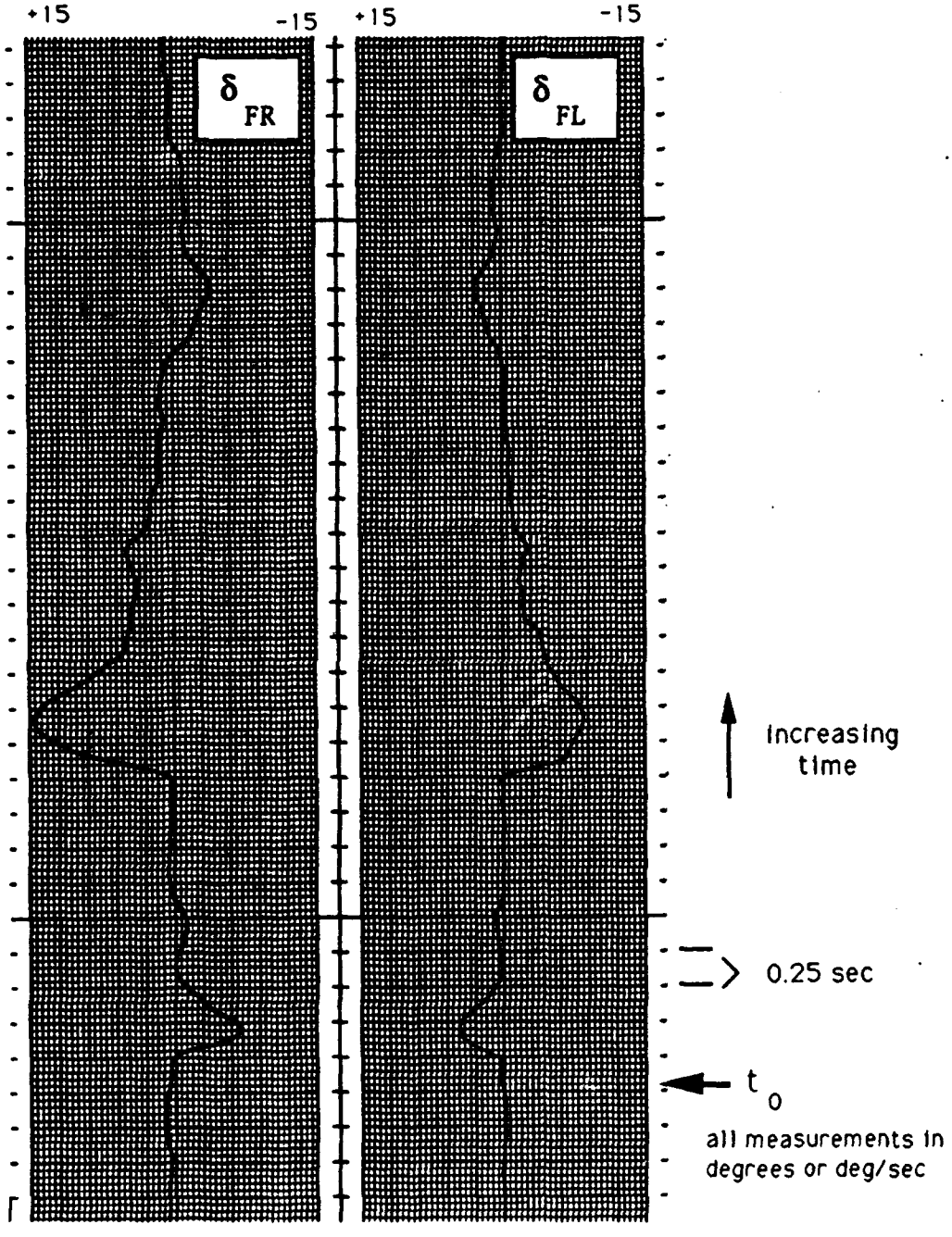


Fig. C.13

FDI of Left Aileron Failure

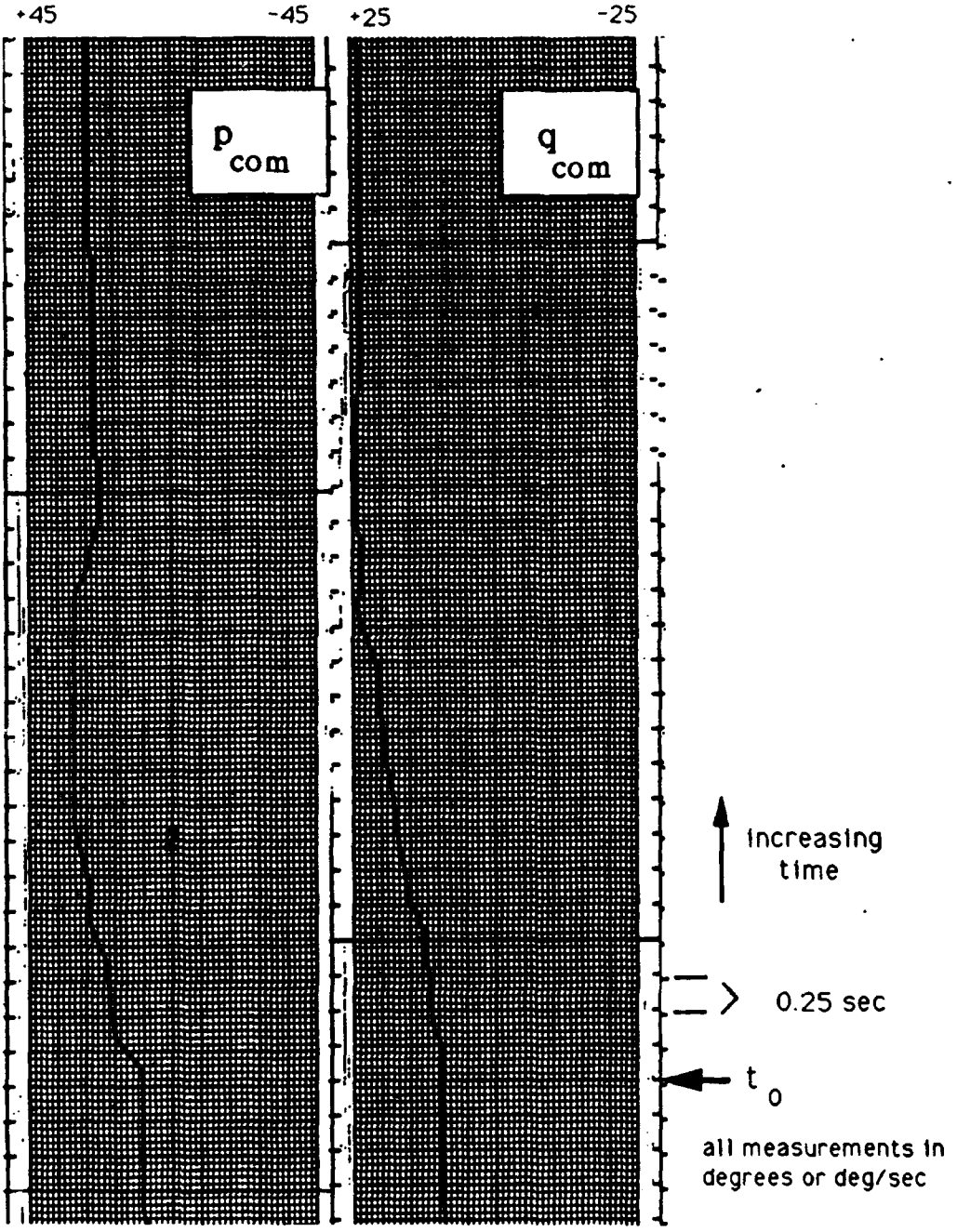


Fig. C.14

FDI Left Aileron Failure

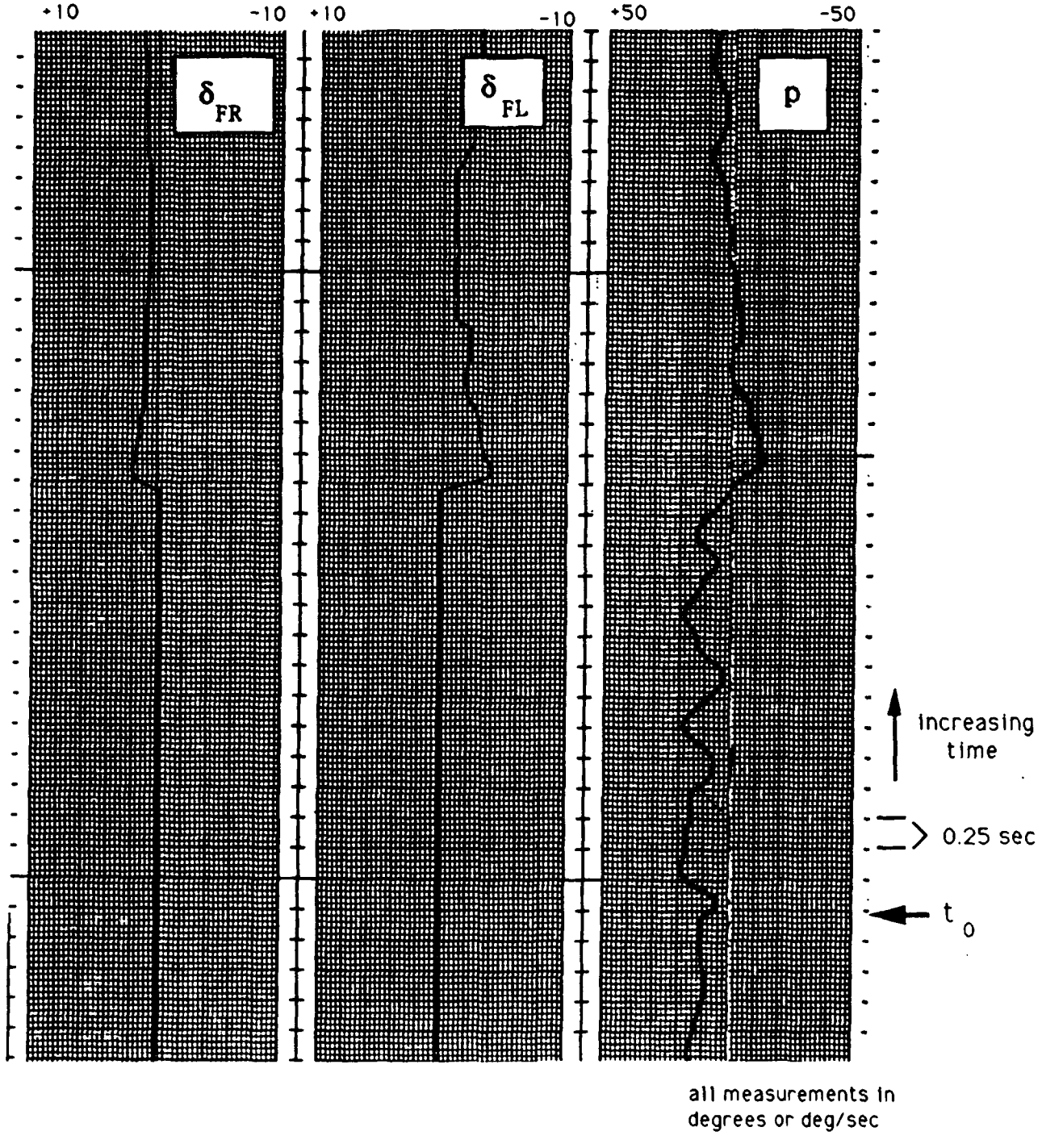


Fig. C.15

FDI of Left Aileron Failure

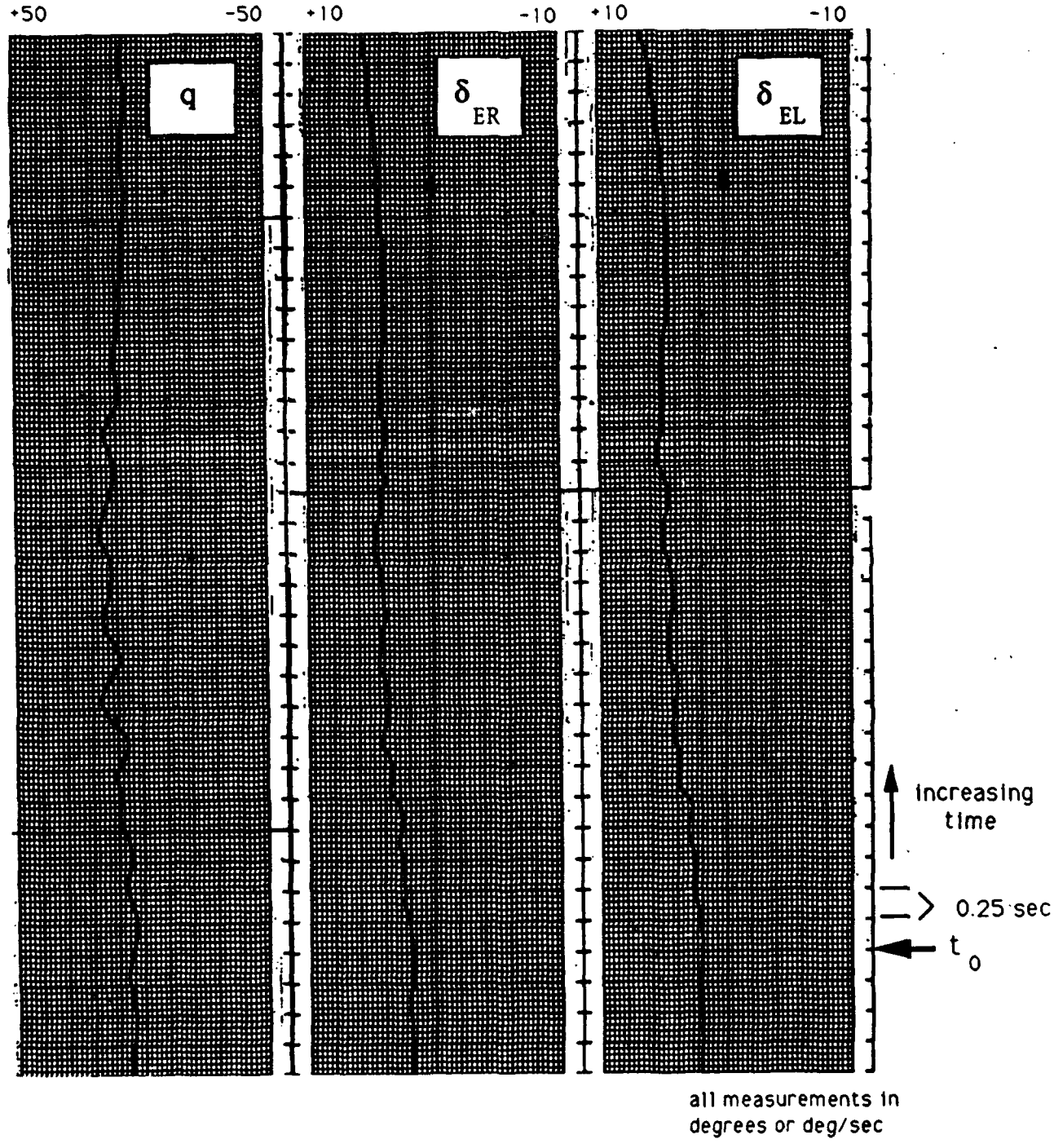


Fig. C.16

FDI of Left Aileron Failure

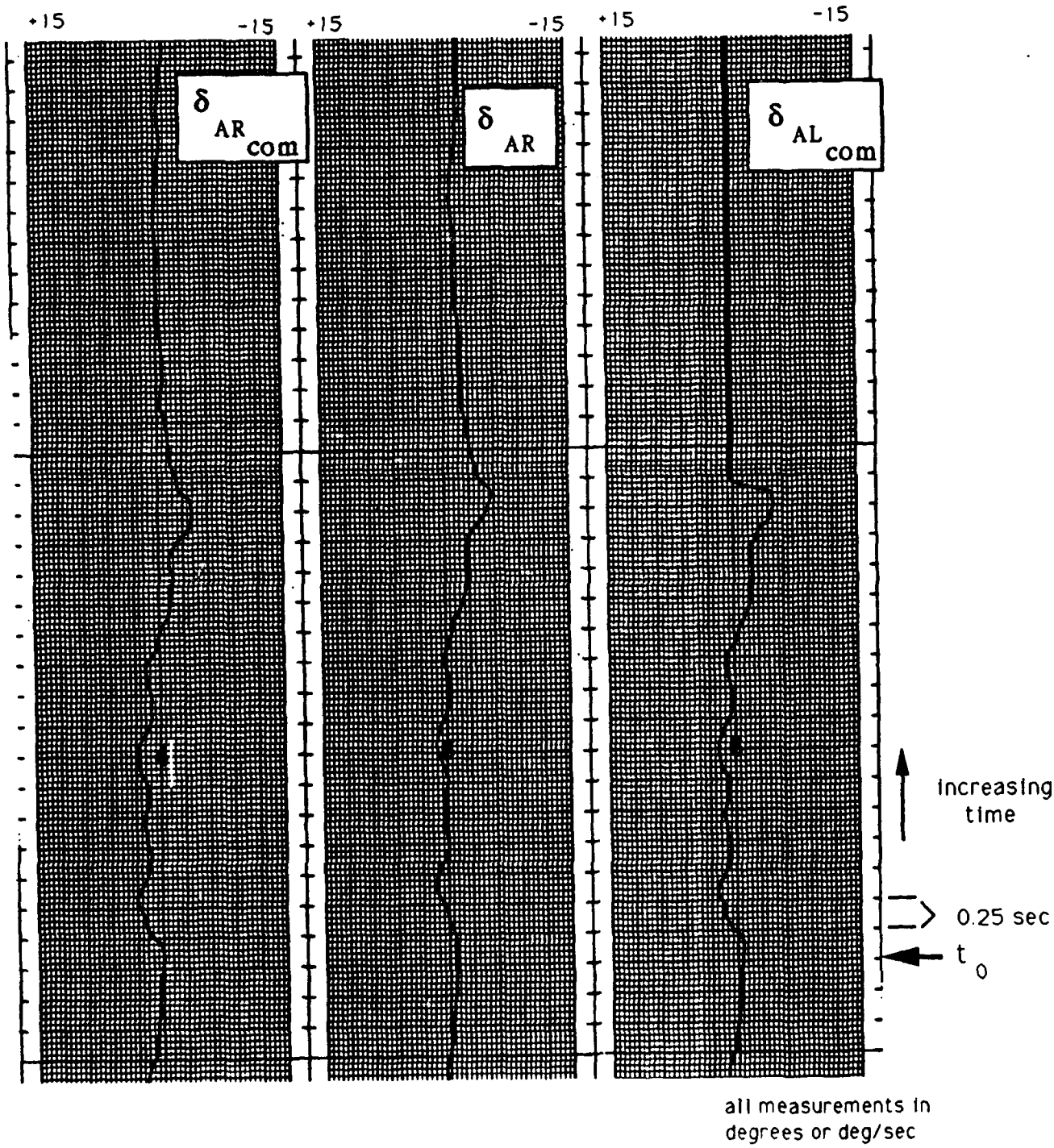


Fig. C.17

FDI for Right Elevator Failed

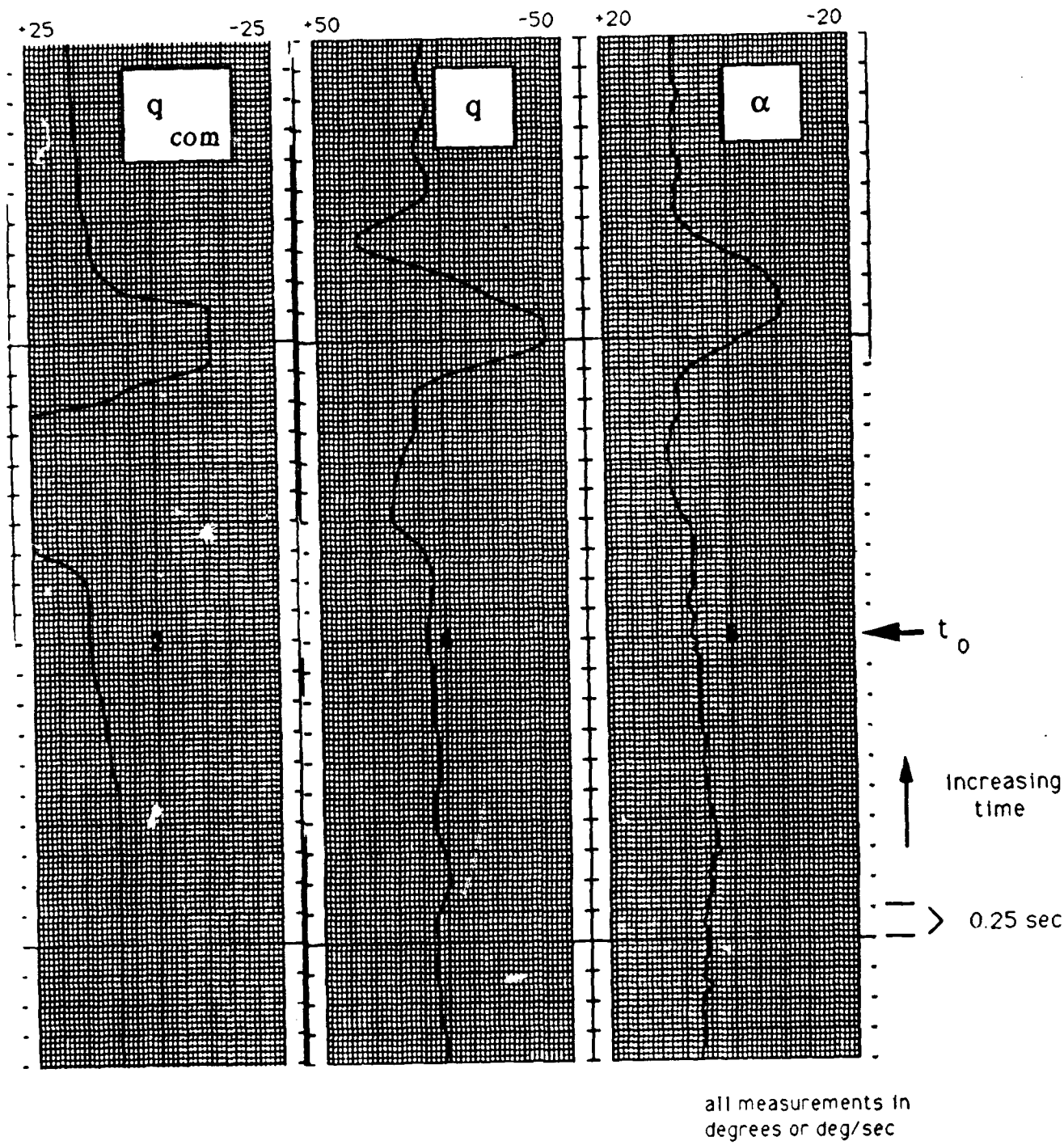


Fig. C.18

FDI for Right Elevator Failed

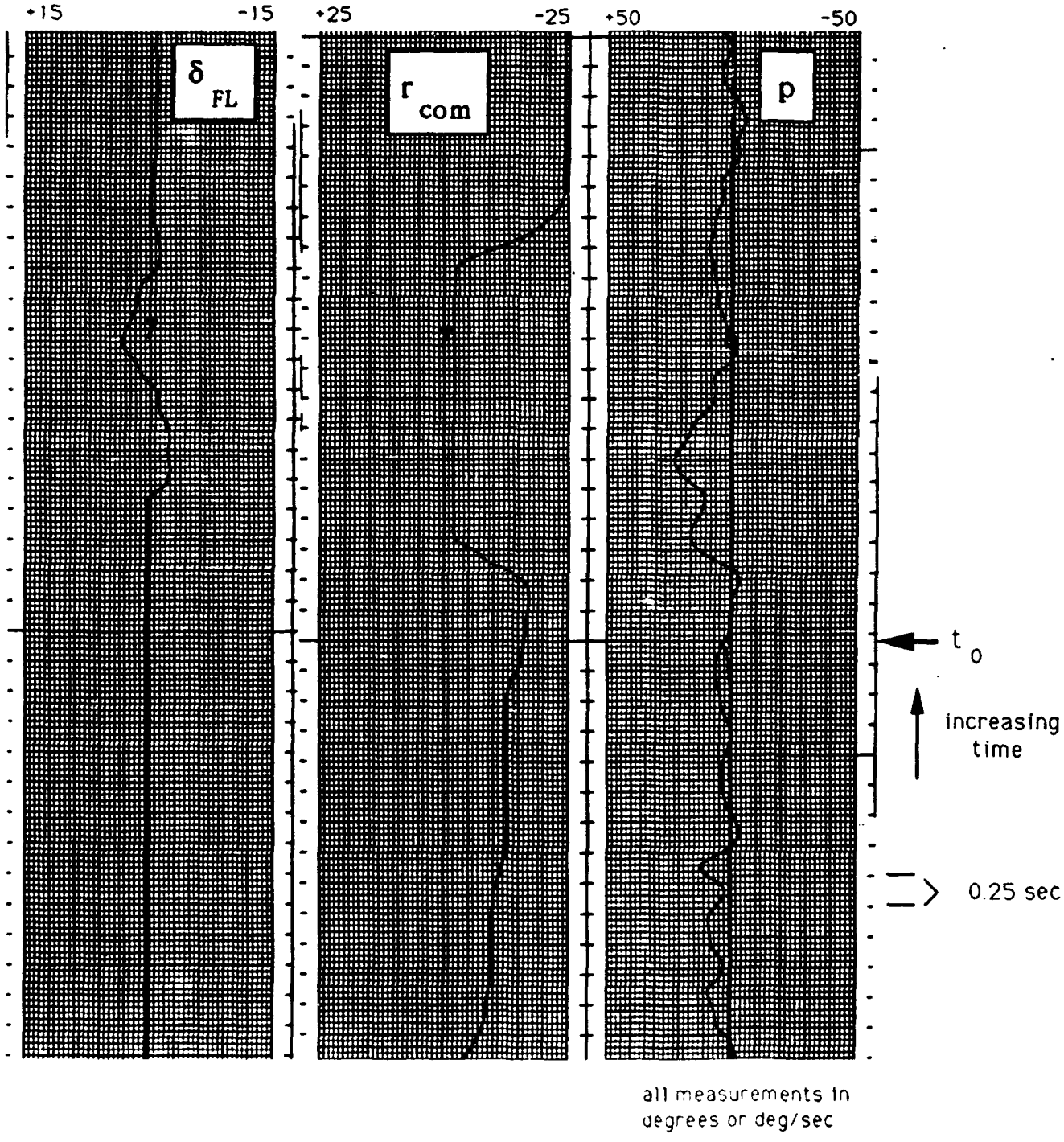


Fig. C.19

FDI for Right Elevator Failed

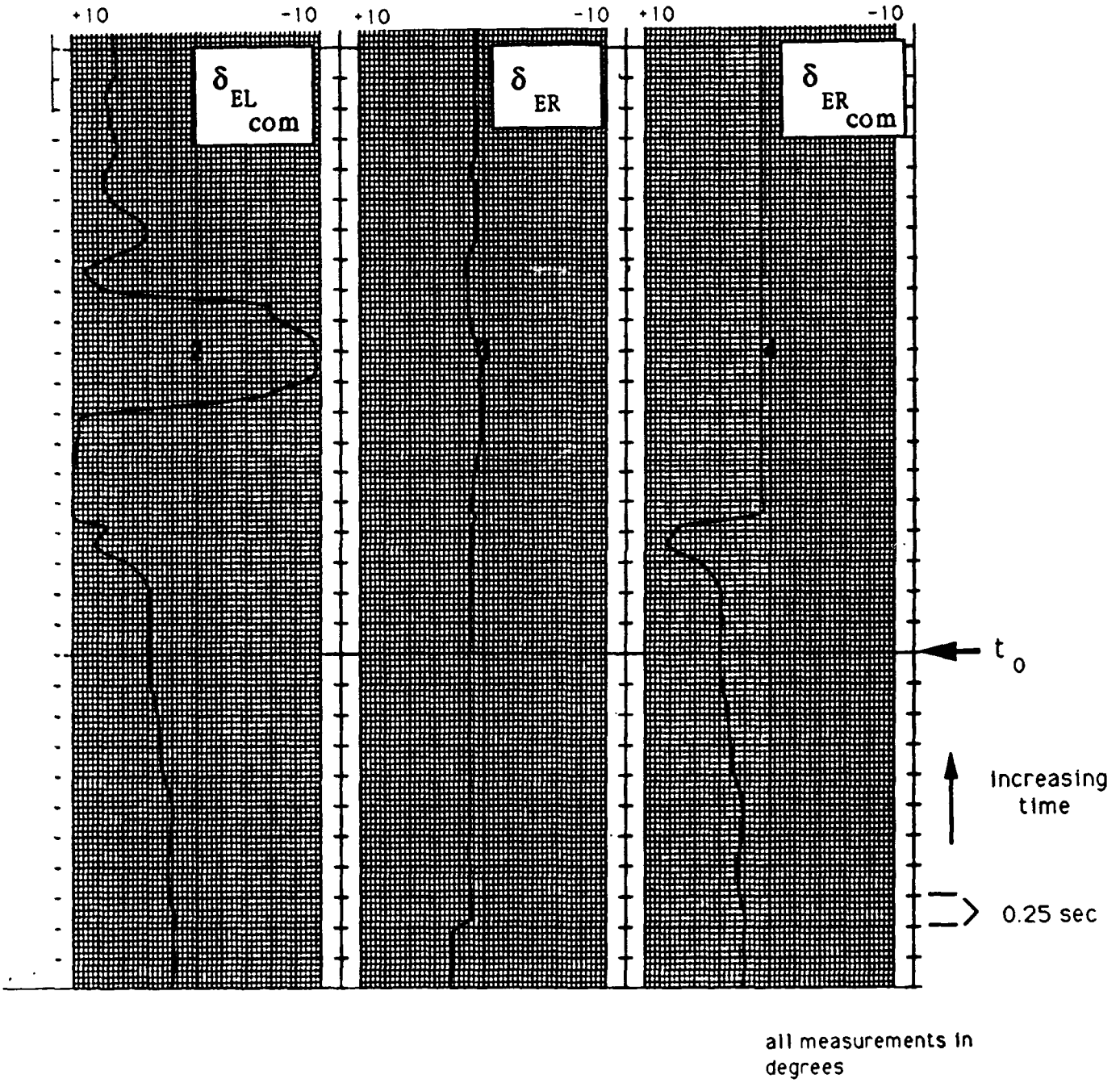


Fig. C.20

FDI for Right Elevator Failed

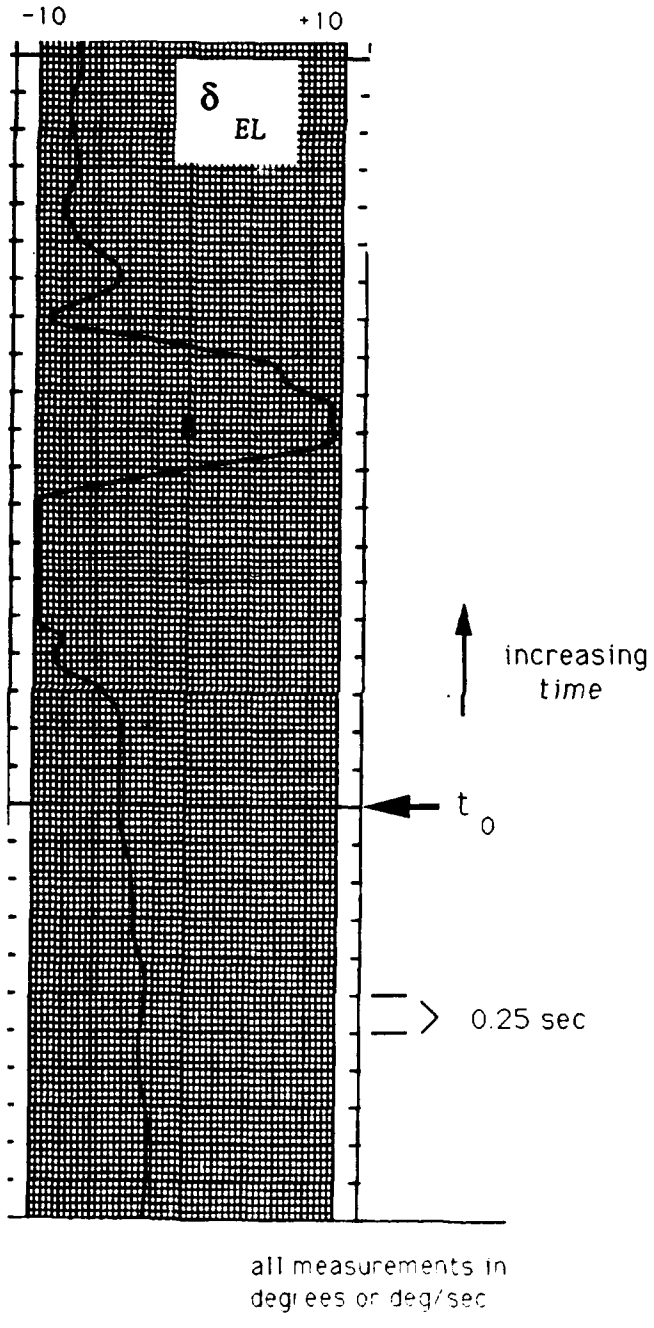


Fig. C.21

FDI for Right Elevator Failed

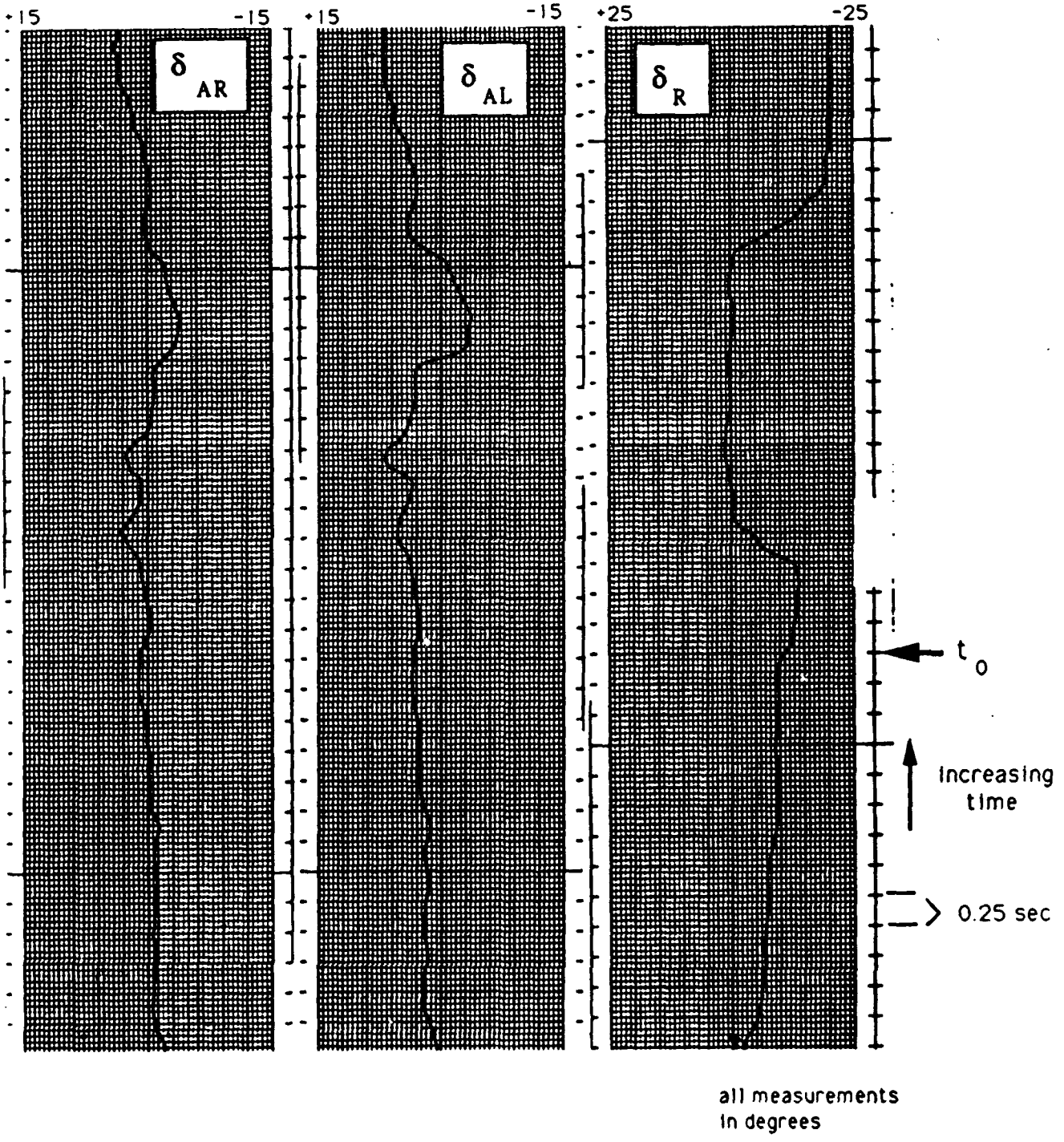


Fig. C.22

Appendix D

Ada Code

The URV flight test of a Self-Repairing Flight Control System (SRFCS) coded

```
with SYSTEM,SD_TYPES,TEXT_IO,FDI_AUX,RECM;
use SYSTEM,SD_TYPES,FDI_AUX;

 procedure FD_USER is

--  Modified Wednesday June 7, 1989 for SD-Ada
--   Wednesday May 30, 1989 Added reconfiguration transient suppression code

--  memory set as Integer 2946, 7122, 1500, and 20000 starting at
--  16#00100030# or hexidecimal B82, 1BD2, 5DC, 4E20, and 10 to produce
--  0.2946, 7.1218, 15.0, 0.2, and 10

    subtype INDX_TYPE is Integer range 1 .. 4;
  KK : INDX_TYPE;

   OFFSET            : constant Integer := 2048;
   FAILURE_STATE     : Integer;
   JAGO              : Integer;
     Transient_Cycles    : Integer;
     Transition_Count    : Integer;
   Fdi_Idling          : Integer;
    Fdi_Running          : Integer;
    Fdi_Looping          : Integer;

  RESET_BUTTON     : CHARACTER;
  MIXER_GAINS_STATE : CHARACTER;
  SURFACE_DATA_STATE : CHARACTER;
  TEMP_DATA_STATE   : CHARACTER;
  INTERRUPT_STATUS  : CHARACTER;
  ENABLED          : CHARACTER := ASCII.eot;
  DISABLED         : CHARACTER := ASCII.nul;

   SURF_CMD_DATA     : array (1 .. MAX_SURFS) of Integer;
   SURF_POS_DATA     : array (1 .. MAX_SURFS) of Integer;
   COEFFICIENTS      : array (INDX_TYPE) of Integer;

  MIXER_GAINS      : URV_MATRIX(1 .. MAX_SURFS, 1 .. 3);
  GAIN_MATRIX      :    MATRIX(1 .. MAX_SURFS, 1 .. 3);

  for FAILURE_STATE      use at SYSTEM.CONVERT_ADDRESS("00100016");
  for RESET_BUTTON       use at SYSTEM.CONVERT_ADDRESS("00100014");
  for MIXER_GAINS_STATE  use at SYSTEM.CONVERT_ADDRESS("00100012");
  for SURFACE_DATA_STATE use at SYSTEM.CONVERT_ADDRESS("00100010");
  for INTERRUPT_STATUS   use at SYSTEM.CONVERT_ADDRESS("00FF0103");

  for SURF_CMD_DATA      use at SYSTEM.CONVERT_ADDRESS("100020");
```

```
   for COEFFICIENTS      use at SYSTEM.CONVERT_ADDRESS("100030");
    for Transition_Count   use at SYSTEM.CONVERT_ADDRESS("100038");
   for Fdi_Idling         use at SYSTEM.CONVERT_ADDRESS("10003A");
   for Fdi_Running        use at SYSTEM.CONVERT_ADDRESS("10003C");
   for Fdi_Looping        use at SYSTEM.CONVERT_ADDRESS("10003E");
   for SURF_POS_DATA      use at SYSTEM.CONVERT_ADDRESS("100040");
   for   MIXER_GAINS      use at SYSTEM.CONVERT_ADDRESS("100060");

  EFFECTIVE_SURFACES    : BOOLEAN_ARRAY(1 .. MAX_SURFS);
  UN_FAILED_SURFACES    : BOOLEAN_ARRAY(1 .. MAX_SURFS);
  MODELED_ACT_RATE      : LINEAR_ARRAY(1 .. MAX_SURFS);
  FILTERED_SURFACE_CMD  : LINEAR_ARRAY(1 .. MAX_SURFS);
  FILTERED_SURFACE_POS  : LINEAR_ARRAY(1 .. MAX_SURFS);
  FILTERED_MODEL_RATE   : LINEAR_ARRAY(1 .. MAX_SURFS);
  FILTERED_SURFACE_RATE : LINEAR_ARRAY(1 .. MAX_SURFS);
  FAULT_WINDOW          : LINEAR_ARRAY(1 .. MAX_SURFS);
   Past_Surface_Cmd     : LINEAR_ARRAY(1 .. MAX_SURFS);
   Delayed_Model_Rate   : LINEAR_ARRAY(1 .. MAX_SURFS);

  SURFACE_COMMAND       : LINEAR_ARRAY(1 .. MAX_SURFS);
  SURFACE_POSITION      : LINEAR_ARRAY(1 .. MAX_SURFS);
  PAST_SURFACE_POS      : LINEAR_ARRAY(1 .. MAX_SURFS);
  DERIVED_SURFACE_RATE  : LINEAR_ARRAY(1 .. MAX_SURFS);
  CONV_FACTOR           : LINEAR_ARRAY(1 .. MAX_SURFS) :=
          ( 0.00391, 0.00391, 0.00586, 0.00586, 0.00586, 0.00586, 0.01220 );

  APU_COEFF             : LINEAR_ARRAY(INDX_TYPE);
  AMPL                  : LINEAR_ARRAY(INDX_TYPE) := ( 0.0001, 0.001, 0.01,
                                                       0.00001);

  FAULT_DETECTED        : BOOLEAN;
  FAULT_QUEUE           : BOOLEAN;
  NOT_DONE              : BOOLEAN;
  No_Faults             : BOOLEAN;
  Stable                : BOOLEAN;

  ELEMENT               : APU_FLOAT;

--  START_TIME   : APU_FLOAT :=0.0;
--  STOP_TIME    : APU_FLOAT :=0.0;
--  ELAPSED_TIME : APU_FLOAT;

  RATE_DT               : constant APU_FLOAT := 28.0;
  XMPL                  : constant APU_FLOAT := 256.0;

   package Flt_Io is new TEXT_IO.Float_Io (APU_FLOAT);

   task PC_INTERFACE is

    entry ACQUIRE_FDI;
    for ACQUIRE_FDI use at SYSTEM.CONVERT_ADDRESS("00000208");
     pragma PRIORITY (15);
```

```
end PC_INTERFACE;

task body PC_INTERFACE is

  procedure DETECT_FAULT is

    begin

-- convert the SURFACE_COMMAND's and SURFACE_POSITION's from Integer
-- format to APU_FLOAT format

      for  II in SURF_CMD_DATA'range(1)
     loop
    CMD_A:
      begin
       SURFACE_COMMAND(II)  :=
         CONV_FACTOR(II)*APU_FLOAT( SURF_CMD_DATA(II) - OFFSET );

       exception

        when others  =>
--        SURFACE_COMMAND(II)   := FILTERED_SURFACE_CMD(II);
        null;
     end CMD_A;
     end loop;

      for  II in SURF_POS_DATA'range(1)
     loop
     POS_A:
      begin
       SURFACE_POSITION(II)  :=
         CONV_FACTOR(II)*APU_FLOAT( SURF_POS_DATA(II) - OFFSET );
       exception

        when others  =>
--        SURFACE_POSITION(II)  := FILTERED_SURFACE_POS(II);
        null;
     end POS_A;
     end loop;

--     SURFACE_DATA_STATE := 'D';

 -- smooth the SURFACE_COMMAND's and SURFACE_POSITION's calculate the
 -- rate and smooth it

--      LOW_PASS_FILTER(UN_FILTERED => SURFACE_COMMAND,
--                      FILTERED => FILTERED_SURFACE_CMD);

     LOW_PASS_FILTER(UN_FILTERED => SURFACE_POSITION,
                     FILTERED => FILTERED_SURFACE_POS);

     RATE_DIFFERENCER(MULTIPLIER => RATE_DT,
           PRESENT_POSITION => FILTERED_SURFACE_POS,
           PREVIOUS_POSITION => PAST_SURFACE_POS,
```

```
                RATE => DERIVED_SURFACE_RATE);

-- LOW_PASS_FILTER(UN_FILTERED => DERIVED_SURFACE_RATE,
--                 FILTERED => FILTERED_SURFACE_RATE);


  ACTUATOR_MODEL(Surface_Not_Failed => UN_FAILED_SURFACES,
                 Model_Act_Rate => MODELED_ACT_RATE,
                   Previous_Act_Cmd => Past_Surface_Cmd,
                    Present_Act_Cmd => Surface_Command,
                    Measured_Act_Pos => Filtered_Surface_Pos,
                     Coef  => APU_COEFF);

     -- smooth the Modeled_Actuator_rate
-- LOW_PASS_FILTER(UN_FILTERED => MODELED_ACT_RATE,
--                 FILTERED => FILTERED_MODEL_RATE);

   if Stable then

      -- **** Modified Tuesday June 6, 1989 for on-going observations ****
         -- Failure Detection Identification  algorithm
  -- ACTUATOR_FDI(SURFACE_NOT_FAILED => UN_FAILED_SURFACES,
   --            MODEL_ACTUATOR_RATE => Delayed_Model_Rate,
   --         MEASURED_ACTUATOR_RATE => FILTERED_SURFACE_RATE,
   --                 FDI_WINDOW => FAULT_WINDOW,
   --                    COEF => APU_COEFF,
   --              FAULTY_ACTUATOR => FAULT_DETECTED);

        -- Failure Detection Identification  algorithm
   ACTUATOR_FDI(SURFACE_NOT_FAILED => UN_FAILED_SURFACES,
             MODEL_ACTUATOR_RATE => Delayed_Model_Rate,
          MEASURED_ACTUATOR_RATE => DERIVED_SURFACE_RATE,
                  FDI_WINDOW => FAULT_WINDOW,
                     COEF => APU_COEFF,
            FAULTY_ACTUATOR => FAULT_DETECTED);

    if FAULT_DETECTED then
         -- The great ALLAH has granted us a failure
      -- set the FAILURE_STATE word
     FAILURE_STATE := 0;
     JAGO  := 1;
        for II in UN_FAILED_SURFACES'range loop
          if not UN_FAILED_SURFACES(II) then
        EFFECTIVE_SURFACES(II) := FALSE;
       FAILURE_STATE := FAILURE_STATE + JAGO;
        end if;
      JAGO := JAGO + JAGO;
       end loop;
         -- Cancel any Rudder failure inserted
     EFFECTIVE_SURFACES(EFFECTIVE_SURFACES'last) := TRUE;

    FAULT_QUEUE     := TRUE;
       Transient_Cycles := 0;
      Stable          := False;
```

```
      end if;

    else
          if Transient_Cycles < Transition_Count then
            Transient_Cycles := Transient_Cycles + 1;
       else
          Transient_Cycles := 0;
         Stable := True;
      end if;
   end if;

       for KK in Delayed_Model_Rate'range(1) loop
         -- Delayed_Model_Rate(KK) := Filtered_Model_Rate(KK);
        Delayed_Model_Rate(KK) := Modeled_Act_Rate(KK);
   end loop;

end DETECT_FAULT;

begin
 loop
   Fdi_Looping := 1;
    accept ACQUIRE_FDI do
    FDI:
        begin

            -- Fdi_Running := Fdi_Running + 1;
       if NOT_DONE then
              -- Interrupt timed out display message
        -- INTERRUPT_STATUS   := DISABLED;
        TEXT_IO.Put (" TIME OUT");
         TEXT_IO.New_line;
        end if;
        Fdi_Looping := 2;

      NOT_DONE  := TRUE;

          if SURFACE_DATA_STATE = 'B' then

             Fdi_Looping := 3;
           SURFACE_DATA_STATE := 'C';
           DETECT_FAULT;
           SURFACE_DATA_STATE := 'D';

           else

           TEMP_DATA_STATE := SURFACE_DATA_STATE;

            TEXT_IO.Put ("          FDI_NOT_B");
            TEXT_IO.New_Line;
           TEXT_IO.Put ("    TEMP_DATA_STATE ");
           TEXT_IO.Put (TEMP_DATA_STATE);
            TEXT_IO.New_Line;
           TEXT_IO.Put (" SURFACE_DATA_STATE ");
           TEXT_IO.Put (SURFACE_DATA_STATE);
```

```
                TEXT_IO.New_Line;

              end if;

            Fdi_Looping := 4;
          NOT_DONE  := FALSE;

            exception

              when NUMERIC_ERROR | CONSTRAINT_ERROR =>

               TEXT_IO.Put (" FDI   NUMERIC_ERROR");
               TEXT_IO.New_Line;

               when others =>

              TEXT_IO.Put (" FDI    OTHER_ERROR");
               TEXT_IO.New_Line;

          end FDI;
        end ACQUIRE_FDI;

      end loop;

     exception
        when others =>
        TEXT_IO.Put (" OTHER_ERROR: PC_INTERFACE");
         TEXT_IO.New_Line;
  end PC_INTERFACE;

 begin
  Fdi_Idling  := 0;
   Fdi_Running := 0;
  Fdi_Looping := 0;
 Stable      := False;
INTERRUPT_STATUS := DISABLED;

OUTER_LOOP:
   begin
  RESET_BUTTON    := 'J';
  FAULT_QUEUE     := FALSE;
 OP_LOOP:
    begin
    loop
     if RESET_BUTTON = 'J' then
      RESET:
         begin
             -- Set initial conditions
          INTERRUPT_STATUS := DISABLED;
            Fdi_Idling  := 0;
             Fdi_Running := 0;
            Fdi_Looping := 0;
           Stable      := False;
         NOT_DONE   := FALSE;
```

```
SURF_CMD_DATA          := ( others => Offset);
SURF_POS_DATA          := ( others => Offset);

  MODELED_ACT_RATE       := ( others => 0.0);
FAULT_WINDOW           := ( others => 0.0);
PAST_SURFACE_POS       := ( others => 0.0);
   Filtered_Surface_Cmd    := ( others => 0.0);
   Filtered_Surface_Pos    := ( others => 0.0);
   Filtered_Surface_Rate   := ( others => 0.0);
   Filtered_Model_Rate     := ( others => 0.0);
  Past_Surface_Cmd        := ( others => 0.0);
  Delayed_Model_Rate      := ( others => 0.0);

      -- Wednesday April 5, 1989 reverse sign of Rt_Elevator
    -- Restored Thursday July 27, 1989
MIXER_GAINS        :=
   (( 256,   0,   0), ( 256,   0,   0),
    (   0, 256,   0), (   0, 256,   0),
    (   0,   0,   0), (   0,   0,   0),
    (   0,   0, 256));

 TEXT_IO.New_Line;
   for KK in APU_COEFF'range loop
   APU_COEFF(KK) := AMPL(KK)*APU_FLOAT(COEFFICIENTS(KK));
  TEXT_IO.Put (" COEFF (" & INDX_TYPE'IMAGE(KK) & ") = ");
    Flt_Io.Put (APU_COEFF(KK));
  TEXT_IO.New_Line;
 end loop;

--    ELAPSED_TIME := STOP_TIME - START_TIME;
--    TEXT_IO.Put(" RECONFIGURE TIME = ");
--      Flt_Io.Put(ELAPSED_TIME);
--     TEXT_IO.New_Line;

  while RESET_BUTTON = 'J' loop
   null;
  end loop;

FAILURE_STATE     := 0;
 EFFECTIVE_SURFACES := ( others => TRUE);
 UN_FAILED_SURFACES := ( others => TRUE);
FAULT_QUEUE       := FALSE;
      -- No_Faults inserted Thursday June 15, 1989 to permit a
   -- single fault
 No_Faults         := TRUE;
   Transient_Cycles   := 0;
INTERRUPT_STATUS   := ENABLED;

 exception

  when   others =>

 TEXT_IO.Put ("   OTHER_ERROR: RESET_LOOP");
```

```
            TEXT_IO.New_Line (2);

          end RESET;
        end if;

        if FAULT_QUEUE then
        FAULT_QUEUE := FALSE;
          if No_Faults then -- Start No_Faults
          No_Faults := False;
--          START_TIME := APU_FLOAT(Calendar.Seconds(Calendar.Clock));

        FAULT:
          begin

        SLO:
          begin

-- call to SLO_GI
          RECM.SLO_GI(SURF_NOT_FAILED => EFFECTIVE_SURFACES,
                      K_I => GAIN_MATRIX);

            while MIXER_GAINS_STATE /= 'H' loop
            null;
           end loop;
          exception

           when NUMERIC_ERROR | CONSTRAINT_ERROR =>

            TEXT_IO.Put (" NUMERIC_ERROR: SLO");
            TEXT_IO.New_Line;

             when others =>

           TEXT_IO.Put ("   OTHER_ERROR: SLO");
            TEXT_IO.New_Line;

         end SLO;

        MIXER:
          begin

          MIXER_GAINS_STATE := 'E';

  -- Convert the Re-configuration matrix (GAIN_MATRIX) to sixteen-bit
  -- Integer format and write into the (MIXER_GAINS) matrix

            for II in GAIN_MATRIX'first(1) .. GAIN_MATRIX'last(1)
           loop
              for JJ in GAIN_MATRIX'first(2) .. GAIN_MATRIX'last(2)
            loop
             ELEMENT          := XMPL*GAIN_MATRIX(II,JJ);
              -- Limit -32760.0 < ELEMENT < 32760.0
              if ELEMENT > 32760.0 then
                MIXER_GAINS(II,JJ) :=  32760;
```

```
                elsif ELEMENT < -32760.0 then
                 MIXER_GAINS(II,JJ) := -32760;
                else
                 MIXER_GAINS(II,JJ) :=
                       Integer(ELEMENT);
                end if;
              end loop;
            end loop;
                -- Thursday July 27, 1989 reverse sign of Lt_Aileron,
             -- Lt_Flap, and Rt_Flap
            for  II in 4 .. 6
            loop
                 for  JJ in GAIN_MATRIX'first(2) .. GAIN_MATRIX'last(2)
              loop
                  MIXER_GAINS(II,JJ) := -MIXER_GAINS(II,JJ);
              end loop;
            end loop;

          MIXER_GAINS_STATE := 'F';

           exception

           when NUMERIC_ERROR | CONSTRAINT_ERROR =>

            TEXT_IO.Put (" NUMERIC_ERROR: MIXER");
            TEXT_IO.New_Line;

            when others =>

           TEXT_IO.Put ("   OTHER_ERROR: MIXER");
            TEXT_IO.New_Line;

         end MIXER;

           exception

           when NUMERIC_ERROR | CONSTRAINT_ERROR =>

            TEXT_IO.Put (" NUMERIC_ERROR: FAULT_LOOP");
            TEXT_IO.New_Line;

            when others =>

           TEXT_IO.Put ("   OTHER_ERROR: FAULT_LOOP");
            TEXT_IO.New_Line;

         end FAULT;
--            STOP_TIME := APU_FLOAT(Calendar.Seconds(Calendar.Clock));

       end if;  -- End No_Faults
      else

      NO_FAULT:
        begin
```

```
            while MIXER_GAINS_STATE  /= 'H' loop
             null;
            end loop;

           MIXER_GAINS_STATE := 'I';

            exception

             when   others =>

            TEXT_IO.Put ("OTHER_ERROR: NO_FAULT_LOOP");
             TEXT_IO.New_Line (2);

         end NO_FAULT;
       end if;

--       exit when RESET_BUTTON = 'Z';

     end loop;

      exception
       when   others =>

         TEXT_IO.Put ("     OTHER_ERROR: OP_LOOP");
          TEXT_IO.New_Line (2);

   end OP_LOOP;


    exception

     when   others =>

        TEXT_IO.Put ("  OTHER_ERROR: OUTER_LOOP");
         TEXT_IO.New_Line (2);

 end OUTER_LOOP;
end FD_USER;

--
*****************************************************************************
*********

package SD_TYPES is

 type    APU_FLOAT    is new FLOAT;
 type     SIXTEEN_BITS  is new INTEGER; --  range -32768 .. 32767;
 subtype  ORDER         is SIXTEEN_BITS  range 0 .. 7;

 APU_ONE   : constant APU_FLOAT := APU_FLOAT(1);
 APU_ZERO  : constant APU_FLOAT := APU_ONE - APU_ONE;
 MAX_SURFS : constant    ORDER  :=  7;
```

```
  type    LINEAR_ARRAY  is array (SIXTEEN_BITS range <>) of APU_FLOAT;
  type    BOOLEAN_ARRAY is array (SIXTEEN_BITS range <>) of BOOLEAN;
  type        MATRIX  is array (SIXTEEN_BITS range <>,SIXTEEN_BITS range <>)
                                of APU_FLOAT;
  type     URV_MATRIX  is array (SIXTEEN_BITS range <>,SIXTEEN_BITS range <>)
                                of SIXTEEN_BITS;
  type   INTEGER_ARRAY  is array (SIXTEEN_BITS range <>) of ORDER;

 -- ---------------- last line package SD_TYPES ------------------
end SD_TYPES;

 --
**********************************************************************
*********

 -- ---------------- first line of package RECM ------------------
 with SD_TYPES;
 use SD_TYPES;

 package RECM is

procedure REDUCE_MATRIX(ABLE_SURFACE : in     BOOLEAN_ARRAY;
                        AA : in    MATRIX;
                        BB :   out MATRIX;
                      ROWS :   out ORDER;
                      COLS :   out ORDER;
                ROW_NOT_NIL : in out BOOLEAN_ARRAY);


procedure    REPLACE_ZEROS(AA : in     MATRIX;
                        BB :   out MATRIX;
             NON_ZERO_ROW : in    BOOLEAN_ARRAY;
             NON_ZERO_COL : in    BOOLEAN_ARRAY);

 procedure TRANS_A_MULT_INV(AA :      in MATRIX;
                      BB :    in MATRIX;
                      CC :   out MATRIX;
                   K_ROWS :    in ORDER;
                   L_COLS :    in ORDER);

  procedure A_MULT_TRANS_A(AA :     in MATRIX;
                      CC :   out MATRIX;
                   K_ROWS :    in ORDER;
                   L_COLS :    in ORDER);

  procedure TRANS_A_MULT_A(AA : in     MATRIX;
                      CC : out   MATRIX;
                   K_ROWS : in    ORDER;
                   L_COLS : in    ORDER);

 procedure INV_MULT_TRANS_A(AA : in     MATRIX;
                      BB : in    MATRIX;
                      CC : out   MATRIX;
                   K_ROWS : in    ORDER;
```

```
                L_COLS : in    ORDER);

    procedure  CROUT_INVERSE(AA : in out MATRIX;
                  UU :   out MATRIX;
                  NN :    in ORDER);

    procedure        LUDCMP(AA : in out MATRIX;
                  NN :    in ORDER;
                 INDX :   out INTEGER_ARRAY);

    procedure        LUBKSB(AA :     in MATRIX;
                   BB : in out LINEAR_ARRAY;
                  NN :    in ORDER;
                 INDX :    in INTEGER_ARRAY);

    procedure  A_MULT_K_BY_L(AA : in      MATRIX;
                  BB : in    MATRIX;
                  CC :   out MATRIX;
                K_ROWS : in    ORDER;
                L_COLS : in    ORDER);

   procedure SLO_GI(SURF_NOT_FAILED : in     BOOLEAN_ARRAY;
                    K_I :   out MATRIX);

   -- ----------------last line of package RECM ----------------------
  end RECM;

  --
  ********************************************************************
  *********

   package body RECM is

   procedure  REDUCE_MATRIX(ABLE_SURFACE : in     BOOLEAN_ARRAY;
                       AA : in    MATRIX;
                       BB :   out MATRIX;
                     ROWS :   out ORDER;
                     COLS :   out ORDER;
                  ROW_NOT_NIL : in out BOOLEAN_ARRAY) is

   --  Procedure REDUCE_MATRIX copies the AA(MAX_SURFS,MAX_SURFS) matrix
   --  into tne BB(K_ROWS,L_COLS) matrix omitting rows consisting of
   --  elements that are insignificant  -NIL_VALUE < element < +NIL_VALUE
   --   and columns representing disabled surfaces

      NIL_VALUE  : constant APU_FLOAT := -0.0001;
     XX       : APU_FLOAT;

      ZERO       : APU_FLOAT renames SD_TYPES.APU_ZERO;

      begin

    SET_ROW:
      begin
```

```
-- The BOOLEAN_ARRAY ROW_NOT_NIL is set to TRUE to record the deleted
--  ROWS for reconstituting a MAX_SURFS by MAX_SURFS pseudo-inverse
--  positions 2 and 5 are set FALSE to indicate that in the initial
--  AA(MAX_SURFS,MAX_SURFS) matrix these ROWS are zero others are
-- set TRUE

      for ROW in AA'first(1) .. AA'last(1) loop

      if ROW_NOT_NIL(ROW) then

-- Test for a ABS(AA(ROW,COL)) greater than +NIL_VALUE in any valid
--  column position

      ROW_NOT_NIL(ROW) := FALSE;
         for COL in AA'first(2) .. AA'last(2) loop
        if ABLE_SURFACE(COL) then
       XX  := AA(ROW,COL);
        if XX < ZERO then
          XX := NIL_VALUE - XX;
          else
          XX := NIL_VALUE + XX;
          end if;

        if XX > ZERO then
         ROW_NOT_NIL(ROW) := TRUE;
         end if;
           exit  when  ROW_NOT_NIL(ROW) = TRUE;
        end if;
       end loop;
      end if;
     end loop;

   end SET_ROW;

-- Copy reduced AA(MAX_SURFS,MAX_SURFS) into BB(ROWS,COLS) and set
-- ROWS and COLS to number of rows and columns

  SET_BB:
     declare
    KK   : ORDER := 0;
    LL   : ORDER;
     begin

      for ROW in AA'first(1) .. AA'last(1) loop

      if  ROW_NOT_NIL(ROW) then
      KK  := KK + 1;
      LL  := 0;
         for COL in AA'first(2) .. AA'last(2) loop
        if ABLE_SURFACE(COL) then
        LL  := LL + 1;
        BB(KK,LL)  := AA(ROW,COL);
        end if;
```

```
          end loop;
        end if;
       end loop;
      ROWS := KK;
      COLS := LL;

    end SET_BB;
 -- ------------- last line of procedure REDUCE_MATRIX --------------
end REDUCE_MATRIX;

procedure REPLACE_ZEROS(AA : in     MATRIX;
                        BB :  out MATRIX;
                NON_ZERO_ROW : in   BOOLEAN_ARRAY;
                NON_ZERO_COL : in   BOOLEAN_ARRAY) is

-- procedure REPLACE_ZEROS copies an AA(I_ROWS,J_COLS) matrix into a
-- BB(MAX_SURFS,MAX_SURFS) matrix and fills with ZERO the ROWS and
-- COLUMNS that have been tagged in the NON_ZERO_ROW and NON_ZERO_COL
-- BOOLEAN_ARRAY as being deleted

    ZERO     : APU_FLOAT renames SD_TYPES.APU_ZERO;

  begin

  COPY_AA:
    declare
    KK  : ORDER := 0;
    LL  : ORDER;
   begin

     for ROW in AA'first(1) .. AA'last(1) loop

-- replace the ZERO_ROWS as indicated by the NON_ZERO_COL BOOLEAN_ARRAY
      if NON_ZERO_COL(ROW) then
         KK := KK + 1;
         LL := 0;
           for COL in AA'first(2) .. AA'last(2) loop
-- replace the ZERO_COLS as indicated by the NON_ZERO_ROW BOOLEAN_ARRAY
          if NON_ZERO_ROW(COL) then
            LL := LL + 1;
            BB(ROW,COL)  := AA(KK,LL);
           else
           BB(ROW,COL)  := ZERO;
          end if;
         end loop;
       else
           for COL in AA'first(2) .. AA'last(2) loop
        BB(ROW,COL)  := ZERO;
         end loop;
      end if;
     end loop;

  end COPY_AA;
-- ------------- last line of procedure REPLACE_ZEROS -------------
```

```
end REPLACE_ZEROS;

procedure A_MULT_TRANS_A(AA : in  MATRIX;
                         CC : out MATRIX;
                       K_ROWS : in ORDER;
                       L_COLS : in ORDER) is
--
--
**********************************************************************
*
--  This procedure calculates the product of a matrix AA(K_ROWS,L_COLS)
--  multiplied by its transpose  The result is returned in the square
-- matrix CC (K_ROWS,K_ROWS)
--
**********************************************************************
*
--
   SUM  : APU_FLOAT;
   ZERO : APU_FLOAT renames SD_TYPES.APU_ZERO;
--
   begin
--
    for I in 1 .. K_ROWS
     loop
       for J in 1 .. K_ROWS
        loop
        SUM := ZERO;
          for K in 1 .. L_COLS
           loop
             SUM := SUM + AA(I,K)*AA(J,K);
           end loop;
         CC(I,J) := SUM;
        end loop;
     end loop;
--
-- -------------- last line of procedure A_MULT_TRANS_A --------------
--
  end A_MULT_TRANS_A;

procedure TRANS_A_MULT_INV(AA : in  MATRIX;
                           BB : in  MATRIX;
                           CC : out MATRIX;
                         K_ROWS : in ORDER;
                         L_COLS : in ORDER) is
--
--
**********************************************************************
*
--  This procedure calculates the product of a transpose matrix
--  AA(K_ROWS,L_COLS) multiplied by a square inverse matrix
--  BB(K_ROWS,K_ROWS) the result is returned in the matrix
-- CC(L_COLS,K_ROWS)
```

```
--
*****************************************************************************
*
--
   SUM  : APU_FLOAT;
   ZERO : APU_FLOAT renames SD_TYPES.APU_ZERO;
--
   begin
--
    for J in 1 .. L_COLS
     loop
       for I in 1 .. K_ROWS
        loop
        SUM := ZERO;
          for K in 1 .. K_ROWS
           loop
             SUM := SUM + AA(K,J)*BB(K,I);
           end loop;
         CC(J,I) := SUM;
        end loop;
      end loop;
--
-- -------------- last line of procedure TRANS_A_MULT_INV ---------------
--
  end TRANS_A_MULT_INV;

 procedure TRANS_A_MULT_A(AA : in  MATRIX;
                          CC : out MATRIX;
                      K_ROWS : in  ORDER;
                      L_COLS : in  ORDER) is
--
--
*****************************************************************************
*
--   This procedure calculates the product of the transpose of a matrix
--   AA(K_ROWS,L_COLS) and the matrix  The reslut is returned in the
--  square matrix CC (L_COLS,L_COLS)
--
*****************************************************************************
*
--
   SUM  : APU_FLOAT;
   ZERO : APU_FLOAT renames SD_TYPES.APU_ZERO;
--
   begin
--
    for I in 1 .. L_COLS
     loop
       for J in 1 .. L_COLS
        loop
        SUM := ZERO;
          for K in 1 .. K_ROWS
           loop
             SUM := SUM + AA(I,K)*AA(K,J);
```

```
          end loop;
        CC(I,J) := SUM;
        end loop;
      end loop;
--
-- ------------- last line of procedure TRANS_A_MULT_A --------------
--
  end TRANS_A_MULT_A;

 procedure INV_MULT_TRANS_A(AA : in  MATRIX;
                            BB : in MATRIX;
                            CC : out MATRIX;
                          K_ROWS : in ORDER;
                          L_COLS : in ORDER) is
--
--
*********************************************************************
*
--   This procedure calculates the product of a square inverse matrix
--   AA(L_COLS,L_COLS) multiplied by the transpose of matrix
--   BB(K_ROWS,K_ROWS) the result is returned in the matrix
-- CC(L_COLS,K_ROWS)
--
*********************************************************************
*
--
   SUM  : APU_FLOAT;
    ZERO : APU_FLOAT renames SD_TYPES.APU_ZERO;
--
   begin
--
    for I in 1 .. L_COLS
     loop
       for J in 1 .. K_ROWS
        loop
        SUM := ZERO;
          for K in 1 .. L_COLS
           loop
             SUM := SUM + AA(I,K)*BB(J,K);
           end loop;
        CC(I,J) := SUM;
        end loop;
     end loop;
--
-- ------------- last line of procedure INV_MULT_TRANS_A --------------
--
  end INV_MULT_TRANS_A;

 procedure A_MULT_K_BY_L(AA :  in MATRIX;
                         BB :  in MATRIX;
                         CC : out MATRIX;
                       K_ROWS :  in ORDER;
                       L_COLS :  in ORDER) is
--
```

```
--
*************************************************************************
*
--    This procedure calculates the matrix product of a square matrix AA
--    multiplied by the (K_ROWS,L_COLS) matrix BB  The result is
--    returned in the square matrix CC (K_ROWS,L_COLS)
--
*************************************************************************
*
--
   SUM  : APU_FLOAT;
   ZERO : APU_FLOAT renames SD_TYPES.APU_ZERO;
--
   begin
--
    for I in 1 .. K_ROWS
     loop
      for J in 1 .. L_COLS
       loop
       SUM := ZERO;
         for K in 1 .. K_ROWS
          loop
            SUM := SUM + AA(I,K)*BB(K,J);
          end loop;
        CC(I,J) := SUM;
       end loop;
     end loop;
--
-- ------------- last line of procedure A_MULT_K_BY_L----------------
--
  end A_MULT_K_BY_L;

 procedure  CROUT_INVERSE(AA : in out MATRIX;
                          UU :    out MATRIX;
                          NN : in     ORDER) is
 --
  NDX :       INTEGER_ARRAY(1 .. MAX_SURFS);
  YY  :       LINEAR_ARRAY(1 .. MAX_SURFS);

  ZERO : APU_FLOAT renames SD_TYPES.APU_ZERO;
  ONE  : APU_FLOAT renames SD_TYPES.APU_ONE;
 --
 begin
 --
 -- procedure LUDCMP over writes the AA-MATRIX and procedure LUBKSB over
 -- writes the YY-VECTOR
 --
 -- enter the LU CROUT algorithm decomposition routine
 --
  LUDCMP(AA,NN,NDX);
 --
 -- do the column by column inverse
 --
   for I in 1 .. NN
```

```
    loop
      for J in 1 .. NN
       loop
        YY(J) := ZERO;
       end loop;
     YY(I) := ONE;
     LUBKSB(AA,YY,NN,NDX);
--
-- build up the UU-MATRIX  column by column
--
      for J in 1 .. NN
       loop
         UU(J,I) := YY(J);
       end loop;
     end loop;
--
 -- ------------last line of procedure CROUT_INVERSE ----------
end CROUT_INVERSE;

  procedure LUDCMP(AA: in out MATRIX;
                   NN: in    ORDER;
                  INDX:   out INTEGER_ARRAY) is

 -- DESCRIPTION: procedure LUDCMP takes an N x N MATRIX with elements
 -- AA(I,J) and uses the CROUT ALGORITHM to replace the elements by the
 -- LU (lower_triangle upper_triangle) decomposition of a rowwise
 -- permutation of itself.

 -- local OBJECTS

  VV      : LINEAR_ARRAY(1 .. MAX_SURFS);

 -- I,FIRST_I,LAST_I      : ORDER;
 -- K,FIRST_K,LAST_K      : ORDER;
  I,J,K,I_MAX          : ORDER;

  SUM,A_MAX,DUM        : APU_FLOAT;

  ZERO      : APU_FLOAT renames SD_TYPES.APU_ZERO;
  ONE       : APU_FLOAT renames SD_TYPES.APU_ONE;
  TINY      : constant APU_FLOAT := 1.0E-6;

  begin

  -- loop over rows to get implicit scaling information

   for I in 1 .. NN
    loop
    A_MAX := ZERO;
      for J in 1 .. NN
       loop
          if ABS(AA(I,J)) > A_MAX then
           A_MAX := ABS(AA(I,J));
         end if;
```

```
      end loop;

-- test for SINGULAR MATRIX (no nonzero largest element) and retain
-- the scaling

    if A_MAX = ZERO then
--      SINGULAR := TRUE;
        VV(I)   := ZERO;
      else
        VV(I)   := ONE/A_MAX;
     end if;
    end loop;

-- loop over columns of CROUT'S method


  for J in 1 .. NN
   loop
     if J > 1 then
--      LAST_I := J - 1;
--       for I in 1 .. LAST_I
       for I in 1 .. (J - 1)
       loop
        SUM := AA(I,J);
        if I > 1 then
--         LAST_K := I - 1;
--          for K in 1 .. LAST_K
           for K in 1 .. (I - 1)
           loop
             SUM := SUM - AA(I,K)*AA(K,J);
           end loop;
         AA(I,J) := SUM;
        end if;
       end loop;
     end if;

 -- initialize A_MAX for the search for the largest pivot element

    A_MAX := ZERO;
     for I in J .. NN
      loop
       SUM := AA(I,J);
       if J > 1 then
--         LAST_K := J - 1;
--          for K in 1 .. LAST_K
          for K in 1 .. (J - 1)
          loop
            SUM := SUM - AA(I,K)*AA(K,J);
          end loop;
        AA(I,J) := SUM;
       end if;

 -- figure of merit for the PIVOT
```

```
        DUM := VV(I)*ABS(SUM);
        if DUM >= A_MAX then
-- it is the best so far
        I_MAX := I;
        A_MAX := DUM;
        end if;
       end loop;

-- is a row interchange indicated ?

      if J /= I_MAX then
       for K in 1 .. NN
        loop
         DUM        := AA(I_MAX,K);
          AA(I_MAX,K) := AA(J,K);
         AA(J,K)    := DUM;
         end loop;
        VV(I_MAX) := VV(J);
      end if;

     INDX(J) := I_MAX;

-- if the PIVOT is 0.0 the matrix is SINGULAR (to the precision of the
-- algorithm). for some applications on SINGULAR matrices, it is
-- desirable to substitute a small value (TINY) for 0.0

      if AA(J,J) = ZERO then
       AA(J,J) := TINY;
      end if;

-- now divide by the PIVOT

      if J /= NN then


      DUM    := ONE/AA(J,J);
--      FIRST_I := J + 1;
--        for I in FIRST_I .. NN
       for I in (J + 1).. NN
       loop
          if I /= J then
           AA(I,J) := AA(I,J)*DUM;
         end if;
        end loop;
      end if;
     end loop;
-- --------------last line of procedure LUDCMP  --------------------
end LUDCMP;

procedure LUBKSB(AA:     in MATRIX;
                 BB: in out LINEAR_ARRAY;
                 NN:    in ORDER;
                 INDX:    in INTEGER_ARRAY) is
```

```
-- DESIGNERS compiler

-- DESCRIPTION: procedure LUBKSB solves a set of N linear equations
-- A * X = B by using foward substitution and backward substitution.
-- A is an N x N  LU (lower triangle upper triangle) MATRIX decomposed
-- by the CROUT algorithm. in the process the MATRIX B is over written

-- local OBJECTS

 KK,LL        : ORDER;
--  FIRST_J,LAST_J  : ORDER;

 NON_ZERO     : BOOLEAN;

 SUM          : APU_FLOAT;

 ZERO         : APU_FLOAT renames SD_TYPES.APU_ZERO;

 begin

-- when NON_ZERO is set to TRUE   J = I becomes the index of the first
 --  non-vanishing element of  B

 NON_ZERO := FALSE;
 KK       := 0;

 -- this loop does the foward substitution

  for I in 1 .. NN
   loop
    LL     := INDX(I);
    SUM    := BB(LL);
     BB(LL) := BB(I);
     if NON_ZERO then
--         LAST_J := I - 1;
--          for J in 1 .. LAST_J
          for J in 1 .. (I - 1)
          loop
            if J > KK then
              SUM := SUM - AA(I,J)*BB(J);
            end if;
          end loop;
       elsif SUM > ZERO then

 -- a non-zero element was encountered. SUM in subsequent looping is
 -- calculated in the above loop.

      NON_ZERO := TRUE;
       KK       := I - 1;
     end if;
    BB(I) := SUM;
   end loop;

 -- this loop does the backward substitution
```

```
   for I in reverse 1 .. NN
    loop
     SUM := BB(I);
      if I < NN then
--       FIRST_J := I + 1;
--        for J in FIRST_J .. NN
--        for J in I .. NN
        for J in (I + 1) .. NN
        loop
          if J > I then
            SUM := SUM - AA(I,J)*BB(J);
          end if;
        end loop;
      end if;
      BB(I) := SUM/AA(I,I);
    end loop;
  -- ---------------last line of procedure LUBKSB ----------------------
end LUBKSB;

procedure   SLO_GI(SURF_NOT_FAILED : in     BOOLEAN_ARRAY;
                   K_I :   out MATRIX) is

-- local OBJECTS

  J_COLS           : constant ORDER := K_I'last(2);
  K_ROWS,L_COLS    : ORDER;  -- K x L MATRIX ------
  MATRIX_ORDER     : ORDER;

--     --- N linear array ---
  ROW_NOT_ZERO     : BOOLEAN_ARRAY(1 .. MAX_SURFS);

--     --- N x N matrices ---
  TEMP1  : MATRIX(1 .. MAX_SURFS, 1 .. MAX_SURFS);
  TEMP2  : MATRIX(1 .. MAX_SURFS, 1 .. MAX_SURFS);
  TEMP3  : MATRIX(1 .. MAX_SURFS, 1 .. MAX_SURFS);
  B_ZERO : MATRIX(1 .. MAX_SURFS, 1 .. MAX_SURFS);

  B_INIT : MATRIX(1 .. MAX_SURFS, 1 .. MAX_SURFS) :=
   (( -0.0034, -0.0034, -0.0022, -0.0022, -0.0040, -0.0040,  0.0000)
   ,( 0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000)
   ,( -0.5812, -0.5812, -0.0481, -0.0481, -0.0660, -0.0660,  0.0000)
   ,( 0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0016)
   ,( 0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000)
   ,(  0.2455, -0.2455,  0.6697, -0.6697,  0.6221, -0.6221,  0.0554)
   ,( -0.0130,  0.0130, -0.0428,  0.0428, -0.0403,  0.0403, -0.1789));

--     --- I x J matrices ---
  K_ZERO         : MATRIX(1 .. MAX_SURFS, 1 .. J_COLS) :=
  (( 1.0,  0.0,  0.0), ( 1.0,  0.0,  0.0), ( 0.0,  1.0,  0.0)
  ,( 0.0, -1.0,  0.0), ( 0.0,  0.0,  0.0), ( 0.0,  0.0,  0.0)
  ,( 0.0,  0.0,  1.0));

  begin
```

```
   -- to include compensation for Aileron-Flap interaction and Rudder effects
 B_ZERO := B_INIT;
  for Surface in 1 .. MAX_SURFS loop
     if not SURF_NOT_FAILED(Surface) then
      case Surface is
      when 1 | 2 =>
       B_ZERO(3,3) := 0.0;
       B_ZERO(3,4) := 0.0;
       B_ZERO(6,5) := 0.0;
       B_ZERO(6,6) := 0.0;
      when 7 =>
       B_ZERO(7,3) := -B_ZERO(7,3);
       B_ZERO(7,4) := -B_ZERO(7,4);
       when others =>
        null;
     end case;
    end if;
  end loop;

-- Initialize ROW_NOT_ZERO BOOLEAN_ARRAY to indicate ROWS 2 and 5 in
-- the B_ZERO matrix are zero (FALSE)

 ROW_NOT_ZERO :=(TRUE, FALSE, TRUE, TRUE, FALSE, TRUE, TRUE);

-- Remove the zero value ROWS and COLUMNS and return
-- TEMP1(K_ROWS,L_COLS)

 REDUCE_MATRIX(ABLE_SURFACE => SURF_NOT_FAILED,
                AA => B_ZERO,
                BB => TEMP1,
              ROWS => K_ROWS,
              COLS => L_COLS,
       ROW_NOT_NIL => ROW_NOT_ZERO);

 if K_ROWS > L_COLS then

-- form the TEMP3(L_COLS,L_COLS) matrix by multiplying the transpose
-- of the TEMP1 matrix by the TEMP1 matrix

   MATRIX_ORDER := L_COLS;
    TRANS_A_MULT_A(TEMP1,TEMP3,K_ROWS,L_COLS);
   else

-- form the TEMP3(K_ROWS,K_ROWS) matrix by multiplying the TEMP1
-- matrix by the transpose of the TEMP1 matrix

   MATRIX_ORDER := K_ROWS;
    A_MULT_TRANS_A(TEMP1,TEMP3,K_ROWS,L_COLS);

  end if;

-- form the TEMP2 matrix by taking the CROUT_INVERSE of TEMP3
-- the procedure CROUT_INVERSE over_writes the TEMP3 matrix
```

```
 CROUT_INVERSE(TEMP3,TEMP2,MATRIX_ORDER);

 if K_ROWS > L_COLS then

-- form TEMP3 by multiplying TEMP2 (the CROUT_INVERSE of TEMP3)
-- by the transpose of TEMP1

    INV_MULT_TRANS_A(TEMP2,TEMP1,TEMP3,K_ROWS,L_COLS);

   else

-- form TEMP3 by multiplying the transpose TEMP1(K_ROWS,L_COLS)
-- by TEMP2 (the CROUT_INVERSE of TEMP3)

    TRANS_A_MULT_INV(TEMP1,TEMP2,TEMP3,K_ROWS,L_COLS);

  end if;

-- replace the the zero value ROWS and COLUMNS that were previously
-- removed  (COLUMNS for ROWS and ROWS for COLUMNS)

 REPLACE_ZEROS(AA => TEMP3,
         BB => TEMP1,
   NON_ZERO_ROW => ROW_NOT_ZERO,
   NON_ZERO_COL => SURF_NOT_FAILED);

-- calculate the B_ZERO K_ZERO product and store in TEMP2

 A_MULT_K_BY_L(B_ZERO,K_ZERO,TEMP2,MAX_SURFS,J_COLS);

-- calculate the new K_I matrix ** TEMP1 multiply TEMP2 **

  A_MULT_K_BY_L(TEMP1,TEMP2,K_I,MAX_SURFS,J_COLS);

end SLO_GI;
 -- ---------------last line of package body RECM -------------------
end RECM;

--
**********************************************************************
*********

with SD_TYPES;
use  SD_TYPES;

package FDI_AUX is

  procedure       RATE_DIFFERENCER(MULTIPLIER : in     APU_FLOAT;
                    PRESENT_POSITION : in    LINEAR_ARRAY;
                    PREVIOUS_POSITION : in out LINEAR_ARRAY;
                           RATE :    out LINEAR_ARRAY);

  procedure       LOW_PASS_FILTER(UN_FILTERED : in     LINEAR_ARRAY;
                          FILTERED : in out LINEAR_ARRAY);
```

```
  procedure ACTUATOR_MODEL( Surface_Not_Failed : in      BOOLEAN_ARRAY;
                            Model_Act_Rate : in out LINEAR_ARRAY;
                            Previous_Act_Cmd : in out LINEAR_ARRAY;
                            Present_Act_Cmd : in     LINEAR_ARRAY;
                            Measured_Act_Pos : in     LINEAR_ARRAY;
                                 Coef : in    LINEAR_ARRAY);

  procedure   ACTUATOR_FDI( SURFACE_NOT_FAILED : in out BOOLEAN_ARRAY;
                       MODEL_ACTUATOR_RATE : in    LINEAR_ARRAY;
                    MEASURED_ACTUATOR_RATE : in    LINEAR_ARRAY;
                            FDI_WINDOW : in out LINEAR_ARRAY;
                              COEF : in    LINEAR_ARRAY;
                       FAULTY_ACTUATOR :   out BOOLEAN);

 -- ------------------- last line of package FDI_AUX ---------------------
end FDI_AUX;

 --
*****************************************************************************
*********

 package body FDI_AUX is

  procedure      RATE_DIFFERENCER(MULTIPLIER : in     APU_FLOAT;
                   PRESENT_POSITION : in    LINEAR_ARRAY;
                   PREVIOUS_POSITION : in out LINEAR_ARRAY;
                           RATE :   out LINEAR_ARRAY) is

 --   Procedure RATE_DIFFERENCER calculates a differenced rate

    begin

      for II in PRESENT_POSITION'range loop

       RATE(II) :=  MULTIPLIER*(PRESENT_POSITION(II)
                    - PREVIOUS_POSITION(II));

      PREVIOUS_POSITION(II) := PRESENT_POSITION(II);
     end loop;
 -- ------------ last line of procedure RATE_DIFFERENCER -------------
 end RATE_DIFFERENCER;

  procedure      LOW_PASS_FILTER(UN_FILTERED : in     LINEAR_ARRAY;
                           FILTERED : in out LINEAR_ARRAY) is

 -- implement low pass filter, corner freq = 20 r/s, digital at 60Hz
 --      C1 : constant APU_FLOAT := 0.7165;
 --      C2 : constant APU_FLOAT := 0.2835;
 -- New Values Inserted Monday April 3, 1989
     C1 : constant APU_FLOAT := 0.4895;
     C2 : constant APU_FLOAT := 0.5105;

    begin
```

```
      for  II in FILTERED'range loop
       FILTERED(II) := C1*FILTERED(II) + C2*UN_FILTERED(II);
      end loop;
 -- ------------ last line of procedure LOW_PASS_FILTER --------------
  end LOW_PASS_FILTER;

  procedure ACTUATOR_MODEL( Surface_Not_Failed : in      BOOLEAN_ARRAY;
                            Model_Act_Rate : in out LINEAR_ARRAY;
                           Previous_Act_Cmd : in out LINEAR_ARRAY;
                           Present_Act_Cmd : in     LINEAR_ARRAY;
                          Measured_Act_Pos : in     LINEAR_ARRAY;
                                Coef : in     LINEAR_ARRAY) is

 --   procedure ACTUATOR_MODEL is the input limited actuator modeled

       Max_Delta : LINEAR_ARRAY(1 .. MAX_SURFS) := ( 1.71429,  1.71429,
                   2.67857,  2.67857,  2.67857,  2.67857,  5.35714);

       Min_Delta : LINEAR_ARRAY(1 .. MAX_SURFS) := ( -1.71429, -1.71429,
                   -2.67857, -2.67857, -2.67857, -2.67857, -5.35714);

        Delta_Position : Sd_Types.Apu_Float;

      begin

       for SURFACE in SURFACE_NOT_FAILED'range loop

 --   Model each actuator that is not tagged as failed

        if SURFACE_NOT_FAILED(SURFACE) then

 --   Limit model actuator input as required

             Delta_Position := Present_Act_Cmd(Surface)
                     - Previous_Act_Cmd(Surface);

            if Delta_Position > Max_Delta(Surface) then
              Delta_Position := Max_Delta(Surface);
              elsif Delta_Position < Min_Delta(Surface) then
               Delta_Position := Min_Delta(Surface);
          end if;

              Previous_Act_Cmd(Surface) := Previous_Act_Cmd(Surface)
                                + Delta_Position;

              Model_Act_Rate(Surface) := Coef(1)*Model_Act_Rate(Surface)
                + Coef(2)*(Previous_Act_Cmd(Surface) - Measured_Act_Pos(Surface));

        end if;
       end loop;

 -- ---------------- last line of procedure ACTUATOR_MODEL -----------------
  end ACTUATOR_MODEL;
```

```
procedure   ACTUATOR_FDI( SURFACE_NOT_FAILED : in out BOOLEAN_ARRAY;
                          MODEL_ACTUATOR_RATE : in    LINEAR_ARRAY;
                       MEASURED_ACTUATOR_RATE : in    LINEAR_ARRAY;
                                 FDI_WINDOW : in out LINEAR_ARRAY;
                                       COEF : in    LINEAR_ARRAY;
                                FAULTY_ACTUATOR :   out BOOLEAN) is

--   procedure ACTUATOR_FDI is the actuator fault detection identifica-
--   tion algorithm
--   constants set to fixed memory location

--      COEF01     : constant APU_FLOAT := 0.6231;
--      COEF02     : constant APU_FLOAT := 4.3437;
--      TRSHAC     : constant APU_FLOAT := 15.0;
--      NSTRSH     : constant APU_FLOAT := 0.2;

--      CPOSMX : LINEAR_ARRAY(1 .. MAX_SURFS) := ( 8.0,  8.0,  12.0,  12.0
--                                    , 12.0,  12.0,  25.0);
--       CPOSMN : LINEAR_ARRAY(1 .. MAX_SURFS) := (-8.0, -8.0, -12.0, -12.0
--                                    ,-12.0, -12.0, -25.0);
       Error  : Sd_Types.Apu_Float;

    begin

   FAULTY_ACTUATOR := FALSE;

     for SURFACE in SURFACE_NOT_FAILED'range loop

         --  examine each actuator that isn't already failed

      if SURFACE_NOT_FAILED(SURFACE) then

          --  this is the error between measured and modeled:

     ERROR := MEASURED_ACTUATOR_RATE(SURFACE)
           - MODEL_ACTUATOR_RATE(SURFACE);

--   if sensor position has exceeded max or min limit, or
--    if error (difference between the model and the sensor measurement)
--   is beyond the threshold:
--     nstrsh = .2, trshac = 15.

--        if MEASURED_SURFACE_POSITION(SURFACE)
--             > CPOSMX(SURFACE) + NSTRSH            or else
--          MEASURED_SURFACE_POSITION(SURFACE)
--             < CPOSMN(SURFACE) - NSTRSH            or else
--           ABS(ERROR(SURFACE))
--             > TRSHAC                        then

        if ABS(ERROR) > COEF(3) then

            -- increment fdi window

         FDI_WINDOW(SURFACE) := FDI_WINDOW(SURFACE) + 1.0;
```

```
          if FDI_WINDOW(SURFACE) > 2.5 then

            -- set SURFACE_NOT_FAILED(SURFACE) to FALSE and
           -- set FAULT_DETECTED to TRUE

                -- modifications to cancel Rudder (Surface = 7) Failure
           SURFACE_NOT_FAILED(SURFACE) := FALSE;
            if SURFACE < SURFACE_NOT_FAILED'last then
           FAULTY_ACTUATOR := TRUE;
            end if;

                -- Allow only one fault to be detected at a time by reseting
            -- all FDI_ WINDOWS to 0.0

              for II in FDI_WINDOW'range loop
             FDI_WINDOW(II) := 0.0;
            end loop;
           end if;
         else
           FDI_WINDOW(SURFACE) := 0.0;
        end if;
      end if;
    end loop;
-- -------------- last line of procedure ACTUATOR_FDI ---------------
 end ACTUATOR_FDI;
-- --------------- last line of package body FDI_AUX ----------------
end FDI_AUX;
```

Appendix E
Matrix Inversion Operations

Backsubstitution and Forward Substitution

Backsubstitution and forward substitution use Eqn. 5.3, with the [A] matrix already in LU decomposed form, to find the $\underline{x}$ vectors. Thus, for the following discussion of backsubstitution and forward substitution, assume that the matrix that is being inverted has been decomposed into lower and upper triangular matrices. That is, the Eqn 5.3 can be written as

$$\underline{y}_{nx1} = ([L] * [U]) * \underline{x}_{nx1} \qquad \text{(E.1)}$$

where

$$([L]_{nxn} * [U]_{nxn}) = A_{nxn} \qquad \text{(E.2)}$$

The procedure used to decompose the [A] matrix into the [L] and [U] matrices will be developed in the next section of this report.

Eqn. E.1 can be written as

$$[L]_{nxn} * ([U]_{nxn} * \underline{x}_{nx1}) = \underline{y}_{nx1} \qquad \text{(E.3)}$$

The [L] and [U] matrices are known because they are derived from [A], $\underline{y}$ is known ($\underline{y}$ is defined as one of the columns of the identity matrix) and $\underline{x}$ is the unknown. If we now let

$$\underline{yy}_{nx1} = ([U]_{nxn} * \underline{x}_{nx1}) \qquad \text{(E.4)}$$

we get

$$[L]_{nxn} * \underline{yy}_{nx1} = \underline{y}_{nx1} \qquad \text{(E.5)}$$

Now the fact that [L] is lower triangular can be used to help solve for the vector $\underline{yy}$. Then the fact that [U] is upper triangular can be used to help solve for $\underline{x}$ using Eqn. E.4.

In general, triangular sets of equations are relatively simple to solve. To illustrate the procedure followed to calculate the inverse, a 4 by 4 example is shown below. However, this procedure is valid for matrices of arbitrarily defined size. Eqn. E.5 can be written as

$$\begin{bmatrix} A_{11} & 0 & 0 & 0 \\ A_{21} & A_{22} & 0 & 0 \\ A_{31} & A_{32} & A_{33} & 0 \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} * \begin{bmatrix} yy_1 \\ yy_2 \\ yy_3 \\ yy_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} \qquad \text{(E.6)}$$

A procedure known as forward substitution is used to solve for vector $\underline{yy}$. From the definition of matrix multiplication, the equation for $\underline{y}$, element y_1 is

$$A_{11} * yy_1 = y_1 \qquad \text{(E.7)}$$

or,

$$yy_1 = y_1/A_{11} \qquad \text{(E.8)}$$

In a similar manner, the equation for element y_2 is written below.

$$A_{21} * yy_1 + A_{22} * yy_2 = y_2 \qquad \text{(E.9)}$$

Solving for yy_2,

$$yy_2 = (y_2 - A_{21} * yy_1) / A_{22} \qquad \text{(E.10)}$$

(Note that yy_1 is known from the previous step.)

If we continue, a pattern will become evident that will suggest a closed form solution to the set of equations defined by the matrix equation. This closed

form provides a solution to systems of arbitrary size and is represented by Eqn. E.15.

$$yy_i = (1/A_{ii})(y_i - \sum_{j=1}^{i-1} A_{ij} * yy_j) \qquad \text{(E.11)}$$

For i = 2,3,...N

It is important to realize that the form of the above equation is such that the i^{th} $\underline{yy}$ value is always known from a previous calculation. In Eqn. E.11, the order of the computations is defined such that the necessary variables are known and available for use by the algorithm. Given the solution for vector $\underline{yy}$, it is now possible to solve for $\underline{x}$.

Eqn. E.4 is the other diagonal system that was generated when Eqn. E.4 was decomposed into two parts. Because Eqn. E.4 is an upper diagonal system the method of forward substitution is not applicable. However, a similar method, referred to as "backsubstitution" can be utilized. Backsubstitution is used to solve upper diagonal systems in the same manner as the method of forward substitution is used to solve lower diagonal systems. The development of the backsubstitution algorithm proceeds in a manner similar to the development of the algorithm for forward substitution.

Eqn. E.4 will be expanded below as a 4 by 4 matrix to illustrate the backsubstitution procedure.

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ 0 & A_{22} & A_{23} & A_{24} \\ 0 & 0 & A_{33} & A_{34} \\ 0 & 0 & 0 & A_{44} \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} \qquad \text{(E.12)}$$

The elements of matrix [U] have been designated A_{ij}. This will be explained in more detail later, but essentially the reason is that the storage of the [U] matrix and the [L] matrix will be in the same memory locations that are used by the A matrix. Therefore, all operations will take place on one matrix.

The equation of $\underline{yy}$ element yy_4 is

$$A_{44} * x_4 = yy_4 \qquad \text{(E.13)}$$

By inspection,

$$x_4 = yy_4 / A_{44} \qquad \text{(E.14)}$$

The equation for yy_3 is

$$A_{33} * x_3 + A_{34} * x_4 = yy_3 \qquad \text{(E.15)}$$

With x_4 obtained, the number of unknowns in the equation for yy_3 has gone from two to one, and it is possible to solve for x_3.

$$x_3 = (yy_3 - A_{34} * x_4) / A_{33} \qquad \text{(E.16)}$$

As with forward substitution, if we continued this process, a pattern becomes evident suggesting a closed form solution for vector $\underline{x}$. The closed form solution is not limited to the 4 by 4 example case, but is applicable to the general case of an n by n upper diagonal system. The closed form solution is

$$x_i = (1/A_{ii}) * (yy_i - \sum_{j=i+1}^{N} A_{ij} * x_j) \qquad \text{(E.17)}$$

For i = 1 ... N

Description of LU Decomposition

The heart of the inverse algorithm being described is the decomposition of a matrix, [A], into a lower triangular matrix [L] and an upper triangular matrix [U]; the [A] matrix represents the $[Bi*Bi^T]$ matrix which must be inverted to find the control mixer gain. This step must take place before the algorithms (backsubstitution and forward substitution) that solve the lower and upper triangular systems can be employed.

The objective is to find a matrix [L] and a matrix [U] such that the following equation holds:

$$[L]_{nxn} * [U]_{nxn} = [A]_{nxn} \qquad \text{(E.18)}$$

As in the development of the algorithms for solving the diagonal systems, the development of the algorithm to decompose a matrix [A] into diagonal matrices [L] and [U] will be illustrated with a 4 by 4 example in Eqn. E.23.

Expanding Eqn. E.18 for illustrative purposes using a 4 by 4 system,

$$\begin{bmatrix} L_{11} & 0 & 0 & 0 \\ L_{21} & L_{22} & 0 & 0 \\ L_{31} & L_{32} & L_{33} & 0 \\ L_{41} & L_{42} & L_{43} & L_{44} \end{bmatrix} * \begin{bmatrix} U_{11} & U_{12} & U_{13} & U_{14} \\ 0 & U_{22} & U_{23} & U_{24} \\ 0 & 0 & U_{33} & U_{34} \\ 0 & 0 & 0 & U_{44} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} \qquad \text{(E.19)}$$

The goal here is to derive an algorithm that will determine the values for the L_{ij} elements and the U_{ij} elements. The values of the A_{ij} are known and are specified.

Before anything else is done, it is necessary to put some constraints on the problem. This becomes evident when examining Eqn. E.19 because there are an infinite number of ways of choosing a lower diagonal [L] matrix and upper diagonal [U] that will suffice.

In general, for an N by N matrix, there are N^2 equations and $(N^2 + N)$ unknowns. However, N of these unknowns are arbitrarily set to a value of one, so there are really only N^2 unknowns in this system of equations. Thus, a unique solution can be determined.

To illustrate that this is indeed the case, consider the following equation derived from matrix Eqn. E.19.

$$L_{11} * U_{11} = A_{11} \qquad \text{(E.20)}$$

In this equation for element A_{11}, there are two unknowns and one known. If one of the unknowns is given an arbitrary value it is then possible to specify a unique value for the other unknown.

However, now the problem is specifying some of the the values of matrices [L] and [U]. Not surprisingly, many mathematicians have worked on this particular problem and many different solutions have been developed. Fortunately, one of these solutions is very applicable to this project. The solution

procedure is an improved Gaussian elimination method, called the Crout algorithm, which is computationally efficient and requires less memory than do some other solutions. Both of these attributes are important in an embedded real-time control application.

The Crout algorithm begins by assigning all diagonal elements of the upper diagonal matrix to one. This eliminates the need for storage of the diagonal elements of the [U] matrix and allows both the upper and lower diagonal matrices to occupy the original [A] matrix. The Crout algorithm performs all the calculations on the original [A] matrix without the need to store any other matrices. Because the numbers are manipulated and stored in overlapping locations in the A matrix, the Crout algorithm is sometimes referred to as an "in-place" algorithm. This type of algorithm will destroy the contents of the [A] matrix, but this application did not require further use of the original [A] matrix.

Because the algorithm is using and writing to locations of the [A] matrix, any computation that requires unaltered components of the [A] matrix must have access to those matrix elements prior to their alteration. Consequently, the "order" of the computations must follow a pattern that never overwrites an [A] matrix element that will be needed in future computations. This point is critical and must be fully taken into account when implementing the algorithm.

The equations of the algorithm are grouped according to the location of their A_{ij} terms in the [A] matrix . The first group of equations has A_{ij} elements that are above the diagonal, that is the i row index is less than the j column index. The second group of equations has A_{ij} elements that are on the diagonal, thus the i row index is equal to the j column index. The third group of equations has A_{ij} elements that are below the diagonal, therefore the i row index is greater than the j column index. The structural form of these three groups of equations are presented below.

For the case of elements above the main diagonal where $i < j$, Eqn. E.21 applies.

$$L_{i1} * U_{1j} + L_{i2} * U_{2j} + \ldots + L_{ii} * U_{ij} = A_{ij} \qquad (E.21)$$

For the main diagonal case where $i = j$, Eqn. E.22 applies.

$$L_{i1} * U_{1j} + L_{i2} * U_{2j} + \ldots + L_{ii} * U_{ij} = A_{ij} \qquad (E.22)$$

For the case of elements below the main diagonal where $i > j$, Eqn. E.23 applies.

$$L_{i1} * U_{1j} + L_{i2} * U_{2j} + \ldots + L_{ij} * U_{jj} = A_{ij} \qquad (E.23)$$

Instead of explicitly solving for a solution to the values of the L's and the U's by using Eqn. E.21, E.22 and E.23, we will first explore finding a solution by using a step by step approach. This is absolutely critical to finding the correct approach to solving this problem. This is because the order of the computations, as previously stated, must meet the condition of not overwriting any data in the [A] matrix that will be needed by any future computations.

Referring to matrix Eqn. E.19, consider the equation that specifies matrix element A11.

$$L_{11} * U_{11} = A_{11} \qquad (E.24)$$

Note first that there is only one unknown in this equation because U11 has been arbitrarily defined to be equal to one. Now, because U11 = 1,

$$L_{11} = A_{11} \qquad (E.25)$$

In other words, the A11 element remains unchanged and an efficient algorithm will not write over this variable.

Continuing the analysis, consider the equation for A21, the next [A] matrix element down the first column.

$$L_{21} * U_{11} + L_{22} * (0) = A_{21} \qquad (E.26)$$

The only unknown in Eqn. E.26 is L21 and solving for L21 yields the following solution.

$$L_{21} = A_{21} / U_{11} \qquad (E.27)$$

and again because we have set $U_{11} = 1$,

$$L_{21} = A_{21} \tag{E.28}$$

As for the last element, the value of the A matrix element A21 does not change value. Consequently, the algorithm computing the values of [L] and [U] does not need to alter the value A_{21} because it is equal to L_{21}.

Furthermore, if we continued we would find that the entire first column of the A matrix does not need to be altered. It follows that the algorithm that is computing the values of [L] and [U] does not need to modify the entire first column of the A matrix and would be wasting time if it did write over the first column. This pattern can be extrapolated to the N by N matrices.

Now consider the next column of the A matrix. The equation for A_{12} is written below.

$$L_{11} * U_{12} + (0) * U_{22} = A_{12} \tag{E.29}$$

Solving for the only unknown, U_{12}, in this equation,

$$U_{12} = A_{12} / L_{11} \tag{E.30}$$

First, note that because of the order of the computations, the other variables in this equation are already known, namely L_{11} has already been determined. Secondly, note that this is the first time in this development that a variable has been computed that is above the diagonal. The form of the solution is slightly different in that there is now a divide operation involved in the solution computation.

Continuing in the same manner, consider the equation for the next [A] matrix element down the second column, namely, A_{22}.

$$L_{21} * U_{12} + L_{22} * U_{22} = A_{22} \tag{E.31}$$

Because of the order of computation, all of the variables in this equation are known except for L_{22}. Therefore L_{22} can be determined and a unique solution found. This is illustrated below.

$$L_{22} = A_{22} - L_{21} * U_{12} \tag{E.32}$$

Recalling that there is a distinction when above, on, or below the diagonal, this last equation involving A_{22}, is on the diagonal.

This form of development can be carried further and an entire 4 by 4 case could be carried through in its entirety. The order of computation is essentially down the columns, starting with the second column, then the third column and so forth. At this point the generalized solutions can be extrapolated from the previous development.

The analytic solutions are as follows:

For $j <= i$ and $i = 1,2,3, \ldots ,n$,

$$L_{ij} = A_{ij} - \sum_{k=1}^{j-1} (L_{ik} * U_{kj}) \qquad \text{(E.33)}$$

For $i <= j$ and $j = 2,3, \ldots ,n$.

$$U_{ij} = (A_{ij} - \sum_{k=1}^{i-1} (L_{ik} * U_{kj})) / L_{ii} \qquad \text{(E.34)}$$

Once the [L] and [U] matrices have been found, they can be used in the backsubstitution and forward substitution algorithms to find the desired matrix inverse.

U.S. GOVERNMENT PRINTING OFFICE 750-113