NOAA Technical Memorandum NMFS

MARCH 1991

DOCUMENTATION OF THE 1980 DATA VERIFICATION PROGRAMS AND COMMON SUBROUTINES FOR FIXED-FORMAT DATA: PORPOISE DATA MANAGEMENT SYSTEM

Charles W. Oliver
Robert L. Butler

NOAA-TM-NMFS-SWFSC-157

U.S. DEPARTMENT OF COMMERCE
National Oceanic and Atmospheric Administration
National Marine Fisheries Service
Southwest Fisheries Science Center

NOAA Technical Memorandum NMFS

This TM series is used for documentation and timely communication of preliminary results, interim reports, or special purpose information; and have not received complete formal review, editorial control, or detailed editing.

MARCH 1991

DOCUMENTATION OF THE 1980 DATA VERIFICATION PROGRAMS AND COMMON SUBROUTINES FOR FIXED-FORMAT DATA: PORPOISE DATA MANAGEMENT SYSTEM

Charles W. Oliver
Robert L. Butler

National Oceanic and Atmospheric Administration
National Marine Fisheries Service
Southwest Fisheries Science Center
8604 La Jolla Shores Drive
La Jolla, California 92037

NOAA-TM-NMFS-SWFSC-157

Table of Contents

Table of Contents

List of Figures

Documentation of the 1980 Data Verification Programs and COMMON SUBROUTINES for Fixed-Format Data: Porpoise Data Management System

Charles W. Oliver and Robert L. Butler

I. INTRODUCTION

This document is provided as a description of the 1980 computer-aided data verification system devised for the Porpoise Data Management Group (PDMG), Southwest Fisheries Science Center, National Marine Fisheries Service, La Jolla, California. The system evolved from simple manual review procedures utilized during the early 1970s to the package of FORTRAN subroutines, edit programs, and associated procedures described herein. Although parts of the system were in use as early as 1974 and there were numerous modifications and expansions of the systems features during 1975-1978, we will describe the system implemented during 1979 and currently in use. The system was implemented during 1979 by the Automated Data Processing Operations group and the Porpoise Data Management group with coding and refinement provided by contract personnel of Potomac Research Incorporated, McLean, Virginia, and Computer Sciences Corporation (CSC), El Segundo, California. Minor revisions to the computer programs and subroutines were made to the 1979 implementation in order to implement the package on different computer systems and FORTRAN compilers, and additional subroutines have been added to the package to facilitate specific data verification needs.

This document describes the approach we developed to assist in the verification of data, and describes the 1980 package of FORTRAN subroutines (referred to by us as COMMON SUBROUTINES) we developed to create computer programs to implement this approach. Additional documention, including a detailed description and listing of each of the 1980 common subroutines, is available in an internal report archived at the SWFSC (Butler and Oliver, 1980). We provide the editing criteria specified for one data form and the edit program code for this application in Appendix 3. More complex applications are available through the Porpoise Data Management Group at the SWFSC.

Our primary objectives in developing this system were to utilize computers to locate and identify potential problems with a variety of data formats used to record field data, and to provide computer programmers with a package of subroutines by which they can easily create, and modify, edit programs for different data formats. The specific editing criteria is obtained from a variety of sources, but the system allows these criteria to be easily altered. The modular approach we implemented applies to the edit criteria as well as to the functions of the common subroutines. Secondary objectives were to provide some insight into the frequency of errors detected and maintain documentation on the actual editing criteria specified by data format and year.

DEVELOPMENTAL HISTORY (1971-1980)

The Porpoise Data Management System developed from an individual research project during the late 1960s (Perrin 1975). The research project identified the incidence of marine mammal mortality which occurred incidental to purse seine sets on porpoise schools by United States fishery for yellowfin tuna. With the enactment of the Marine Mammal Protection Act of 1972, the NMFS was directed to increase efforts at monitoring the incidence of porpoise mortality, investigate methods of reducing the kill, and assessing the populations of marine mammals affected by the fishery.

Beginning in late 1971, additional observers were placed aboard fishing and research vessels to gather data on the number of marine mammals which were killed during each fishing net set. The observed data for each cruise was compiled into a "cruise report", and at the end of each trip, mortality tables were produced and typed from the cruise reports. The first observers were placed aboard tuna vessels during 1971, when six fishing and research trips were observed. During 1972, a total of 13 cruises had accompanying observers. In 1973, the number rose to 24 fishing and the increase in the number of cruises each year brought the realization that too much time was required for the preparations and typing of morality tables. By 1974 the number of observed trips had risen to the point where it was no longer efficient to manually perform the cruise report calculations or to type the mortality tables.

As a result of this large increase in the number of observed trips, the first computer compatible data collection forms were provided to the observers fielded in 1974. At the end of each cruise, the coded data was punched and computer programs were used to produce the mortality tables. Ensuring the accuracy of the observer data in 1974 was a major obstacle to effective utilization of the databases. In the later part of 1974, a data base management system was developed which included computer verification of each data set. The data verification routines, written in 1975 and used until 1978, underwent major revisions each year to accommodate both changes in the collection/input forms and the refinement of verification needs. These programs were never fully documented and were found to be nearly incomprehensible to other than the original programmer. The original program listings are archived at the SWFC, and instructions in how to read (interpret) the computer-coded logic that performs the actual error checking is included in Appendix 1 of this documentation. Specific editing criteria are separately archived, by year, in a series of internal reports at the SWFSC.

The 1979 programs and subroutines were written in the FORTRAN-66 language on Computer Science Corporation's INFONET Telecommunications network. They take significant advantage of many system dependent features although the basic coding plan is applicable to other computers and languages. The data verification system was designed to provide a standardized, general approach for the verification of fixed-field, formatted data. This approach is described and we provide, as an example, an application we implemented on the CSC INFONET computer system, including the program code for the sequencing and main verification programs written by the Porpoise Data Management Group.

During late 1979 and early 1980, the group of "COMMON SUBROUTINES" utilized by the edit programs were modified to meet FORTRAN-77 and ANSI standards, and to facilitate the implementation of the edit programs on a wider range of computer systems. In late 1981, further modifications were made in order to implement the edit programs on the UCSD VAX computer system (Appendix 2). None of these modifications has changed the basic functions of the common subroutines, nor the hierarchical relationship between the subroutines. The purpose of this document is to describe the common subroutines developed and implemented during and after 1979, including their function and hierarchical relationship. Further revisions may occur in order to enhance the execution of programs or to implement the package on another computer system. An example of a relatively simple edit program is included in Appendix 3 and includes the data collection form, the specific editing criteria applied to the data, and the edit program code. More complex applications and the coded edit programs for these applications are archived at the SWFSC.

II. DATA VERIFICATION ON FIXED-FIELD FORMATS

Prior to 1974, researchers at the SWFC had performed analyses using calculators and mainframe computers. Data were compiled from field notebooks, and data forms, and transferred (coded) onto FORTRAN coding sheets. The coded data were then keypunched onto 80-column cards or "physical records", with each physical record containing a unique identifier associated with an event, or "logical record". In many cases, all data associated with a logical record could be coded onto a single physical record. In cases where more than one physical record was required to incorporate all the data associated with a logical record, each physical record contained the same unique identifier, and the physical records (cards or card images on computer disks) were numbered sequentially (e.g., 01,02,...,10,11). Thus, one or more physical records (cards) represented a single logical record, and logical records were grouped together into data files. Each data file represents a unique type of event or activity and is associated with a unique coding format. Data files were sometimes separated into subgroups by calendar year.

A variety of data files were in use during the early 1970s and additional logical records were appended to data files as new data was received. A listing of the keypunched cards was printed and the coded, keypunched values were reviewed. Errors in either the coding or keypunching of these data resulted in the card containing the error being removed from the data file, and a new card was keypunched. These data files were used as input to computer programs which produced summary results, analyses, etc. Each time a computer program was executed, the data file(s) were included in the jobstream.

With increases in the amount of data being collected that occurred beginning in 1973, and the development of coded data forms sent directly to be keypunched, we began using a few of the BMDP package of software programs to assist in the verification of the keypunched data. We primarily used the program which produced the frequency of values for fields identified in a format statement. We were thus able to quickly identify data files which contained values that were suspect (e.g. Day > 31; Time > 2400). These "range checks" were helpful, but somewhat cumbersome to execute and review, and the physical records containing the out-of-range values were not identified.

During 1974, we began developing computer programs using the FORTRAN-66 language which would perform range checks similar to what we had obtained from the BMDP software, and identify the physical and logical records where these "errors" occurred. We incorporated additional features into these computer edit programs including the identification of fields which did not contain any data (when we felt there should be data), and records which failed previously defined criteria comparing the values of one or more fields. These edit programs were implemented during 1974, and utilized on a number of data files collected during the year. A specific edit was developed for each data file, although each program performed similar functions: 1) sequence of physical records associated with logical records, and sequence of logical records within the data file, 2) blank checks, 3) range checks, and 4) logical checks. The specification criteria for blank, range, and logical checks was embedded in the programs in numerous arrays. We wrote new edit programs for each data file during 1975 and 1976, because the data formats changed each year. We realized that although separate edit programs were required for each data file, because of specific physical and logical record formats, and blank, range, and logical criteria, many of the edit program functions were common to all edits.

During late 1976, we began developing new edit programs for use with data to be collected during 1977. We initiated efforts to develop subroutines which could be used directly, or with only minor alterations, by all programs to perform functions common to all edits. A sequencing-function subroutine was developed for each 1977 edit and implemented as a subroutine called by the main program. The specification criteria for blank, range, and logical checks were embedded within subroutines containing arrays of these values. These arrays were used by other subroutines which performed the actual edit-checks on the data. We developed additional subroutines which created output files intended to document the actual blank, range, and logical checks performed by each main edit program. The documentation files for the blank and range checks identified whether "blank" was an acceptable value for a variable, and the range of acceptable values for a variable. The documentation file for the logical checks identified the criteria and variables associated with each logical error check, although interpretation of the documentation remained difficult. We wrote "How to read intra-variable logic" in an attempt to elucidate the meaning of this documentation, but our efforts were inadequate (Appendix 1). We wrote edits using this revised package of subroutines during 1977 and 1978.

We developed and wrote new subroutines to perform the edit functions using contracted programmers beginning in mid-1978. The new package of subroutines and edit programs were implemented in 1979, and for the most part, are still in use. We removed most of the blank and range criteria from arrays, and incorporated these checks as parameters in calls to newly created subroutines that performed the function (e.g., blank check or range check). We developed less obtuse coding of the logical checks in the edit programs, and created a database of logical error statements which provided an "English-like" definition of the actual error check. Finally, we created subroutines to tally the frequency of blank, range, and logical error checks that occurred each time and edit was executed on data. This latter function provided some insight into the data which we used to modify data definitions, training classes, or just to inform researchers of the occurrences.

The remainder of this documentation on data verification reflects the programming perspective and design that was implemented in

1979-1980 and which is essentially in use today. The needs of an error report will be discussed and then constraints on the computer program created by the nature of the input data will be described.

ERROR REPORT OUTPUT

The net outcome of a data verification program is the error report. It is this report which communicates the error(s). There are different ways and needs of communication and different kinds of errors. An error report should relatethe following: 1) the name of the verification program, 2) the name of the file being verified, 3) the date and time the program was executed, and the sequential page number of the report listing. All this can be suitably formatted as a page header. This information will prevent confusion as to when the verification was performed, what files were used, and what version of the verification program was executed. Additional items necessary to the report are the location of the error (record number and columns affeced), the type of error detected, and a display of the record(s) in which the error(s) occurred. Listing the record(s) involved in the error provide the data editor with a updated listing of the data file thus preventing confusion which can result from use of an old listing. The ability to relate the position of the error and to list the error records is dictated by the type of error. There are two main error classes: field errors and logical errors.

Field errors are those errors in which a specific field is tested for its contents against some previously defined criteria. The physical record number in error, the beginning and ending columns of the field (for the physical record), and the type of field error (e.g., blank or range error), are the basic reporting requirements. The field may be required to:

1. be blank or not be blank: (e.g., yes or no)

2. be within a range of values: (e.g., 25-43)

3. contain only certain characters: (e.g., A,X,z,/,9)

4. be a valid numeric value: (e.g., 23 or 1.234)

5. be one of several code values (e.g., 1,10,A,E)

6. other: (?)

Logical errors generally require more than one field to be tested, although this is not always the case. A data field may be tested against the file name, or some input parameter. Within the logical error class, fields are summed or compared to other fields. Logical errors may involve only one input record or several records. These records may be within the same record group or other record groups. Multiple input files may be involved. Logical errors have more complex reporting requirements in that fields in multiple records may be involved. The error message for a logical error may consist of a detailed error message with direct references to various fields and their contents, or a code for the error. In either case a record(s) must be selected for display.

We found that it was most useful to display all error messages relating to a particular data record, followed by a listing of the record. All data records within a record group were processed and if any one record in the record group had an error, the entire record group was printed on the error report. A record group is a set of records which have a common feature which ties them together as a group. The record group may be a functional unit described by the input data forms or be suggested by the types of logical errors being tested. The error report now consists of:

1. PAGE HEADER

2. Error statement(s) : blank, range, logical
3. Error Record number and listing

4. Error statement(s) : blank, range, logical
5. Error Record number and listing

... continues for all Records in a Record Group

6. Record Group listing

INPUT REQUIREMENTS TO EDIT PROGRAMS

There are several constraining factors defined by the nature of the input data. Since the role of the data verification program is to relate the data errors, errors within the data should not cause the program to blow-up (abnormally terminate). Both logical and field error types contribute to this problem, although in different ways.

A problem presented by input data is the input format required by the READ and FORMAT statements of the FORTRAN compiler, and data conversions required by the FORMAT statement. When a field needs to be numerically converted upon input, it may be invalid. Thus, it is best to read in an input record unformatted, and then validate and translate each field separately. This allows determination of the validity of the record length and variable values by the program logic, and prevents the abnormal termination of the program because of the FORTRAN compiler's error handling function. We refer to this data file testing as Phase 1.

Because logical errors may require comparisons between record groupings which may or may not be directly related to the data file format, the content of the appropriate record groupings must be determined before these logical error checks can be performed. Some record group contain multiple physical or even multiple logical records, while others are either single physical or logical records. There may be one record in the input file, several single-card format records, several groups of multiple-card records, or even several groups of single and multiple-card records. Record groupings can be arbitrary in that the user, or programmer, can decide what constitues a group. For example, a record grouping could be defined as all records (physical, or logical, or both) which occur on a single day. Logical error checks could then be performed on these groupings. This differs from "groups" of physical records which, by definition, make up a logical record.

A program which attempts to assess data file integrity (sequence of records) and perform logical errors checks is complicated and difficult to update. Changes in data file formats are relatively infrequent, and when they are required, it is relatively simple to modify the edit program to accommodate the new record groups. Thus, logical errors relating to the data file integrity (sequence of records) are best dealt with before entering the main logical edit functions of the edit program. This is why we developed subroutines to check the sequence of physical and logical records in a data file (referred to as Phase 2) prior to performing the other error-checking functions (blank, range, and logical checks), which we refer to as Phase 3 testing.

III. THE DATA VERIFICATION SYSTEM FOR THE PORPOISE DATA MANAGEMENT SYSTEM

The data verification system developed for the Porpoise Data Management System evolved into three verification phases (Phase 1, 2, and 3). An error in one phase prevents advance to the next phase. The main edit program (Phase 3) calls the sequencing subroutine which then executes the Phase 1 tests, and if successful, the Phase 2 testing. If both Phase 1 and Phase 2 testing are successful, the main edit program then performs the Phase 3 testing (blank, range, and logical error checks).

Phase 1 identifies record length and variable format type errors, or the absence of these errors, associated with how a FORTRAN compiler handles READ, FORMAT, EOF, and various "type formats" (INTEGER, REAL, CHARACTER). This phase locates the first record number in the data file being edited which fails the test, and informs the user of this location. Corrections to the data file are required before the Phase 1 tests are redone. It is sometimes necessary to repeatedly execute the edit program (Phase 1), correct identified Phase 1 errors, and re-run the edit program again until all Phase 1 problems with the data file are corrected.

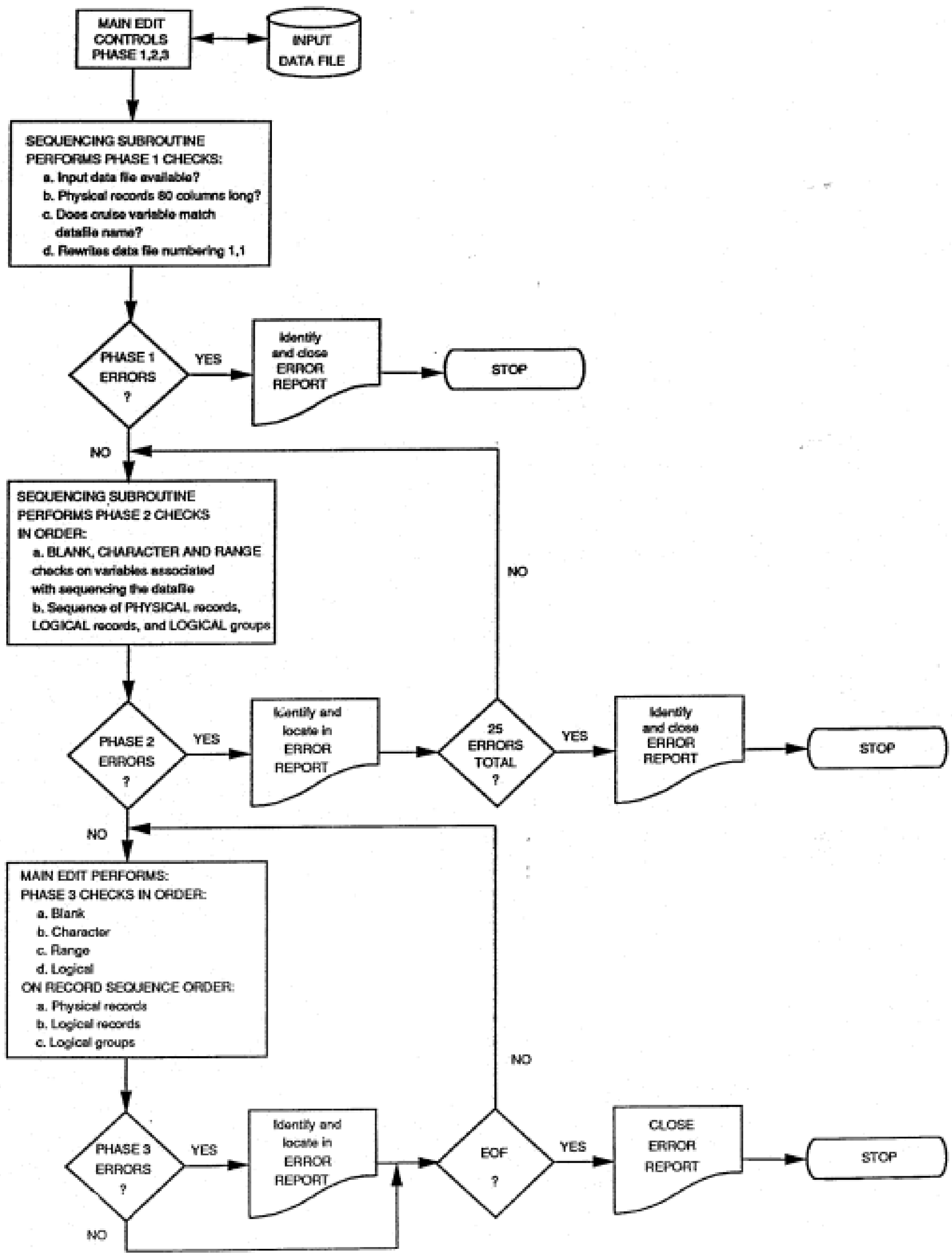
Phase 2 testing checks the sequential order of physical records and logical records and, in some edits, ascertains that variables such as DATE and TIME do not decrease between sequentially-ordered physical and logical records. Phases 1 and 2 were implemented during 1979 in the subroutines that perform the sequencing checks on a data file. Prior to 1979, separate sequencing programs were developed and executed prior to execution of the main edit program for each data file.

Phase 3 testing provides detailed, cross-field verifications (blank, range, and logical error checks). Separating the program logic, and subroutines, into these three phases provided simpler program structures which were easier to revise and update, communicated the errors to data editors in a step by step manner, and provided clear, concise documentation within the program code.

There were two main programs per data file to be edited prior to the 1979 implementation: 1) The combination of Phase 1 and Phase 2 tests formed the sequencing program. Phase 3 tests were performed by the main edit program. With the 1979 implementation, a single main edit program was developed for each data file with the sequencing (Phases 1 and 2) tests performed by a subroutine called by the main edit program. The main edit program (Phase 3) calls the sequencing subroutine which then executes the Phase 1 tests, and if successful, the Phase 2 testing. If both Phase 1 and Phase 2 testing are successful, the main edit program then performs the Phase 3 testing (blank, range, and logical error checks). Thus, when the main edit is executed, it always performs a check of the physical record sequence first.

The data verification system is comprised of a main program (edit) which sequentially progresses through each of three phases. If errors are detected during either Phase 1 or Phase 2, the program can terminate depending upon the users predefined criteria for termination and output. Once the program progresses to Phase 3, the edit processes all records in the data file regardless of the number and type of errors detected. All three phases utilize the common subroutines and functions. The progression through the three phases and the termination criteria we defined are shown in Figure 1. A diagram relating the calling sequence, by phase, for all common subroutines is shown in Figure 2, and a brief description of each of the common subroutines is presented in section IV. Some routines are called once and others repeatedly as indicated. The common subroutines relate to one another via parameter lists or variables defined in the named common block called ERROR (section V). Figure 3 provides a cross reference of the variables within the named common block ERROR directly referenced by the various subroutines. Section VI provides additional documentation of the logical hierarchy of the common subroutines ERRLOG.DP, ERRFIL, ERRCHK.DP, and WRAPUP.DP which are called by either the main edit program or other common subroutines. Section VII provides additional information on how data is input into the edit program, subsequently passed between the various common subroutines, and included in the error report listing. Section VIII describes the databases which were developed to archive and update logical error statements (ES80DB) and counts (EC80DB).

Figure 1. Data verification calling structure and hierarchy flowchart.



MAIN EDIT CONTROLS PHASE 1,2,3
INPUT DATA FILE
SEQUENCING SUBROUTINE PERFORMS PHASE 1 CHECKS:
a. Input data file available?
b. Physical records 80 columns long?
c. Does cruise variable match datafile name?
d. Rewrites data file numbering 1,1
PHASE 1 ERRORS ?
YES
Identify and close ERROR REPORT
STOP
NO
SEQUENCING SUBROUTINE PERFORMS PHASE 2 CHECKS IN ORDER:
a. BLANK, CHARACTER AND RANGE checks on variables associated with sequencing the datafile
b. Sequence of PHYSICAL records, LOGICAL records, and LOGICAL groups
NO
PHASE 2 ERRORS ?
YES
Identify and locate in ERROR REPORT
25 ERRORS TOTAL ?
YES
Identify and close ERROR REPORT
STOP
NO
MAIN EDIT PERFORMS: PHASE 3 CHECKS IN ORDER:
a. Blank
b. Character
c. Range
d. Logical
ON RECORD SEQUENCE ORDER:
a. Physical records
b. Logical records
c. Logical groups
NO
PHASE 3 ERRORS ?
YES
Identify and locate in ERROR REPORT
EOF ?
YES
CLOSE ERROR REPORT
STOP
NO

THE ERROR REPORT FORMATS FOR PHASES 1, 2, AND 3

The error report formats differ between Phase 1, Phase 2, and Phase 3 testing. Phase 1 verifications involve only errors in relation to one record. The report format allows for the error messages for the record, followed by the record. The format is defined to fit on an 80 column terminal.

Phase 2 verifications involve the order and number of records. The report format allows for the error messages for the record, followed by a suitable range of records before, after, and including the error record. The format is designed to fit on an 80 column terminal.

Phase 3 verification types are mixed but usually relate to fields within a given record group. All error messages for a record are printed followed by a listing of the record. When all records within a record group are processed, the records within the record group are printed sequentially with a column delineator.

In addition to the printed error report, a database, EC80DB, is updated with the number of errors detected during Phase 3 execution, by error type and data file type. Analysis of this data base can help in identifying problems in data definition and collection, and allow quality control to be more dynamic. Records in this database contained the frequency of errors, by error code, encountered each time the main edit was executed on a data file. Error codes include blank, range, and logical errors and are differentiated by data file type (e.g., SL for Set Log). The logical error statement is written in the form of a conditional statement followed by a resultant clause. The logical error statements are written out on the Error Message Collection Forms which are then keypunched and loaded to the yearly database (e.g., ES80DB) using FORTRAN program UPEROR.MN.

There are two general classes or types of errors: LOGICAL and FIELD. Both classes are tallied in database EC80DB. Logical error definitions are contained in database ES80DB. Both data bases have the same KEY format, consisting of the following.

DATASETID:	Columns 1-8 identifies the field collection or coding format by year. CSEDIT80 SLEDIT80 VAEDIT80 MMEDIT80 BLEDIT80 LHEDIT80 ASEDIT80
GRPCOD:	Column 9 identifies multi-format field collection or coding formats.

'N'	-	Non-porpoise	Set Log	SLEDIT
'P'	-	Porpoise	Set Log	SLEDIT
'E'	-	Effort		MMEDIT
'S'	-	Sightings		MMEDIT
' '	-	all others		VAEDIT, BLEDIT, CSEDIT

ERRORTYPE: Column 10 identifies the nature of the field error or logical error.

B	-	Blank error	(Field)
C	-	Character error	(Field)
L	-	Logical error	(Logical)
R	-	Range error	(Field)

CARDSEQ: Columns 11-12 for FIELD errors, the card sequence of a field (e.g., "01") within a multicard format data set. For a single card format the code is "01" (Length 2). Blank for LOGICAL errors.

BEGINCOL: Columns 13-14 for FIELD errors, the beginning column number of (e.g., "29") the field having an error. (Length 2): Blank for LOGICAL errors.

ENDINGCOL: Columns 15-16 for FIELD errors, the ending column of a field (e.g., "34") having an error. (Length 2). Blank for LOGICAL errors.

ERROR NUMBER: Columns 17-24 identifies the particular error for a given data file. Blank for FIELD errors. Always ERROR"___" where "___" is some number between 1-999.

The KEY value is built by the Phase 3 coding. Either, or both, of the databases can be accessed via the same key. There are "DUMP" programs (DERCT.80 and DERST.80) for EC80DB and ES80DB respectively. Access to these data bases (section VIII) occurs through subroutine ERRLOG.DP called by subroutines BLANKT.DP, RANGIT.DP, VALUIT.DP, and VERFIT.DP when an error is detected (sections IV and VI).

the logical variable ANYERR is set to TRUE and then checked:

Phase 1 Processing

The Main edit program calls the Phase 1 coding, passing the character string, HEAD, which is then used as the header for the top of each output page of the error report. Prior to 1980 there were two different Phase 1 entries within subroutine CSEQP1: CSEQP1 and CSEQP2. During the 1980 modifications to the common subroutines, we incorporated these two entries, along with other Phase 1 and 2 functions into a single subroutine (the "sequencing subroutine") specific to each edit program on the CSC system. These "sequencing subroutines" perform the functions desribed below that were previously performed by entries CSEQP1 or CSEQP2 in the subroutine CSEQ.P1. In late 1980, further modifications to this area of the common subroutine package was required in order to implement the package on another computer. The common subroutine CSEQ.P1 (entries CSEQP1 and CSEQP2) which is CSC specific, was incorporated into a "sequencing subroutine". The "sequencing subroutines" are referenced, for example, as CSEQ81 (for Cruise Specifications sequencing subroutine for 1981).

Entry CSEQP1 was called for the following data sets: Cruise Specifications, Fishing Mode, Shipboard Mammal Watch Daily Effort, Shipboard Mammal Watch Sighting, and Porpoise and School Fish Set Logs during 1975-1979. CSEQ.P1 was utilized during 1980 on the CSC system.

The input data file is rewritten to renumber the file by 1.
The cruise number is crossed checked between the file name and the input file.
The input records are verified to be of length 80.

Entry CSEQP2 was called for the Marine Mammal Bridge Log data files for 1975-1979. CSEQ.P1 was utilized during 1980 on the CSC system.

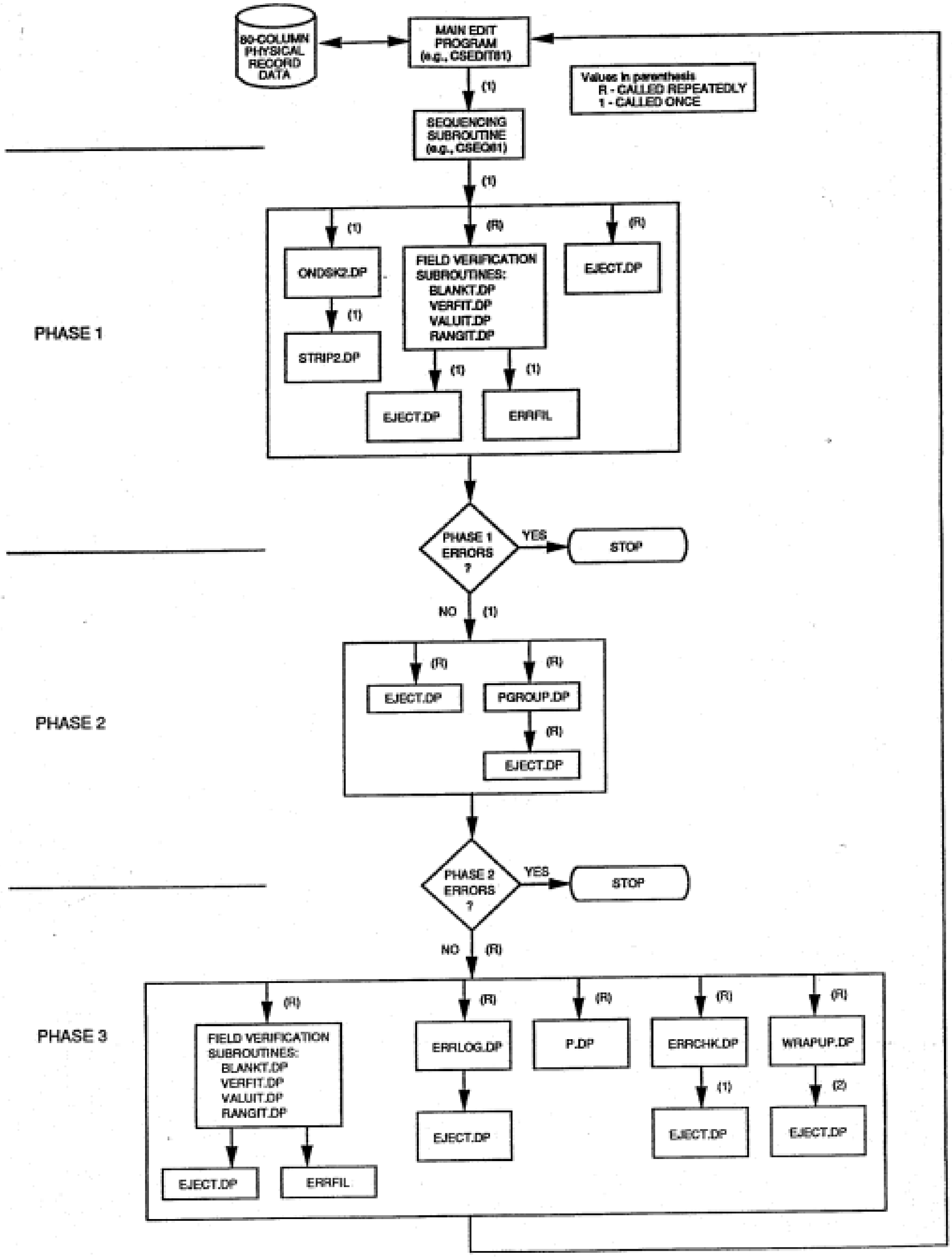
The input data file is rewritten to renumber the file by 1.
The input records are verified to be of length 80.

Phase 1 prompts for an input data file name. The input file name is concatenated with HEAD and passed by EJECT.DP, the paging routine. EJECT.DP performs a top of form stating the card sequence program name, the input file name, the date, time, and page number.

Subroutine ONDSK2.DP is called to verify the occurrence of the input file on the users disk. STRIP2.DP is called by ONDSK2.DP to prepare the input file name for use in a system routine which does the verification of the file on disk. If the file is not on disk, execution terminates. If the file is on disk, the appropriate verifications and renumbering are performed.

The Phase 1 program rewrites the file so that the key begins with 1 and increments by 1. The input file is rewound. Each record is input and a verification is performed. If an error is detected

Figure 2. Common subroutine calling sequence and hierarchy flowchart.



If ANYERR is TRUE:

subroutine EJECT.DP is called for paging.

the error message is written because the logical variable ANYERR is checked and found to be TRUE.

ANYERR is reset to FALSE, and TOTERR is set to TRUE.

If more than 25 errors are encountered, the program will terminate immediately. If all records are processed and 25 or less errors were detected, TOTERR is checked. If TRUE, an error was detected during Phase 1 testing and a stop is performed rather than continuing on to Phase 2 testing. If no errors were detected, the program procedes to Phase 2.

Phase 2 Processing

If there are no Phase 1 errors, Phase 2 verifications are performed. Phase 2 coding verifys the order and number of records in the data file. Phase 2 coding is written for each data file type. Phase 2 coding can exist as both main programs and subroutines. The only difference between the main programs and subroutines is that the latter have the SUBROUTINE statement and a RETURN statement; main programs have a STOP statement and no SUBROUTINE STATEMENT.

All Phase 2 coding has the following basic structure. A record is input and a verification is performed. If an error is detected, subroutine EJECT.DP is called with the number of output lines for the error message, the error message is written, and logical variable ANYERR set to TRUE. Following all verifications for the record, logical variable ANYERR is checked. If ANYERR is TRUE, subroutine PGROUP.DP is called to print a set of records before, after, and including the record in error. Then ANYERR is set to FALSE and logical variable TOTERR set to TRUE. When all records are read, the Phase 2 coding finishes with either a STOP or RETURN, depending upon whether it is a main program or subroutine.

Phase 2 coding may make calls to the field verification routines. These routines check for blanks (BLANKT.DP), range (RANGIT.DP), character type (VERFIT.DP), and retrieve a numeric from a string (VALUIT.DP). The fields verified would be those fields which are used to check the sequence of the records. A general description of these routines is given in section IV.

Phase 3 Processing

When a RETURN is made to the main verification program, common logical variable TOTERR is examined. If TRUE, an error was encountered in Phase 2 and Phase 3 writes a message and then terminates the program. If TOTERR is FALSE (meaning no Phase 1 or 2 errors), then Phase 3 prompts for "NO EDIT" or "GO EDIT". The user response indicates whether or not the program should terminate, or continue with a main edit. By incorporating this prompt into the Phase 3 coding, we have eliminated the need for stand alone Phase 1 and 2 coding (previously the "sequencing" programs. All sequencing requirements (Phase 1 and 2 coding) are incorporated into Phase 3 coding. When Phase 3 begins, the EC80DB data base is opened with a call to OPLGCT (see ERRLOG.DP in sections IV and VI). The actual database name (i.e. EC80DB) is passed through the call as variable ECNAME. This allows us to only change the MAIN programs once a year, and leave the COMMON SUBROUTINES intact.

Variables ELEVAL and DATSET are assigned to enable errors to be tallied on EC80DB (section VIII). The disk sequence number (variable DSKSEQ) is set to zero. A prompt is made to set the named common block ERROR variable ALLBLK to TRUE or FALSE. If ALLBLK is TRUE, all of the input data file, regardless of error conditions, will be printed. If ALLBLK is FALSE, only those record groups which have an error will be printed. Now we are ready to proceed.

As in Phase 2, there are field verifications and logical verifications. The field verifications are handled using the field verification subroutines. The subroutines used are: BLANKT.DP, RANGIT.DP, VALUIT.DP, and VERFIT.DP. All the field verifications for a given record type are performed prior to proceeding to the logical verifications. Generally, if a field requires a blank check or character check, and range check the blank or character check is done first, and if the blank or character check results in an error, no range check is performed (example 1). If a field does not require a blank or character check (e.g., it's okay if it is blank), the blank or character checking subroutine is called, and only if no "error condition" results will a range check be performed (example 2).

Example 1

```
C
      CALL BLANKT(14,4)           Blank check cols. 14-17
      IF( NOGO ) GOTO 38          if not blank do Range check
C                                 if blank do Character check
C
      CALL VERFIT(14,4,DIGITS)    Character check
   38 CONTINUE
      CALL RANGIT(14,4,850,1100)  Range 850-1100 on cols. 14-17
C
```

Example 2

```
C
      CALL RANGIT(14,4,850,1100)  Range 850-1100 on cols. 14-17
C
```

The logical verifications comprise the bulk of the Phase 3 coding. Each logical verification is assigned a unique 8 character error code. When the logical verification is performed, the logical error is reported using this 8 digit code by calling subroutine ERRLOG.DP. ERRLOG.DP prints an error message containing the sequence number of the input record and the error code and tallies database EC8ODB with the occurrence of the error. Logical error checks are performed after blank, character, and range checks have been performed. It is generally more difficult to locate specific problems with data resulting from a logical error occurrence, because there are multiple fields involved. Therefore, we found it was easier to locate "errors" by sequentially checking for blank, character, range, and then logical errors. Logical errors are coded as a group within an edit program such that they can be modified, added, or removed easily. They can be simple or complex (examples 3-4).

Example 3

```
C
      IF (P(STRING(1)(10:11)) .LE. '60') GO TO 100    cols 10-11
      IF (P(STRING(1)(12:15)) .LT. '0400') GO TO 100  cols 12-15
      IF (P(STRING(1)(16:16)) .EQ. '3') GO TO 100     col  16
      CALL ERRLOG('ERROR001')                         error 001
  100 CONTINUE
C
```

Example 4

```
C
      IF (P(STRING(1)(10:11)) .LE. '60') .AND.        cols 10-11
      IF (P(STRING(1)(12:15)) .LT. '0400') .AND.      cols 12-15
      IF (P(STRING(1)(16:17)) .EQ. '03') .OR.         cols 16-17
      IF (P(STRING(1)(16:17)) .EQ. '99') .OR.         cols 16-17
      IF (P(STRING(1)(72:72)) .NE. 'X') GO TO 100     col  72
      CALL ERRLOG('ERROR112')                         error 112
  100 CONTINUE
C
```

All verifications are performed for a record. Then subroutine ERRCHK.DP is called to check if any errors have occurred (ANYERR is TRUE or FALSE) for the record. If ANYERR=TRUE, the record is printed. Following all verifications for a record group, subroutine WRAPUP.DP is called to check if any records within the group had an occurrence of error (TOTERR or ALLBLK are TRUE or FALSE). If TOTERR=TRUE, or logical variable ALLBLK is TRUE, the entire data group is printed as a block. See sections IV and VI for a more detailed explanation of how to use ERRLOG.DP, ERRCHK.DP, WRAPUP.DP, and the field verification. Each logical verification occupies a separate block of code which is entered from the top and exited from the bottom. If a block is removed, it will not affect the rest of the program. This form of coding is called block structuring. A block is never entered from other than the top, and is never left other than through the end of the block. A labeled CONTINUE statement is coded at the bottom of some blocks to allow for a jump out of the block.

IV. Brief description of 1980 common subroutines and databases.

This section lists the 1980 subroutines and databases that comprise the data verification package. A brief description of the COMMON SUBROUTINES and databases is provided. Appendix 2 provides additional information useful to programmers considering implementation of the package of common subroutines on another computer system.

Subroutine	Purpose
BLANKT.DP	Field verification routine used for blankness. The field specified by passed parameters (beginning column number and field width within STRING) is tested to be non-blank. If the field is blank, an error message is written and optionally, EC80DB data base is tallied with the error occurrence.
_SEQ80	The "sequencing subroutine" for a particular year and edit where "_" is replaced by the edit type (e.g., C for Cruise Specifications). Prior to 1979, the "sequencing subroutine" was a separate program executed independent of the main edit program. These separate programs used entries CSEQP1 and CSEQP2 within a common subroutine named CSEQP1. In 1980, we incorporated these functions with the "sequencing subroutines" through calls to CSEQ.P1 on the CSC system.
CSEQ.P1	Phase 1 verification subroutine called by the "sequencing subroutine" for an edit. Prior to 1979 CSEQP1 and CSEQP2 were entries within the common subroutine CSEQP1. In 1980, we modified the subroutine and eliminated the need for both entries. CSEQ.P1 is CSC specific (see Appendix 2). The input files name is prompted for and then all physcial records in the data file are renumbered beginging with 1 and incremented by 1. The first 3 characters of each record are cross checked against the first 3 characters of the input file name (generally the NMFS Cruise number assigned to the observer trip data file(s) undergoing edit. The record length is verified to be 80 characters.
CSEQP2	Phase 1 verification entry accessed through the "sequencing subroutine" for an edit. Prior to 1979 CSEQP1 and CSEQP2 were entries within the common subroutine CSEQP1. In 1980, we modified the subroutine CSEQP1 and eliminated the need for both entries. CSEQ.P1 is CSC specific (see Appendix 2). This subroutine performs the same functions as entry CSEQP1, except that records are not cross checked against the input file name.

DERCT.80	ALADIN program used to provide a report listing from the database EC80DB containing the frequency of errors by error code and data file type.
DERST.80	ALADIN program used to provide a report listing from the database ES80DB containing the logical error statements, by error code and data file type. This program produces a disk file which can be printed and transformed into the annual LOGICAL ERROR STATEMENT book.
DOERCT.80	ALADIN program which declares the error count database EC80DB.
DOERST.80	ALADIN program which declares the error statement database ES80DB.
EC80DB	ALADIN database used to store a tally of error occurrences by error code for a data file type.
EJECT.DP	Paging routine called whenever printed output is desired. When initialized, a page is ejected with a header consisting of a user specified 60 character message, the date, the time, and the page number of the report, followed by 2 blank lines. Subsequent calls to EJECT.DP are for the purpose of passing the number of output lines to be written. When insufficient space is available on the current page, a top of form is performed and the line number on the page is reset. Various entry points are available.
ERRCHK.DP	Subroutine called after all verifications for a specific record have been performed. If the record had errors, (ANYERR is TRUE), ERRCHK.DP prints the record contained in STRING, ANYERR is set to FALSE and TOTERR is set to TRUE. If no error existed for the record (ANYERR is FALSE), the subroutine returns to the calling program.
ERRFIL	An entry point to ERRLOG.DP. It is called by the field verification routines BLANKT.DP, RANGIT.DP, VALUIT.DP, and VERFIT.DP to tally the frequency of range, character, and blank errors in the database EC80DB, and to set variables NOGO and ANYERR to TRUE.
ERRLOG.DP	Subroutine called to handle logical errors. It can write 3 forms of error messages (long, short, and ?), tallies the frequencies of logical errors encountered in the database EC80DB, and sets variables ANYERR and NOGO to TRUE.
ES80DB	ALADIN database used to store logical error statements by logical error code for a data file type.

GENER1.DP	Subroutine to provide integer functions that are equivalent to the INFONET computer system (Computer Sciences Corporation) functions FIVAL$, FVRFY$, and FBRKC$. Used for implementations of edit programs using FORTRAN where these CSC system functions are unavailable.
GENER2.DP	Subroutine to provide a character function that closely simulates the INFONET computer system (Computer Sciences Corporation) function FSTR$. Used for implementations of edit programs using FORTRAN where the CSC system function is unavailable.
GENER3.DP	Subroutine to provide a single interface to the INFONET computer system (Computer Sciences Corporation) functions FDEFN$, DOY, TOD, and UDAT2$. Used for implementations of edit programs using FORTRAN where these CSC system functions are unavailable.
ONDSK2.DP	A logical function which tests to see if the passed file name is on the computer disk. The function is set to TRUE if the file is on disk and to FALSE if not.
P.DP	A function used to left pad zeroes on a passed numeric string field. For valid numeric fields, the function is assigned the field with zeroes appended as needed. For non-numeric or blank fields, the function is assigned the field unchanged.
PGROUP.DP	A subroutine used to print a record containing an error(s) and associated records located before and after the record containing an error(s) by accessing a disk file.
POST	Subroutine used to calculate a geographic position (latitude and longitude) for a time, using a starting position and time, speed, and bearing (course).
RANGIT.DP	Field verification routine for integer range (can be modified to handle real values). The field specified by the passed parameters beginning column and width within STRING, is verified to be numeric and then numerically tested to be within or equal to the passed parameter range values. If the field is out of range, an error message is written and optionally, the database EC80DB is tallied with the error occurrence.
STRIP2.DP	A subroutine to prepare a file name for use by the subroutine called by ONDSK2.DP to test the file to be on a computer disk.
UPERST.80	ALADIN program used to add, delete, and change

	records on the database ES80DB.
VALUIT.DP	Field verification routine. The field specified by the passed parameters, beginning column and field width within STRING, is tested to be a valid integer value (can be modified to handle real values). The character field is translated and assigned to integer variable VALUE. If the field is invalid, an error message is written and optionally, the database EC80DB is tallied with the error occurrence.
VERFIT.DP	A field verification routine. The field specified by passed parameters, beginning column and field width within STRING, is validated against a truth-set character string passed as a parameter. If the field is invalid, an error message is written and optionally, the database EC80DB is tallied with the error occurrence.
WRAPUP.DP	A subroutine used to print out a record group, passed as a parameter, when the value of TOTERR is TRUE; TOTERR is then set to FALSE.
YESNO	Subroutine to prompt for a YES or NO response to a question.

V. Named common block ERROR

The named common block ERROR is the central means of communication between the various subroutines used in the data verification system. The common block statement and variable declaration appears in all of the COMMON SUBROUTINES listed in sections IV. Any program or subroutine which directly references any of these variables must contain the common block statement and must declare the variables properly. The common block, variable declarations, and a description of the variables follows.

Named common block ERROR is used by all common subroutines. A cross reference of the variables and the routines is shown in Figure 3. Even though each variable is not directly used, each variable must have type specification to avoid serious storage problems. Below is an example of the FORTRAN coding for the common area, the type specification, and variable description.

```
C
C
      COMMON/ERROR/ANYERR, CARSEQ, DATSET, DSKSEQ, ELEVAL, GRPCOD,
     1NOGO, STRING, TEMP, VALUE, TOTERR, ALLBLK, INFILE
C
      CHARACTER GRPCOD*1, STRING*80, TEMP*80, DATSET*8, INFILE*20
C
      INTEGER CARSEQ, DSKSEQ, ELEVAL, VALUE
C
      LOGICAL ANYERR, NOGO, TOTERR, ALLBLK
C
C
```

There are several categories of variables. Variables STRING, TEMP, and VALUE make data available to the calling routine and the field verification subroutines.

Variables NOGO, ANYERR, and TOTERR inform the users routines and system subroutines (ERRCHK and WRAPUP) as to the occurrence of an error within a field, a record, or a group of records, respectively.

Variables DATSET, GRPCOD, CARSEQ, and DSKSEQ serve as field identifiers: DATSET, denoting the type of data; GRPCOD a further distinction in situations involving multiple-multicard format being verified simultaneously; CARSEQ distinguishing records within multi-record formats; and DSKSEQ relating the absolute record number within the input file (base 1, increment 1).

Variables ELEVAL and ALLBLK control the error report characteristics. ELEVAL controls the complexity of the error message and ALLBLK controls whether all data groups are to be printed regardless of number or types of errors detected.

Figure 3. Variable cross reference for named common block ERROR and the common subroutines.

"A" indicates variable is assigned a value within the phase or subroutine.
"R" indicates variable is referenced within the phase or subroutine.

				Referencing COMMON SUBROUTINES					
				BLANKT.DP			ERRLOG.DP		
VARIABLE NAME	Phase 1	Phase 2	Phase 3	VERFIT.DP	RANGIT.DP	VALUIT.DP	ERRFIL.DP	ERRCHK.DP	WRAPUP.DP
ALLBLK			A						R
ANYERR	A R	A R	A R				R	A R	
CARSEQ	A	A	A	R	R	R			
DATSET			A				R		
DSKSEQ	A R	A R	A R	R	R	R	R		
ECNAME			A				R		
ELEVAL			A				R		
ESNAME			A				R		
INFILE	A R								
GRPCOD			A	R	R	R			
NOGO	R	R	R	A	A R	A	A		
STRING	A R	A R	A R	R	R	R		R	
TEMP				R	R	R			
TOTERR	A R	A R	A R					A	A R
VALUE	R	R	R		R	A			

ALLBLK: This logical variable is set to TRUE when it is desired that subroutine WRAPUP.DP prints all record groups regardless of error condition. A value of FALSE causes WRAPUP.DP to print only those record groups meaning TOTERR is true when record groups (i.e., TOTERR is TRUE when WRAPUP.DP is called).

ANYERR: Logical variable set to TRUE to indicate that a record contains an error and left FALSE when a record does not have an error. Subroutines ERRCHK.DP, ERRFIL, and ERRLOG.DP, and the Phase 1 and Phase 2 coding make direct references to this variable. ERRCHK.DP prints the record stored in STRING if ANYERR is TRUE and then sets it to FALSE. Entry ERRFIL and ERRLOG.DP, which are called for field and logical error conditions respectively, explicitly set ANYERR to TRUE. Phase 1 and Phase 2 coding sets ANYERR to TRUE for an error condition. When all verifications for a record are complete, ANYERR is checked. If TRUE, the record in error is printed.

CARSEQ: The card sequence number within a data set. This variable is assigned the character(s) from a record which contains a sequence number within a multicard format. CARSEQ need not be numeric. For non-multicard formats it should be assigned blank(s). CARSEQ is used by the field verification routines to help specify the location of a field containing the error. The error message written by these routines and the record created on database EC80DB via the call to ERRFIL (an entry point to ERRLOG.DP) contain CARSEQ. This variable must be assigned by the user when the field verification routines are called.

DATSET: An 8 character string used to identify the data being verified. This variable is concatenated with the field and logical error specification code to create a key for EC80DB or ES80DB to indicate what data set type and file contained the error. DATSET contains the data set type identifier code (a string of characters) that indicate the type of data file that is being processed (e.g., SLEDIT80 for 1980 Set Log data). DATSET is used by ERRFIL (entry point to ERRLOG.DP) and ERRLOG.DP. DATSET need only be assigned if EC80DB is opened in preparation to store records. Columns 1-8 of DATSET are concatenated with the logical error code by ERRLOG.DP to create a key used to retrieve records from ES80DB database. ES80DB database contains both long and short error messages.

DSKSEQ: The absolute record sequence number which begins with zero and is incremented by one each time a record is read from the input data file. This value is used to indicate which record contains the error. It appears in the error messages written by the field verification routines and ERRLOG.DP. It is used to indicate the record number by ERRCHK.DP and WRAPUP.DP. It is used for the same purposes in Phase 1 and Phase 2 coding. The value of DSKSEQ must equal the record number of the record currently being verified when calls are made to the field verification routines, ERRLOG.DP, or ERRCHK.DP. It must equal the record number of the last record in a record group when WRAPUP.DP is called.

ELEVAL: The variable codes for the type of logical error message written by ERRLOG.DP, controls whether EC80DB database should be updated by ERRLOG.DP, and controls whether error messages are written by VALUIT.DP.

ELEVAL	MESSAGE?	UPDATED?	MESSAGE?
-1	Code (1)	Yes (4)	No
0	No	No	Yes
1	Code (1)	Yes (4)	Yes
2	Short (2)(5)	Yes (4)	Yes
3	Long (3)(5)	Yes (4)	Yes

Notations for the numbers in parentheses are:

(1) CODE - is an error message consisting of the record number (DSKSEQ) and a message stating that logical error number "ERRORCODE" has failed.

(2) SHORT- is a 4 character error message, retrieved from ES80DB data base element "SHORT MESSAGE" printed in addition to the CODE error area message.

(3) LONG- is a multiple of 4 character error message records retrieved from the ES80DB database element "LONG MESSAGE" printed in addition to the code error message. A maximum of 99 error message records is allowed for each multiple record set.

(4) If the database EC80DB is to be updated, OPLGCT (entry to ERRLOG.DP) must be called prior to calling the data verification subroutines, or ERRLOG.DP and CLLGCT (entry to ERRLOG.DP) should be called at the close of the program. These routines open and close data bases EC80DB or ES80DB, respectively.

(5) To retrieve the short and long error messages, ES80DB database must be on disk pack and subroutine OPLGST (entry to ERRLOG.DP) called prior to the first call to ERRLOG.DP.

GRPCOD: A character string, concatenated as part of the field verification error code, used to identify the data set of a record having a field error. When more than one data set or card format is being simultaneously verified some distinction must be made to avoid ambiguity. GRPCOD becomes part of the key used when storing a field error on EC80DB database. The field verification routines create this key and then call ERRFIL (entry to ERRLOG.DP) to update EC80DB. The user should assign GRPCOD as blank if not otherwise needed. If needed, it should be assigned any code desired, prior to calling the field verification routines for a given record type.

INFILE: A 20 character string containing the input data file name. It is assigned by the Phase 1 coding.

NOGO: A logical variable set to TRUE to indicate that a field, being verified by a field verification routine, is invalid. When a field verification routine is called, NOGO is set to FALSE. If the field is invalid, NOGO is set to true when ERRFIL (entry to ERRLOG.DP) is called by the field verification routines. In this way, the calling program has an indication of whether the field is invalid, and can branch if necessary.

STRING: An 80 character string into which input records are stored. The field verification routines verify STRING. When ERRCHK.DP is called, the record printed out is stored in string.

TEMP: An 80 character string to which is stored the field being verified. The field verification routines extract the field being verified from STRING and assign it to TEMP. **NOTE! Since the length function is not associated with TEMP the length of the field placed there is not known.**

TOTERR: In Phase 3 coding this logical variable is set to TRUE to indicate that a record group contains a record which had an error occurrence. It is also set to TRUE by Phase 1 and Phase 2 coding when any error occurs, as an indication to not go to the next phase. Within Phase 3 coding, TOTERR is set to TRUE by ERRCHK.DP when Any error is TRUE. TOTERR is set to FALSE when WRAPUP.DP is called. Phase 1 and Phase 2 coding require explicit assignments for TOTERR.

VALUE: When subroutine VALUIT.DP is called explicitly by the calling program, or implicitly by calling RANGIT.DP, the integer translation of the field being verified is assigned to VALUE. If the field is invalid or blank, VALUE is assigned the value of zero.

VI. The logical hierarchy for the field verification common subroutines ERRLOG.DP, ERRFIL, ERRCHK.DP, WRAPUP.DP and calling programs.

A hierarchical structure relates the field verification subroutines ERRLOG.DP, ERRFIL (an entry point within ERRLOG.DP), ERRCHK.DP, WRAPUP.DP, and any calling programs. This structure performs the function of informing the different routines about the occurrence of errors at the field level, physical record level, and logical record group level. Logical variables NOGO, ANYERR, and TOTERR (variables within labeled common block ERROR), are set to TRUE or FALSE by these subroutines to control the printing of error records, and to inform the calling subroutines. The structure is designed to reduce the amount of coding during development of edit programs.

The field verification subroutines are used to perform checks on field (variables) as part of the editing process. As each subroutine is called, the logical variable NOGO is set to FALSE. If a field error is detected by any of the field subroutines, ERRFIL is called and ERRFIL sets logical variables NOGO and ANYERR to TRUE. By setting NOGO to TRUE, the calling program is informed that a field had an error. When a logical error is detected, ERRLOG.DP is called to report the error, and ERRLOG.DP sets logical variable ANYERR to TRUE. Following all verifications for a given record, subroutine ERRCHK,DP is called. This subroutine checks logical variable ANYERR. If ANYERR is TRUE, then the character variable STRING containing the input record being edited is printed. Logical variable TOTERR is set to TRUE and logical variable ANYERR to FALSE.

The calling program or edit program is then ready for the next physical record to be edited. When all verifications of a record group have been completed, subroutine WRAPUP.DP is called. WRAPUP.DP prints out the entire record group as a block of records if either of the logical variables ANYERR or ALLBLK (see sections VI and VII) were reset to TRUE. The logical variable TOTERR is then set to false, and the cycle is completed. The main edit program must initialize NOGO, ANYERR, and TOTERR at the beginning of execution, and prior to the next cycle of records to be edited.

IX. Data flow and field reference sequence.

There are fundamental error report format requirements and input data error ramifications which help to mold a program structure. It is the purpose of this section to provide a detailed description of how data is input, data is referenced, and how the report is written. We refer to these three aspects as the program data structure.

Each phase reads from a disk file equated to FORTRAN logical unit 10. Each record is read into STRING, a variable of type character, length 80. The field verification routines do their verifications of a field, described by beginning passed parameter column number and field width within variable STRING. Logical verifications make direct references to fields within STRING. The ANSI standard syntax for a field reference is of the following form.

```
STRING (BEG:END) where  BEG is the beginning column number and
                        END is the ending column number.

e.g.,     IF (STRING(4:6) .GT. STRING(13:15)) GO TO 100
```

Reference of this nature is self-documenting. The field which is referenced is known simply by looking at the input data format layout. References of this format are useful for comparisons in which no arithmetic is required. If a field needs to be added, subtracted, multiplied, divided, or some other computation performed, the data field has to be translated to its numeric equivalent. This utility is provided by the field verification routines RANGIT.DP and VALUIT.DP. Both these routines assign common block variable value with the integer, numeric equivalent of the field, providing the field is valid numeric. Following a validation, a user designated variable may be assigned a value via the arithmetic assignment statement.

```
MYVAR = VALUE
```

MYVAR is available for computations as necessary.

```
CALL RANGIT (5,3,0,500)
MYVAR = VALUE

     or

CALL RANGIT (8,2,0,200)
IPOST = VALUE
```

It is in the subsequent validations where the three phases differ. In phase 1, a record with errors is printed out by phase 1 coding using a write statement. In phase 2, STRING is passed to subroutine PGROUP.DP where it is printed out. In phase 3, subroutine ERRCHK.DP is called to print the record with errors (the record being stored in STRING). In addition, the record stored in STRING is assigned to a dimensioned character string of length 80. This type of variable has the capacity to store many record images. When all of the record group has been processed, the record stored in the dimensioned character string, is passed along with a count of the records to subroutine WRAPUP.DP which then selectively prints the record group. Figures 4, 5, and 6 show the data flow for phases 1, 2, and 3, respectively.

The perspective presented thus far is as if all logical verifications are performed on a single, 80-column long, physical record stored in the variable STRING. This is not always the case for phases 2 and 3. Logical verifications can be performed between the current record stored in STRING and previous records. The previous records were saved in a numeric or character variable. When possible, the references were made to a field within a record, using the beginning and ending columns. References can be made of the form:

```
   STORE  (3)  (5:8)
```

The field referred to is the fifth through eighth column of the third record stored in the record group storage area called STORE.

```
   IF (STRING (7:9) .GT. STORE(2) (6:8)) GO TO 20
```

All of the logical operators. LT, LE, EQ, NE, GT, and GE are permitted for comparing character strings. But, since not all fields are zero filled, comparing a field which is blank filled with one which is zero filled will produce an erroneous result. The common subroutine function P.DP was used to temporarily pad a field with leading zeroes prior to performing a field edit verification.

```
      IF ( P(STRING (28:29)) .EQ. '02') GO TO 20
```

Figure 4. Phase 1 flowchart.

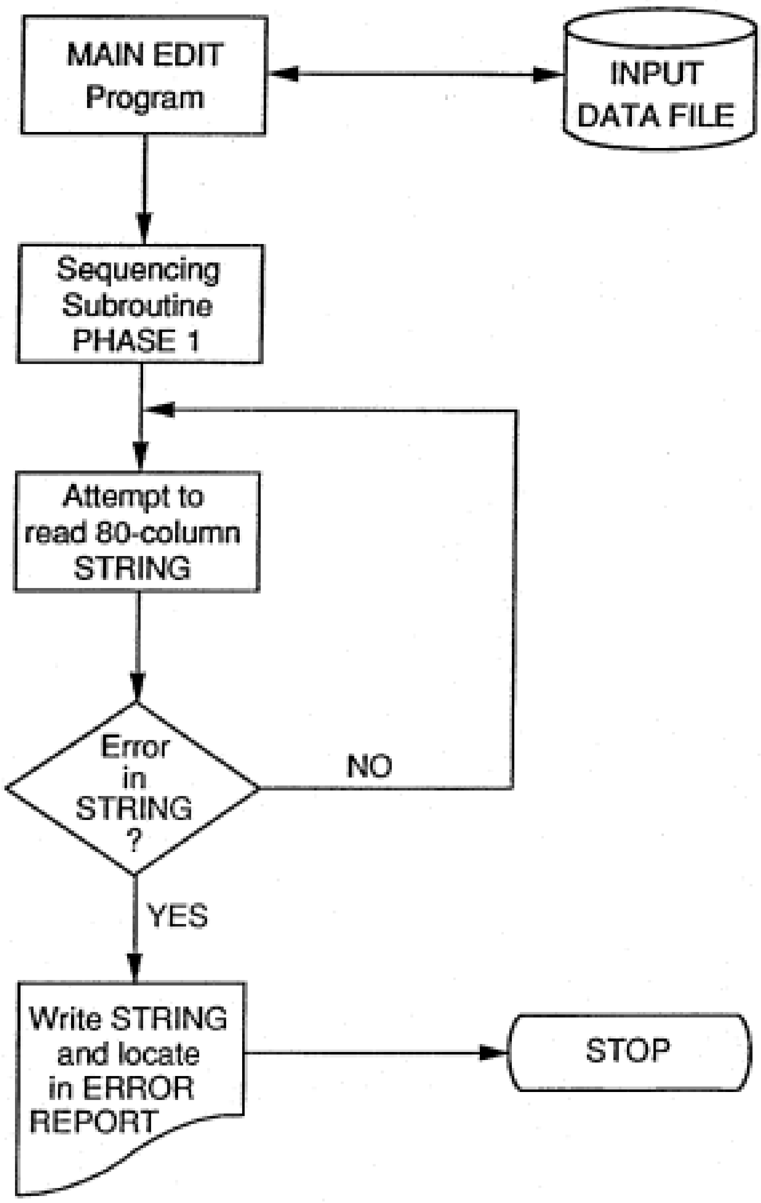

MAIN EDIT Program
INPUT DATA FILE
Sequencing Subroutine PHASE 1
Attempt to read 80-column STRING
Error in STRING ?
NO
YES
Write STRING and locate in ERROR REPORT
STOP

Figure 5. Phase 2 flowchart.

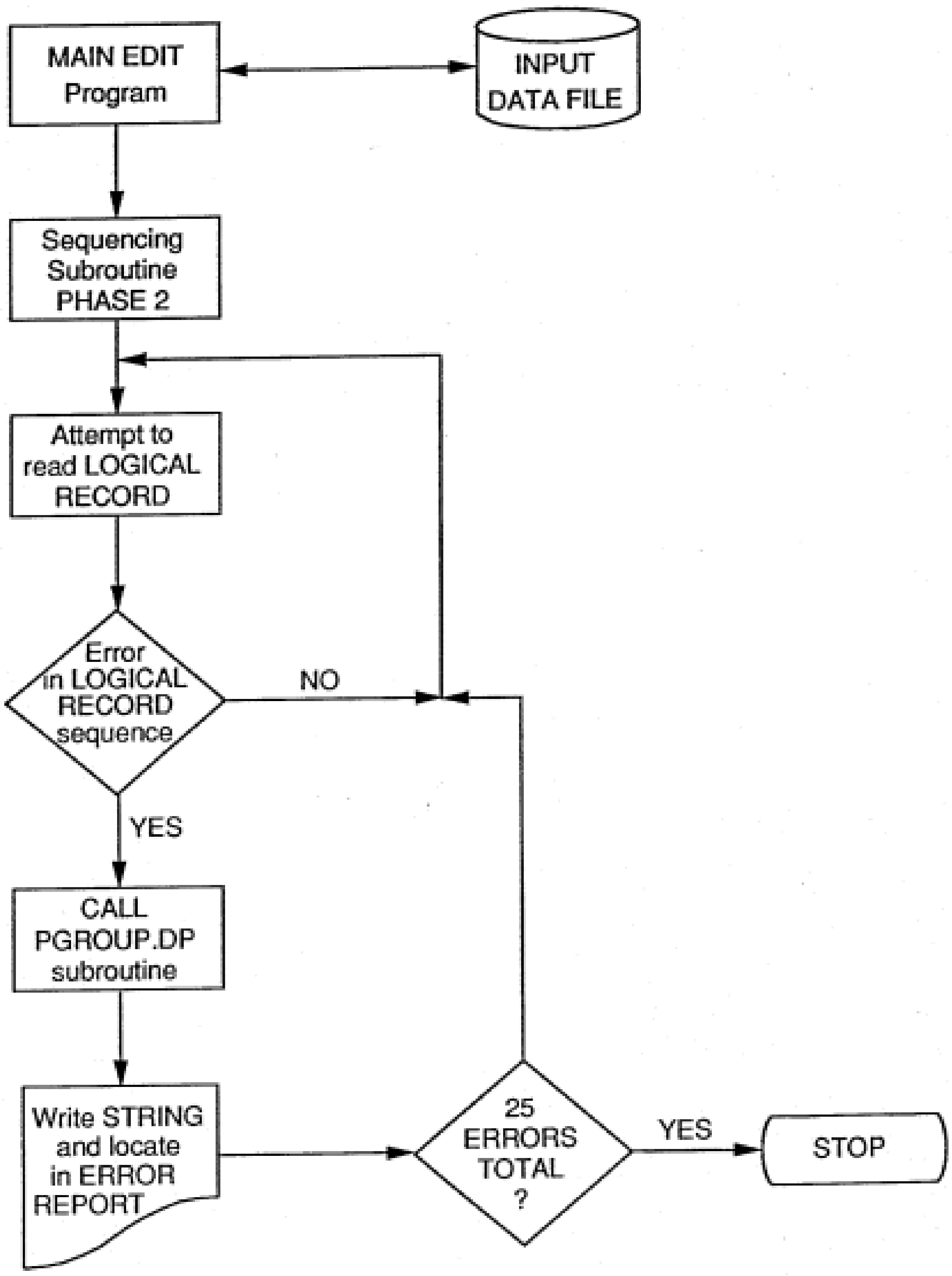
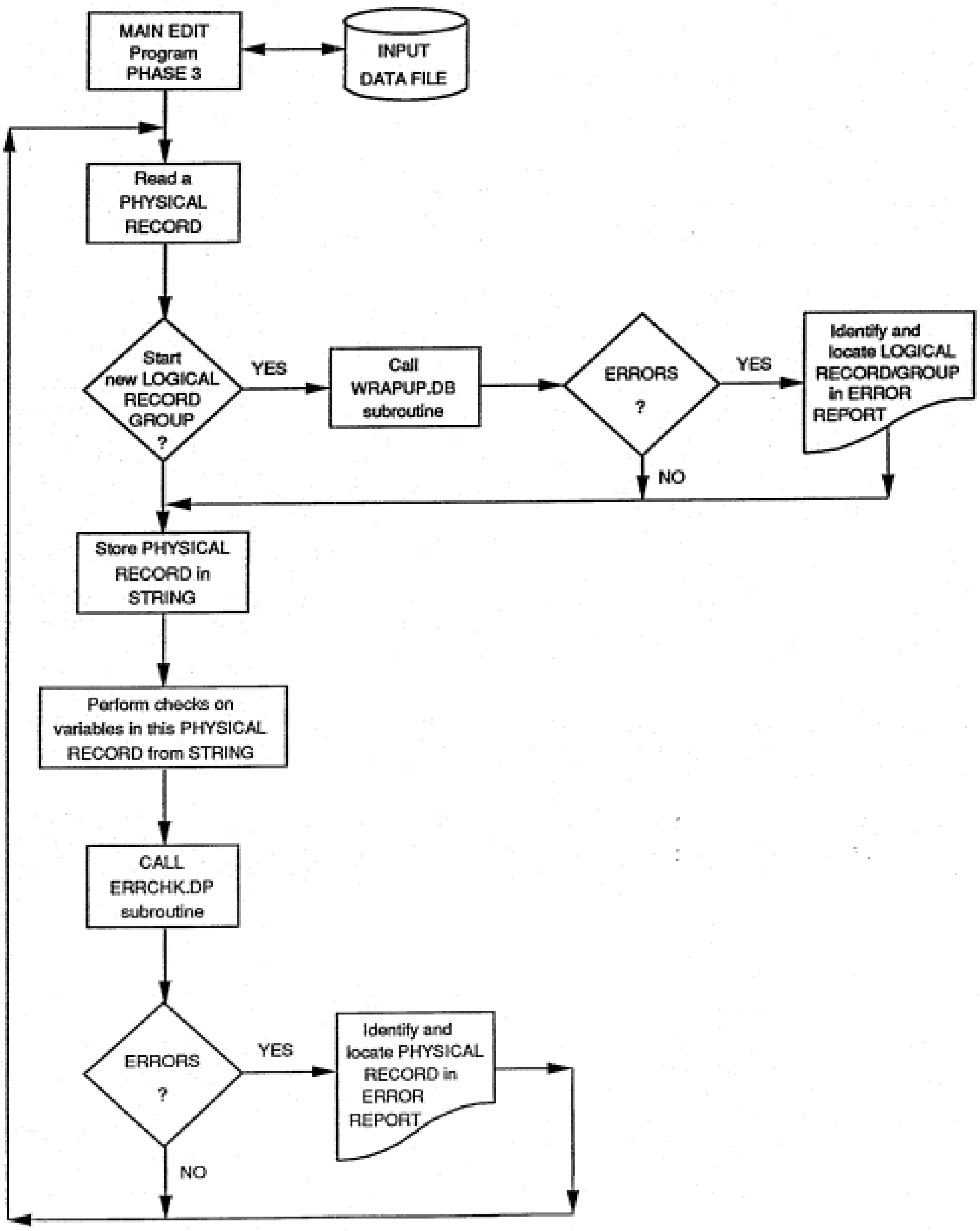

MAIN EDIT Program
INPUT DATA FILE
Sequencing Subroutine PHASE 2
Attempt to read LOGICAL RECORD
Error in LOGICAL RECORD sequence
NO
YES
CALL PGROUP.DP subroutine
Write STRING and locate in ERROR REPORT
25 ERRORS TOTAL ?
YES
STOP

Figure 6. Phase 3 flowchart.


MAIN EDIT Program PHASE 3
INPUT DATA FILE
Read a PHYSICAL RECORD
Start new LOGICAL RECORD GROUP ?
YES
Call WRAPUP.DB subroutine
ERRORS ?
YES
Identify and locate LOGICAL RECORD/GROUP in ERROR REPORT
NO
Store PHYSICAL RECORD in STRING
Perform checks on variables in this PHYSICAL RECORD from STRING
CALL ERRCHK.DP subroutine
ERRORS ?
YES
Identify and locate PHYSICAL RECORD in ERROR REPORT
NO

VIII. Error Counts (EC80DB) and Error Statements (ES80DB) databases.

Two ALADIN databases were created to contain a tally of the number of error occurrences (EC80DB) or a record of the logical error statements (ES80DB). The error count database (EC80DB) records the number of error occurrences (field and logical) detected each time the edit program is executed. Tallies are maintained by data file type (e.g., Set Log, Marine Mammal Effort and Sightings), subsets of data file type (e.g., Porpoise versus Non-porpoise Set Logs), errortype (blank, range, logical, or character), and year. Retrieval from this database is accomplished using the ALADIN program DERCT.80 which produces either a formatted computer disk file of the frequency of error tallies, or a printer report of the same information.

The error statement database (ES80DB) contains both long and short error messages that describe the error detected by the edit program. A listing of the contents of this database is printed and archived as the Logical Error Statements book for all edit programs developed during a calendar year. It serves to document the criteria used to code the logical errors, and identifies the specific logical error detected during execution of the edit program. Records are appended to the database ES80DB using the FORTRAN program UPEROR.MN, or the ALADIN program UPERST.80, and an input file of statements described below. Many of the error statements remained unchanged between years, and the archived records within ES80DB, for instance, can be used to create records for 1981 edit programs (ES81DB). Retrieval from this database is accomplished using the ALADIN program DERST.80 which produces either a formatted computer disk file of the frequency of error tallies, or a printer report of the same information.

There is a 24-character length key for both databases, constructed using parameters passed by the subroutines, or contained in the labeled common block ERROR. The key is built using the following parameters.

DATASETID: Columns 1-8 identifies the field collection or coding format by year. Passed from labeled common block ERROR as DATSET (MUST BE ONE OF THE FOLLOWING).

CSEDIT80
SLEDIT80
VAEDIT80
MMEDIT80
BLEDIT80
LHEDIT80
ASEDIT80

GRPCOD: Column 9 identifies multi-format field collection or coding formats. Passed from labeled common block ERROR as GRPCOD (MUST BE ONE OF THE FOLLOWING).

'N'	-	Non-porpoise	Set Log	SLEDIT
'P'	-	Porpoise	Set Log	SLEDIT
'E'	-	Effort		MMEDIT
'S'	-	Sightings		MMEDIT
' '	-	all others		VAEDIT, BLEDIT, CSEDIT

ERRORTYPE: Column 10 identifies the nature of the field error or logical error. Passed from labeled common block ERROR as ERRORTYPE (MUST BE ONE OF THE FOLLOWING).

B	-	Blank error	(Field)
C	-	Character error	(Field)
L	-	Logical error	(Logical)
R	-	Range error	(Field)

CARDSEQ: Columns 11-12 for FIELD errors, the card sequence of a field (e.g., "01") within a multicard format data set. For a single card format the code is "01" (Length 2). Blank for LOGICAL errors. Passed from named common block ERROR as CARSEQ.

BEGINCOL: Columns 13-14 for FIELD errors, the beginning column number of (e.g., "29") the field having an error. (Length 2): Blank for LOGICAL errors. Passed by the calling subroutine.

ENDINGCOL: Columns 15-16 for FIELD errors, the ending column of a field (e.g., "34") having an error. (Length 2). Blank for LOGICAL errors. Passed by the calling subroutine.

ERRORCODE: Columns 17-24 for LOGICAL errors contain the logical error number of the statement. Passed by the calling subroutine as ERRCODE.

Logical error statements are coded onto FORTRAN coding forms which are 80 columns in length. To accommodate statements that are longer than the SHORT STATEMENT length of 54 characters, columns 25-26 of the coding format are reserved to indicate the sequential number of coding lines needed for any length statement (up to 99 lines of 54 characters each). When the keypunched data is used as input to the update procedures (ALADIN program UPERST.80 or the FORTRAN program UPEROR.MN, the sequential numbers in columns 25-26 are utilized but not stored in the database ES80DB. The ALADIN program DERST.80 restores the sequential numbers to columns 25-26 when a "dump" is made of the database.

ERROR COUNT DATABASE (EC80DB)

We desired to keep track of which errors were most frequently detected in the input data to assist in subsequent training of observers and to address possible ambiguities with data definitions. The two different error types, field and logical, are recorded via the calls to the field verification subroutines (BLANKT.DP, RANGIT.DP, VALUIT.DP, VERFIT.DP) or the logical error type subroutine ERRLOG.DP. When a field verification routine is called and an error is detected, an error code "key" is constructed and passed as a parameter via a call to ERRFIL. The "key" is constructed to access the ALADIN database EC80DB for the particular error type, etc. detected.

Subroutine ERRFIL, all entry point of ERRLOG.DP, concatenated a key consisting of the data file type, the error code, the data file name code, and blanks. The data file type is derived from columns 1-8 of labeled common block ERROR element DATSET. The data file name code is columns 9-12 of DATSET. A retrieval from EC80DB is attempted using this key. If the record already exists, the occurrence is tallied. If a record does not exist, a new record is created and the tally set to 1. In this way field errors are tallied in the ERROR COUNT database.

Examples of valid keys for the database EC80DB are:

(DATASETID+GRPCOD+ERRORTYPE+CARDSEQ+BEGINCOL+ENDCOL+ERRORCODE)

'SLEDIT80NL ERROR112'	=	1980 non-porpoise setlog logical error 112
'SLEDIT80PR010306 '	=	1980 setlog range error for physical card 1, columns 3-6

The formatted structure of the database EC80DB is as follows:

Element name	Columns	Element type
ERROR KEY	1 - 24	Character
DATASETID	1 - 8	Character
GRPCOD	9 - 9	Character
ERRORTYPE	10 - 10	Character
CARDSEQ	11 - 12	Character
BEGINCOL	13 - 14	Character
ENDINGCOL	15 - 16	Character
ERRORCODE	17 - 24	Character
COUNT	25 - 28	Integer

ERROR STATEMENT DATABASE (ES80DB)

Logical errors are tallied the same as the field errors, the only difference being the user supplies the error code. When a logical error is detected, subroutine ERRLOG.DP is called and passed the user specified eight character logical error code (e.g., "ERROR001", "ERROR002"). Error codes correspond to logical error statements which are stored within the database ES80DB.

ES80SB database is a storage area for the logical error messages corresponding to the logical error verifications performed within the main edit programs (Phase 3). The error messages were the basis by which the logical verifications were coded by programmers. They are the explanation of the logical error code which is printed on the error report, and documentation of the logical verifications performed. They may be accessed and printed directly on the error report or printed for reference.

When logical errors are reported via the call to ERRLOG.DP, the logical error code is passed as a parameter. Optionally, ERRLOG.DP may be told to retrieve the long or short message from ES80DB and display it or the printer. This action is controlled by the value of ELEVAL (see sections VI and VII). What ERRLOG.DP does to retrieve the necessary message is to form a key consisting of the data file identifier code (columns 1-8 of DATSET), the group code (GRPCOD), the error type (ERRORTYPE), six blanks, and the 8 character logical error code (e.g., ERROR001 or ERROR002). This key is used to access the database ES80DB. If a record does not exist for that key, ERRLOG.DP prints of that information. Examples of logical error statements are included in Appendix 3 of this document.

Examples of valid keys for the database ES80DB are:

```
(DATASETID+GRPCOD+ERRORTYPE+'      '+ERRORCODE

'SLEDIT80NL      ERROR112' = 1980  non-porpoise setlog  logical
                             error 112

'SLEDIT80PL      ERROR001' = 1980  porpoise setlog  logical
                             error 001
```

The formatted structure of the database ES80DB is as follows:

<u>Element name</u>	<u>Columns</u>	<u>Element type</u>
ERROR KEY	1 - 24	Character
DATASETID	1 - 8	Character
GRPCOD	9 - 9	Character
ERRORTYPE	10 - 10	Character
blanks	11 - 16	blanks
ERRORCODE	17 - 24	Character
SHORT STATEMENT	25 - 78	Character
LONG STATEMENT	79 -133	Character

IX. Acknowledgements

There have been a great many persons involved in the evolution of the porpoise data management system. Dr. W.F. Perrin (SWFC) encouraged us to apply computer capabilities to these data and provided the initial support and direction in our doing so. D. Roll and D. Mackett (SWFC) provided subsequent support and guidance in development of the first edit programs. Many of the observers who collected data while aboard tuna purse-seiners provided useful comments on the data forms, data definitions, and editing criteria. The data groups at the SWFC and at the SWR office were an integral part of this development (F. Ralston and J. Scordino). We also thank the numerous computer programmers who have developed the edit programs using the common subroutines, and who in some cases, have recommended improvements and additional "common subroutines". Special recognition is deserved by K. Wallace (SWFC) who worked as an observer, data editor, computer programmer, and who during the 1980s, implemented the common subroutine package on yet another computer system. Ken also provided helpful comments during the drafting of this report.

References

Butler, R.L. and C.W. Oliver. 1980. Program descriptions, listings, and documentation for the common edit subroutines: Porpoise Data Management System. Southwest Fisheries Center internal reports. Southwest Fisheries Center, La Jolla, California. 100-plus pages.

Oliver, C.W. 1983. Documentation of aerial survey sighting and transect forms for the 1977 and 1979 eastern tropical Pacific cetacean surveys. Southwest Fisheries Center Admin. Report No. LJ-83-20. Southwest Fisheries Center, La Jolla, California. 35p.

Perrin, W.F. 1975. Variation of the spotted and spinner porpoise (genus Stenella) in the eastern tropical Pacific and Hawaii. Bull. Scripps Inst. Oceanography, Univ. of Calif. 206pp.

Appendix 1. How to read intra-variable logic: explains the blank, range, logical error coding utilized during 1974-1978 for edit programs.

HOW TO READ INTRA-VARIABLE-LOGIC

Intra-variable logic is a coined expression for the Boolean "IF, THEN" type error checking system designed to allow complex variable relationships to be handled without new program coding. Its positive attributes include ease of coding, generalized error reporting, and documentation of the errors checked with a subroutine called TRCWOB that prints the error checks performed in an easily read form. This document is provided to clarify the syntax of the output provided by the program TRCWOB, which provides a list of all error checked for by INTRA-VARIABLE LOGIC code. Some terms should clarified:

CARD AND COLUMN (CARD.COL):

A symbolic means of identifying variables associated with a logical record consisting of one or more 80-column physical records. Variables are located on a certain card within multi-card formats and begin with a certain column. A contraction was formed (e g., 2.05 for the variable located on card 2 beginning in column 5).

VARIABLES

The input data is contained within array locations. Certain constants are also stored within the array string. Thus data and constants may be referenced from the same string. The term "Data-variable" coins for an element of the array whose value arises from the input data source. The term "Constant-variable" coins for constants.

Within this document, variables are referenced by card and column number, and all variables have their array index values listed at the end of the statement line following dashes and a referencing number on the statements which are referenced by the error reports themselves.

Four Boolean logical operators are used: LT (less than), EQ (equal), GT (greater than), and NE (not equal). The logical connectors "OR" and "AND" are also used. Values within parentheses contain data or "????". Data exists for constant-variables, and "????" for data-variables.

The logic flow is entered systematically although each variable may, or may not, have logical relationships coded with dependencies upon its value. However, the logic structure may only be reached when the variable is non-blank and within range. Each logic construct begins with a card-column value to the far left identifying the begin variable. Following are the two types of statements:

1. 1.01 IF VAR# 1.01 IS EQ VAR# 2.01 (????)--1--20

2. 1.01 SINCE VAR# 1.01 IS WITHIN BOUNDS--1

The latter statements merely reflects that since variable 1.01 is non-blank and within bounds, there is going to be some resultant relationship, to be fulfilled, while the former sets up an "IF" relationship to build upon. Following each expression will be one or more resultant contingencies of the form:

VAR# 2.15 MUST BE GT VAR# 1.15 (????)--41--7

Thus an "IF, THEN" expression may be formed as follows:

1.01 IF VAR# 1.01 IS EQ VAR# 2.01 (????)--1--20

VAR# 2.15 MUST BE GT VAR# 1.15 (????)--41--7

The above translate to say "IF variable 1.01 is equal to the stored value of the data-variable 2.01, THEN the stored data-variable 2.15 must be greater than the stored value of data-variable 1.15).

Use of "AND" or "OR".

The word "AND" is used to express additional "IF" statements within two areas.

1. **Compound "IF":** An "IF" directly followed by 1 or more "AND IF" expressions means that both (all) "IF" relationships must be valid before the resultant clause becomes active. The resultant clause may indicate an error or no error depending upon the construct of the expression.

   IF VAR# 1.24 is NE VAR# 2.78 ( )--10--70

   AND IF VAR# 1.17 is EQ VAR# 1.24 (????)--8--10

   VAR# 2.13 must be GT VAR# 1.13 (????)--40--6--

2. **Additional "IF"s** : An "AND IF" following an "IF" interspersed by 1 or more resultant clauses expresses an entirely new "IF" contingency.

   IF VAR# 2.17 IS EQ VAR# 2.24 (????)-42-44

   VAR# 1.52 MUST BE EQ VAR# 2.84(5)--23--76--

   AND IF VAR# 2.24 IS NE VAR# 2.78 ( )--44--70

   VAR# 2.24 MUST BE EQ VAR# 2.17 (????)--44--42--

```
     OR VAR# 2.24 MUST BE GT VAR# 2.17 (????)--44--42--
```

The logical connector "OR" is used to connect resultant clauses IL# as seen In the last example. However, the "OR" is only in effect for the resultants having the "OR" and for the first previous resultant expression before the expression having the "OR".

```
     IF VAR# 2.48 IS EQ VAR# 2.79 (1)--54--71

        VAR# 2.51 MUST BE NE VAR# 2.78 (    )--55--70--

        VAR# 2.49 MUST BE GT VAR# 2.80 (0 )--55--72 --

     OR VAR# 2.54 MUST BE GT VAR# (2.80 (0 )--56--72 --
```

Only the last two resultants are connected. The first resultant must be valid regardless of the validity of the following two resultants or an error report would occur.

To make use of this readable intra-variable logic, one looks at the "Begin variable" on the edit program output associated with the error statement. This begin will be found on the HUMN output listing on the far left of the page (e.g., SLHUMN for Set Log data). To find a specific error statement, the edit program output has a reference number (the one with the dashes). This number will also be found to be the last number on the associated THEN statement.

R.W. Butler second draft (8/25/75); revised C.W. Oliver

Appendix 2. Conversion considerations for the COMMON SUBROUTINES written in Computer Sciences Corporation (CSC) Fortran language

CSC FORTRAN CONVERSION EFFORT (1979-1980)

In order to convert CSC INFONET Fortran to some other system's Fortran, the programmer should be aware of a number of problem areas. Of primary concern is that the new Fortrans have string capabilities CHARACTER, concatenation (//), substring (STRVAR (5:7)), character functions, passed length string variables to functions and subroutines (FUNCTION A(STRING); CHARACTER STRING*(*)), and passed length string functions (CHARACTER *(*) FUNCTION STRING (A,B)).

The Porpoise Data Verification System "COMMON SUBROUTINES" used do develop edit programs were modified so that they never interface directly with CSC system routines, except for programs GENER1, GENER2, or GENER3 and those routines that are dependent on CSC data access or naming conventions; ONDSK2, STRIP2, ERRLOG, and CSEQ.P1 (CSEQ.P2 has been eliminated).

GENER1 has as entries, integer functions as follows:

1. FYRFYX - identical to CSC's FVRFY$.
2. FBRKCX - identical to CSC's FBRKC$.
3. FIVALX - identical to CSC's FIVAL$.

For these routines to be used in a program, they must be defined as INTEGER (i.e. INTEGER FVRFYX, FIVALX). There is a logical variable in GENER1 called CSC, which is currently set to .TRUE. and which, for efficiency, causes these functions to involve the appropriate CSC functions. On converting to a new system, the variable CSC needs to be set .FALSE., and all function references within GENER1 that reference CSC routines need to be made comments or the section of code removed.

GENER2 has a single entry as a character function, FSTRX. This function returns a character string equivalent of an integer with leading zeros. The program which involves this function must specify the character length of the returned string (i.e. CHARACTER FSTRX*5, will return an integer right-justified in a five-character string with leading zeros).

The variable CSC (see above) will determine if CSC functions will be involved or if an "ENCODE" will convert the string. The form of the ENCODE may have to be changed to conform with the system being used.

GENER3 has as subroutine entries:

1. FDEFNX - identical to four argument call to CSC's FDEFN$.

2. FDEFN5 - identical to five argument call to CSC's FDEFN$.

3. DOYX - identical to CSC's DOY.

4. TODX - identical to CSC's TOD.

5. UDAT2X - identical to six argument call to CSC's UDAT2$.

These subroutines should have, on any system to which conversion is made, equivalent functions or subroutines. These special entries were created only to eliminate the need for a conversion programmer to have to find all references to these routines in the various edit packages,

The programs ONDSK2, STRIP2, ERRLOG, and CSEQ.P1 have not been modified, nor have their calls, even though they are CSC dependent. These routines depend on CSTS file naming conventions, MANAGE data base capabilities or special capabilities of CSC's editor. On conversion, they will have to be specially rewritten.

Corrections that were made to the Porpoise Data Verification System COMMON SUBROUTINES used with edit programs are:

1. Variable length character functions P and FSTR$. Function P remains, but the routine that involves P must specify a fixed length (i.e. CHARACTER P*8). References to FSTR$ were changed to FSTRX with a fixed length specification (i.e. CHARACTER FSTRX*5) and any programming changes required to handle a fixed length return string with leading zeroes.

2. Variable length character strings were changed to fixed length.

3. The intrinsic function SUBSTR. All substring references of the form SUBSTR (STRNG,N,L) were changed to the ANSI standard of the form STRNG(N:M), where N is the position of the beginning character and M the ending character position within the string STRNG of the needed substring.

4. ENCODE's and DECODE`s. Since the syntax of ENCODE and DECODE is different for different system Fortrans, they were eliminated from the porpoise Data System edit routines by using other available routines for converting strings to numeric and visa versa, mainly FVRFYX and FSTRX.

5. READ with PROMPT. All reads with prompt as in;

```
READ(5,*,PROMPT='ENTER  DATA-') INPV
```

   were eliminated with combination writes and reads.

6. CSC Routine FVRFY$, FBRKC$, FIVAL$, FDEFN$, DOY, TOD, and UDAT2$. These routine calls (or function references) were changed to FVRFYX, FBRKCX, FIVALX, FDEFNX (for a 4 argument

call or FDEFN5 for a 5 argument; all), DOYX, TODX, and UDAT2X. In addition FVRFYX, FBRKCX, and FIVALX needed an integer type specification in the calling routine (i.e. INTEGER FVRFYX, FBRKCX).

In addition possible problems may exist with other systems with Fortran programs if a conversion from CSC is attempted. Some of these potential problem areas are:

1. CSC Routines FSEQ$, FCHR$, FVAL$, FDVAL$, FTRMM$, FTRMB$, FTRIM$, FPAD$, and OBEY.

2. Read from internal storage (internal file) as in:

   ```
   READ (DATA, 10) VAL
   ```

   This type statement can normally be replaced with a DECODE statement.

3. Any logic which depends on 6 characters per word.

Note: All of the above mentioned special CSC routines or statements can be referenced in CSC's Fortran Reference Manual and related Network Release Manual's, or the CSC Technical Notes.

Appendix 3. The main edit program (CSEDIT.81) and sequencing subroutine (CS00CS.81) utilized on data collected on the 1981 Cruise Specification Record data form.

The 1981 Cruise Specification Record edit program was written during 1980 for use with data to be collected during 1981. The common subroutines initially used were the 1980 versions described within this document. During the conversion of the edit programs (and the common subroutines) to the UCSD VAX computer system, some of the common subroutines were subsequently modified. This example represents the coding which was performed using the 1980 subroutines on the CSC computer system during 1980.

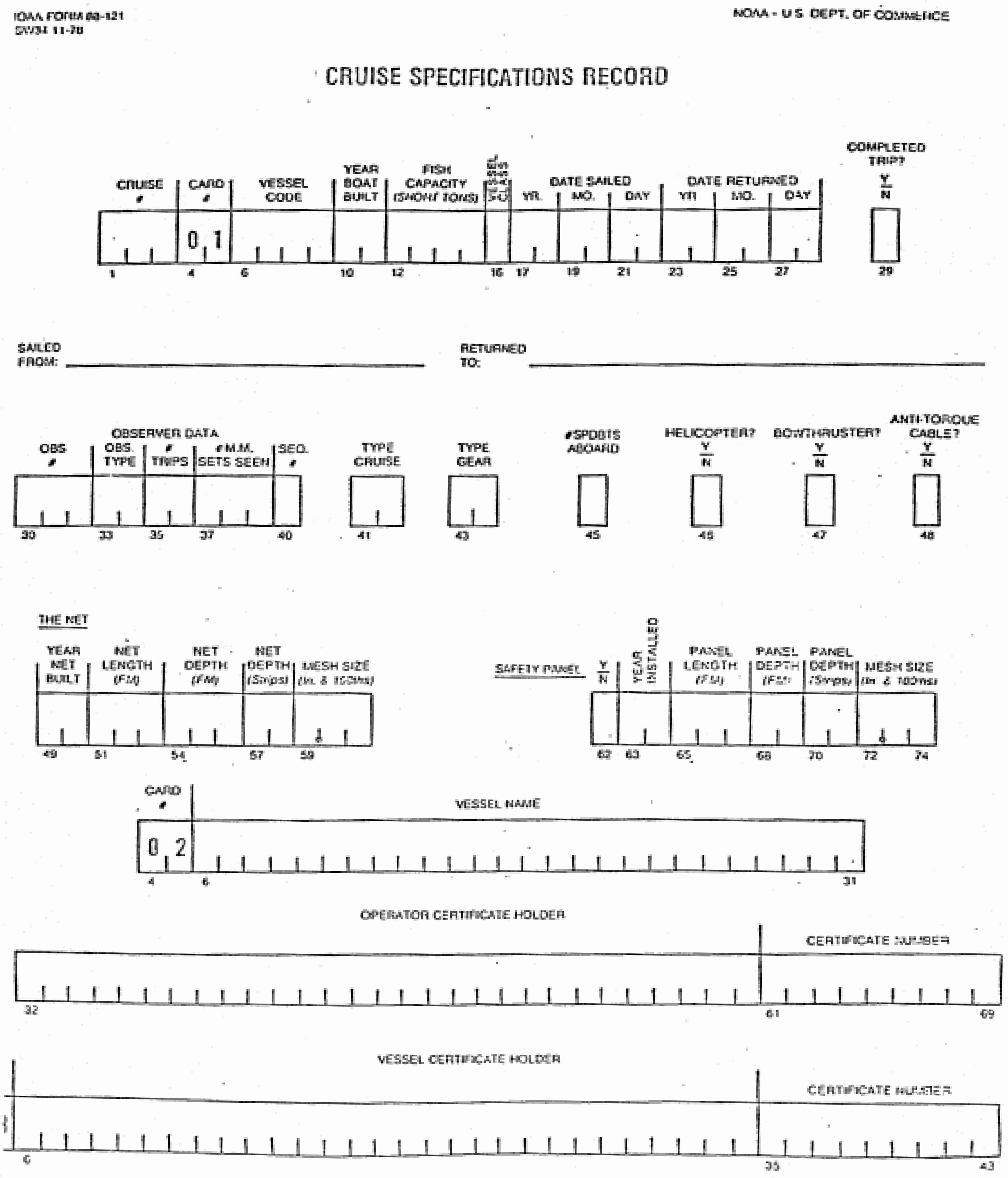
The 1981 Cruise Specification Record data form (Figure 1) was completed for each observed trip (IATTC or NMFS observer). A logical record consists of three 80-column long physical records. The physical records undergo edit by the main edit program (CSEDIT.81). When a logical record is considered "ALL OK", it is appended to the archival database for the year (e.g. CS81DB.DAT).

This appendix is provided to demonstrate the program code for a fixed-format logical record consisting of three physical records which has no relationship with another logical record. This is a relatively simple record to program an edit for because there is a single logical record grouping to edit, consisting of three physical records.

The blank and range criteria used to implemented within the edit program code is provided in Table 1, and the logical error criteria in Table 2. The coding format and data definitions for this record is available in the 1981 Observers Field Manual archived at the SWFSC, La Jolla, California.

The program listings are provided for the main edit (CSEDIT81.FOR) and sequencing subroutine (CSSEQ81.FOR).

Figure 7. 1981 Cruise Specification Record data form.



NOAA FORM 88-121
SW34 11-78

NOAA - U.S. DEPT. OF COMMERCE

CRUISE SPECIFICATIONS RECORD

CRUISE #	CARD #	VESSEL CODE	YEAR BOAT BUILT	FISH CAPACITY (SHORT TONS)		DATE SAILED YR.	DATE SAILED MO.	DATE SAILED DAY	DATE RETURNED YR	DATE RETURNED MO.	DATE RETURNED DAY	COMPLETED TRIP? Y/N
	0 1											
1	4	6	10	12	16	17	19	21	23	25	27	29

SAILED FROM: ______________________ RETURNED TO: ______________________

OBS #	OBSERVER DATA OBS. TYPE	OBSERVER DATA # TRIPS	OBSERVER DATA # M.M. SETS SEEN	SEQ. #	TYPE CRUISE	TYPE GEAR	#SPDBTS ABOARD	HELICOPTER? Y/N	BOWTHRUSTER? Y/N	ANTI-TORQUE CABLE? Y/N
30	33	35	37	40	41	43	45	46	47	48

THE NET

YEAR NET BUILT	NET LENGTH (FM)	NET DEPTH (FM)	NET DEPTH (Strips)	MESH SIZE (in. & 100ths)
49	51	54	57	59

SAFETY PANEL

Y/N	YEAR INSTALLED	PANEL LENGTH (FM)	PANEL DEPTH (FM)	PANEL DEPTH (Strips)	MESH SIZE (in. & 100ths)
62	63	65	68	70	72 74

CARD #	VESSEL NAME
0 2	
4	6 31

OPERATOR CERTIFICATE HOLDER	CERTIFICATE NUMBER
32	61 69

VESSEL CERTIFICATE HOLDER	CERTIFICATE NUMBER
6	35 43

1981 Cruise Specifications Record Form
Data Element Blankness and Range Editing Specifications

Record 1 of 3

Data Element	Columns	Character type	Blankness OK?	Range Lower	Range Upper
Cruise Number	1-3	N	No	670	785
Record Number	4-5	N	No	1	1
Vessel Code	6-9	N	No	29	460
Year Boat Built	10-11	N	No	44	80
Fish Capacity (short tons)	12-15	N	No	401	2000
Vessel Class	16	N	No	2	3
Date Sailed -Yr	17-18	N	No	81	81
Date Sailed -Mo	19-20	N	No	1	12
Date Sailed -Day	21-22	N	No	1	31
Date Returned -Yr	23-24	N	No	81	81
Date Returned -Mo	25-26	N	No	1	12
Date Returned -Day	27-28	N	No	1	31
Completed Trip ? Y/N	29	N	No	1	2
Observer Number	30-32	N	No	34	370
Observer Type	33-34	N	No	1	6
Number of Trips	35-36	N	No	0	6
Number of marine mammal sets seen	37-39	N	No	0	300
Sequence Number	40	N	No	1	3
Type Cruise	41-42	N	No	1	3
Type Gear	43-44	N	No	1	8
Number of Speedboats	45	N	No	4	6
Helicopter ? Y/N	46	N	No	1	2
Bowthruster ? Y/N	47	N	No	1	2
Anti-torque Cable ? Y/N	48	N	No	1	2
Year Net Built	49-50	N	No	59	80
Net Length (FM)	51-53	N	No	480	800
Net Depth (FM)	54-56	N	No	48	90
Net Depth (STRIPS)	57-58	N	No	8	15
Mesh Size (In. & 1000ths)	59-61	N	No	375	425
Safety Panel ? Y/N	62	N	No	1	2
Year Panel Installed	63-64	N	No	76	81
Panel Length (FM)	65-67	N	No	162	200
Panel Depth (FM)	68-69	N	No	6	20
Panel Depth (STRIPS)	70-71	N	No	1	4
Mesh Size (In. & 100ths)	72-74	N	No	113	125

A = Alpha　　B = Blank　　N = Numeric

1981 Cruise Specifications Record Form
Data Element Blankness and Range Editing Specifications

Record 2 of 3

Data Element	Columns	Character type	Blankness OK?	Range Lower	Range Upper
Cruise Number	1-3	N	No	670	785
Record Number	4-5	N	No	2	2
Vessel Name	6-31	A	Yes		
Subpart of name	6-10	A	No		
Operator Certificate Holder	32-60	A	Yes		
Subpart of holder	32-34	A	No		
Certificate Number	61-69	A	No		
Subpart of certificate	61-63	A	No		
Subpart of certificate	64-66	N	No	812	812
Subpart of certificate	67-69	N	No	501	640

A = Alpha B = Blank N = Numeric

1981 Cruise Specifications Record Form
Data Element Blankness and Range Editing Specifications

Record 3 of 3

Data Element	Columns	Character type	Blankness OK?	Range Lower	Range Upper
Cruise Number	1-3	N	No	670	785
Record Number	4-5	N	No	3	3
Vessel Certificate Holder	6-34	A	Yes		
Subpart of holder	6-8	A	No		
Certificate Number	35-43	A	No		
Subpart of certificate	35-37	A	No		
Subpart of certificate	38-40	N	No	812	812
Subpart of certificate	41-43	N	No	1	140

A = Alpha B = Blank N = Numeric

1981 Cruise Specifications Data (1981 format): BLANK, RANGE, and LOGICAL error criteria.

CSEDIT81 L

ERROR001	IF THE YEAR BOAT BUILT IS GREATER THAN 1960 , AND THE FISH CAPACITY IS EQUAL TO OR GREATER THAN 0400 TONS... THE VESSEL CLASS MUST BE EQUAL TO 3.
ERROR002	IF THE YEAR BOAT BUILT IS LESS THAN 1961, AND THE FISH CAPACITY IS EQUAL TO OR GREATER THAN 0400 TONS ... THE VESSEL CLASS MUST BE EQUAL TO 2.
ERROR003	IF THE FISH CAPACITY IF LESS THAN 0400 TONS ... THE VESSEL CLASS MUST BE EQUAL TO 1.
ERROR004	IF THE PRESENT OCCURRENCE OF OBSERVER (1.30) IS NOT EQUAL TO THE PREVIOUS OCCURRENCE OF OBSERVER ... THE PRESENT OCCURRENCE OF TOTAL OBSERVERS MUST BE EQUAL TO THE PREVIOUS OCCURRENCE OF TOTAL OBSERVERS + 1.
ERROR005	IF THE PRESENT OCCURRENCE OF TOTAL OBSERVERS (1.40) IS NOT EQL TO THE PREVIOUS OCCURRENCE OF TOTAL OBSERVERS ... THE PRESENT OCCURRENCE OF OBSERVER (1.30) MUST BE NOT EQUAL TO THE PREVIOUS OCCURRENCE OF OBSERVER.
ERROR006	IF THE NUMBER OF PORPOISE SETS SEEN (1.37) IS GREATER THAN ZERO ... THE NUMBER OF TRIPS (1.35) MUST BE GREATER THAN ZERO.
ERROR007	IF THE GEAR TYPE IS 03 ... THE PANEL STRIP DEPTH MUST BE EQUAL TO 02.
ERROR008	IF THE GEAR TYPE IS 03 ... THE PANEL MESH SIZE MUST BE EQUAL TO OR LESS THAN 125 INCHES.
ERROR009	IF THE GEAR TYPE IS 04 ... THE PANEL DEPTH MUST BE EQUAL TO 20 FATHOMS.
ERROR010	IF THE GEAR TYPE IS 04 ... THE PANEL STRP DEPTH MUST BE EQUAL TO 04.
ERROR011	IF THE GEAR TYPE IS 04 ... THE PANEL MESH SIZE MUST BE EQUAL TO OR LESS THAN 125 INCHES.
ERROR012	IF THE GEAR TYPE IS 05 ... THE PANEL STRIP DEPTH MUST BE EQUAL TO 01.
ERROR013	IF THE GEAR TYPE IS 05 ... THE PANEL MESH SIZE MUST BE EQUAL TO OR LESS THAN 125 INCHES.
ERROR014	IF THIS CRUISE SPECIFICATIONS FILE CONTAINS MORE THAN ONE LOGICAL RECORD SET ... TRIP COMPLETED (1.29) FOR ALL BUT THE LAST OF THESE LOGICAL RECORD SETS SHOULD BE EQUAL TO 2.
ERROR015	WITHIN THIS FILE, THE LAST OCCURRENCE OF TRIP COMPLETED (1.29) SHOULD BE EQUAL TO 1 ... IF NOT , THIS MAY CONSTITUTE AN ERROR.
ERROR016	IF THIS CRUISE SPECIFICATIONS FILE CONTAINS MORE THAN ONE LOGICAL RECORD SET ... THE ELEMENTS : VESSEL CODE (1.06) , YR BOAT BUILT (1.10) , FISH CAPACITY (1.12) , AND VESSEL CLASS (1.16) FOR EACH OF THE LOGICAL RECORD SETS SHOULD BE IDENTICAL.
ERROR017	THE PRESENT OCCURRENCE OF DATE RETURNED (1.23) SHOULD BE GTR THAN THE PRESENT OCCURRENCE OF DATE DEPARTED (1.17) ... IF IT IS NOT , THIS CONSTITUTES AN ERROR.
ERROR018	THE PRESENT OCCURRENCE OF DATE DEPARTED (1.17) SHOULD BE GTR THAN THE PREVIOUS OCCURRENCE OF DATE RETURNED

(1.23) ... IF IT IS NOT , THIS CONSTITUTES AN ERROR.

ERROR019 IF THE INITIAL OCCURRENCE OF TOTAL OBSERVERS (1.40) IS NOT EQUAL TO 1 ... THIS CONSTITUTES AN ERROR.

ERROR020 IF THE PRESENT OCCURRENCE OF OBSERVER TYPE (1.33) IS EQUAL TO 04 ... THE FOLLOWING ELEMENTS SHOULD BE BLANK : VESSEL CODE (1.06), FISH CAPACITY (1.12), VESSEL CLASS (1.16), TRIP COMPLETED (1.29), GEAR TYPE (1.43), NUM SPEEDBOATS (1.45), BOWTHRUSTER (1.47), ANTITORQ CABLE (1.48), YEAR NET BUILT (1.49), NET LENGTH (1.51), NET DEPTH (1.54), NET STRIP DEPTH (1.57), NET MESH SIZE (1.59), PORP PANEL (1.62), YR PANEL INSTALD (1.63), PANEL LENGTH (1.65), PANEL DEPTH (1.68), PANEL STRIP DEPTH (1.70), PANEL MESH SIZE (1.72), OPER CERT HOLDER (2.32), AND OPER CERT NUMBER (2.61).

```
*****************************************************************
Beginning of the Cruise Specification Main Edit program for 1981 data.


C***   PROGRAM                 CSEDIT.81
C
C***   PURPOSE                 THE PURPOSE OF THIS PROGRAM IS TO PERFORM
C                              FIELD AND INTER-VARIABLE LOGIC CHECKS ON
C                              THE DATA CONTAINED IN THE CRUISE SPECIFI-
C                              CATIONS FILE.  ALSO A CARD SEQUENCE/DECK
C                              INTEGRITY CHECK IS CARRIED OUT BY SUBROUTINE
C                              CSOOCS.
C
C***   LANGUAGE                FORTRAN IV (CSC INFONET)
C
C***   PROGRAMMER              R. GRISMORE; C. LONG
C
C***   DATE                    SEPTEMBER 1980
C
C*********************************************************************
C*********************************************************************
C                              EXECUTION INSTRUCTIONS
C
C                              !CSEDIT.81
C                              CSXXX              :FILE NAME
C     NO EDIT                :EDIT OPTION
C
C                                 OR
C
C                              !CSEDIT.81
C                              CSXXX              :FILE NAME
C                              GO EDIT            :EDIT OPTION
C                              TRUE  OR  FALSE    :LISTINGS OPTION
C
C
C    OPTIONS NOTE:  NO EDIT  - USED WHEN ONLY A DATA FILE
C                              SEQUENCING CHECK IS DESIRED.
C
C                   GO EDIT  - USED WHEN FIELD OR LOGICAL
C                              ERROR CHECKS ON THE FILE ARE
C                              DESIRED.
C
C                   TRUE     - USED WHEN A LISTING OF ALL
C                              LOGICAL RECORD GROUPINGS
C                              IS DESIRED.
C
C                   FALSE    - USED WHEN A LISTING OF ONLY
C                              LOGICAL RECORD GROUPINGS IN
C                              ERROR ARE DESIRED.
C
C*********************************************************************
C
C***   INPUT FILES:
C         CS### - CRUISE SPECIFICATIONS INPUT FILE.
C            CS - PREFIX NAMING CONVENTION FOR INPUT FILES.
```

```
C          ### - CRUISE NUMBER.
C          (READ FROM LOGICAL DEVICE = 10.)
C        PROMPT DEVICE - SOURCE OF ALL USER COMMANDS (LOGICAL DEVICE = 5).
C
C***   OUTPUT FILES:
C        PRINTER - DESTINATION OF ALL PROGRAM OUTPUT (LOGICAL DEVICE = 6).
C
C***   SUBROUTINES:
C        ADP SUBROUTINES:
C           BLANKT - VERIFIES THAT A FIELD IS NON-BLANK.
C           CLLGCT - CALLED TO CLOSE THE DATA BASE HOLDING THE ERROR COUNTS.
C           CS00CS - PRE-EDIT PROGRAM FOR THE CRUISE SPECIFICATIONS FILE.
C           EJECTB - CAUSES LINE COUNT TO BE SET AT 54 TO FORCE STARTING
C                       A NEW PAGE.
C           EJECTH - STORES HEADER INFORMATION TO BE PRINTED PREPARATORY
C                       TO STARTING A NEW PAGE.
C           EJECTR - CAUSES A LINE COUNT TO BE MAINTAINED IN ORDER TO
C                       PRINT HEADING INFORMATION (ARGUMENT - NUMBER
C                       OF LINES TO BE PRINTED).
C           ERRCHK - PRINTS AN ERROR MESSAGE AND THE PHYSICAL RECORD IN
C                       ERROR WHEN LOGICAL VARIABLE ANYERR IS FOUND TO
C                       BE TRUE.
C           ERRLOG - OBTAINS AN ERROR MESSAGE FROM THE ERROR DATA BASE
C                       AND POSTS THE TYPE OF ERROR AND THE NUMBER OF
C                       OCCURRENCES IN ANOTHER DATA BASE.
C           OPLGCT - CALLED TO OPEN THE DATA BASE HOLDING THE ERROR COUNTS.
C           P(ARG) - FUNCTION SUBPROGRAM WHICH CHECKS THE FIELD ARG FOR
C                       NUMERIC DATA, AND ZERO FILLS IT TO THE LEFT IF IT
C                       IS VALID NUMERIC.
C           RANGIT - VERIFIES THAT THE VALUE OF A FIELD FALLS WITHIN
C                       SPECIFIED UPPER AND LOWER LIMITS.
C           VERFIT - VERIFIES THAT ALL CHARACTERS OCCURRING IN ONE STRING
C                       OCCUR IN ANOTHER.
C           WRAPUP - PRINTS OUT THE DATA GROUP.
C        INFONET SUBROUTINES:
C           FDEFNX - DEFINES AND ALLOCATES LOGICAL FILES.
C           FSTRX  - FUNCTION SUBPROGRAM WHICH CONVERTS ITS ARGUMENT FROM
C                       AN ARITHMETIC VALUE TO A CHARACTER STRING
C                       REPRESENTATION.
C           FBRKCX - FUNCTION WHICH DETERMINES FIRST OCCURANCE IN
C                    ARG 1 WHICH ALSO OCCURES IN ARG 2.
C
C***   PRIMARY VARIABLES:
C        ALPHAS - DATA STRING CONTAINING ALPHABET & CERTAIN SPECIAL CHARACTERS
C        ALLBLK - USER-ENTERED LOGICAL VARIABLE SET TO TRUE IF ALL DATA BLOCKS
C                    ARE TO BE PRINTED.
C        ALLSTR - CONTAINS ENTIRE DATA BLOCK.
C        ANYERR - SET TO TRUE ON THE OCCURRENCE OF ANY ERROR.
C        CARSEQ - CONTAINS THE CURRENT CARD SEQUENCE NUMBER.
C        CUROBS - CURRENT OBSERVER SEQUENCE NUMBER.
C        DATSET - CONTAINS THE PARTIAL KEY FOR THE ERROR DATA BASE.
C        DIGITS - DATA STRING CONTAINING THE NUMERIC DIGITS.
C        DSKSEQ - CONTAINS THE CURRENT PHYSICAL RECORD NUMBER.
C        ELEVAL - CODE USED TO SPECIFY THE LENGTH OF ERROR MESSAGE TO PRINT.
C        GRPCOD - EQUALS 'P' OR 'N' AS SET IS MARINE MAMMAL OR NOT.
```

```
C          HEAD   - CONTAINS PAGE-HEADING CHARACTER STRING.
C          INFILE - CONTAINS INPUT FILE NAME.
C          IRET   - RETURN CODE FROM INFONET SUBROUTINE FDEFNX.
C          LASSEQ - NUMBER OF PHYSICAL RECORDS IN THE DISK FILE.
C          LOGGRP - DATA GROUP IDENTIFIER.  BASE 1, INCREMENT 1, FOR EACH
C                     DATA SET.
C          NOGO   - SET TO TRUE WHEN ERRLOG IS CALLED.
C          OLDARR - ARRAY OF DIMENSION 3 CONTAINING THE LAST PREVIOUS DATA BLOCK
C          OUROBS - PREVIOUS OBSERVER SEQUENCE NUMBER.
C          STRING - CONTAINS THE CURRENT INPUT DATA RECORD.
C          TOTERR - SET TO TRUE IF PRE-EDIT FAILED; HALTS EXECUTION.
C          VALUE  - CONTAINS INTEGER VALUE RESULTING FROM A CALL TO RANGIT.
C
C***  COMMON AREA DEFINITION.
C
      COMMON/ERROR/ ANYERR, CARSEQ, DATSET, DSKSEQ,
     1              ELEVAL, GRPCOD, NOGO,   STRING,
     2              TEMP,   VALUE,  TOTERR, ALLBLK,
     3              INFILE
C
C***  PROGRAM VARIABLE SPECIFICATION STATEMENTS.
C
      CHARACTER     GRPCOD*1, STRING*80, TEMP*80, DATSET*8,
     1              CARSEQ*2, INFILE*20, ALLSTR(3)*80,
     2              OLDARR(3)*80,
     3              ALPHAS*30/" ABCDEFGHIJKLMNOPQRSTUVWXYZ.'&"/,
     4              DIGITS*10/'0123456789O'/, HEAD*60,
     5              P*8, FSTRX*6, NUMST*6
C
      INTEGER       DSKSEQ, ELEVAL, VALUE, IRET, CUROBS, OUROBS
      INTEGER FBRKCX, STARTC
C
      LOGICAL       ANYERR/.FALSE./, TOTERR, ALLBLK, NOGO
C
      LENGTH P(*)
C
      CHARACTER GOEDIT*7
C
      DATA ELEVAL/0/, GRPCOD/' '/
C
C***  SECTION 1 ***************************************************************
C***  BEGIN MAIN PROGRAM LOGIC. *********************************************
C
C
C***  CALL THE CARD-SEQUENCING PRE-EDIT PROGRAM.
C
      CALL CS00CS
      IF ( TOTERR ) GO TO 199
      ELEVAL = 1
C
C***  OPEN EC81DB DATA BASE.  (THIS IS AN ENTRY POINT WITHIN ERRLOG.)
C
      CALL OPLGCT('EC81DB  ')
C     CALL OPLGST('ES81DB  ')
      CALL EJECTR(1)
```

```
C
C    PROMPT USER TO GO/NOGO EDIT.
C
      WRITE(6,3)
    3 FORMAT(' ENTER ''GO EDIT'' OR ''NO EDIT''.')
      READ(5,4) GOEDIT
    4 FORMAT(A7)
      IF( GOEDIT .EQ. 'GO EDIT' ) GO TO 6
      IF( GOEDIT .EQ. 'NO EDIT' ) GO TO 99
      WRITE(6,5) GOEDIT
    5 FORMAT(' RESPONSE - ',A7,' NOT ''GO EDIT'' OR ''NO EDIT''.')
      GO TO 99
    6 CONTINUE
      REWIND 10
C
C***  ESTABLISH DATA BASE KEY.
C
      CALL FDEFNX(IRET,10,INFILE,'STD')
      DATSET(1:8)='CSEDIT81'
C
C***  PROMPT USER FOR VALUE OF ALLBLK.
C
      WRITE(6,2)
    2 FORMAT(' ENTER TRUE/FALSE PRINT ALL DATA GROUPS.' )
      READ(5,*) ALLBLK
      LASSEQ = DSKSEQ
      DSKSEQ = 0
C
C***  TOP OF READ LOOP.
C
   20 CONTINUE
      DO 30 I=1,3
      READ(10,1,END=99) ALLSTR(I)
    1 FORMAT(A80)
   30 CONTINUE
      LOGGRP = LOGGRP + 1
C
C    DETERMINE NUMBER OF SIGNIFICANT CHAR'S IN LOGGRP.
C
      NUMST = FSTRX(LOGGRP)
      STARTC = FBRKCX(NUMST,'123456789')
      IF( STARTC .EQ. 0 ) STARTC = 1
C
C***  HEAD UP OUTPUT PAGE.
C
      HEAD = '*** CSEDIT.81 --- INPUT FILE:'//INFILE(1:8)//
     1' LOGICAL GROUP '// NUMST(STARTC:)
      CALL EJECTH(HEAD)
      CALL EJECTB(54)
C
C***  SECTION 2 *************************************************************
C***  VERIFY THE FIELDS OF CARD #1. ******************************************
C
C
C***  BEGIN WITH BLANKNESS AND RANGE CHECKS.
```

```
C
      CARSEQ='01'
      DSKSEQ=DSKSEQ+1
      STRING=ALLSTR(1)
C
      CALL BLANKT(1,3)
      CALL RANGIT(1,3,670,785)
C
      CALL BLANKT(4,2)
      CALL RANGIT(4,2,1,1)
C
      CALL BLANKT(6,4)
      CALL RANGIT(6,4,0029,0460)
C
      CALL BLANKT(10,2)
      CALL RANGIT(10,2,44,80)
C
      CALL BLANKT(12,4)
      CALL RANGIT(12,4,0401,2000)
C
      CALL BLANKT(16,1)
      CALL RANGIT(16,1,2,3)
C
      CALL BLANKT(17,2)
       IF ( NOGO ) GO TO 35
       CALL VERFIT(17,2,DIGITS)
   35 CONTINUE
      CALL RANGIT(17,2,81,81)
C
      CALL BLANKT(19,2)
       IF ( NOGO ) GO TO 36
       CALL VERFIT(19,2,DIGITS)
   36 CONTINUE
      CALL RANGIT(19,2,1,12)
C
      CALL BLANKT(21,2)
       IF ( NOGO ) GO TO 37
       CALL VERFIT(21,2,DIGITS)
   37 CONTINUE
      CALL RANGIT(21,2,1,31)
C
      CALL BLANKT(23,2)
       IF ( NOGO ) GO TO 40
       CALL VERFIT(23,2,DIGITS)
   40 CONTINUE
      CALL RANGIT(23,2,81,81)
C
      CALL BLANKT(25,2)
       IF ( NOGO ) GO TO 41
       CALL VERFIT(25,2,DIGITS)
   41 CONTINUE
      CALL RANGIT(25,2,1,12)
C
      CALL BLANKT(27,2)
       IF ( NOGO ) GO TO 42
```

```
        CALL VERFIT(27,2,DIGITS)
   42 CONTINUE
      CALL RANGIT(27,2,1,31)
C
      CALL BLANKT(29,1)
      CALL RANGIT(29,1,1,2)
C
      CALL BLANKT(30,3)
        IF ( NOGO ) GO TO 45
        CALL VERFIT(30,3,DIGITS)
   45 CONTINUE
      CALL RANGIT(30,3,34,370)
C
      CALL BLANKT(33,2)
      CALL RANGIT(33,2,01,06)
C
      CALL BLANKT(35,2)
      CALL RANGIT(35,2,00,06)
C
      CALL BLANKT(37,3)
      CALL RANGIT(37,3,000,300)
C
      CALL BLANKT(40,1)
      CALL RANGIT(40,1,1,3)
        CUROBS = VALUE
C
      CALL BLANKT(41,2)
      CALL RANGIT(41,2,01,03)
C
      CALL BLANKT(43,2)
      CALL RANGIT(43,2,01,08)
C
      CALL BLANKT(45,1)
      CALL RANGIT(45,1,4,6)
C
      CALL BLANKT(46,1)
      CALL RANGIT(46,1,1,2)
C
      CALL BLANKT(47,1)
      CALL RANGIT(47,1,1,2)
C
      CALL BLANKT(48,1)
      CALL RANGIT(48,1,1,2)
C
      CALL BLANKT(49,2)
      CALL RANGIT(49,2,59,80)
C
      CALL BLANKT(51,3)
      CALL RANGIT(51,3,480,800)
C
      CALL BLANKT(54,3)
      CALL RANGIT(54,3,048,090)
C
      CALL BLANKT(57,2)
      CALL RANGIT(57,2,08,15)
```

```
C
      CALL BLANKT(59,3)
      CALL RANGIT(59,3,375,425)
C
      CALL BLANKT(62,1)
      CALL RANGIT(62,1,1,2)
C
      CALL BLANKT(63,2)
      CALL RANGIT(63,2,76,81)
C
      CALL BLANKT(65,3)
      CALL RANGIT(65,3,162,200)
C
      CALL BLANKT(68,2)
      CALL RANGIT(68,2,06,20)
C
      CALL BLANKT(70,2)
      CALL RANGIT(70,2,01,04)
C
      CALL BLANKT(72,3)
      CALL RANGIT(72,3,113,125)
C
C***  BEGIN INTER-VARIABLE LOGIC CHECKS ON FIELDS OF CARD #1.
C
C
C***  ERROR001
C
      IF (P(ALLSTR(1)(10:11)) .LE. '60')   GO TO 100
      IF (P(ALLSTR(1)(12:15)) .LT. '0400') GO TO 100
      IF ( (ALLSTR(1)(16:16)) .EQ. '3')    GO TO 100
      CALL ERRLOG('ERROR001')
  100 CONTINUE
C
C***  ERROR002
C
      IF ( (ALLSTR(1)(10:11)) .GE. '61')   GO TO 110
      IF (P(ALLSTR(1)(12:15)) .LT. '0400') GO TO 110
      IF ( (ALLSTR(1)(16:16)) .EQ.  '2')   GO TO 110
      CALL ERRLOG('ERROR002')
  110 CONTINUE
C
C***  ERROR003
C
      IF (P(ALLSTR(1)(12:15)) .GE. '0400') GO TO 120
      IF ( (ALLSTR(1)(16:16)) .EQ. '1')    GO TO 120
      CALL ERRLOG('ERROR003')
  120 CONTINUE
C
C***  NEXT TWO CHECKS INVOLVE PRESENT AND PREVIOUS DATA SETS.  SKIP THEM IF
C***  NO PREVIOUS DATA SET.
C
      IF (P(OLDARR(1)(4:5)) .NE. '01') GO TO 145
C
C***  ERROR004
C
```

```
      IF (P(ALLSTR(1)(30:32)) .EQ. P(OLDARR(1)(30:32))) GO TO 130
       OUROBS = OUROBS + 1
      IF (OUROBS .EQ. CUROBS)                           GO TO 130
      CALL ERRLOG('ERROR004')
  130 CONTINUE
C
C***  ERROR005
C
      IF (P(ALLSTR(1)(40:40)) .EQ. P(OLDARR(1)(40:40))) GO TO 140
      IF (P(ALLSTR(1)(30:32)) .NE. P(OLDARR(1)(30:32))) GO TO 140
      CALL ERRLOG('ERROR005')
  140 CONTINUE
C
C***  END OF PRESENT VS. PREVIOUS DATA SET CHECKS.
C
  145 CONTINUE
C
C***  ERROR006
C
      IF (P(ALLSTR(1)(37:39)) .LE. '000') GO TO 150
      IF (P(ALLSTR(1)(35:36)) .GT. '00' ) GO TO 150
      CALL ERRLOG('ERROR006')
  150 CONTINUE
C
C***  ERROR007
C
      IF (P(ALLSTR(1)(43:44)) .NE. '03') GO TO 170
      IF (P(ALLSTR(1)(70:71)) .EQ. '02') GO TO 160
      CALL ERRLOG('ERROR007')
  160 CONTINUE
C
C***  ERROR008
C
      IF ( (ALLSTR(1)(72:74)) .EQ. '125') GO TO 170
      CALL ERRLOG('ERROR008')
  170 CONTINUE
C
C***  ERROR009
C
      IF (P(ALLSTR(1)(43:44)) .NE. '04' )  GO TO 200
      IF ( (ALLSTR(1)(68:69)) .EQ. '20' )  GO TO 180
      CALL ERRLOG('ERROR009')
  180 CONTINUE
C
C***  ERROR010
C
      IF (P(ALLSTR(1)(70:71)) .EQ. '04') GO TO 190
      CALL ERRLOG('ERROR010')
  190 CONTINUE
C
C***  ERROR011
C
      IF (P(ALLSTR(1)(72:74)) .LE. '125') GO TO 200
      CALL ERRLOG('ERROR011')
  200 CONTINUE
```

```
C
C***   ERROR012
C
       IF (P(ALLSTR(1)(43:44)) .NE. '05') GO TO 220
       IF (P(ALLSTR(1)(70:71)) .EQ. '01') GO TO 210
       CALL ERRLOG('ERROR012')
  210  CONTINUE
C
C***   ERROR013
C
       IF (P(ALLSTR(1)(72:74)) .LE. '125')  GO TO 220
       CALL ERRLOG('ERROR013')
  220  CONTINUE
C
C***   ERROR014
C
       IF ((LASSEQ - DSKSEQ) .LT. 3 )  GO TO 230
       IF (ALLSTR(1)(29:29) .EQ. '2' ) GO TO 230
       CALL ERRLOG('ERROR014')
  230  CONTINUE
C
C***   ERROR015
C
       IF ((LASSEQ - DSKSEQ) .GT. 2 )  GO TO 240
       IF (ALLSTR(1)(29:29) .EQ. '1' ) GO TO 240
       CALL ERRLOG('ERROR015')
  240  CONTINUE
C
C***   ERROR016
C
       IF (P(OLDARR(1)(4:5)) .NE. '01')           GO TO 250
       IF (ALLSTR(1)(6:16) .EQ. OLDARR(1)(6:16)) GO TO 250
       CALL ERRLOG('ERROR016')
  250  CONTINUE
C
C***   ERROR017
C
       IF (P(ALLSTR(1)(23:28)) .GT. P(ALLSTR(1)(17:22))) GO TO 260
       CALL ERRLOG('ERROR017')
  260  CONTINUE
C
C***   ERROR018
C
       IF (P(OLDARR(1)(4:5)) .NE. '01')              GO TO 270
       IF (ALLSTR(1)(17:22) .GT. OLDARR(1)(23:28)) GO TO 270
       CALL ERRLOG('ERROR018')
  270  CONTINUE
C
C***   ERROR019
C
       IF (DSKSEQ .GT. 1)               GO TO 280
       IF (ALLSTR(1)(40:40) .EQ. '1') GO TO 280
       CALL ERRLOG('ERROR019')
  280  CONTINUE
C
```

```
C***   END OF INTER-VARIABLE LOGIC CHECKS ON CARD #1.
C
C
C***   WRITE ANY RECORDS WHICH ARE IN ERROR.
C
       CALL ERRCHK
C
C***   SECTION 3 ************************************************************
C***   VERIFY THE FIELDS OF CARD #2. ********************************************
C
       CARSEQ='02'
       STRING=ALLSTR(2)
       DSKSEQ=DSKSEQ+1
C
       CALL BLANKT(1,3)
       CALL RANGIT(1,3,670,785)
C
       CALL BLANKT(4,2)
       CALL RANGIT(4,2,2,2)
C
       CALL BLANKT(6,5)
        IF ( NOGO ) GO TO 300
       CALL VERFIT(6,26,ALPHAS)
  300  CONTINUE
C
       CALL BLANKT(32,3)
        IF ( NOGO ) GO TO 305
       CALL VERFIT(32,29,ALPHAS)
  305  CONTINUE
C
       CALL BLANKT(61,3)
        IF ( NOGO ) GO TO 310
       CALL VERFIT(61,3,ALPHAS)
  310  CONTINUE
C
       CALL BLANKT(64,3)
        IF ( NOGO ) GO TO 315
       CALL VERFIT(64,3,DIGITS)
  315  CONTINUE
       CALL RANGIT(64,3,812,812)
C
       CALL BLANKT(67,3)
        IF ( NOGO ) GO TO 320
       CALL VERFIT(67,3,DIGITS)
  320  CONTINUE
       CALL RANGIT(67,3,501,640)
C
C***   WRITE ANY RECORDS WHICH ARE IN ERROR.
C
       CALL ERRCHK
C
C***   SECTION 4 ************************************************************
C***   VERIFY THE FIELDS OF CARD #3. ********************************************
C
       CARSEQ='03'
```

```
      STRING=ALLSTR(3)
      DSKSEQ=DSKSEQ+1
C
      CALL BLANKT(1,3)
       IF ( NOGO ) GO TO 400
      CALL RANGIT(1,3,670,785)
  400 CONTINUE
C
      CALL BLANKT(4,2)
      CALL RANGIT(4,2,3,3)
C
      CALL BLANKT(6,3)
      CALL VERFIT(6,29,ALPHAS)
C
      CALL BLANKT(35,3)
       IF ( NOGO ) GO TO 405
      CALL VERFIT(35,3,ALPHAS)
  405 CONTINUE
C
      CALL BLANKT(38,3)
       IF ( NOGO ) GO TO 410
      CALL VERFIT(38,3,DIGITS)
  410 CONTINUE
      CALL RANGIT(38,3,812,812)
C
      CALL BLANKT(41,3)
       IF ( NOGO ) GO TO 415
      CALL VERFIT(41,3,DIGITS)
  415 CONTINUE
      CALL RANGIT(41,3,1,140)
C
C***  BEGIN INTER-VARIABLE LOGIC CHECKS ON FIELDS OF CARDS #1 & 2.
C
C
C     ERROR020
C
      IF ( P( ALLSTR(1)(33:34)) .NE. '04' )  GO TO 420
      IF    (( ALLSTR(1)( 6:9 )  .EQ. '  ' ) .AND.
     *        ( ALLSTR(1)(12:15)  .EQ. '  ' ) .AND.
     *        ( ALLSTR(1)(16:16)  .EQ. '  ' ) .AND.
     *        ( ALLSTR(1)(29:29)  .EQ. '  ' ) .AND.
     *        ( ALLSTR(1)(43:44)  .EQ. '  ' ) .AND.
     *        ( ALLSTR(1)(45:45)  .EQ. '  ' ) .AND.
     *        ( ALLSTR(1)(47:47)  .EQ. '  ' ) .AND.
     *        ( ALLSTR(1)(51:52)  .EQ. '  ' ) .AND.
     *        ( ALLSTR(1)(54:56)  .EQ. '  ' ) .AND.
     *        ( ALLSTR(1)(57:58)  .EQ. '  ' ) .AND.
     *        ( ALLSTR(1)(59:61)  .EQ. '  ' ) .AND.
     *        ( ALLSTR(1)(62:62)  .EQ. '  ' ) .AND.
     *        ( ALLSTR(1)(63:64)  .EQ. '  ' ) .AND.
     *        ( ALLSTR(1)(65:67)  .EQ. '  ' ) .AND.
     *        ( ALLSTR(1)(68:69)  .EQ. '  ' ) .AND.
     *        ( ALLSTR(1)(70:71)  .EQ. '  ' ) .AND.
     *        ( ALLSTR(1)(72:74)  .EQ. '  ' ) .AND.
     *        ( ALLSTR(2)(32:60)  .EQ. '  ' ) .AND.
     *        ( ALLSTR(2)(61:69)  .EQ. '   ' )) GO TO 420
      CALL ERRLOG('ERROR020')
  420 CONTINUE
C
C***  END OF INTER-VARIABLE LOGIC CHECKS ON CARDS #1 & 2.
C
C
C***  WRITE ANY RECORDS WHICH ARE IN ERROR.
C
      CALL ERRCHK
C
C***  SECTION 5 ************************************************************
C***  COMPLETE PROCESSING OF DATA SET JUST EDITED, AND RETURN TO START ON ***
C***  NEXT ONE. ************************************************************
C
C
C***  MOVE PRESENT DATA SET INTO PREVIOUS DATA STORAGE AREA.
C
      DO 500 I=1,3
      OLDARR(I) = ALLSTR(I)
  500 CONTINUE
      OUROBS = CUROBS
C
C***  FINISH PRINTING PAGE, AND GO BACK TO READ NEXT DATA BLOCK.
C
      CALL WRAPUP(ALLSTR,3)
      GO TO 20
C
C***  SECTION 6 ************************************************************
C***  NORMAL AND ABNORMAL TERMINATION PROCEDURES. ***************************
C
   99 WRITE(6,1099)
 1099 FORMAT(1X)
C
C***  CLOSE EC81DB DATA BASE.  (THIS IS AN ENTRY POINT WITHIN ERRLOG.)
C
      CALL CLLGCT('EC81DB  ')
C     CALL CLLGST('ES81DB  ')
      STOP 'END OF CSEDIT.81'
C
C***  TERMINATION FOR CASE OF PRE-EDIT FAILURE.
C
  199 WRITE(6,1099)
      STOP 'PRE-EDIT FAILED - FIX DATA AND RE-RUN.'
      END
```

End of the Cruise Specification Main Edit program for 1981 data.
****

```
*********************************************************************
Beginning of the Cruise Specification Sequencing program for 1981 data.


      SUBROUTINE CS00CS
C ***
C
C*** PROGRAMMER:      C. LONG.
C*** PROGRAM:         CSEQCS.81 HERE CALLED CS00CS.81
C
C*** DATE:            OCTOBER, 1978.
C
C *** PURPOSE                                                      ***
C *       THIS PROGRAM EDITS THE FOLLOWING DATA WHICH IS            *
C *    CONTAINED IN THE CRUISE FILE.                                *
C *              1.) CRUISE NUMBER                                  *
C *              2.) DATE SAILED/PREVIOUS DATE RETURNED             *
C *              3.) COMPLETED-TRIP FIELD ON PREVIOUS SET = '2'     *
C *              4.) VESSEL CHARACTERISTICS MUST MATCH              *
C ***                                                              ***
C*** INPUT FILES:
C        CS### - INPUT CRUISE FILE.
C           CS - PREFIX NAMING CONVENTION FOR INPUT FILES.
C          ### - CRUISE NUMBER.
C        LOGICAL DEVICE = 10.
C
C        PROMPT DEVICE - SOURCE OF USER COMMANDS.
C        LOGICAL DEVICE = 5.
C
C*** OUTPUT FILES:
C        PRINTER - DESTINATION OF ALL PROGRAM OUTPUT.@
C        LOGICAL DEVICE = 6.
C
C*** SUBROUTINES:
C        ADP SUBROUTINES:
C           VERFIT - VERIFIES THAT ALL CHARACTERS OCCURING IN ONE
C                       STRING OCCUR IN ANOTHER.
C
C        INFONET SUBROUTINES:
C           FDEFNX - ALLOCATES AND DEFINES LOGICAL FILES.
C
C*** PRIMARY VARIABLES:
C        STRING - CONTAINS CURRENT DATA RECORD.
C        TEMP   - HAS PART OF DATA RETURNED BY VERFIT.
C        DSKSEQ - CONTAINS CURRENT LOGICAL RECORD NUMBER.
C        CARSEQ - CONTAINS CURRENT CARD SEQUENCE NUMBER.
C        ANYERR - SET TO TRUE IF ANY ERROR IS ENCOUNTERED.
C        TOTERR - SET TO TRUE IF ANYERR IS EVER TRUE AND THEN PASSED.
C        INFILE - CONTAINS THE INPUT FILE NAME.
C        PREVON - STRING CONTAINING THE PREVIOUS CARD SEQ 01
C                 AN ASTERICK IS PLACED IN COLUMN 1.  WHEN REMOVED
C                 THIS SIGNIFIES A PREVIOUS CARD 1 IS PRESENT.
C
C*** DESCRIPTION
C        THIS PROGRAM'S PRIMARY CONCERN IS THAT THE CARD SEQUENCE OF THE
```

```
C        CRUISE SPECIFICATION DECK IS CORRECT.  THE ERROR CHECKS ARE
C        PERFORMED PRIMARILY WITH DIRECT RECORD COMPARISONS USING THE
C        STRINGING FUNCTIONS AVAILABLE IN FORTRAN.  THE FOLLOWING IS
C        PERFORMED.
C
C         1) SUBROUTINE CSEQP1 IS CALLED TO:
C           A) PROMPT FOR THE INPUT FILE NAME AND EQUATE TO UNITS 10 AND 12
C           B) RENUMBER THE INPUT FILE TO HAVE STANDARD KEYS BEGINNING
C              WITH 1 AND INCREMENTING BY 1.
C           C) CHECK THE CRUISE NUMBER, COLUMNS 1-3, TO MATCH THE CRUISE
C              NUMBER OF THE FILE NAME, COLUMN 3-5.
C           D) CHECK FOR RECORDS WHOSE LENGTH IS NOT EQUAL TO 80 COLUMNS.
C         2) THE CARD SEQUENCE NUMBER, COLUMN 5, IS CHECKED TO BEGIN WITH
C            1 AND INCREMENT BY 1 FOR EACH 3 CARD DATA SET.
C         3) THE FOLLOWING FIELDS ARE VERIFIED TO INCLUDE ONLY THE DIGITS
C            0123456789
C          A) THE CRUISE NUMBER. COLUMNS 1-3
C          B) THE CARD SEQUENCE NUMBER. COLUMNS 4-5
C
C
C
C*** COMMON AREA DEFINITION.
C
      COMMON/ERROR/ ANYERR, CARSEQ, DATSET, DSKSEQ, ELEVAL,GRPCOD,
     X      NOGO, STRING, TEMP, VALUE, TOTERR, ALLBLK, INFILE
C
C *** PROGRAM VARIABLE DEFINITIONS.
C
      CHARACTER      GRPCOD*1, STRING*80, DATSET*8, CARSEQ*2,
     X               TEMP*80,  INFILE*20, PREVCD*80,   PREVON*80/'*'/,
     X               HEAD*60/'*** CS00CS.81 --- INPUT FILE:      '/,
     X               DIGITS*10/'0123456789'/
C
      LOGICAL        ANYERR/.FALSE./, NOGO, TOTERR, ALLBLK
C
      INTEGER        DSKSEQ, ELEVAL, VALUE,
     X               IRET
C
 5    FORMAT(A80)
    9 FORMAT(/20X,'RECORD NUMBER= ',I4,'. ***',
     1' INVALID CARD CODE ENCOUNTERED.')
C
C*** ACTUAL BEGINNING OF PROGRAM LOGIC.
C
 119  CALL CSEQP1(HEAD)
      CALL FDEFNX(IRET,10,INFILE,'STD')
C
C*** GET FIRST INPUT DATA RECORD.
C
 10   READ(10,5)STRING
      CARSEQ=STRING(4:5)
      DSKSEQ=1
      IF(CARSEQ.EQ.'01') GO TO 40
         CALL EJECTR(2)
         WRITE(6,9) DSKSEQ
```

```
C
C*** PGROUP IS CALLED TO PRINT OUT THE RECORD IN ERROR AND THE
C             RECORDS BRACKETING THE RECORD IN ERROR.
C
   20 IF(.NOT.ANYERR) GO TO 12
      CALL PGROUP(10,DSKSEQ,3,3,STRING)
      CALL EJECTB(54)
      ANYERR=.FALSE.
      TOTERR=.TRUE.
 12   PREVCD=STRING
C
C*** GET NEXT INPUT DATA RECORD.
C
      READ(10,5,END=99)STRING
      CARSEQ=STRING(4:5)
      DSKSEQ=DSKSEQ+1
      CALL VERFIT(1,3,DIGITS)
      CALL VERFIT(4,2,DIGITS)
C
C*** IS THE CARD TYPE VALID?
C
      IF(PREVCD(4:5) .EQ. STRING(4:5)) GO TO 34
      IF(PREVCD(4:5) .EQ. '01') GO TO 31
      IF(PREVCD(4:5) .EQ. '02') GO TO 32
      IF(PREVCD(4:5) .EQ. '03') GO TO 33
      PREVCD(4:5)=STRING(4:5)
      GO TO 40
 31   IF (STRING(4:5) .EQ. '02') GO TO 12
      GO TO 34
 32   IF (STRING(4:5) .EQ. '03') GO TO 12
      GO TO 34
 33   IF (STRING(4:5) .EQ. '01') GO TO 40
   34 CONTINUE
      CALL EJECTR(2)
      WRITE(6,9) DSKSEQ
      ANYERR=.TRUE.
      GO TO 20
   40 CONTINUE
C
C SAVE THE CARD SEQUENCE 01 RECORD FOR LATER COMPARISONS
C
   75 PREVON=STRING
      GO TO 20
C
C THIS STATEMENT IS REACHED AT THE END OF FILE ON UNIT 10
C
   99 CONTINUE
         BACKSPACE 10
C
C CHECK THAT THE LAST CARD READ WAS A CARD 3
C
      IF(STRING(5:5).EQ.'3') GO TO 100
         CALL EJECTR(3)
         WRITE(6,1100)
 1100 FORMAT(/' THE LAST CARD READ WAS NOT A CARD 3'/
     1 ' THE FOLLOWING GROUP IS PRINTED OUT FOR CLARITY')
         CALL PGROUP(10,DSKSEQ,3,3,STRING)
      CALL EJECTB(54)
  100 CONTINUE
  110 CONTINUE
      CALL EJECTR(1)
      PRINT *,'END OF CS00CS.81'
      RETURN
      END
```

End of the Cruise Specification Sequencing program for 1981 data.
****