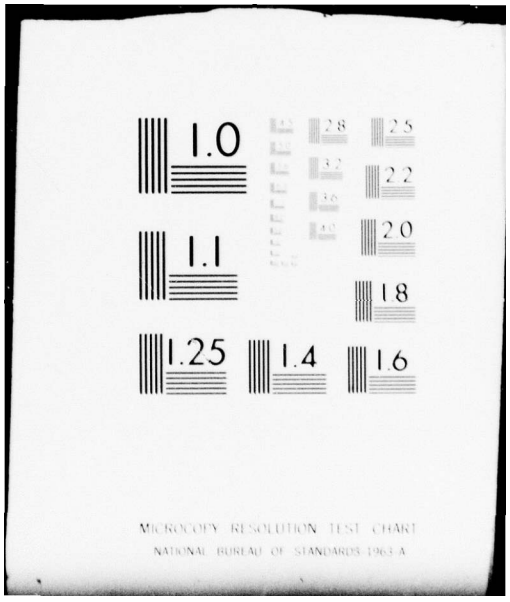
AD-A071 005 RAND CORP SANTA MONICA CALIF F/G 9/2
AN INTRODUCTION TO THE ISIS INTERACTIVE INFORMATION SYSTEM.(U)
APR 79 H J SHUKIAR, C H BUSH, R C GAMMILL F49620-77-C-0023
UNCLASSIFIED RAND/R-2435-AF NL

1 OF 2
AD A071005




1.0
1.1
1.25
1.4
1.6
2.8
2.5
3.2
2.2
3.6
2.0
4.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

RAND/R-2435-AF AN INTRODUCTION TO THE ISIS INTERACTIV

AD A071005

LEVEL II

B.S. 12

R-2435-AF
April 1979

An Introduction to the ISIS Interactive Information System

Herbert J. Shukiar, Charles H. Bush, Robert C. Gammill

DDC FILE COPY

A Project AIR FORCE report
prepared for the
United States Air Force

DDC
RECEIVED
JUL 11 1979
D

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

Rand
SANTA MONICA, CA. 90406

79 07 17 014

The research reported here was sponsored by the Directorate of Operational Requirements, Deputy Chief of Staff/Research, Development, and Acquisition, Hq USAF, under Contract F49620-77-C-0023. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

Shukiar, H J

An introduction to the ISIS interactive information system.

([Report] - Rand Corporation; R-2435-AF)
" A Project Air Force report prepared for the United States Air Force."
1. ISIS (Computer system). 2. Data base management. 3. Interactive computer systems. I. Bush, Charles H., joint author. II. Gamill, Robert C., joint author. III. Title. IV. Series: Rand Corporation. Rand report; R-2435-AF.
QA76.6.S55 001.6'4 79-12502
ISBN 0-8330-0121-3

The Rand Publications Series: The Report is the principal publication documenting and transmitting Rand's major research findings and final research results. The Rand Note reports other outputs of sponsored research for general distribution. Publications of The Rand Corporation do not necessarily reflect the opinions or policies of the sponsors of Rand research.

Published by The Rand Corporation

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1 REPORT NUMBER: R-2435-AF	2 GOVT ACCESSION NO.
3 RECIPIENT'S CATALOG NUMBER	
4 TITLE (and Subtitle): An Introduction to the ISIS Interactive Information System.	5 TYPE OF REPORT & PERIOD COVERED: Interim
	6 PERFORMING ORG. REPORT NUMBER
7 AUTHOR(s): H. J. Shukiar, C. H. Roach, R. C. Gammill	8 CONTRACT OR GRANT NUMBER(s): F49620-77-C-0023
9 PERFORMING ORGANIZATION NAME AND ADDRESS: The Rand Corporation, 1700 Main Street, Santa Monica, CA 90406	10 PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11 CONTROLLING OFFICE NAME AND ADDRESS: Project AIR FORCE Office (AF/RDQA), Directorate of Operational Requirements, Hq USAF, Washington, D.C. 20330	12 REPORT DATE: April 1979
	13 NUMBER OF PAGES: 93
14 MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15 SECURITY CLASS (of this report): UNCLASSIFIED
	15a DECLASSIFICATION/DOWNGRADING SCHEDULE

16 DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; Distribution Unlimited

17 DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

No Restrictions

18 SUPPLEMENTARY NOTES

19 KEY WORDS (Continue on reverse side if necessary and identify by block number)

ISIS
Information Systems
Tactical Command and Control
Tactical Air Support

20 ABSTRACT (Continue on reverse side if necessary and identify by block number)

SEE REVERSE SIDE

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

296 690

R-2345-AF An Introduction to the ISIS in Storage Information System. H. J. Shukiar, C. H. Bush, R. C. Gammill.

Describes in a tutorial manner ISIS, an interactive data base management system developed on the PDP11 computer under the UNIX time-sharing system. ISIS, designed for computer-naive users, uses an English-like command language to give the user the ability to view, modify, and otherwise reorganize his data base. ISIS operates on data bases of modest size, exploiting the unique characteristics associated with such data bases. The tutorial focus is tactical command and control, but ISIS is more generally applicable. The report introduces the ISIS command language in stages, and the user with access to Rand's PDP11/70 can try ISIS out on the command and control data base utilized in the report. The report also discusses the system development philosophy adopted during ISIS implementation. (Author)

Accession For
NTIS GRA&I
DDC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or special
A

R-2435-AF
April 1979

An Introduction to the ISIS Interactive Information System

Herbert J. Shukiar, Charles H. Bush, Robert C. Gammill

A Project AIR FORCE report
prepared for the
United States Air Force

DDC
RECEIVED
JUL 11 1979
D

Rand
SANTA MONICA, CA. 90406

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

PREFACE

Rand is engaged in research on issues of tactical command and control under Project AIR FORCE. Part of this research is examining how state-of-the-art computer technology can be applied toward the more effective management of tactical assets in the command and control environment. The interactive "In Storage Information System" (ISIS), which is reported on here, was developed in part to support this research.

This report provides a tutorial introduction to ISIS. Using the tactical command and control environment as a focus, it presents examples of ISIS statements. The data base utilized in this manual was taken from the Air Tasking Order (or "frag") of a military exercise, and the tutorial is geared toward activities surrounding the Tactical Air Control Center or TACC as it executes and monitors the execution of the air tasking order. Readers with access to Rand's PDP11/70 computer system will be able to try ISIS out while reading this manual.

Although the emphasis here is on tactical command and control, ISIS has been designed to be of more general applicability. We believe that this generality makes ISIS of interest to a much broader audience than the tactical command and control community.

The report should be of interest to Air Force agencies responsible for providing computing support, agencies with an interest in aircraft operations and maintenance scheduling, and agencies wanting rapid interactive access to data bases of modest size. It should also be of general interest to organizations responsible for the design and implementation of interactive information systems.

This work was performed in part under the Project AIR FORCE research project "Command and Control: Issues During a Crisis in Europe."

SUMMARY

This report is a tutorial on ISIS, an interactive system developed on Rand's PDP11/70 computer under the UNIX operating system. ISIS, which stands for In Storage Information System, allows the user to interactively view, modify, and otherwise reorganize data bases that are of modest size, with hundreds rather than hundreds of thousands of data objects. The objective of this interactive observation is to give the user a better understanding of the data base contents.

The ISIS command language is English-like, and the user can ask ISIS to display the records in a data base that satisfy certain user-determined characteristics. For example, the data base may contain employee records, and the user may wish to determine which employees were hired prior to a certain point in time, or which employees received large increases in salary during a specified interval. Another data base may represent corporate accounting ledgers, and the user may wish to determine which ledgers have had entries of greater than a specific amount during a given period, or which ledgers have seen no activity during the period.

This report employs a tactical command and control data base, one that represents the air tasking order for a given day of a military exercise. Both fighter missions and refueling missions are included. The environment is the Tactical Air Control Center, which has responsibility for the implementation and monitoring of the air tasking order. The examples presented are taken from this environment, and the reader with access to Rand's PDP11/70 will be able to try these examples out as he reads the report.

While the report employs a tactical command and control environment for presenting the ISIS command language, ISIS is designed to be more generally applicable. For example, the user must define the structure of an ISIS data base and the data base display formats. This enables the user to tailor ISIS to a specific application, by both defining the data base and preparing only desired data base displays. Furthermore, the user can write ISIS "programs" that can be executed interactively. This capability allows the user to prepare more general "commands" tailored to a particular application. Examples of some other possible applications include the development, maintenance, and updating of bibliographies; department accounting and personnel records; and aircraft and/or aircrew records.

An English-like command language was chosen for two reasons. First we wished to develop a system that is easy for a computer-naive user to learn, and we believe an English-like command language supports such a user's intuition. Second, we wanted the observer unfamiliar with ISIS to still get a flavor of what the ISIS user is

PRECEDING PAGE BLANK

trying to accomplish during an ISIS session, and we believe the command language supports this objective as well. Additionally, because ISIS allows the user to build computer files of ISIS "selection criteria," that is, files that contain ISIS statements permitting the selection of those data records satisfying the criteria, we envision users' building libraries of such files that can be reviewed and critiqued by others unfamiliar with ISIS. We believe that such libraries can also serve as a valuable record-keeping and historical device for ISIS users new to a particular application.

ISIS was designed to operate on data bases of modest size, with hundreds rather than hundreds of thousands of records. Such data bases present unique opportunities for the user. For example, even though a data base with hundreds of records may be too large for the user to deal with manually, he might still want to query, modify, and otherwise reorganize the data base. ISIS allows quick reorganization of a data base to suit specific and perhaps transient needs.

The ISIS development effort had several objectives. In addition to those stated above, we wanted to address our attention to data bases of modest size. The intent was to provide the user with rapid access to his data base at modest cost so that he could exploit the computer's power without a significant investment in capital and human resources.

Second, we wanted to develop a system that the computer-naive user would find comfortable, easy to learn, and easy to use. The structure of an ISIS data base, therefore, had to be straightforward, and the ISIS command language had to be easy to understand.

Another objective arose: At the outset, we had an unclear picture of all the ISIS functional requirements. Stated differently, the uncertainty surrounding the application area did not permit the specification at the outset of all the functions we wished ISIS to perform. Therefore, we wanted to employ an "evolutionary" design philosophy, with implementation progressing in stages. The product of each stage was a usable version of ISIS. Use of these early versions of ISIS helped to uncover deficiencies and thereby additional functional requirements. We expect this evolution to continue. Furthermore, we believe that evolutionary design is a highly desirable approach to the implementation of systems characterized by uncertain functional requirements.

ACKNOWLEDGMENTS

It would be impossible to acknowledge all those who have contributed their time and ideas to the development of ISIS. However, we would be remiss if we did not note the contributions of several Rand and Air Force associates.

We are indebted to Stephen Drezner of Rand and Louis Miller of The Wharton School at the University of Pennsylvania, as well as to Lt. Col. Billy Boles and Major Larry Rinderknecht, formerly of DCS/Manpower and Personnel, for their active participation in the early stages of this project.

John Service of DCS/Manpower and Personnel, who was the first ISIS user, also deserves special mention. His willingness to put up with the problems that accompany a system under development and his many helpful suggestions and constructive criticisms proved most valuable.

Richard Hillestad of Rand was instrumental in overcoming several technical obstacles in the system during the project. He also suggested the application of ISIS to the command and control area.

Finally, we owe special thanks to Sidney Miller of Rand, whose advice, counsel, and encouragement throughout the development were greatly appreciated.

CONTENTS

PREFACE iii

SUMMARY v

ACKNOWLEDGMENTS vii

Section
I. INTRODUCTION 1
ISIS Users 1
Some ISIS Definitions 2
The Tutorial Environment--Tactical Command and Control 5
Tutorial Organization 6

II. BASIC ISIS 7
Getting Started--An Easy Way to Define the Data Structure 7
The Load Command 11
The Display/Show Command 13
The Sort Command 21

III. INTERMEDIATE ISIS 23
The Keep Command 23
The Remove Command 25
The Move Command 26
The Save Command 27
The Copy Command 28
Display Formats 29
A Further Discussion of Templates (Data Types) 39

IV. ADVANCED ISIS 42
Domains: Their Definition and Use 42
Logical Expressions (Requirement Lists) 45
Modification of Attribute Values: The "For" Statement 51
Adding New Attributes and Objects: The Insert Command 57

V. THE PERFORM STATEMENT AND ISIS PROGRAMMING 59
The Perform Statement 59
Special Load, Append, and Reset Statements 63

VI. MISCELLANEOUS STATEMENTS 65
Display Size 65
Display Sets 66
Display Formats 66

Display Templates and Display Template
Called <Template Name> 67
Display {Encrypted} File Called <File Name> 67
Edit {Encrypted} File Called <File Name> 67
Remove File Called <File Name> 67
%--Getting to UNIX While in ISIS 67
Hard Copy Output 67
External Data Structure 68
Logging Out of ISIS 69

VII. ISIS SYSTEM DEVELOPMENT 70
Computer-Naive User 70
Modest Data Base Size 72
Incomplete Functional Requirements 72

Appendix
A. QUICK REFERENCE TO ISIS COMMAND LANGUAGE SYNTAX 75

B. SOME PERFORM FILE EXAMPLES 78

C. INDEX TO STATEMENT DESCRIPTIONS 92

I. INTRODUCTION

This report presents a tutorial introduction to ISIS (In Storage Information System), an interactive system developed on the PDP11 computer under the UNIX operating system.* ISIS was developed at Rand to give users interactive access to relatively modest sized data bases, i.e., those containing hundreds rather than hundreds of thousands of records or objects.

ISIS facilities exist to retrieve, organize, and selectively filter objects in a data base of interest to the user. For example, the data base may be a publications bibliography, and the user may wish to search the bibliography for publications written by a specific author, or for publications from a particular subject area, or both. Other example data bases might contain employee records, research projects, or corporate accounting ledgers.

By giving the user the ability to select from and reorganize objects in a data base, we hope he will be able to develop a more intuitive feel for the data base contents and relationships. We further hope that the user will be more readily able to get answers to questions about the data base, e.g., "Which employees are scheduled for salary increases during the next three months," or "What projects are in danger of overrunning their budgets."

This report is designed to aid users of ISIS by presenting and explaining examples of statements from the ISIS command language. It is not, nor is it intended to be, a complete guide to the ISIS command language. The report is semi-tutorial in nature in that some detailed command language syntax is presented along with examples of ISIS statements.

Interactive system designers should find the last section of this report of particular interest. In that section we discuss the design philosophy adopted during the development of ISIS. We also present the rationale for some of the major design decisions made.

ISIS USERS

ISIS users fall into three general categories. The first category includes those responsible for developing an ISIS

*The PDP11 computer is manufactured by Digital Equipment Corporation. The UNIX operating system was developed by Bell Telephone Laboratories.

application. These users would have the job of defining (or directing the definition of) the data structure and display formats, and of developing procedures, either within or outside ISIS, for preparing an ISIS data base. Such users would require an understanding of ISIS facilities. But more important such users would need complete familiarity with the need for applying ISIS to the data base. They would have to understand why automated access to the data base is important.

The second class of users would include those responsible for actual preparation of an ISIS data base. Such a user would need a broad understanding of ISIS facilities and might need some detailed knowledge of how an ISIS data base is formatted on disk.

Users in the third category would be responsible for the day-to-day operation of ISIS in support of the application area. This user would need a more detailed understanding of ISIS facilities than users in the second category, but would not need the intimate understanding required of the first type of user. This report is aimed for the most part at the third type of user. However, material is provided in later sections to assist users belonging in the other two categories.

SOME ISIS DEFINITIONS

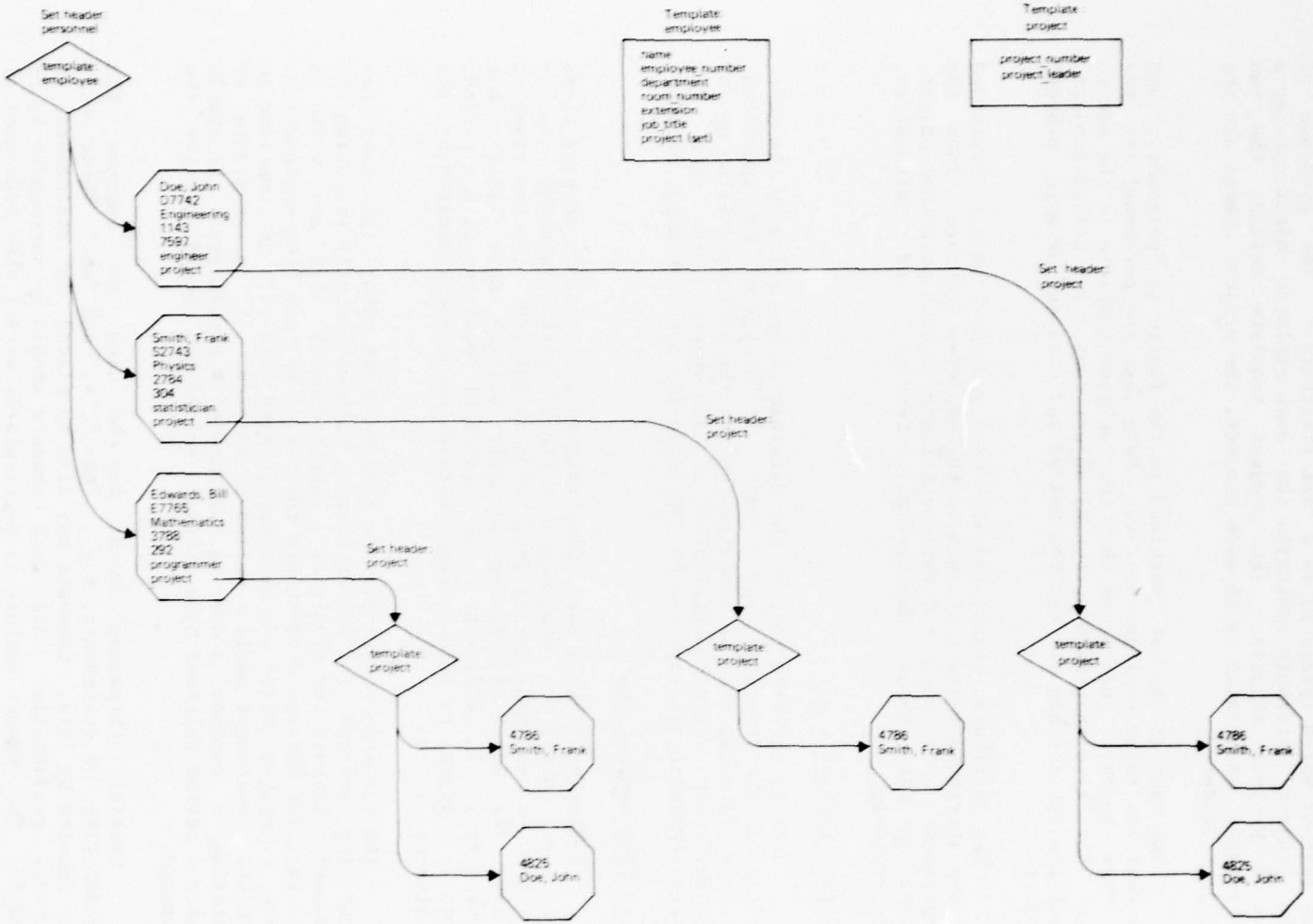
Objects and Sets

ISIS views the world as collections of "objects," and each object has a number of attributes. An example of an object might be an employee, containing attributes such as name, employee_number, department, room_number, extension, and job_title. Objects can be grouped into "sets," as, for example, the "set" of all employees in a given department, or the set of all employees with a given job title. Typically, the objects are found in disk files, and ISIS provides commands that "load" the objects from disk into the computer's main memory and "save" objects onto disk from computer memory.

Templates

In ISIS the structure of an object must be defined by the user. In other words, the user must indicate to ISIS what attributes an object has or which objects it will be called upon to deal with. This is done with a "template," which indicates the attributes associated with objects.

The ISIS data structure consists of objects collected in sets, each object in a set having the same attributes. It is possible, however, for an individual object to own a set in its own right, as for example the set of projects an employee is currently working on. The following figure, which contains two templates and a simple set, illustrates this. The two templates define the structure of employee and project objects. The employee template lists seven attributes,



An example ISIS data structure

the last of which (project) is a "set attribute." The presence of the project attribute indicates that each employee object can own a set of project objects. The project template defines the two attributes associated with each project, the project_number and the project_leader.

The name of the set contained in the figure is "personnel," and it contains three employee objects. Note that the personnel set has a "set header," indicating that the employee template is the one to use. Note also that each personnel object owns a set called project. Each project set has its own set header and owns one or more project objects.

The attributes associated with each personnel object correspond to the attribute names contained in the employee template. Thus the department attribute, the third attribute in each personnel object, takes on the value Engineering, Physics, and Mathematics, respectively.

Display Formats

Just as the structure of the data can be specified by the user, so too can the structure and format of "displays" be specified. Display formats can be constructed by the user by calling up for display only certain attributes of an object. In this way special-purpose displays can be developed for specific applications.

Some ISIS Conventions

Before presenting each ISIS statement, we discuss several items that apply to all the statements. First, each statement can be written on as many (or as few) lines as desired. (In the examples below, for ease of reading we will utilize more lines than necessary.) In addition, each statement must end with a period. This is a signal to ISIS to begin taking the action indicated by the statement.

The carriage return signals ISIS to check what the user has thus far entered for syntax errors, without actually executing the command. Thus in the middle of a long statement ISIS may write a syntax error message even though the period has not been encountered. Once a syntax error has been encountered ISIS will not continue to process a statement until a period is entered. If, in the middle of entering a command, a user is informed of a syntax error, he should enter a period followed by a carriage return and should reenter the command.

Several "throwaway" words may be used to improve the readability of statements, e.g., "the," "a," and "an." These words are ignored by ISIS. Comments may also be placed in statements to improve readability. All such comments should be surrounded by /* and */. The comment feature is particularly useful when sequences of

statements are placed in a file for later callup and execution by ISIS (called "perform" files). Finally, in the examples that follow, several conventions have been adopted to indicate segments of a statement that are optional--enclosed in braces ({})--and those that are mandatory--enclosed in angle brackets (<>).

Unless enclosed in quotation marks, all upper case letters are converted to lower case by ISIS. For example ISIS will interpret the character string F100 as f100. If an upper case F is desired, "F100" should be written. Additionally, several characters have special meaning in ISIS, and if they are desired in character strings, the strings should be enclosed in quotation marks. The special characters are:

```
+ - / * . = { } [ ] ( ) $ ^ %
```

The blank character is also a special character. Character strings that begin with a numeric character must be enclosed in quotation marks also.

THE TUTORIAL ENVIRONMENT--TACTICAL COMMAND AND CONTROL

The examples in this report are based on a demonstration data base developed for Rand's tactical command and control project. This is not to say that ISIS's sole application area is tactical command and control. The system was designed to be and is applicable to a wide variety of areas. We follow a tutorial scenario to introduce the command language, but the detailed syntax of the various language statements is also presented. Since many readers may not be familiar with tactical command and control in general or with the specific elements of the tactical command and control situation dealt with in this tutorial, we will first present a brief description of the tutorial environment.

The tactical command and control arena centers on providing an Air Force commander with the information needed to control and coordinate the implementation of an air tasking order or "frag." The frag, which is the name we use in this report, contains the daily flying orders for the various air wings under the commander's control.

The flying order for a given wing indicates the specific missions that wing is tasked to fly during a given day, and includes detailed information about each mission. This information includes, for example, the number and type of aircraft to fly, the type of ordnance to carry, the mission objective, the target to be struck, any forward air controller or air-to-air refueling data, and the estimated time the aircraft flying the mission should arrive over the target.

Questions that might arise in the course of "executing the frag" revolve around the status of each mission, the status of

targets to be struck, management of airspace, and the avoidance of airspace conflict. For example, we may wish to determine which missions have been canceled and why. Or we may wish to identify those aircraft scheduled to have already returned from a mission that have not as yet returned.

Consistency among various sections of the air tasking order is also important. For example, we would like to insure that aircraft scheduled to strike targets outside the aircraft's range have been furnished air-to-air refueling rendezvous. Furthermore, the air-to-air refueling mission must know about the aircraft scheduled to be refueled.

The tutorial is built around an ISIS data base designed to deal with two frags--the fighter frag and the air-to-air refueling frag. During the course of the tutorial we introduce and define to ISIS the data structures for the two frags and the display formats developed for displaying the frags, and in general we use ISIS statements to order and reorganize the frags.

TUTORIAL ORGANIZATION

The remaining sections are organized so as to present ISIS in increasing levels of difficulty. The next section presents basic ISIS statements that enter, reorder, and display the frags. In Sec. III we introduce moderately complex statements that permit the reorganization of ISIS sets and the saving of those sets in disk files. We also present a detailed treatment of display formats and a discussion of how the user can construct formats tailor-made to his needs. Section IV delves into some of the more complex features of ISIS that allow the user to make more meaningful queries about the data base and also help the user understand how specific objects in a set satisfy a given set of logical expressions. A detailed treatment of logical expressions is also presented, and statements that permit the modification of attribute values are introduced. Section V deals with ISIS programming and special commands designed for use with ISIS programming. In Sec. VI we discuss the format and construction of ISIS data bases and a number of miscellaneous ISIS statements and topics. Finally, Sec. VII discusses the design philosophy employed and the system development capabilities available during the implementation of ISIS.

Three appendixes are also provided. The first contains a quick reference to ISIS command language syntax. The second presents examples of ISIS "perform" procedures--ISIS programs that in essence provide "commands" tailored to the tactical command and control environment. The third appendix contains an index to command language descriptions.

II. BASIC ISIS

Anyone having access to Rand's PDP11 should be able to execute the commands found in this report as he reads them. Once logged into the PDP11, the user should enter the following two statements to execute ISIS in the tactical command and control environment:

```
chdir /rnd/herb/isis.demo
isis
```

These are statements to the UNIX operating system rather than to ISIS. The first instructs UNIX to place the user in the file system segment that supports the tactical command and control demonstration. The second statement instructs UNIX to execute ISIS. Note that while ISIS does not distinguish between upper case and lower case characters, UNIX does. Therefore, in typing the above two statements, only lower case characters should be used.

The gothic font used for the above two statements indicates that the statements can be typed into the PDP11 computer. All such statements in this manual will use that gothic font.

GETTING STARTED--AN EASY WAY TO DEFINE THE DATA STRUCTURE

ISIS has just been invoked, and the ISIS prompt character (?) has just appeared on the terminal. In addition, the prompt character's appearance was accompanied by an audible beep from the terminal. This sound will always occur when the prompt character appears.

The first step is to define the data structure of the objects and the display formats to be used when displaying the objects. For the command and control data bases, the data structure and display formats can be found in a "perform" file called frag_template. The frag_template file contains ISIS statements to define the data structure and display formats.

To instruct ISIS to "execute" the file, we say

```
Perform frag_template.
```

When the prompt character appears again the data structure and display formats will have been defined.

Let us now examine the data structures. We can do this by typing

```
Show the templates.
```

The response will be similar to the following, showing that eight data structures have been defined, i.e., that eight types of object have been defined.

```
distance
targets
control
rendezvous
aar_frags
frags
t_a_s
domain
```

To begin with, we will concentrate only on frags, aar_frags, and rendezvous. To examine the structure of "frags" objects, we type

```
Show the template called frags.
```

and the ISIS response will be similar to the following, indicating the name and data "type" of each attribute associated with a "frags" object.

```
template for set frags (
( unit,          symbol  )
( missn_no,      integer )
( aircraft,      symbol  )
( num_ac,        integer )
( call_sign,     symbol  )
( tot,           time    )
( request_no,    text    )
( missn_type,    symbol  )
( prim_target,   symbol  )
( sec_target,    symbol  )
( fac_sign,      text    )
( ord_load,      symbol  )
( iff_sif_comm,  text    )
( ecm,           text    )
( remarks,       text    )
( aar_time,      time    )
( tanker_sign,   text    )
( aar_alt,       symbol  )
( aar_dur,       time    )
 );
```

The items appearing on the right, e.g., symbol, integer, and text, indicate the "data types" of each attribute. These data types are discussed in detail on p. 39.

Note that we said "frags" instead of "frag" in the above statement. This is because the template name is frags. Whenever referring to this template, the template name must be entered exactly as it was defined. The plural was chosen instead of the singular form to improve the readability of the statements.

When entering the statement into ISIS, the user will notice that the ISIS output differs slightly from the response shown in the report. For example, in ISIS the data types do not line up evenly in a column; instead only one blank space follows each comma. We have taken this minor editorial liberty to improve the readability of this example. We might add also that the above is similar to the statement contained in the frag_template file to define the frags data structure. That is, a "template" statement will be found in the frag_template file (see p. 81) that looks much like the ISIS response from the "Show the template called frags." command.

While most of the attributes in the frags template are not directly used in the tutorial, we feel it necessary to define each attribute in detail. Those definitions appear below, but the reader need not understand the definitions completely.

unit	the organizational unit from which the sortie is to originate, e.g., the 49th Tactical Fighter Wing (49 TFW).
missn_no	a number assigned to this particular mission e.g., 1, 2, ...
aircraft	the type of aircraft to use on this mission, e.g., F-100 or F-4.
num_ac	the number of aircraft to fly on the mission.
call_sign	the call sign associated with the mission for radio communications.
tot	the time over target, that is, the time at which the mission is scheduled to arrive over the target area.
request_no	the identifying number assigned to any request for a mission coming from a non-Air Force organizational unit.
missn_type	the type of mission to be flown, e.g., close air support or counter air.
prim_target sec_target	the names or identifiers of the primary and secondary targets to be attacked during the mission.
fac_sign	the call sign and frequency of the forward air controller, in case coordination with a forward air controller is necessary.

ord_load	the type of ordnance to be carried on the mission.
iff_sif_comm	the settings to be used on the IFF (Identify Friend or Foe) gear found on the aircraft, i.e., the "squawk box" settings.
ecm	the types of electronic countermeasures to be employed.
remarks	any textual remarks or commentary about the mission.
aar_time	the time the mission is scheduled to rendezvous with the tanker sortie if air-to-air refueling will be required.
tanker_sign	the call sign of the tanker sortie.
aar_alt	the altitude at which the mission is to rendezvous with the tanker.
aar_dur	the expected duration of the refueling operation.

Similarly defined are the aar_frags (the air-to-air refueling sorties), as the following statement indicates:

```
Show the template called aar_frags.
```

with the resulting display being:

```
template for set aar_frags (
( unit, symbol )
( missn_no, integer )
( aircraft, symbol )
( num_ac, integer )
( call_sign, symbol )
( aar_area, symbol )
( altitude, symbol )
( rendezvous, set )
( remarks, text )
( offload_time, integer )
);
```

We will not discuss these attributes except to point out the presence of an attribute called rendezvous, which is defined as a "set" attribute. This set of rendezvous, owned by an aar_frag, indicates which frags the aar_frag is to rendezvous with and refuel. The rendezvous set contains rendezvous objects, whose attributes can be displayed in response to the following statement:

```
Show the template called rendezvous.

template for set rendezvous (
( call_sign, symbol )
( aircraft, symbol )
( num_ac, integer )
( aar_time, time )
( time_per_ac, integer )
);
```

The aar_time attribute in this object is the time at which the aar_frag is to rendezvous with the frag.

THE LOAD COMMAND

The load command permits the creation in computer memory of sets of objects from files located on disk. The file contains the objects in a specific external format (described in Sec. VI), and the load command transfers those objects from the file into the computer's main memory. ISIS must have a template for the objects in order to make this translation.

In this and most other statement discussions in this report, we first present the detailed syntax of the statement followed by examples of statement usage. This should help the reader, when looking at statement examples, to understand how those examples fit into the more general statement syntax.

Command Syntax

```
Load {and {dont} display} <template name> {requirements list}
        from {encrypted} <file name> file.

Load {and {dont} display} <set name> using template
        <template name> {requirements list} from
        {encrypted} <file name> file.
```

Examples

```
Load the frags from the fighter_frag file.
```

The above statement will place into a set called frags (whose template is called frags) all the objects from the file named fighter_frag.

```
Load the frags whose aircraft is f4 from the
        fighter_frag file.
```

Execution of the above two statements will cause the "double" loading into computer storage of each frag whose aircraft is f4. The first statement will load all the frags from the fighter_frag file, including all the f4 frags. The second statement will then load only the f4 frags from the fighter_frag file. One way to avoid this double loading is to direct ISIS to load objects into a set whose set name differs from the template name. For example,

```
Load the f100s using template frags whose aircraft is f100
      from the fighter_frag file.
```

places into a set called f100s (using template frags) all the f100 frags from the fighter_frag file.

In Sec. III, we will encounter statements that permit the purging of unwanted objects from a set, including the extra f4 objects in our frags set.

```
Load and display the aar_frags from the aar_frag file.
```

places into the aar_frags set the aar_frags from the aar_frag file. The "and display" instruction causes the aar_frags to be displayed as they are loaded.

```
Load and display the f4_aar_frags using template aar_frags
      for which there is a rendezvous whose
            aircraft is f4 and
            aar_time > `1200 and
            aar_time < `1300
      from the aar_frag file.
```

This statement places into a set called f4_aar_frags (using template aar_frags) those aar_frags scheduled to rendezvous with f4 sorties between 12:00 and 13:00. Recall that each aar_frags object represents an air-to-air refueling mission, and the rendezvous set contains objects that indicate the fighter missions with which the air-to-air refueling mission is to rendezvous. Furthermore all such aar_frags are displayed as they are loaded. Note that an aar frag is loaded only if it "owns" at least one f4 rendezvous object that is scheduled to link up with the aar frag between 12:00 and 13:00. In other words, ISIS is being asked to examine the attribute values of "subobjects" in determining whether or not to load a "primary" object. The "`" in `1200 and `1300 denotes a "time" constant (see p. 39). Typing this statement into ISIS gives the following response:

```
AIR TO AIR REFUELING FRAG

                    AIRCRAFT  CALL   AAR          OFFLOAD
UNIT   MISSN  NO    TYPE      SIGN   AREA   ALT   TIME    REMARKS

2 bw     1     1    kc        tokay  blue   f1      40
                    135       41     anchor 260

          RENDEZVOUS WITH

          CALL   AIRCRAFT   AAR    TIME PER
          SIGN   NO  TYPE   TIME   AIRCRAFT

          anker  2   f4     12:25      5
          21
          anker  2   f4     12:40      5
          31
          anker  2   f4     12:55      5
          41
          anker  2   f4     13:10      5
          51
1 of 2 loaded.
```

The above display occurred because the load command contains the "and display" instruction. The display format employed was defined when we performed frag_template. Display formats will be discussed in detail on p. 29, but we should point out here that in the above display both the primary object (the aar frag) and the subobjects (the rendezvous) are displayed. Furthermore, the aar frag is scheduled to rendezvous with f4 sorties not only between 12:00 and 13:00 but at 13:10 also. Three f4 sorties will rendezvous with the aar sortie between 12:00 and 13:00 also. Finally, the last line of the display (1 of 2 loaded.) indicates that there were two aar frag objects in the aar_frag file, and only one of them satisfied the selection criteria in the ISIS statement. That one, along with its owned set, is the object that was loaded.

Another point needs mention. Saying

for which there is a rendezvous

tells ISIS to shift its attention from the f4_aar_frags object to the set of rendezvous owned by the f4_aar_frags object. All attributes mentioned after this phrase will refer to the rendezvous template rather than the aar_frags template. See the discussion on logical expression in Sec. IV for more details.

THE DISPLAY/SHOW COMMAND

The display command permits the display of objects that have been entered into ISIS, i.e., into the computer's memory or primary storage. The display is guided by display formats developed by the

user (see Display Formats, p. 29). The word "show" is a synonym for "display."

Command Syntax

Display <set name> {rqmt list} {using format <format name>}.

Show <set name> {rqmt list} {using format <format name>}.

Examples

Before discussing the display command, we will need a "clean" version of ISIS, one with no loaded objects. Enter the following:

```
Stop.
isis
Perform frag_template.
```

We now execute the following two statements:

```
Load the frags from the fighter_frag file.
Load the aar_frags from the aar_frag file.
```

The results will be that 28 frags and two aar_frags will be loaded into the frags and aar_frags sets, respectively.

We will now give some examples of display commands on the two sets.

```
Show the frags.
```

will result in the display on the following page.

The display format used is the first "frags" display format ISIS encountered when performing frag_template. There are 28 frags in the set, but ISIS stopped after the first six were displayed, asking for either a carriage return (CR) to continue the display, a break (BRK) to indicate that display is to be suspended, or a "print" to indicate that the screen's contents are to be printed on a hard copy device (a special printer is needed for this option--see Hard Copy Output in Sec. VI). The display format controls the number of objects to be printed before ISIS issues the request for a response. To terminate a display before the prompt is reached, or if one is not expected, the user can hit the <BRK> key.

FIGHTER FRAGS

UNIT	MSN	ACRFT NO	TYPE	CALL SIGN	MISSN TYPE	TARGETS PRIMARY	TARGETS SECOND	TOT	ORD	IFF/SIF/ COMM	FAC CALL SIGN
178 tfgp	1	3	f100	bryan 01	close air support	armor veh pk 125675		12:00	a-3	mode ii 3321	alfalfa 366.3(p) 366.9(s)
178 tfgp	2	3	f100	bryan 11	close air support	armor veh pk 154693		13:00	a-4	mode ii 3322	alfalfa 366.3(p) 366.9(s)
178 tfgp	3	3	f100	bryan 21	close air support	veh pk 115567		13:50	a-3	mode ii 3522	alfalfa 366.3(p) 366.9(s)
178 tfgp	4	2	f100	bryan 31	close air support	armor veh pk 093554		14:00	a-4	mode ii 3524	alfalfa 366.6(p) 366.9(s)
178 tfgp	5	2	f100	bryan 41	close air support	armor veh pk 973607		14:30	a-3	mode ii 3223	alfalfa 366.3(p) 366.9(s)
178 tfgp	6	2	f100	bryan 51	close air support	armor veh pk 043667		15:00	a-3	mode ii 3425	alfalfa 366.3(p) 366.9(s)

Waiting for cr, brk, or 'print'?

If we wish to examine only the frags from the 49th TFW (in the demonstration data base the unit attribute is 49 tfw for those frags originating from the 49th TFW) that require air-to-air refueling, we can type the following statement:

```
Show the frags whose
        unit is "49 tfw" and
        tanker_sign is not ""
        using format show_fighter_aar.
```

We note three things in this statement. First, we note that in

```
unit is "49 tfw"
```

"49 tfw" is a text string, but all text strings need not be enclosed in quotation marks. This one is for two reasons. It begins with a numeric character, and it contains an embedded blank. The quotation marks are used to let ISIS know that the item is a text string. Were they not present ISIS would read the 49 and interpret it as an integer. The tfw would then be unexpected, and ISIS would generate a syntax error.

The second thing to note deals with

```
tanker_sign is not ""
```

This means look for frags whose tanker_sign attribute has a value. The "" is the way to represent an "empty" text or symbol attribute, and therefore

```
not ""
```

denotes "nonempty" text or symbol attributes, i.e., text or symbol attributes that have a value.

The last item to note is

```
using format show_fighter_aar
```

This instructs ISIS to use the display format called

```
show_fighter_aar
```

The following display will be produced in response to the above display statement.

FIGHTER FRAG AAR SCHEDULE

UNIT	MISSN NO	CALL SIGN	AIRCRAFT NO	AIRCRAFT TYPE	TANKER CALL SIGN	RENDEZVOUS TIME	AAR ALTITUDE	AAR DURATION
49 tfw	3	anker 21	2	f4	tokay 41 235.1(p) 254.6(s)	12:25	fl 260	0:10
49 tfw	4	anker 31	2	f4	tokay 41 235.1(p) 254.6(s)	12:40	fl 260	0:10
49 tfw	5	anker 41	2	f4	tokay 41 235.1(p) 254.6(s)	12:55	fl 260	0:10
49 tfw	6	anker 51	2	f4	tokay 41 235.1(p) 254.6(s)	13:10	fl 260	0:10

It should be clear by now that a display format need not provide for the display of all attributes in an object. But one format has been developed that contains the display of all attributes in a frags object. The following statement illustrates:

```
Show the frags whose
        call_sign contains anker and
        missn_no is 3
        using format full_fighter_frag.
```

One thing should be noted in the above statement, namely,

```
call_sign contains anker
```

Here ISIS is being instructed to look at the call_sign attribute value to determine if the text string "anker" appears anywhere within it. Execution of this statement results in the following display.

MISSION IDENTIFICATION SECTION

UNIT:	49 tfw
MISSION NUMBER:	3
CALL SIGN:	anker 21
AIRCRAFT NUMBER/TYPE:	2/f4
MISSION TYPE:	counter air
REMARKS:	

TARGET SPECIFICATION SECTION

PRIMARY:	ta 128
SECONDARY:	ta 127
TIME OVER TARGET:	12:00
ORDNANCE:	c-18

REQUEST/FAC/IFF SECTION

REQUEST NUMBER:	
FAC CALL SIGN:	
IFF/SIF:	mode ii 4156

AIR TO AIR REFUELING SECTION

TANKER CALL SIGN:	tokay 41 235.1(p) 254.6(s)
ALTITUDE:	fl 260
RENDEZVOUS TIME:	12:25
AAR DURATION:	0:10

Note that all the alphabetic characters in the text strings displayed by ISIS are lower case--no capitals are present. For example,

tokay 41 235.1(p) 254.6(s)

contains no capital letters--but the descriptive text

TANKER CALL SIGN:

that goes along with it in the above display does. The reason for this is that ISIS, whenever it encounters a capital letter that is not surrounded by quotation marks, changes the capital letter to lower case. The attribute being displayed is tanker_sign, and its value is

tokay 41 235.1(p) 254.6(s)

While there must be quotation marks surrounding this value because of the presence of blank and special characters, only lower case letters were used. However, the display format called full_fighter_frag does contain text strings surrounded by quotation marks. Thus the "pretext"

TANKER CALL SIGN:

is displayed in capital letters. The full_fighter_frag format, as it appears in the frag_template file, is listed on p. 84. Display formats are discussed in detail on p. 29.

It is possible to have attribute values that contain capital letters, so long as the attribute values are enclosed in quotation marks in the disk files that contain them--see External Data Structure in Sec. VI. In the demonstration data base, and for ease of typing, we have refrained from using capital letters in attribute values.

As a last example of the display command, the following will cause the display of the two objects contained in the aar_frags set.

Show the aar_frags.

and the resulting display is:

AIR TO AIR REFUELING FRAG

UNIT	MISSN	AIRCRAFT NO	AIRCRAFT TYPE	CALL SIGN	AAR AREA	ALT	OFFLOAD TIME	REMARKS
2 bw	1	1	kc 135	tokay 41	blue anchor	fl 260	40	

RENDEZVOUS WITH

CALL SIGN	AIRCRAFT NO	AIRCRAFT TYPE	AAR TIME	TIME PER AIRCRAFT
anker 21	2	f4	12:25	5
anker 31	2	f4	12:40	5
anker 41	2	f4	12:55	5
anker 51	2	f4	13:10	5

Waiting for cr, brk, or 'print'?

AIR TO AIR REFUELING FRAG

UNIT	MISSN	AIRCRAFT NO	AIRCRAFT TYPE	CALL SIGN	AAR AREA	ALT	OFFLOAD TIME	REMARKS
2 bw	2	1	kc 135	tokay 42p	blue anchor	fl 260	24	drogue sortie. confirm that tokay 41 has departed prior to entering anchor. times can slide. heads up. falcon 269

RENDEZVOUS WITH

CALL SIGN	AIRCRAFT NO	AIRCRAFT TYPE	AAR TIME	TIME PER AIRCRAFT
traps 31	4	f100	13:45	3
traps 41	4	f100	14:00	3

THE SORT COMMAND

The sort command allows the user to order the objects in a set according to the value of the object attributes. For example, employee records can be sorted alphabetically, and frags can be sorted by tot (estimated time over target) within unit.

Statement Syntax

Sort <set name> by <attribute name> {alpha} {reversed}.

The type of sorting performed is governed by the data type of the attribute. Data types are described fully in Further Discussion of Templates (Data Types) found in Sec. III. Table 1 indicates the type of sorting that will be done.

Table 1

DATA TYPE SORTING CONVENTIONS

Data Type	Type of Sorting
integer, date, time or fulldate	descending order (highest to lowest value)
text	alphabetically
symbol	position in symbol table (highest to lowest)

To reverse the order of the sort, the user may simply add the word "reversed" to the sort statement. To sort a symbol attribute alphabetically, the word alpha can be added to the sort statement.

Examples

To arrange the frags in ascending order by time over target, we use the following statement:

```
Sort the frags by tot reversed.
```

The following statement sorts the frags alphabetically by unit:

```
Sort the frags by unit alpha.
```

improve the readability of the statements.

To sort the frags by tot within unit, we use the following statements:

```
Sort the frags by tot reversed.
Sort the frags by unit alpha.
```

To sort the frags by primary target, we type:

```
Sort the frags by prim_target alpha.
```

This statement is useful if more than one mission is scheduled to strike the same target, and it allows the user to see the ordering of the missions over a target, e.g., the time interval between the strikes.

III. INTERMEDIATE ISIS

It is important to distinguish between ISIS objects in the computer's memory and the external representation of those objects in a file. With the exception of the load statement, the statements we have been dealing with act on ISIS objects as they exist in memory. The display statement allows for the selective display of objects in a set--where by set we mean that the objects are in memory. The sort statement allows for the physical reordering of the objects in a set. Stated differently, if we visualize the set as a line-up of different objects, the sort statement rearranges the line based on attribute value.

Other statements also exist to alter the organization of objects in sets. Objects can be removed from sets or moved from one set to another. Just as is the case with the sort statement, this removal or movement affects only the objects in the computer's memory. The disk file from which the objects were originally loaded is not altered by these statements. Statements do exist, however, that can modify the contents of the disk file as well.

In this section we will present the statements that permit the reorganization of sets in computer storage. We will also introduce statements that permit alteration of the disk files from which the objects were loaded as well as the creation of new disk files. Next we will examine in detail the display formats and ISIS features that allow the user to tailor displays to satisfy particular needs. Finally we will discuss templates and data types in detail.

THE KEEP COMMAND

The keep command permits the selective purging of objects from a set. Only those objects in a set that satisfy the command's requirement list will not be purged. All other objects will be deleted from the set.

It is important to note that once an object is purged from a set it is no longer in computer memory. This in no way affects the object's status in the disk file from which it was loaded.

Command Syntax

Keep <set name> {rqmt list}.

Examples

```
Keep the frags whose
        unit is "181 tfgp" and
        fac_sign is not "".
```

This command will cause ISIS to look at all objects in the frags set, "removing" from the set all objects that do not satisfy the requirements. Thus, only an object whose unit is the 181st TFGP and whose FAC call sign is not "empty" (that is, a forward air controller is needed) will remain in the frags set. All other objects in the frags set will be deleted from storage. Note that this command affects only objects in the computer's memory and does not affect the contents of the file from which the object was entered into the system.

Until now, only one example has been given that refers to owned sets, e.g., sets such as rendezvous which are owned by an object. We have shown an example of how the attribute values in an owned set can be examined to determine whether the owning object satisfies a logical expression or requirement list--see the last load command example on p. 12. But it is possible to affect the contents of an owned set without altering the owning object. The following statement illustrates this:

```
Keep the rendezvous whose
        aircraft is f100
        within aar_frags.
```

This statement will cause the removal from each rendezvous set owned by an object in the aar_frags set all those objects whose aircraft attribute value is not f100. Only the rendezvous objects whose aircraft attribute value is f100 will remain in rendezvous sets. The key part of the statement is

```
within aar_frags.
```

because it indicates that the rendezvous sets are owned by objects in the aar_frags set. We could have qualified the statement further by restricting attention only to certain objects in the aar_frags set, as, for example,

```
Keep the rendezvous whose
        aircraft is f100
        within aar_frags whose seq is 1.
```

In this case, only the first aar_frags object will be affected, where seq means "sequence in the set."

The following statement would affect only rendezvous objects owned by the last aar_frags object:

```
Keep the rendezvous whose
        aircraft is f100
        within aar_frags whose seq is last.
```

The logical expressions discussion in Sec. IV provides a detailed treatment of the types of selection criteria that can be used.

THE REMOVE COMMAND

The remove command is the reverse of the keep command, permitting the review of objects in a set and elimination of those satisfying the specified requirements.

Command Syntax

```
Remove <set name> {rqmt list}.
```

Examples

```
Remove the frags whose
    unit is not "181 tfgp" or
    fac_sign is "".
```

```
Keep the frags whose
    unit is "181 tfgp" and
    fac_sign is not "".
```

The remove statement on the left is equivalent to the keep statement on the right. The keep statement is the first example in the keep statement section above.

The two statements below are also equivalent:

```
Remove the rendezvous whose
        aircraft is not f100
        within aar_frags.
```

```
Keep the rendezvous whose
        aircraft is f100
        within aar_frags.
```

the statement on the right being the second keep statement example.

The remove statements associated with the third and fourth keep statement examples are presented below:

```
Remove the rendezvous whose
        aircraft is not f100
        within aar_frags
        whose seq is 1.
```

```
Keep the rendezvous whose
        aircraft is f100
        within aar_frags
        whose seq is 1.
```

```
Remove the rendezvous whose
        aircraft is not f100
        within aar_frags
        whose seq is last.
```

```
Keep the rendezvous whose
        aircraft is f100
        within aar_frags
        whose seq is last.
```

Note in these two statement pairs that the phrases

within aar_frags whose seq is 1

within aar_frags whose seq is last

are the same in both the remove and keep statements. These phrases indicate to ISIS which owning objects to consider, and therefore only the rendezvous sets associated with the owning objects up for consideration are filtered. Stated differently, the remove and keep processes apply to the owned sets rather than the owning sets. To apply the process to the owning set we might say

Remove the aar_frags whose seq is last.

This statement causes the removal from computer memory of the last aar_frags object as well as its rendezvous set.

THE MOVE COMMAND

This command permits the movement of objects from one set to another. The command is useful when trying to isolate, by placing into one set, those objects that satisfy a requirements list while retaining in ISIS (i.e., by retaining in computer storage) the other objects.

Command Syntax

Move <set name> {requirement list} into <set name>.

Examples

The examples presented below will parallel the keep statement examples presented above.

```
Move the frags whose
    unit is not "181 tfgp" or
    fac_sign is ""
    into no_fac_181.
```

```
Keep the frags whose
    unit is "181 tfgp" and
    fac_sign is not "".
```

The above statement will place all the frags satisfying the selection criteria into the set named no_fac_181. The frags set will contain the remaining objects, i.e., those that do not satisfy the requirements list. The effect of the statement, therefore, is to keep in the frags set only those 181st tfgp objects requiring FAC interaction, but the other objects are still available in the set no_fac_181.

```
Move the rendezvous whose
        aircraft is not f100
        within aar_frags
        into f4_rendezvous.
```

```
Keep the rendezvous whose
        aircraft is f100
        within aar_frags.
```

This statement will move all "owned" rendezvous whose aircraft are not f100 into the set named f4_rendezvous. Note that the f4_rendezvous set is not owned by an object but is itself a primary set. The objects in the owned rendezvous sets that do not satisfy the selection criteria will remain unaffected.

```
Move the rendezvous whose
        aircraft is not f100
        within aar_frags
        whose seq is 1
        into f4_rendezvous.
```

```
Keep the rendezvous whose
        aircraft is f100
        within aar_frags
        whose seq is 1.
```

```
Move the rendezvous whose
        aircraft is not f100
        within aar_frags
        whose seq is last
        into f4_rendezvous.
```

```
Keep the rendezvous whose
        aircraft is f100
        within aar_frags
        whose seq is last.
```

These statements operate on the first and last aar_frags objects, respectively, appropriately moving the non-f100 objects into the f4_rendezvous set. The remaining rendezvous objects are undisturbed.

THE SAVE COMMAND

This command allows the user to write onto a disk file a set of objects. It is useful for saving working sets from one ISIS session to another. The statement does not alter the set in memory in any way. It merely creates a file containing the contents of the set.

Command Syntax

Save <set name> {rqmt list} into {encrypted} <file name > file.

Examples

```
Save the frags into the fighter_frag file.
```

All the objects in the frags set will be written into the fighter_frag file unless the file already exists. If the file does already exist, then ISIS will respond with the following:

```
File fighter_frag already exists.  Enter option desired.
  'rewrite<CR>'  if file is to be rewritten.
  'append<CR>'   if file is to be appended.
  '<CR>'         if file is to be left unchanged.
  option?
```

Responding to the option prompt with

```
rewrite
```

will cause the contents of the frags set to replace the contents of the fighter_frag file. The previous file contents will be lost.

Responding with

```
append
```

will cause the contents of the frags set to be added to the contents of the fighter_frag file, provided that the fighter_frag file currently contains a valid externally formatted ISIS set. The objects currently in the fighter_frag file will not be altered.

Responding with a carriage return will cause no action to be taken.

It should be noted that the objects in storage are not affected by the statement. They remain in the frags set.

We should also point out that if the UNIX write protection bits do not permit the writing of the file, even though the response is append or rewrite no writing will take place.

Other examples of the save statement are listed below:

```
Save the frags whose unit is "49 tfw" into the
        frag_49tfw file.
```

This will place into the frag_49tfw file the objects in the frags set associated with the 49th TFW.

```
Save the aar_frags
        for which there are rendezvous whose
                call_sign contains anker
        into the frag_49tfw_aar file.
```

This will place into the frag_49tfw_aar file the aar_frags objects associated with sorties scheduled to fly from the 49th TFW--their call signs contain anker.

THE COPY COMMAND

This command permits the user to selectively copy objects from one file into another. It is useful when the user expects that the objects satisfying the selection criteria will be too numerous to fit into computer storage all at once were the load command to be used (the demonstration objects do not have this problem, but if the number of objects were large--say 400 or 500--then there probably would not be enough room in storage to hold all of them). The command is also useful if several subfiles are to be created from a master file, for example, the creation of files containing only the frags for a specific unit. The copy command is similar to the load command, except that objects satisfying the criteria are not kept in primary storage but are written into another file.

Command Syntax

```
Copy {and {dont} display} <template name> {rqmt list}
        from {encrypted} <old file name> file
        into {encrypted} <new file name> file.
```

Examples

```
Copy the frags whose unit is "178 tfgp"
        from the fighter_frag file
        into the frag_178_tfgp file.
```

Assuming that the fighter_frag file contains externally formatted frags objects, then all frags in the file whose unit is the 178th TFGP will be written into the frag_178_tfgp file. If the file already exists, then ISIS will present the same options as with the save statement, namely, rewrite, append, and <CR>. The contents of the fighter_frag file is unaltered.

```
Copy and display the aar_frags whose
        aircraft is a c135 and
        for which there is a rendezvous whose
                aircraft is an f4
        from the aar_frag file
        into the c135_f4 file.
```

This statement will cause the creation of a file containing each aar_frags object whose aircraft is a c135 provided that the aar_frags object is scheduled to rendezvous with an f4 sortie. All such aar_frags will be displayed as well. Note that both aar_frags and rendezvous have aircraft attributes, and that "a" and "an" are throwaway words that were inserted in the statement for clarity just as "the" has been.

DISPLAY FORMATS

Display formats were mentioned on several occasions during the discussion, and display formats were referred to in several examples. We will now examine display formats in some detail, first presenting some display format examples, and then examining the formats defined when frag_template is performed. We have chosen not to present the detailed statement syntax first because we do not feel that such presentation would aid in understanding display formats.

If we wish to display just the aircraft type associated with the objects in the frags set, the following display format could be employed:

```
Format show_aircraft for the set frags {
        ( , aircraft, , , )
}.
```

This statement is a valid ISIS statement, and an example below will indicate what the extra commas are for in the "attribute line."

Once this statement is entered into ISIS we can type

```
Show the frags using format show_aircraft.
```

with the following not too desirable result.

```
f100f100f100f100f100f100f100f100f100f100f100f100f100f100f100
f100f4f4f4f4f4f4f4f4f4f4
```

The aircraft attribute values are being displayed, but they are difficult to read. To make the display more legible, the following format could be used:

```
Format show_aircraft for the set frags {
        (" ", aircraft, , , )
}.
```

Note the difference between this and the first format, namely, the presence of the

```
" "
```

in the field before the attribute field in the attribute line. This first field of the attribute line is called the "pre-text" field, and any text string appearing in that field will be displayed prior to the display of each object's attribute value.

When we now type

```
Show the frags.
```

we get the following display:

```
f100 f100 f100 f100 f100 f100 f100 f100 f100 f100 f100 f100 f100
f100 f100 f100 f100 f4 f4 f4 f4 f4 f4 f4 f4 f4 f4 f4
```

We did not have to use the format name in the above display statement, because the first display statement mentioned it, and the same format name will be used until a display statement calls for a different format. In this display, each aircraft attribute value is preceded by one blank character.

Alternatively, we could have used the following format to separate the attribute values:

```
Format show_aircraft for set frags {
        ( , aircraft, 10, , )
}.
```

This format will display the aircraft attribute values in a ten character field with the text left-justified, i.e., the text string will begin at the extreme left of the ten character field, and those right-hand characters of the field that are not used will be filled with blanks. The statement

```
Show the frags.
```

yields the following:

```
f100    f100    f100    f100    f100    f100    f100
f100    f100    f100    f100    f100    f100    f100
f100    f100    f100    f4      f4      f4      f4
f4      f4      f4      f4      f4      f4      f4
```

The following format also provides a type of spacing:

```
Format show_aircraft for set frags {
        ( "
",          aircraft, , , )
}.
```

The point to note here is the presence of

```
          "
"
```

in the pre-text field, which will cause a carriage return to be executed prior to the display of each attribute value.

The display statement

```
Show the frags.
```

yields:

```
f100
f100
f100
f100
f100
f100
f100
f100
f100
f100
f100
f100
f100
f100
f100
f100
f4
f4
f4
f4
f4
f4
f4
f4
f4
f4
f4
```

Here each attribute value appears on a separate line.

A user will notice when typing these statements into ISIS that the displays in the report are slightly different from those on the terminal screen. The terminal screen displays use 80 columns, whereas we chose to shorten the display lines in the report. In fact, in all but the last example ISIS has generated "only one line of display" for each statement, but that line was much more than 80 columns in length. The terminal's wrap-around capability make it appear as if more than one line is being generated.

So far, we have been displaying only one attribute; now we will show some examples with more than one attribute displayed per object. The following format permits the display of three attributes. The call_sign is displayed on a new line--because of the pre-text carriage return--in a field that is ten characters long. Next, the num_ac attribute value is displayed with no pre-text, and only in the exact field needed to display the full number--because of the absence of a field width entry. Finally the aircraft attribute value is displayed, preceded by a "/", and in exactly the field width needed.

```
Format show_aircraft for set frags {
        ( "
",             call_sign, 10, ,  )
        (   , num_ac   ,   , ,  )
        ( "/", aircraft ,   , ,  )
}.
```

The resulting display is:

```
bryan 01  3/f100
bryan 11  3/f100
bryan 21  3/f100
bryan 31  2/f100
bryan 41  2/f100
bryan 51  2/f100
bryan 61  3/f100
bryan 71  3/f100
bryan 81  3/f100
bryan 91  3/f100
traps 11  4/f100
traps 21  4/f100
traps 31  4/f100
traps 41  4/f100
traps 51  4/f100
traps 61  4/f100
traps 71  3/f100
anker 11  2/f4
anker 13  2/f4
anker 21  2/f4
anker 31  2/f4
anker 41  2/f4
anker 51  2/f4
anker 61  2/f4
anker 71  3/f4
anker 81  3/f4
anker 91  3/f4
anker 01  2/f4
```

This display contains more information, in that we have a frag's call sign, aircraft number, and type all on one line.

When typing long, complex statements into ISIS, the user may encounter ISIS syntax errors. To help correct errors, the user can type the statements into a file by employing the Rand Editor,* instead of typing them directly into ISIS. Typing the statement

```
Edit the file called zformat.
```

invokes the Rand Editor for the file named zformat. The format can be typed into the file, then the user can exit the editor and resume the ISIS session.

Typing

```
Perform zformat.
```

will result in the display format's entry into ISIS. Even if the format had been previously entered, performing zformat will replace the old format with the new one.

The next example shows a heading for each of the fields as well as a shortened field width for the call_sign attribute value.

```
Format show_aircraft for set frags {
        ( heading, "
CALL    AIRCRAFT
SIGN    NO  TYPE
        " )
        ( "
",              call_sign, 5, , )
        ( ,     num_ac   , 4, , )
        ( "/", aircraft ,  , , )
}.
```

The statement

```
Show the frags whose seq < 8.
```

yields

*The Rand Editor is a "two dimensional," full page editor developed at Rand. It is described in Walter Bilofsky, The CRT Text Editor NED--Introduction and Reference Manual, The Rand Corporation, R-2176-ARPA, December 1977; and in Jeanne Kelly, A Guide to NED: A New On-Line Computer Editor, The Rand Corporation, R-2000-ARPA, July 1977.

```
CALL    AIRCRAFT
SIGN    NO  TYPE

bryan    3/f100
01
bryan    3/f100
11
bryan    3/f100
21
bryan    2/f100
31
bryan    2/f100
41
bryan    2/f100
51
bryan    3/f100
61
```

Several points should be noted about this display. First, the heading was produced because the following appeared in the format, with everything between the quotes being used as the heading:

```
        ( heading, "
CALL    AIRCRAFT
SIGN    NO  TYPE
        " )
```

The call sign appeared on a new line and took two lines because of the contents of the following attribute line:

```
        ( "
",              call_sign, 5, , )
```

Once again we see the quotation mark, carriage return, quotation mark causing the call sign attribute to be displayed on the new line. But note also that the field width has been specified as five characters. All of the call signs in the fighter_frag file are more than five characters long, so ISIS continues the call sign display on the next line in the same five character field, as in

```
bryan
01
```

It is also possible to specify the maximum number of lines of continuation that will be permitted. For example, if the following had appeared in the format only one line would be used:

```
        ( "
",              call_sign, 5, , 1 )
```

and the resulting call_sign attribute display would have been

```
bryan
```

The attribute value display would have been truncated after one line.

The examples thus far have included all but one of the following fields in the attribute line, namely, the pre-text field, the attribute name field, the width field, and the "depth" field. The last field in the attribute line is the "subformat" field and is used when, during the display of an object, we wish to display those objects in the primary object's owned set. To provide an example of the use of subformats, we will examine two formats that appear in the frag_template file, the file that was "performed" at the start of this ISIS session. In the last example in the display statement discussion, the aar_frags were displayed making use of these two formats.

```
     Format show_aar for the set aar_frags {
        ( display, 1 )                                          <-------
        ( heading, "
AIR TO AIR REFUELING FRAG

             AIRCRAFT  CALL  AAR          OFFLOAD
UNIT  MISSN  NO  TYPE  SIGN  AREA   ALT   TIME   REMARKS
" )
        ( "
"               , unit          , 4, , )
        ("  "   , missn_no      , 3, , )
        (       , num_ac        , 6, , )
        ("  "   , aircraft      , 4, , )
        ("  "   , call_sign     , 5, , )
        (" "    , aar_area      , 6, , )
        (" "    , altitude      , 3, , )
        (       , offload_time, 7, , )
        ("    " , remarks       ,30, , )
        ("
"               , rendezvous    ,  , show_rendezvous, )     <-------
     }.

     Format show_rendezvous for the set rendezvous {        <-------
              ( heading, "
        RENDEZVOUS WITH

        CALL  AIRCRAFT  AAR   TIME PER
        SIGN  NO  TYPE  TIME  AIRCRAFT
" )
        ( "
        "     , call_sign    , 5, , )
        (     , num_ac       , 3, , )
        ("  ", aircraft      , 4, , )
        ("  ", aar_time      , 5, , )
        (     , time_per_ac  , 6, , )
     }.
```

Three lines in the above format displays have been marked with arrows. The last two will be discussed first, since they provide an example of the use of subformats. The first format refers to aar_frags and the second to rendezvous, and recall that each aar_frags object can own a set of rendezvous. In the aar_frags format, called show_aar, an attribute line makes reference to rendezvous, and the subformat field contains

```
show_rendezvous
```

This is the name of the rendezvous format, and the attribute line instructs ISIS that the show_rendezvous format should be used to display the contents of the owned rendezvous set. Thus, if we execute the statement

```
Show the aar_frags using format show_aar.
```

ISIS begins the display of each aar_frags object in the show_aar format until it comes to the rendezvous attribute line. It then shifts to the show_rendezvous format displaying the objects in the owned rendezvous set. When the display of the rendezvous set is completed, ISIS returns to the show_aar format and continues the display of aar_frags.

The first line marked with an arrow in the show_aar format contains

```
( display, 1 )
```

This instructs ISIS to stop after the display of each aar_frags object, giving the user the "should I continue the display" prompt. Any integer can be placed in the display line, with a zero entry meaning that all the objects in the set are to be displayed, and a nonzero meaning that the "display continue" prompt should be requested after the indicated number of objects is displayed.

Format Command Syntax

Not all of the options available in the format statement have been demonstrated in the examples above. More detailed format options will be described below.

```
Format <format name> for set <template name> {
( heading, <"text string"> )                          /* optional */
( display, <number of objects to be displayed>,
                                  <formfeed> )        /* optional */
( select,  <"select" attribute name> )                /* optional */
   <attribute list--see below>
}.
```

The braces {} are required in this statement. The heading, display, and select lines are optional.

The three optional entries--heading, display, select--provide some display flexibility. The heading entry permits the placement of heading information prior to the display of any objects. The display entry permits control of the number of objects to be displayed after which ISIS asks if the display should continue, e.g., (display, 10) causes the display of 10 objects followed by a query from ISIS as to whether or not more objects should be displayed. The select entry (which we did not use in the examples discussed above) allows for the display of only those objects for which a particular attribute value is nonzero; e.g.,

```
( select, tot )
```

would cause the display of only those objects whose tot attribute (time over target for frags) is nonzero.

We will now turn attention to the attribute list. The format of each entry in the "attribute list" is

```
( {"pre-text"}, {attribute name}, {field width},
                {subformat}     , {field depth} )
```

The optional pre-text field is displayed prior to the display of the attribute value. The attribute name--which is required most of the time, but can be optional if a subformat is present--must be a valid attribute name appearing in the template statement for the type of object being displayed. The field width indicates the number of characters to use when displaying the attribute value--if not provided then enough characters will be used to display the entire attribute value.

The subformat entry can be used in two different situations. If the attribute is a "set" attribute, as is the case with the set of rendezvous owned by an aar-frags, then the subformat must be provided to display the member objects of the subset (e.g., the rendezvous objects). If no attribute name is provided, then the subformat must refer to the same object type as the format.

The depth field applies to text and symbol attributes. If the field width is shorter than the length of the text string or symbol, the field depth indicates the number of lines of continuation to use to print the text string. The additional lines will line up right under the first line of the field, left-justified. If a zero is entered, or if no field depth entry is provided, then as many lines as necessary will be used. If the number of lines specified in the field depth is insufficient to display the attribute value, then the text will be truncated accordingly.

A FURTHER DISCUSSION OF TEMPLATES (DATA TYPES)

Templates have already been discussed in this report, and we noted that templates are the way the data structure is defined. In fact, the three template statements shown in Table 2 below are taken from the frag_template file--they are valid ISIS template statements. Each attribute associated with an object must be defined in the template statement defining the object's data structure, along with the "data type" of the attribute. This subsection will describe the data types.

Table 2

TEMPLATES USED TO DEFINE THE frags, aar_frags, AND rendezvous OBJECTS

```
Template for the set frags (
   ( unit,          symbol  )
   ( missn_no,      integer )
   ( aircraft,      symbol  )
   ( num_ac,        integer )
   ( call_sign,     symbol  )
   ( tot,           time    )
   ( request_no,    text    )
   ( missn_type,    symbol  )
   ( prim_target,   symbol  )
   ( sec_target,    symbol  )
   ( fac_sign,      text    )
   ( ord_load,      symbol  )
   ( iff_sif_comm,  text    )
   ( ecm,           text    )
   ( remarks,       text    )
   ( aar_time,      time    )
   ( tanker_sign,   text    )
   ( aar_alt,       symbol  )
   ( aar_dur,       time    )
).
```

```
Template for the set aar_frags (
   ( unit,          symbol  )
   ( missn_no,      integer )
   ( aircraft,      symbol  )
   ( num_ac,        integer )
   ( call_sign,     symbol  )
   ( aar_area,      symbol  )
   ( altitude,      symbol  )
   ( rendezvous,    set     )
   ( remarks,       text    )
   ( offload_time,  integer )
).
```

```
Template for the set rendezvous (
   ( call_sign,     symbol  )
   ( aircraft,      symbol  )
   ( num_ac,        integer )
   ( aar_time,      time    )
   ( time_per_ac,   integer )
).
```

The valid data types are listed below:

integer	date
text	time
symbol	fulldate
set	domain

Integer attributes can take on integer values between -32767 and 32767 (the limits are dictated by the size of the PDP11 word).

Text attributes can hold any text string, as we saw in some of the examples above. A text string must be enclosed in quotation marks if it contains blanks or special characters, e.g., + - / * . = $ % ^ ( ) { } [ ]. The underscore character _ is not considered special. Note that ISIS transforms all capital letters to lower case unless they appear within quotes, e.g., "Smith, John."

Symbol attributes also hold textual information, e.g., f4 or f100. But symbol attributes provide storage economies in that the actual textual information appears in storage only once, in a symbol table. For symbols the attribute value is a pointer to the symbol table entry. Symbol attributes are appropriate if it is expected that many occurrences of the text strings associated with that attribute will be encountered. If the user expects a particular text string to appear repeatedly in a file, it should be made into a symbol.

Set attributes are used to indicate that the object whose structure is being defined is to own a set of objects in its own right. As was the case with the rendezvous set, the owned set must have a template.

Date, time, and fulldate attributes are related. A date attribute is used to hold calendar dates, and in ISIS dates are specified by using date constants, e.g.,

```
$780721
```

means July 21, 1978. $7807 mean "between" June and July 1978. And $78 means between 1977 and 1978. Note that

```
$780630 < $780700 < $780701

$771231 < $780000 < $780101
```

For convenience, we can write $7807 instead of $780700 and $78 instead of $780000.

The general form of a date constant is

```
$yymmdd
```

where yy refers to year, mm to month, and dd to day of the month. Date constants must always be preceded by a $ to distinguish them from integers.

Time attributes are used to hold the time of the day, and they too have a special form:

^hhnn

where hh is the hour and nn the minute of the hour. Thus

^0624

refers to 6:24 AM, and

^1751

refers to 5:51 PM. Time constants must always be preceded by the caret character ^ to distinguish them from integers.

Fulldate attributes combine both date and time, and their general form is

$yymmdd^hhnn

Examples of fulldate constants are

$820701^0723	7:23 AM on July 1, 1982
$8207010723	Same as above--the caret need not be specified when all of the date is there, but the example is probably more easily interpreted if the ^ is there.
$8207^1841	6:41 PM on July "00", 1982

The remaining data type is the domain data type. This data type, described in detail in the next section, permits the user to define groups of symbols so that queries can be phrased in terms of relationships among the symbols in such a group.

IV. ADVANCED ISIS

The ISIS facilities introduced thus far permit the user to create and reorganize sets of objects, and to display selected attributes from those objects. With the exception of the discussion of display formats, we have avoided going into a detailed discussion of statement syntax. In addition, we have glossed over some of the details associated with the specification of requirement lists. In this section and in Sec. V, we will deal more completely with these topics. While we will continue the tutorial format, our emphasis will be on completeness rather than illustration.

DOMAINS: THEIR DEFINITION AND USE

The last data type is "domain," and we have not presented any examples of domains thus far. But generally, domains allow the specification of relationships between various symbols. An example unrelated to command and control will be used to illustrate this.

Suppose that we have a file of employees, each with three attributes, as the template below indicates.

```
Template for the set employees (
        ( name,          text    )
        ( specialty,     domain academic_area )
        ( job_level,     domain level )
).
```

Suppose further that the following are valid specialties and job_levels, respectively:

Specialties	Job Levels
mathematician	junior_staff
statistician	senior_staff
info_scientist	project_leader
compu_scientist	program_director
anthropologist	
archaeologist	

In ISIS we can write statements like the following:

```
Show the employees whose
        job_level > senior_staff and
        specialty is related by academic_area to
        antiquity_studies.
```

provided that the academic_area domain and the level domain have been properly specified. Example specifications are given below:

```
Ordered domain level (
        junior_staff,
        senior_staff,
        project_leader,
        program_director
).
```

The ordered domain statement defines the domain "level" to contain the junior_staff, senior_staff, project_leader, and program_director symbols, with junior_staff being "less" than senior_staff, which in turn is a lower level than project_leader, which itself is lower than program_director. The ordered domain statement, therefore, defines the "precedence relationship" between the four members of the domain.

```
Domain academic_area (
        mathematician,
        statistician,
        info_scientist,
        compu_scientist,
        anthropologist,
        archaeologist,
        quantitative_studies,
        computer_studies,
        antiquity_studies,
        information_sciences
) {
        < mathematician,   quantitative_studies >
        < statistician,    quantitative_studies >

        < info_scientist,  computer_studies     >
        < compu_scientist, computer_studies     >

        < anthropologist,  antiquity_studies    >
        < archaeologist,   antiquity_studies    >

        < mathematician,   information_sciences >
        < statistician,    information_sciences >
        < info_scientist,  information_sciences >
        < compu_scientist, information_sciences >
}.
```

The academic_area domain is defined to contain the ten symbols from mathematician to information_sciences. Furthermore, the symbols in this domain are grouped into classes, with mathematician and statistician belonging to the quantitative_studies grouping, info_scientist and compu_scientist belonging to the computer_studies

grouping, anthropologist and archaeologist belonging to the antiquity_studies grouping, and mathematician, statistician, info_scientist, and compu_scientist belonging to the information_sciences grouping. Note that a symbol can belong to more than one grouping, as for example is the case for the mathematician symbol.

Once these domains are defined, statements such as the following can be written:

```
Show the employees whose specialty is related by
        academic_area to information_sciences.

Remove the employees whose
        job_level < senior_staff or
        specialty is not related by academic_area
        to quantitative_studies.
```

The phrases

```
is related by <domain name> to <text string>
```

and

```
is not related by <domain name> to <text string>
```

indicate to ISIS that the specified domain name is to be searched for the indicated text string.

The user should use caution if domains are to be employed. First, the following templates must be defined prior to the domain definitions:

```
Template for the set "domain" (
        ( name,         symbol )
        ( lowest,       symbol )
        ( highest,      symbol )
        ( parent,       symbol )
        ( type,         integer)
        ( relation,     set    )
).

Template for the set relation (
        ( left,         symbol )
        ( right,        symbol )
).
```

These will allow ISIS to define sets that will hold the information contained in the domain statements, and will also allow ISIS, when the loading of objects is taking place, to make sure that all symbols being loaded into domain attributes actually are in the domains.

The domain template allows ISIS to create a domain object for each defined domain, with attribute values indicating, for example, the name of the domain and the location in the ISIS symbol table of the first (lowest) and last (highest) domain entries. Additionally, the relation template permits the creation of objects that indicate the groupings of different domain entries, e.g., those domain entries associated with quantitative studies.

Finally, the domain statements must be processed prior to the loading of any files and the processing of any template and format statements (except the two template statements appearing above). The domain statements in effect place the symbols associated with the domain in the symbol table in the correct order. If files or template statements are processed prior to the domain statements, the necessary ordering may be impossible to establish.

LOGICAL EXPRESSIONS (REQUIREMENT LISTS)

In many of the examples given above, logical expressions were included, as were entries of the form {rqmt list} in the statement syntax sections. These logical expressions or requirement lists provide the ability to select objects possessing certain characteristics from a larger set of objects, e.g., to select those frags possessing certain ordnance requirements, or those scheduled to rendezvous with a given aar sortie.

There are two general ways to enter requirement lists into an ISIS statement, one of which has been illustrated in the examples above. The first is to specify the requirements directly in the form of logical expressions, as in

```
whose tot > ^1200 and aar_time < ^1300
```

or

```
for which there is a rendezvous whose
        call_sign contains anker and
        aar_time >= ^1245
```

The first example applies to the primary object, while the second example refers to the set of owned objects. The expression

```
for which there is a rendezvous whose
```

is the key to getting from the primary object to its owned set.

The following statement will display only those aar_frags scheduled to rendezvous with both f4 and f100 sorties.

```
Show the aar_frags
        for which there are rendezvous whose
                aircraft is f4, and                    <-------
        for which there are rendezvous whose
                aircraft is f100.
```

Note the presence of the comma in the flagged line. That comma is there to instruct ISIS to return to the primary object before applying the next part of the selection criteria. Had the comma not been there ISIS would have tried to move down to a set owned by rendezvous whose name is rendezvous. Since none exists, an error message would have been generated. The comma's presence causes ISIS to return to the aar_frags object, and then the next part of the statement brings ISIS back into the rendezvous set once again. The effect will be that no aar_frag will be displayed unless it has an f4 rendezvous and an f100 rendezvous.

Flagging Objects That Satisfy Selection Criteria

The second way to write logical expressions is a little cumbersome at first, but it provides two rather useful facilities. First of all the user can place in a file selection criteria of arbitrary complexity. Not only does this eliminate the need to type some rather long expressions, but by having the selection criteria in a file, the user can thereby save for future use frequently employed criteria. Libraries of such criteria can thus be created and printed for general reference and critique.

The second facility provided by the more cumbersome method allows for the flagging of objects satisfying portions of the selection criteria. This enables the user to get an intuitive grasp of his data base contents without a detailed look at each object. For example, the user can create the selection criteria file so as to flag aar_frags having f4 rendezvous differently than aar_frags having f100 rendezvous. Specifically, if the objects and subobjects in question are also given a "desirability" attribute, ISIS will set the value of that attribute in accord with instructions contained in the selection criteria file.

The following examples should clarify this. The statement

```
Show the aar_frags for f4_and_f100.
```

assumes the existence of a file named "f4_and_f100" that contains specially structured logical expressions. An example of such a file that would have an effect identical to the display statement on p. 46 is

```
/* file f4_and_f100 */

[ 0:  there is a rendezvous whose aircraft is f4, and
      there is a rendezvous whose aircraft is f100
]
```

Recall that /* and */ denote a comment. The 0 on the first line of the requirement indicates that the requirement is "mandatory." Nonmandatory requirements look like:

```
/* file f4_and_f100 */

[ 0:  there is a rendezvous whose
                aircraft is f4    or
                aircraft is f100;

  1:  there is a rendezvous whose aircraft is f4;

  2:  there is a rendezvous whose aircraft is f100;
]
```

This file has the mandatory requirement (the two lines following the 0:) and two "desirable" requirements. One of the desirable requirements has "desirability" 1, and the other has "desirability" 2.

Note that the mandatory requirement in this file is different from the one in the previous example. The mandatory requirement will select aar_frags having either an f4 or f100 rendezvous, while in the previous example both types of rendezvous are required.

To clarify the use of "desirable" requirements, and to illustrate how to selectively flag objects, we will re-examine the aar_frags and rendezvous templates. Notice that in Table 3 a desirability attribute has been added to each template (marked by *).

Table 3

MODIFIED aar_frags AND rendezvous TEMPLATES

```
Template for the set aar_frags (
   ( unit,          symbol  )
   ( missn_no,      integer )
   ( aircraft,      symbol  )
   ( num_ac,        integer )
   ( call_sign,     symbol  )
   ( aar_area,      symbol  )
   ( altitude,      symbol  )
   ( rendezvous,    set     )
   ( remarks,       text    )
   ( offload_time,  integer )
   ( desirability,  integer )   *
).
```

```
Template for the set rendezvous (
   ( call_sign,     symbol  )
   ( aircraft,      symbol  )
   ( num_ac,        integer )
   ( aar_time,      time    )
   ( time_per_ac,   integer )
   ( desirability,  integer )   *
).
```

If the following statement is executed:

```
Load the aar_frags for
  f4_and_f100 from the
  aar_frag file.
```

then any aar_frags with either an f4 or f100 rendezvous would be loaded, and the desirability attributes of the loaded objects would be set to values dictated by how they satisfy the desirable criteria.

For example, suppose there are three aar_frags, and the first has four f4 rendezvous, the second four f100 rendezvous, and the third two f4 and two f100 rendezvous, as given below:

```
A1
  4 f4 rendezvous

A2
  4 f100 rendezvous

A3
  2 f4   rendezvous
  2 f100 rendezvous
```

Upon loading of these aar_frags under control of the f4_and_f100 file, the desirability attribute of each f4 rendezvous will be assigned the value

```
11000      (10000 + 1000)
```

and the desirability attribute of each f100 rendezvous will be given the value

```
10100      (10000 +  100)
```

The 10000 part comes from satisfying the mandatory criteria in the f4_and_f100 file. The 1000 part comes from satisfying the "desirability 1" criterion. And the 100 part comes from satisfying the "desirability 2" criterion.

The desirability attribute of the aar_frags objects will be assigned the values

```
A1    4000    (1000 + 1000 + 1000 + 1000)
A2     400    ( 100 +  100 +  100 +  100)
A3    2200    (1000 + 1000 +  100 +  100)
```

respectively.

We will first explain where the rendezvous desirability values come from. Any rendezvous that satisfies the mandatory requirement will be set to 10000. If a rendezvous satisfies the desirability 1 requirement, it will have 1000 added to it. If it satisfies the desirability 2 requirement, it will have 100 added to it. Thus the f4 rendezvous satisfy both the mandatory and the desirability 1 criterion, and their desirability attributes are set to 11000 or 10000 + 1000. And the f100 rendezvous satisfy both the mandatory and the desirability 2 criteria, giving their desirability attributes a value of 10100 or 10000 + 100.

The following display should help clarify the discussion. The desirability level in the f4_and_f100 file represents a digit location in the desirability attribute of the affected object, with mandatory criteria (or desirability 0 criterion) holding the first of the five positions, the desirability 1 criterion holding the second position, all the way down to desirability 4 and above holding the fifth position in the attribute value.

```
DESIR
01234

11000   f4 rendezvous

10100   f100 rendezvous
```

The aar_frags desirability value is determined based on how the objects in its rendezvous set satisfy the selection criteria in the f4_and_f100 file. The following display shows the desirability values associated with each rendezvous in the three aar-frags. Recall that the first aar frag has four f4 rendezvous, the second has four f100 rendezvous, and the third has two f4 and two f100 rendezvous.

4 f4 (A1)	4 f100 (A2)	2 f4 & 2 f100 (A3)
11000	10100	11000
11000	10100	11000
11000	10100	10100
11000	10100	10100

The aar_frags desirability values are determined by adding each of the rendezvous values together ignoring the mandatory digit. Thus the A1 desirability value is

```
4000 = 1000 + 1000 + 1000 + 1000
```

and similarly the A2 and A3 desirability values are 400 and 2200, respectively.

The desirability feature is provided in ISIS to allow the user to see at a glance how objects satisfy the selection criteria. The desirable selection criteria need not be the same as the mandatory criteria. In this example, the user could display the aar_frags after they were loaded under control of the f4_and_f100 file, and would see instantly how each compares with the f4 and f100 frags--provided, of course, that the display format used calls out the desirability attribute for display.

The display statement does not result in changed desirability values, but the load, copy, save, move, keep, and remove statements do. In other words, "Show the aar_frags for f4_and_f100." will not cause the setting of desirability values, but statements such as "Move the aar_frags for f4_and_f100 into these_frags." will cause such alteration.

Arithmetic Operations in Logical Expressions

While we have not shown this in the examples, it is possible to have arithmetic expressions in logical expressions, as for example

```
Load the frags whose
        aar_time is not ^0000 and
        tot is not ^0000 and
      ( aar_time - tot < ^0030 or tot - aar_time < ^0030 ).
```

In this case, only those frags having both nonzero aar_times and tots such that the tot is within 30 minutes of the aar_time will be loaded.

Care must be taken when employing arithmetic expressions to avoid the mixing of attribute types. Dates, times, and fulldates can be arithmetically combined but integers should not be combined with dates, times, or fulldates.

Several functions, provided to facilitate the use of mixed attribute types in arithmetic expressions, are listed below:

```
years  ( any date, time or fulldate expression )
months ( any date, time or fulldate expression )
days   ( any date, time or fulldate expression )
hours  ( any date, time or fulldate expression )
minutes( any date, time or fulldate expression )
```

These functions take date, time, or fulldate expressions as input and return integer results. No rounding of the result is done. For example,

```
years ($020918) =   2
months($020918) =   2*12 + 9 = 33
```

Note that in converting dates into days, ISIS assumes that each month has the same number of days in it, namely,

```
365/12 = 30.4167
```

Care must be taken to avoid the conversion of expressions that will result in integers that are larger than the capacity of the PDP11 16 bit word, or 32767. When such conversion takes place the result is unpredictable. This will usually be a problem only when relatively large date constants are converted to integer hours or minutes. The hours and minutes functions probably are most useful when converting times into integer hours or minutes, and we therefore do not expect the overflow problem to be a major one.

MODIFICATION OF ATTRIBUTE VALUES: THE "FOR" STATEMENT

The examples given so far have focused on the selection of objects with specific characteristics, but in only one case has such selection caused an attribute value to be modified--the modification of the desirability attribute to reflect how an object compares with a particular set of selection criteria. We now introduce a new statement, the "for" statement, which permits the display and modification of attribute values.

Command Syntax

```
For <set name> {rqmt list} {action clauses}
        {within <set name> {rqmt list} {action clauses}
                {within <set name> {rqmt list} {action clauses}
                 ...} }
```

where the action clauses can be one or more of the following:

```
show <attribute name>
show <text string>
show <arithmetic expression>

set <attribute name> = <expression>

prompt for <attribute name>
prompt for attributes
```

Recall that "display" and "show" are synonymous.
The three action clause verbs (show, set, and prompt) allow the user to do the following:

show	display a given attribute value or set of attribute values in an unformatted mode
set	alter an attribute value or set of attribute values
prompt	allow user input of attribute values from the terminal

In the syntax, "within" can be thought of as meaning "owned by." Action clauses must be separated by semicolons, and more than one attribute, expression, or text string can appear within a given action clause provided they are separated by commas.

Examples

The following statement will display the call_sign of each of the aar_frags.

```
For aar_frags show call_sign.
```

The results will be

```
call_sign:  tokay 41
call_sign:  tokay 42p
```

The following statement will show the call signs of all frags scheduled to fly from the 49th TFW:

```
For frags whose unit is "49 tfw" show call_sign.
```

and will result in the following display:

```
call_sign:  anker 11
call_sign:  anker 13
call_sign:  anker 21
call_sign:  anker 31
call_sign:  anker 41
call_sign:  anker 51
call_sign:  anker 61
call_sign:  anker 71
call_sign:  anker 81
call_sign:  anker 91
call_sign:  anker 01
```

If we wish to see the call signs of the frags scheduled for air-to-air refueling, we could use the following statement:

```
For rendezvous show call_sign within aar_frags.
```

Note that rendezvous is a set owned by aar_frags. The statement above will display, for each of the aar_frags, the call signs of those rendezvous owned by the aar_frags.

The following statement will display the call signs of those frags whose scheduled aar time is within 30 minutes of the scheduled time over target.

```
For frags whose
        aar_time > ^0000 and
        tot > ^0000 and
        minutes(aar_time - tot) <= 30
        show call_sign.
```

Alternatively, if the aar_time must be no less than 30 minutes after time over target, the following statement will see that this is the case by altering the aar_time attribute.

```
For frags whose
        aar_time > ^0000 and
        tot > ^0000 and
        minutes(aar_time - tot) < 30
        set aar_time = tot + ^0030.
```

Note that the above statement has altered the value of the aar_time attribute. If the aar_time is within 30 minutes of the tot, then ISIS will set the aar_time to be exactly 30 minutes after the tot. Of course, if this statement is executed the user must also make sure that the appropriate aar_frags object is modified to reflect the changed rendezvous time, and that the change will not be in conflict with other rendezvous.

The following statement allows the user to type in directly the new aar_time value.

```
For frags whose
        aar_time > 0000 and
        tot > 0000 and
        minutes(aar_time - tot) < 30
        show "
        The frag whose call sign is
        given below has an aar_time
        that is within 30 minutes of
        its time over target. Please
        enter the desired aar_time.",
        call_sign;
        prompt for aar_time.
```

Several things should be noted about the statement. First of all note that a text string is being displayed that is several lines long. The string begins at the first quotation mark which is on the line containing "show", and ends on the line just above the last one in the display on the right. The text string is followed by a comma, and then the call_sign attribute appears. The effect of this show clause is to place the textual information on the screen followed by the value of the call_sign attribute.

```
show "
The frag whose call sign is
given below has an aar_time
that is within 30 minutes of
its time over target. Please
enter the desired aar_time.",
call_sign;
```

The next point to notice is the prompt at the end of the for statement. This in effect asks the user to enter the desired aar_time value. Notice, too, that the prompt clause is separated from the show clause by a semicolon. Different action clauses in the for phrase must be separated by semicolons, and different attributes or expressions within an action clause must be separated by commas. The above statement will generate the following displays (the arrows are not part of the displays and have been added to point out where the prompts take place).

```
:
The frag whose call sign is
given below has an aar_time
that is within 30 minutes of
its time over target.  Please
enter the desired aar_time.
call_sign: anker 21
aar_time:  12:25 = ?     <----------------------

:
The frag whose call sign is
given below has an aar_time
that is within 30 minutes of
its time over target.  Please
enter the desired aar_time.
call_sign: anker 31
aar_time:  12:40 = ?     <----------------------

:
The frag whose call sign is
given below has an aar_time
that is within 30 minutes of
its time over target.  Please
enter the desired aar_time.
call_sign: anker 41
aar_time:  12:55 = ?     <----------------------

:
The frag whose call sign is
given below has an aar_time
that is within 30 minutes of
its time over target.  Please
enter the desired aar_time.
call_sign: anker 51
aar_time:  13:10 = ?     <----------------------
```

ISIS will wait until the user responds to each prompt before continuing. Several responses are possible (see Table 4 below). First of all, the user can simply enter the desired value, such as 1300. This will replace the current attribute value. If an improper value is provided ISIS will so inform the user and will reprompt.

The user can cause an attribute to be "emptied" by simply responding with a carriage return. This instructs ISIS that no value is desired for this attribute, and ISIS will store an "empty" value.

If the user wishes not to alter an attribute value he simply responds with the <BRK> character. This causes ISIS to leave the current attribute value unaltered. If the user wishes to terminate all prompts within the set he responds with \<BRK>. This will cause ISIS to leave the current set. If the current set is owned by an object in another set, ISIS will return to that level and will process any remaining segments of the for phrase associated with the owning set. If the user wishes to terminate processing for the owning set as well he can simply respond to the owned set's prompt with \\<BRK>. If the prompt is several sets removed from the primary set and the user wishes to terminate execution of the for phrase, he need only type \\\...\\\<BRK> (an arbitrary number of back-slashes followed by the <BRK>) to exit the for phrase.

Table 4

POSSIBLE RESPONSES TO A PROMPT
IN THE FOR STATEMENT

Enter the desired value, e.g., 1300 followed by a carriage return
Respond with a carriage return only, which will cause the attribute to be set to the "empty" value
Respond with a <BRK>, which will cause the current attribute value to be preserved (note the current attribute value is printed at prompt time)
Respond with \<BRK>, which will preserve the current attribute value and terminate execution of the for phrase for the current set. If the prompt is within an owned set, then control will pass to the next object in the owning set. Responding the \\<BRK> will get you past the owning set.

Three Other Arithmetic Functions

Three functions are provided, in addition to the years, months, days, hours, and minutes functions, as listed below:

```
min( any expression )
max( any expression )
avg( any expression )
```

These three functions, different in scope than the five date-integer conversion functions, allow the comparison of expressions among the objects of a set. For example,

```
For frags whose
        aar_time > ^0000 and
        tot      > ^0000
        show min( aar_time - tot ),
             max( aar_time - tot ),
             avg( aar_time - tot ).
```

will display for each frags the minimum, maximum, and average (arithmetic mean) "thus far" of aar_time - tot. Since there are six frags with non-empty aar_time and tot attribute values, six sets of minimum, maximum, and average values will be displayed.

If all six displays are not wanted, but only the last one, then the following can be used:

```
For frags whose
        aar_time > ^0000 and
        tot      > ^0000
        set imin = min( aar_time - tot ),
            imax = max( aar_time - tot ),
            iavg = avg( aar_time - tot );
        show last imin, imax, iavg.
```

In this statement imin, imax, and iavg are defined as "local" variables that exist only for the duration of the statement, and the "last" that appears on the final line of the statement instructs ISIS to display the values of imin, imax, and iavg on only the last of the objects in the frags set.

For statements can be rather cumbersome to write, and in fact their value comes into play more when they are used in perform files. They then can be constructed with the assistance of an online text editor. Additionally, altering attribute values may not always be desirable, since the attribute's current value would thereby be lost. The next subsection will describe how attributes can be added to templates and therefore their objects for the duration of an ISIS session, and further how new objects can be created.

ADDING NEW ATTRIBUTES AND OBJECTS: THE INSERT COMMAND

The insert statement can be used to add a new attribute to a template. For example,

```
Insert the integer attribute called desirability in frags.
```

The syntax of this statement is,

Insert <data type> attribute called <attribute name>
in <template name>.

The statement appends the new attribute on the end of the template. The statement is useful if the user wishes to perform some computations on an object and to save the results temporarily. Care must be taken, if a set that uses the modified template is saved onto a disk file, to make sure that when the set is reloaded at some future point in time the added attribute is present in the template.

The insert statement can also be used to add an object to an existing set. For example,

Insert an object after frags.

A new object will thereby be placed at the end of the frags set. It can be assigned attribute values by the for statement, as, for example,

For frags whose seq is last prompt for attributes.

ISIS will then ask for a value to be provided for each attribute in the new frags.

The syntax of the insert statement is

Insert object after <set name> {rqmt list}.

or

Insert object before <set name> {rqmt list}.

Thus we can say

Insert an object after frags whose seq is 3.

and

Insert an object before frags whose call_sign is "anker 11".

V. THE PERFORM STATEMENT AND ISIS PROGRAMMING

An example of the perform statement was discussed at the beginning of this report when we talked about an easy way to define the data structure for the command and control example. We stated that the data structure and display format definition statements can be placed into a file for later callup during ISIS sessions by using the perform statement (e.g., Perform frag_template.). It is possible to place perform statements in files that themselves are to be performed, and furthermore, the execution of a perform file can be invoked conditionally based on attribute values. This provides a modest ISIS programming capability that has proven to be rather useful. For example, data base update procedures and special computation and display procedures have been implemented. One will be illustrated in this section.

In addition to the perform statement, several other statements particularly useful when programming in ISIS will be described in this section.

THE PERFORM STATEMENT

Statement Syntax

Perform <file name> {while <rqmt list>}.

The while clause in the perform statement allows the execution of a perform file as long as the requirement list is true. The requirement list is "tested" prior to the execution of the perform statement, and if found to be true the file is executed. At the conclusion of the file's execution the requirement list is again tested, and if found to be true the file is executed once again. The file will continue to be executed as long as the requirement list is true.

Note that once control is passed to the perform file, the perform file retains that control until all statements in the perform file are executed. Thus, even if the first statement in the perform file causes the controlling while clause to become false, the perform file will continue in control until all statements are executed. We should also point out that the only way to iteratively execute statements is by using the perform file, i.e., looping can take place only by using perform files. The fact that all statements in a perform file must be executed before the perform file surrenders control has not proven to be a difficulty.

Examples

We will illustrate how the perform statement can be used for ISIS programming by presenting a simple example. This example loads each frag from disk and writes that frag onto one of several files depending on the unit from which the frag is scheduled to fly. The procedure would be invoked by typing

Perform split_frags.*

The split_frags file contains three statements. The first one, the template statement, defines an object that can be used to control the execution of the second file. That is to say, the first statement defines the data structure called frag_file, and frag_file will be used to control the execution of the second perform file. The second statement, the load statement, causes the loading of a frag_file object, and furthermore the status attribute of the object is set to the symbol "open."

```
/* split_frags */

   Template for the set frag_file (
           ( status, symbol )
   ).
   Load frag_file {(open)}.

   Perform frags_to_units while
       status of frag_file is open.
```

Notice the difference between the load statement in the split_frags file and the load statements we have seen thus far. The other load statements contained references to files, as in

Load the frags from the fighter_frag file.

Were we to look at the fighter_frag file, we would find that the file begins with "{" and ends with "}". This is the external representation of the beginning and end of a set. Each object in the set begins with "(" and ends with ")". The items between the opening and closing parentheses represent attribute values associated with

*The files referenced in the example, namely, split_frags and frags_to_units, are not included in the demonstration directory, and therefore the perform statement cannot be executed. This is why it appears in the normal type font.

the object. If the set is small enough, it is possible to load the objects directly, and this is what we have done in the

```
Load frag_file {(open)}.
```

statement. One object is loaded, and the value of its status attribute is set to "open."

The final statement in the split_frags file

```
Perform frags_to_units while status of frag_file is open.
```

invokes the frags_to_units file as long as the status of the frag_file object is open. As we will discuss below, the frags_to_units file will change the status of the frag_file object to "closed" when the last object in the fighter_frag file is processed.

The requirement list in the while clause is slightly different than what has been thus far presented. Not only is the attribute name mentioned (status), but a set name is also provided (frag_file). Specifically, the clause

```
status of frag_file
```

refers to the value of the status attribute in the "first" object in the frag_file set. In the example, only one object will be found in the frag_file set. But even when more than one object appears, the clause

```
<attribute name> of <set name>
```

always refers to the attribute value in the first object of the set. This type of clause can appear in any requirement list, as, for example,

```
Show the frags whose
        unit     is unit     of this_frag or
        aircraft is aircraft of this_frag.
```

In this case, all frags with either the same aircraft or the same unit as the first frag in the this_frag set will be displayed.

We are now ready to examine the frags_to_units file. Recall that the objective is to load each frag from the fighter_frag file and to save it into one of three files, depending on the unit from which the mission is scheduled to fly. This file contains seven ISIS statements. The first loads the "next" frag from the fighter_frag file. Note the difference between this load statement and the previous ones we have encountered. The "next" is the key here. The "load next" statement was provided to facilitate programming with ISIS perform files, and permits the loading of only one object at a time. More will be said about the statement in the next subsection.

```
/* frags_to_units */

    Load the next frags from the
        fighter_frag file.

    For frag_file
        set status = closed.
    For frags set
        status of frag_file = open.

    Append the frags whose
        unit is "49 tfw" into the
        frag_49 file.

    Append the frags whose
       unit is "178 tfgp" into the
        frag_178 file.

    Append the frags whose
        unit is "181 tfgp" into the
        frag_181 file.

    Remove the frags.
```

The next two statements in the file are the ones that provide control of the file's execution. The status of the frag_file object is set to "closed." Then, if there is an object in the frags set, the status of the frag_file object is set back to open. Thus, as long as the "load next" statement actually results in the loading of a frag, i.e., as long as we have not reached the end of the fighter_frag file, the status of the frag_file object will remain "open." Once there are no frags to be loaded from the fighter_frag file, the status of the frag_file object will be changed to "closed," and execution of the governing

```
Perform frags_to_units while status of frag_file is open.
```

statement will be suspended. Note that the "Remove the frags." statement at the end of the frags_to_units file insures that at most one frag will be in storage at a given time. When the second and third statements are executed, the frags set will be empty only when the end of the fighter_frag file is reached.

The next three statements, the "append" statements, save the frag into the appropriate unit file, either frag_49 for the 49th TFW, frag_178 for the 178th TFGP, or frag_181 for the 181st TFGP. The append statement was provided to facilitate ISIS programming. It is equivalent to a save statement with the "append" response. The remove statement deletes the frag since it is no longer needed.

More complicated procedures can be written, procedures that can ask the user for instructions by using the prompt command, that can display results of relatively complex computations, and that can in general allow the user to tailor ISIS to perform specialized tasks.

We have found that the ability to do ISIS programming has allowed us in effect to develop specially designed additional commands tailored for specific application areas. In the command and control application, these files provide for the updating of frags objects, checking for consistency between frags objects and aar_frags objects, to name but two. Appendix B provides some examples of these perform files which can be invoked by the user.

SPECIAL LOAD, APPEND, AND RESET STATEMENTS

Examples of the "load next" and "append" statements have already been introduced. This subsection will provide the detailed syntax associated with the statements. In addition, the "reset" statement will introduced as well.

Statement Syntax

```
Load next <template name> {rqmt list}
        from {encrypted} <file name> file.

Load next <set name> using template <template name>
        {rqmt list} from {encrypted} <file name> file.

Append <set name> {rqmt list}
        into {encrypted} <file name> file.

Reset file called <file name>.
```

Statement Descriptions

The "load next" statement provides for the loading of one object from the file, the next object satisfying the requirement list. If no object satisfies the requirement list, or if no objects are left in the file, the statement has no effect. The encryption key need be given only once, when the first object is read from the file. After that, ISIS keeps track of the key.

The append statement is similar to the save statement, except that no prompting of the user is performed. It is assumed that the objects in the named set are to be added at the end of the named file.

During the execution of the "load next" command, some status information about the referenced file is preserved, and the file is kept open. The reset statement restores a file to its initial unopened state. If a file's status is other than at its initial state, then a load of the file will begin at the point just after the location of the object just loaded by the "load next" statement.

VI. MISCELLANEOUS STATEMENTS

Several statements, very useful in practice, will be described here. They are

```
Display size.
Display sets.
Display formats.
Display format called <format name>.
Display templates.
Display template called <template name>.
Display {encrypted} file called <file name>.
Edit {encrypted} file called <file name>.
Remove file called <file name>.
"%" or getting to UNIX while in ISIS
```

DISPLAY SIZE

ISIS operates on data held in the computer's primary storage. The amount of storage is limited, and it is possible to ask ISIS to perform some function that causes all ISIS storage to be used up. The "Display size." command indicates the "high water mark" or maximum amount of primary storage thus far used during an ISIS session. If all sets are removed from ISIS, and primary storage is relatively empty, the "Display size." command will still show the largest amount of storage thus far used. The maximum storage available to ISIS is about 60,000 units.

Note that when an ISIS session begins, about 28,000 units (bytes) are consumed right away to hold various work tables needed by ISIS. The following are examples of the "Display size." command output before and after execution of "Perform frag_template."

Before "Perform frag_template."

```
Boundary check - 177777

Curr top= 28054.  Symbol count 7/500.
Curr dynamic alloc. Avail:   724. Used:  1324. Num allocs:     9. F# 2.
Elapse time (1/60th secs). User:      1.  Sys:     58.  Tot:     59.
```

After "Perform frag_template."

```
Boundary check - 177777

Curr top= 32150.  Symbol count 46/500.
Curr dynamic alloc. Avail:   774. Used:  5370. Num allocs:    69. F# 4.
Elapse time (1/60th secs). User:     38.  Sys:     38.  Tot:     76.
```

The "boundary check" display is provided to assist in debugging. It should always be 177777. The "current top" indicates the current maximum amount of storage used by ISIS. The "symbol count" indicates the number of symbols thus far placed in the symbol table. Note that in the implementation of ISIS that generated these displays, the symbol table can hold 500 symbols.

The "current dynamic allocation" entries show how much storage ISIS has been called upon to consume over and above the initial 28,054. As can be seen, before the "Perform frag_template." statement, 2048 bytes were consumed (724 + 1324), 1324 of which are in use by ISIS and 724 of which are available for allocation. After the "Perform frag_template." statement, 6144 bytes have been consumed (774 + 5370), with 774 of them still available for allocation. The "number of allocations" and "#F" fields indicate how broken up or "fragmented" the computer storage is, with low numbers indicating less fragmentation. Highly fragmented storage increases the probability of unusable storage, i.e., storage that is broken into fragments too small to be utilized. It is therefore desirable to keep the fragmentation number as low as possible. This can be done by saving the sets being worked on into temporary files, logging out of ISIS, and then logging back in.

The "elapsed time" entries indicate how much computer time has been consumed by ISIS since the last "Display size." statement. As can be seen, it took a little over one second to process the "Perform frag_template." statement, split equally between ISIS (or user) processing and operating system processing.

DISPLAY SETS

The "display sets" command lists the names of each set (but not any subsets owned by objects in the sets) entered into ISIS along with the number of objects currently in each set.

DISPLAY FORMATS

This statement is used to display the names of the formats currently entered into ISIS.

Display format called <format name>.

This statement lists the named format.

DISPLAY TEMPLATES AND DISPLAY TEMPLATE CALLED <TEMPLATE NAME>

These statements, similar to the format statements above, list, respectively, all the templates entered into ISIS and the contents of the indicated template. They were illustrated at the beginning. of this report.

DISPLAY {ENCRYPTED} FILE CALLED <FILE NAME>

This statement can be used to display the contents of a disk file. When the statement is executed, ISIS will tell the user about the options available to him in displaying a file; namely, ISIS can list a certain number of lines, move to a given line, find the occurrence of a character string, or stop the display.

EDIT {ENCRYPTED} FILE CALLED <FILE NAME>

This statement allows the user to use the Rand Editor (ned) from ISIS. Note that, when an encrypted file is edited, the backup file remains unencrypted.

REMOVE FILE CALLED <FILE NAME>

This statement, when executed, deletes the named file from the disk.

%--GETTING TO UNIX WHILE IN ISIS

It is sometimes desirable to perform some UNIX function while in ISIS (e.g., seeing who is logged in, or looking at the names of files by using the "dir" process). To do this, the user types % followed by a carriage return. ISIS will return control to UNIX for the execution of one UNIX command, and then will return to ISIS after the UNIX command is executed. Note that it is permissible to invoke a process that requires interaction with the user (such as write). When the process terminates, control will be transfered back to ISIS.

HARD COPY OUTPUT

If the terminal being used is an Ann Arbor 1902 terminal (one that allows the contents of the screen to be printed on an attached printer), then ISIS has several commands designed to generate hard copy output. Their syntax follows:

Display and print <set name> {rqmt list} {using format <format name>}.

Display and print the {encrypted} file called <file name>.

When displaying sets, care must be taken to make sure that the number of objects displayed at one time (as dictated by the display clause in the format statement) is small enough to be held on one screen. If this is not the case, the hardcopy will be incomplete. When printing a file's contents, ISIS knows how many lines of text can be placed on the screen.

EXTERNAL DATA STRUCTURE

Mention has been made of the "external" representation of the objects on disk. It is now time to look at the external representation. On the right are the templates that define the aar_frags data structure and the data structure of its owned rendezvous set. Below that the contents of the file that holds the aar_frags are also listed (the aar_frag file). Note that the aar_frag file begins with an "{". This signifies the beginning of a set. The beginning of a new object is signified by a "(", and then the attribute values of the aar frag object are listed in the order in which they appear in the template, with succeeding attribute values separated by commas.

Table 5

THE aar_frags AND rendezvous TEMPLATES

```
Template for the set aar_frags  (
   (unit,           symbol  )
   ( missn_no,      integer )
   ( aircraft,      symbol  )
   ( num_ac,        integer )
   ( call_sign,     symbl   )
   ( aar_area,      symbol  )
   ( altitude,      symbol  )
   ( rendezvous,    set     )
   ( remarks,       text    )
   ( offload_time, integer )
).
```

```
Template for the set rendezvous (
   ( call_sign,     symbol  )
   ( aircraft,      symbol  )
   ( num_ac,        integer )
   ( aar_time,      time    )
).
```

Notice that the rendezvous set also begins with an "{", and ends with a "}", with each rendezvous object in the set beginning and ending with a "(" and ")", respectively. The aar_frags objects end with ")", and the aar_frags set ends with "}". Thus all objects are surrounded by parentheses, and all sets are surrounded by braces. Attribute values are separated by commas, and attributes whose values are empty must still have a comma as a place holder (see Table 6).

Table 6

CONTENTS OF THE aar_frag FILE

```
{  /* begin aar_frags set */

   (  /* begin aar_frags object */

      "2 bw",      1,              "kc 135", 1,
      "tokay 41", "blue anchor", "fl 260",

      {  /* beginning of rendezvous set */

         ( "anker 21", "f4", 2, ^1225, 5 )
         ( "anker 31", "f4", 2, ^1240 ,5 )
         ( "anker 41", "f4", 2, ^1255 ,5 )
         ( "anker 51", "f4", 2, ^1310 ,5 )

      }, /* end of rendezvous set */

      "", 40

   )  /* end of aar_frags object */

   (  /* begin aar_frags object */

      "2 bw",      2,              "kc 135", 1,
      "tokay 42p", "blue anchor", "fl 260",

      {  /* beginning of rendezvous set */

         ( "traps 31", "f100", 4, ^1345, 3 )
         ( "traps 41", "f100", 4, ^1400, 3 )

      }, /* end of rendezvous set */

   "drogue sortie.  confirm that tokay
   41 has departed prior to entering
   anchor.  times can slide.  heads up.
   falcon 269", 24

   )  /* end of aar_frags object */

}  /* end of aar_frags set */
```

LOGGING OUT OF ISIS

The stop command is the only way to log out of ISIS. To log out, type

Stop.

VII. ISIS SYSTEM DEVELOPMENT

While the main thrust of this report has been to introduce the ISIS command language, we would like now to describe some characteristics of the application environment in which ISIS was developed and the ISIS design philosophy adopted to deal with those characteristics.

Three characteristics stand out with respect to the application environment. First of all, ISIS was designed for computer-naive users. The user is expected to be quite knowledgable in the application area but with only scant exposure to state-of-the-art interactive computer technology. Second, the user's data environment is assumed to be of modest size, where modest means hundreds rather than hundreds of thousands of records. We feel that many data bases, which people currently access manually, satisfy this characteristic, and that by exploiting the small size of such data bases we can take advantage of interactive computer technology while paying only a modest cost. Third, while we were fairly confident at the outset of the development that we could bring interactive computer technology to bear on modest sized data bases, we were not sure exactly what the "final" set of ISIS functional requirements ought to be. We in fact wanted to develop ISIS in such a way that ISIS itself could assist in identifying those requirements.

COMPUTER-NAIVE USER

Several decisions were made to deal with the computer naivety of the user. First, a command language was developed that was English-like and supportive of the user's intuition. A menu selection approach was considered and rejected as being too restrictive. The English-like command language has undesirable features, such as the need occasionally for excessive wordiness, but the advantages, in our judgment, outweighed the disadvantages.

One advantage is that the user can be "eased" into the computer technology. To do simple tasks, the user does not need to understand the more complex aspects of that technology. Further, as the user becomes familiar with ISIS, that familiarity provides a foundation for the introduction of added complexity. Use of ISIS, in other words, should help the user grow toward and understand the more complex technology.

Another advantage afforded by the English-like command language is that those unfamiliar with the details of ISIS can still get a flavor of what the user is trying to accomplish. For example, ISIS permits the development of files containing "selection criteria," i.e., logical expressions written in "ISIS English." A selection criteria file can be employed, for example, to isolate data records possessing only the requirements specified in the file. Not only does this ease the typing burden on the user, but the selection criteria can also be reviewed and critiqued by others for completeness and accuracy. In this manner, even those unfamiliar with ISIS are able to lend their expertise to problem solution.

An additional advantage is that the selection criteria files can be saved for later use. Those new to the application area can gain insight into how a new problem might be solved by seeing how a previous similar problem was attacked.

A second decision made as a result of the computer naivety of the user dealt with how an ISIS data base should be structured. We wanted a data structure that is relatively straightforward to grasp and deal with, and that additionally is suitably robust in a modest sized data base environment. We chose a structure based on the SIMSCRIPT* "world view," where each data record is an object that possesses a specific set of attributes, and where objects can be organized into sets. This type of data structure was successfully employed in the DOSS system,** which also was developed for computer-naive users. Furthermore, the RITA system*** successfully employs a similar data structure and utilizes an English-like command language. Our past experience and the successful application of the object/attribute/set data structure in the DOSS and RITA systems led to the adoption of this type of data structure in ISIS.

This is not to say that more traditional data base management systems do not employ the same data structuring concepts, however. Most if not all do in that data bases are usually defined in terms of records and fields within records. Extract files can also be generated from such data bases. Our goal, however, was to choose a way to describe the world more in concert with a computer-naive user's intuition.

---

*See P. J. Kiviat, R. Villanueva, and H. M. Markowitz, The SIMSCRIPT II Programming Language, Prentice Hall, Inc., Englewood Cliffs, N.J., 1968.

**See Richard Hillestad, "Decision Oriented Scheduling: A Man-Machine Approach for Improving Resource Allocation in Large Organizations," TIMS Conference, Athens, Greece, July 1977.

***See R. H. Anderson and J. J. Gillogly, Rand Intelligent Terminal Agent (RITA): Design Philosophy, The Rand Corporation, R-1809-ARPA, February 1976. Also see R. H. Anderson et al., The RITA Reference Manual, The Rand Corporation, R-1808-ARPA, September 1977.

MODEST DATA BASE SIZE

The second characteristic of the user environment we want to touch on is the nature of the user's "data base." The data base is assumed to be small and is managed for the most part manually. The modest size affords unique opportunities. First of all, it eliminates the information retrieval problems associated with more traditional data base management systems, i.e., problems associated with organization of the data base on disk for rapid access and retrieval. Hence the "In Storage" decision. ISIS data bases reside on disk as sequential files, and the user must bring into computer storage those objects of interest to him. In the computer's storage the user can select, sort, filter, and otherwise reorganize the data base and structure to suit his particular needs.

In addition, ISIS allows the user to become intimate with the data base in a way that would not be possible were the data base accessible only manually. The ready availability of the data base in ISIS permits the user to consider queries previously thought to be too time consuming to investigate manually. Such queries would allow the user to gain insights previously not possible.

INCOMPLETE FUNCTIONAL REQUIREMENTS

The fact that the nature of the potential user's "final" set of functional requirements was not known at the outset of system development led to the choice of a particular development philosophy. We wanted ISIS to help the user understand the nature of the modest sized data base area. Furthermore, we wanted to minimize the development cost associated with the implementation of ISIS features that turned out to be less useful than originally anticipated.

These requirements led us to employ an evolutionary development approach to ISIS. ISIS was therefore developed in stages, the product of each stage being a usable version of ISIS that could be tried out on real world problems. Experience with each early version helped identify system deficiencies and therefore functional requirements to be implemented in later stages.

It was important for the time interval between successive stages to be small, so that deficiencies could be eliminated in a timely manner. Not only did we want rapid feedback from the user about ISIS deficiencies, but we also wanted to be able to respond to that feedback quickly. Therefore, the system development environment had to be supportive of interactive computer system development. The UNIX time-sharing system* on the PDP11/70 afforded such an

*See D. M. Ritchie and K. Thompson "The UNIX Time-Sharing System," The Bell System Technical Journal, Vol. 57, No. 6, Part 2, July-August 1978.

environment, in that it was designed for interactive system development and use and afforded system development tools that proved particularly well suited to our needs.

We employed the C Programming Language* and the YACC** parser generator. The C Programming Language proved to be easy for project members to learn. In addition, we found that programming and debugging time in C was much less than our past experience led us to expect. Furthermore, the C code was compact and easy to read and modify. The YACC parser generator permitted the timely modification and extension of the ISIS command language by providing an easy to use framework for the definition of that command language. The burden of command language interpretation and translation was thereby lifted from the system developers' shoulders. Use of UNIX in general, and C and YACC in particular, in our judgment significantly reduced system development time and cost.***

*See D. M. Ritchie et al., "The C Programming Language," The Bell System Technical Journal, Vol. 57, No. 6, Part 2, July-August 1978.

**See S. C. Johnson and M. E. Lesk "Language Development Tools," The Bell System Technical Journal, Vol. 57,No. 6, Part 2, July-August 1978.

***For further details see R. C. Gammill and H. J. Shukiar, "An Interactive System For Aiding Management Decision Making," Proceedings of the 1977 National Computer Conference, Dallas, Texas, 13-16 June 1977.

Appendix A

QUICK REFERENCE TO ISIS COMMAND LANGUAGE SYNTAX

The following is a quick reference to ISIS commands for those already familiar with the command language. No description of the statements is provided.

CONVENTIONS

The following conventions apply in this section:

< >	Required in the statement.
{ }	Optional in the statement.
"{" "}" "<" ">"	The character in quotation marks is required as part of the statement syntax. The quotes themselves are not required.

Note that in ISIS commands the words "show" and "display" are synonymous. Additionally, all ISIS statements must end with a period (.).

STATEMENT SYNTAX GUIDE

Append <set name> {rqmt list} into {encrypted} <file name> file.

Copy <template name> {rqmt list} from {encrypted} <file name> file
into {encrypted} <file name> file.

Display {encrypted} file called <file name>.

Display formats.

Display format called <format name>.

Display {and print} <set name> {rqmt list}.

Display sets.

Display size.

Display templates.

Display template called <template name>.

PRECEDING PAGE BLANK

```
Domain <domain name> ( {symbol,} ... {symbol} )
 "{"
        "<" <symbol>, <domain class> ">"
                .
                  .
                    .
 "}".

Edit {encrypted} file called <file name>.

For <set name> {rqmt list} <action clauses>
        {within <set name> {rqmt list} {action clauses}} ...

Format <format name> for set <set name> "{"
        ( display,   <number>       {, formfeed} )
        ( select,    <attribute name> )
        ( heading,   <text string> )
        ( {pretext}, {attribute name}, {field width},
                     {subformat},      {field depth} )
                .
                  .
                    .
"}".

Insert <data type> attribute called <attribute name> in <set name>.

Insert object after <set name> {rqmt list}.

Insert object before <set name> {rqmt list}.

Keep <set name> {rqmt list}.

Load {and {dont} display} <template name> {rqmt list}
        from {encrypted} <file name> file.

Load {and {dont} display} <set name> using template <template name>
        {rqmt list} from {encrypted} <file name> file.

Load {and {dont} show} <template name> {rqmt list}
        "{" (...) ... (...) "}".

Load {and {dont} show} <set name> using template <template name>
        {rqmt list} "{" (...) ... (...) "}".

Load next <template name> {rqmt list}
        from {encrypted} <file name> file.

Load next <set name> using template <template name> {rqmt list}
        from {encrypted} <file name> file.
```

```
Move <set name> {rqmt list} into <set name>.

Ordered domain <domain name> (
        {symbol,} ... {symbol}
).

Perform <file name> { while <rqmt list>}.

Remove <set name> {rqmt list}.

Reset file called <file name>.

Save <set name> {rqmt list} into {encrypted} <file name> file.

Show (see Display).

Sort <set name> by <atb name> {reversed} {alpha}.

Stop.

Template for set <template name> (
        < ( <atb name>, <data type> ) >
                .
                  .
                    .
).

%
```

Appendix B

SOME PERFORM FILE EXAMPLES

This appendix briefly describes seven perform files developed for the tactical command and control project. In addition, listings of four of the files are reproduced as additional examples of ISIS programming. The seven files are described below, and listings of the first four follow.

FILE NAME	FILE DESCRIPTION
frag_startup	Defines all templates and formats for fragging example, and loads the following sets: frags aircraft aar_frags targets
frag_template	Defines all templates and formats for fragging example.
count_frags	Produces summary counts of all frags (fighter frags, that is). Two summaries are produced, one showing frags by unit, and the other showing frags by mission type.
check_aar	Checks for consistency between frags and aar_frags, e.g., if an aar_frag is scheduled to rendezvous with a frag, this file checks to see that both the frag and aar_frag so indicate. When one or the other does not, the user is notified.
distance_check	Compares target distances with aircraft ranges for a specified frag.
target_check	Performs two target checking functions. First, those frags having secondary targets identical to primary targets of other frags are displayed for airspace conflict management purposes. Next those frags having targets that are out of range of the aircraft and having no air to air refueling scheduled are displayed.
update_frag	Permits user updating of a frag.

THE frag_startup FILE

The frag_startup file defines four types of object and provides for the loading of those objects. We have discussed frags and aar_frags but not aircraft and targets. These two sets contain contrived data about the aircraft and targets appearing in the frags set so that perform files can be developed to make sure that the targets are within the range of the aircraft. Additionally, the frag_startup file provides some display formats for the aircraft and targets sets.

This file is more convenient to use than the frag_template file simply because it provides for the loading of all the objects needed as well as the display formats. The frag_startup file performs the frag_template file. The listings of both files follow.

```
/* frag_startup */

        Perform frag_template.

        Load control {()}.

        For control show "

I'm loading the frags.".
        Load the frags from the fighter_frag file.
        For control show "

I'm loading the aar_frags.".
        Load the aar_frags from the aar_frag file.
        For control show "

I'm loading the aircraft.".
        Load the aircraft from the frag_aircraft file.
        For control show "

I'm loading the targets.".
        Load the targets from the frag_target file.
        Remove control.

        Format show_range for set aircraft {
        ( heading, "
AIRCRAFT   RANGE
  TYPE    (MILES)
" )
        ( "
  ",       type, 10, , )
        (, range,  , , )
        }.
```

```
        Format show_distance for set distance {
        ( "
        ", unit,  8, , )
        ( , miles, 5, , )
        }.

        Format target_distance for set targets {
        ( display, 5 )
        ( heading, "
DISTANCE OF TARGETS FROM FLYING UNITS

TARGET
" )
        ("
",          id, , , )
        ("

",          distance, , show_distance, )
        }.
```

```
/* frag_template */

     Template for the set frags (
        ( unit,          symbol  )
        ( missn_no,      integer )
        ( aircraft,      symbol  )
        ( num_ac,        integer )
        ( call_sign,     symbol  )
        ( tot,           time    )
        ( request_no,    text    )
        ( missn_type,    symbol  )
        ( prim_target,   symbol  )
        ( sec_target,    symbol  )
        ( fac_sign,      text    )
        ( ord_load,      symbol  )
        ( iff_sif_comm,  text    )
        ( ecm,           text    )
        ( remarks,       text    )
        ( aar_time,      time    )
        ( tanker_sign,   text    )
        ( aar_alt,       symbol  )
        ( aar_dur,       time    )
     ).

     Template for the set aar_frags (
        ( unit,          symbol  )
        ( missn_no,      integer )
        ( aircraft,      symbol  )
        ( num_ac,        integer )
        ( call_sign,     symbol  )
        ( aar_area,      symbol  )
        ( altitude,      symbol  )
        ( rendezvous,    set     )
        ( remarks,       text    )
        ( offload_time,  integer )
     ).

     Template for the set rendezvous (
        ( call_sign,     symbol  )
        ( aircraft,      symbol  )
        ( num_ac,        integer )
        ( aar_time,      time    )
        ( time_per_ac,   integer )
     ).

     Template for the set control (
        ( status,        symbol  )
     ).
```

```
      Template for the set targets (
         ( id,           symbol  )
         ( distance,     set     )
      ).
      Template for the set distance (
         ( unit,         symbol  )
         ( miles,        integer )
      ).

      Template for the set aircraft (
         ( type,         symbol  )
         ( range,        integer )
      ).

      Format show_frag for the set frags {
         ( display, 6 )
         ( heading, "
FIGHTER FRAGS

          ACRFT  CALL  MISSN     --------TARGETS-------        IFF/SIF/    FAC
UNIT MSN NO TYPE SIGN  TYPE      PRIMARY  SECOND    TOT ORD      COMM  CALL SIGN
" )
         ("
"              , unit          , 5, , )
         (     , missn_no      , 2, , )
         ("  ", num_ac         , 2, , )
         (" " , aircraft       , 5, , )
         (     , call_sign     , 5, , )
         (" " , missn_type     , 7, , )
         ("  ", prim_target    , 8, , )
         (" " , sec_target     , 7, , )
         (" " , tot            , 5, , )
         (" " , ord_load       , 6, , )
         (     , iff_sif_comm  , 8, , )
         (     , fac_sign      , 8, , )
         ("

                  ", remarks      ,60, , )
      }.

      Format show_rendezvous for the set rendezvous {
               ( heading, "
         RENDEZVOUS WITH
```

```
        CALL  AIRCRAFT  AAR   TIME PER
        SIGN  NO  TYPE  TIME  AIRCRAFT
" )
        ( "
        "     , call_sign    , 5, , )
        (     , num_ac       , 3, , )
        ("  ", aircraft     , 4, , )
        ("  ", aar_time     , 5, , )
        (     , time_per_ac , 6, , )
     }.

     Format show_aar for the set aar_frags {
        ( display, 1 )
        ( heading, "
AIR TO AIR REFUELING FRAG

             AIRCRAFT  CALL  AAR          OFFLOAD
UNIT  MISSN  NO  TYPE  SIGN  AREA   ALT   TIME   REMARKS
" )
        ( "
"               , unit         , 4, , )
        ("  "   , missn_no     , 3, , )
        (       , num_ac       , 6, , )
        ("  "   , aircraft     , 4, , )
        ("  "   , call_sign    , 5, , )
        (" "    , aar_area     , 6, , )
        (" "    , altitude     , 3, , )
        (       , offload_time, 7, , )
        ("    " , remarks      ,30, , )
        ("
"               , rendezvous   ,  , show_rendezvous, )
     }.

     Format show_fighter_aar for the set frags {
        ( heading, "
FIGHTER FRAG AAR SCHEDULE

      MISSN  CALL  AIRCRAFT   TANKER     RENDEZVOUS     AAR        AAR
UNIT   NO    SIGN  NO  TYPE  CALL SIGN      TIME      ALTITUDE  DURATION
" )
        ( display, 6 )
        ("

"               , unit        , 4, , )
```

```
        (          , missn_no      5, , )
        ("    "    , call_sign   , 5, , )
        (          , num_ac      , 3, , )
        ("  "      , aircraft    , 4, , )
        ("  "      , tanker_sign, 9, , )
        (          , aar_time    , 9, , )
        ("      "  , aar_alt     , 6, , )
        (          , aar_dur     , 8, , )
    }.

    Format full_fighter_frag for set frags {
        ( display, 1, formfeed )
        ("
MISSION IDENTIFICATION SECTION

                    UNIT:  ", unit         ,    , , )
        ("
         MISSION NUMBER:  ", missn_no     ,    , , )
        ("
              CALL SIGN:  ", call_sign    ,    , , )
        ("
   AIRCRAFT NUMBER/TYPE:  ", num_ac       ,    , , )
        ("/"                           , aircraft    ,    , , )
        ("
           MISSION TYPE:  ", missn_type   ,    , , )
        ("

                  REMARKS:  ", remarks      , 40, , )
        ("

TARGET SPECIFICATION SECTION

                  PRIMARY:  ", prim_target  ,    , , )
        ("
                SECONDARY:  ", sec_target   ,    , , )
        ("

              TIME OVER TARGET:  ", tot          ,    , , )
        ("
                    ORDNANCE:  ", ord_load     ,    , , )
        ("

REQUEST/FAC/IFF SECTION

         REQUEST NUMBER:  ", request_no   ,    , , )
        ("
           FAC CALL SIGN:  ", fac_sign     ,    , , )
        ("
                   IFF/SIF:  ", iff_sif_comm,    , , )
        ("
```

```
AIR TO AIR REFUELING SECTION

        TANKER CALL SIGN: ", tanker_sign ,    , , )
        ("
                ALTITUDE: ", aar_alt      ,    , , )
        ("
         RENDEZVOUS TIME: ", aar_time     ,    , , )
        ("
            AAR DURATION: ", aar_dur      ,    , , )
    }.
```

THE count_frags PERFORM FILE

This perform procedure (it contains a number of perform files) produces summary counts of fighter frags by both unit and mission type.

```
/* count_frags */

     Template for the set ac_counter (
        ( unit,          symbol  )
        ( aircraft,      symbol  )
        ( missn_type,    symbol  )
        ( count,         integer )
     ).

     Format show_ac_frag_count for the set ac_counter {
        ( "
"           , aircraft, 10, , )
        (   , unit    , 10, , )
        (" ", count   ,  3, , )
     }.

     Format show_missn_frag_count for the set ac_counter {
        ( "
"           , missn_type, 15, , )
        (   , aircraft  , 15, , )
        (" ", count     ,  3, , )
     }.

     Load ac_counter {()}.
     Load    ac_frag_set using template control {(not_empty)}.
     Load missn_frag_set using template control {(not_empty)}.
     Load       frag_set using template control {(not_empty)}.

     Move the frags whose aircraft is "" into completed_frags.
     Perform ac_frags while status of frag_set is not_empty.
     For ac_counter show "
".

     Move completed_frags whose aircraft is not "" into frags.
     For frag_set set status = not_empty.
     Perform missn_frags while status of frag_set is not_empty.
     For ac_counter show "
".

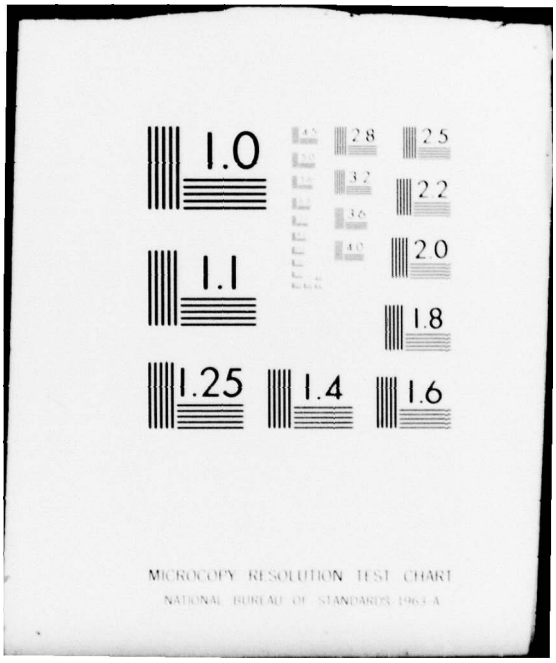
     Remove ac_counter.
     Remove ac_frag_set.
     Remove frag_set.
     Move completed_frags into frags.
```

AD-A071 005 RAND CORP SANTA MONICA CALIF F/G 9/2
AN INTRODUCTION TO THE ISIS INTERACTIVE INFORMATION SYSTEM.(U)
APR 79 H J SHUKIAR, C H BUSH, R C GAMMILL F49620-77-C-0023
UNCLASSIFIED RAND/R-2435-AF NL

2 OF 2
AD A071005



END
DATE FILMED
8-79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

```
/* ac_frags */

     Move the frags whose seq is 1 into ac_frags.
     Move the frags whose aircraft is aircraft of ac_frags
        into ac_frags.

     For ac_counter
        set aircraft = aircraft of ac_frags,
            unit     = "ALL UNITS",
            count    = 0.

     For ac_frags
        set ix                  = count of ac_counter,
            count of ac_counter = ix + num_ac.
     For ac_counter show "
".
     Show ac_counter using format show_ac_frag_count.

     For ac_frag_set set status = not_empty.
     Perform count_ac_units while status of ac_frag_set is not_empty.

     For frag_set set status = empty.
     For frags set status of frag_set = not_empty.




/* count_ac_units */

     Move the ac_frags whose seq is 1 into ac_unit_frags.
     Move the ac_frags whose unit is unit of ac_unit_frags
        into ac_unit_frags.

     For ac_counter
        set unit     = unit of ac_unit_frags,
            aircraft = "",
            count    = 0.

     For ac_unit_frags
        set ix                  = count of ac_counter,
            count of ac_counter = ix + num_ac.
     Show ac_counter using format show_ac_frag_count.

     Move ac_unit_frags into completed_frags.
     For ac_frag_set set status = empty.
     For ac_frags set status of ac_frag_set = not_empty.
```

```
/* missn_frags */

     Move the frags whose seq is 1 into missn_frags.
     Move the frags whose missn_type is missn_type of missn_frags
        into missn_frags.

     For ac_counter
        set missn_type = missn_type of missn_frags,
            aircraft   = "ALL AIRCRAFT",
            count      = 0.

     For missn_frags
        set ix                    = count of ac_counter,
            count of ac_counter = ix + num_ac.
     For ac_counter show "
".
     Show ac_counter using format show_missn_frag_count.

     For missn_frag_set set status = not_empty.
     Perform count_ac_missns while status of missn_frag_set is not_empty.

     For frag_set set status = empty.
     For frags set status of frag_set = not_empty.



/* count_ac_missns */

     Move the missn_frags whose seq is 1 into missn_ac_frags.
     Move the missn_frags whose aircraft is aircraft of missn_ac_frags
        into missn_ac_frags.

     For ac_counter
        set missn_type = "",
            aircraft = aircraft of missn_ac_frags,
            count    = 0.

     For missn_ac_frags
        set ix                    = count of ac_counter,
            count of ac_counter = ix + num_ac.
     Show ac_counter using format show_missn_frag_count.

     Move missn_ac_frags into completed_frags.
     For missn_frag_set set status = empty.
     For missn_frags set status of missn_frag_set = not_empty.
```

THE check_aar FILE

This perform procedure compares the frags with the aar_frags, making sure that rendezvous times shown in the frags set are the same as the rendezvous times given the aar_frags set.

```
/* check_aar */

        Load rendezvous_set     using template control {(not_empty)}.
        Load aar_frag_set       using template control {(not_empty)}.
        Load got_a_fighter_frag using template control {()}.
        Load process_state      using template control {()}.

        Perform this_aar_frag while status of aar_frag_set is not_empty.

        Move the frags whose tanker_sign is not "" into these_frags.
        For these_frags whose seq is 1 show "

    The following fighter sorties include aar, but no aar frags have
been provided
".
        Show these_frags using format show_fighter_aar.
        For these_frags whose seq is 1 show "
".

        Move completed_fighter_frags into frags.
        Move these_frags into frags.
        Move completed_aar_frags into aar_frags.

        Remove rendezvous_set.
        Remove aar_frag_set.
        Remove got_a_fighter_frag.
        Remove process_state.



/* this_aar_frag */

        Move the aar_frags whose seq is 1 into this_aar_frag.
        Move the rendezvous within this_aar_frag into rendezvous.
        Sort rendezvous by aar_time reversed.

        For rendezvous_set set status = not_empty.
        Perform this_aar_check while status of rendezvous_set is not_empty.

        Move completed_rendezvous into rendezvous within this_aar_frag.
        Move this_aar_frag into completed_aar_frags.

        For aar_frag_set set status = empty.
        For aar_frags set status of aar_frag_set = not_empty.
```

```
/* this_aar_check */

/* debug    For process_state show "*** debug  enter this_aar_check".   */


        Move frags whose call_sign is call_sign of rendezvous into
                this_fighter_frag.
        Move the rendezvous whose seq is 1 into this_rendezvous.

        For got_a_fighter_frag set status = no.
        For this_fighter_frag set status of got_a_fighter_frag = yes.

        Perform no_fighter_frag while
                status of got_a_fighter_frag is no.

        Perform check_rendezvous while
                status of got_a_fighter_frag is yes.

/* debug    for process_state show "*** debug  return from check_rendezvous".
*/
        Move this_rendezvous into completed_rendezvous.
        Move this_fighter_frag into completed_fighter_frags.

        For rendezvous_set set status = empty.
        For rendezvous_set status of rendezvous_set = not_empty.

/* debug    For process_state show "*** debug  leave this_aar_check".   */
```

```
/* check_rendezvous */

        For this_rendezvous whose aar_time is not aar_time of
                this_fighter_frag show "

    The fighter sortie and aar sortie listed below have inconsistent
rendezvous times.".

        Show this_fighter_frag whose aar_time of this_rendezvous is not
                                        aar_time of this_fighter_frag
                using format show_fighter_aar.

        For got_a_fighter_frag whose aar_time of this_rendezvous is not
                                        aar_time of this_fighter_frag
                show "
".

        Show this_aar_frag whose aar_time of this_rendezvous is not
                                aar_time of this_fighter_frag.

        Show this_rendezvous whose aar_time of this_rendezvous is not
                                  aar_time of this_fighter_frag.

        For got_a_fighter_frag set status = done.

        For got_a_fighter_frag whose aar_time of this_rendezvous is not
                                        aar_time of this_fighter_frag
                show "


";
                prompt for status.



/* no_fighter_frag */

        For this_aar_frag show "

The aar sortie whose call sign is
",              call_sign, "
has a rendezvous scheduled with a fighter sortie whose call sign is
",
                call_sign of this_rendezvous, "
but I can't find a frag for the fighter sortie";
                set status of got_a_fighter_frag = done.
```

Appendix C

INDEX TO STATEMENT DESCRIPTIONS

APPEND STATEMENT 63
ATTRIBUTES 2, 8
ATTRIBUTE MODIFICATION (FOR STATEMENT) 51
AVG FUNCTION 56
COPY STATEMENT 28
DATA STRUCTURE 39
DATA STRUCTURE (EXTERNAL) 68
DATA TYPE SORTING CONVENTIONS 21
DATA TYPES 39
DATE DATA TYPE 39
DAY FUNCTION 51
DESIRABILITY ATTRIBUTE 48
DISPLAY AND PRINT STATEMENT 67
DISPLAY <ATB NAME> 51
DISPLAY FILE STATEMENT 67
DISPLAY FORMATS 4, 29, 66
DISPLAY FORMAT CALLED <FORMAT NAME> 66
DISPLAY FORMAT STATEMENT 66
DISPLAY SETS STATEMENT 66
DISPLAY SIZE STATEMENT 65
DISPLAY STATEMENT 13
DISPLAY TEMPLATE CALLED <TEMPLATE NAME> 8, 67
DISPLAY TEMPLATES STATEMENT 7, 67
DOMAIN DATA TYPE 39, 42
EDIT FILE STATEMENT 34, 67
EXTERNAL DATA STRUCTURE 68
FORMATS 4, 29
FOR STATEMENT 51
FULLDATE DATA TYPE 39
FUNCTIONS FOR DATE AND TIME CONVERSION 51
FUNCTIONS OVER OBJECTS 56
HARD COPY OUTPUT 67
HOUR FUNCTION 51
INSERT STATEMENT 57
INTEGER DATA TYPE 39
ISIS PROGRAMMING (PERFORM FILES) 59, 78
KEEP STATEMENT 23
LOAD NEXT STATEMENT 63
LOAD STATEMENT 11, 60
LOGGING OFF (STOP STATEMENT) 69
LOGICAL EXPRESSIONS 45
MAX FUNCTION 56
MIN FUNCTION 56
MINUTE FUNCTION 51

MONTH FUNCTION 51
MOVE STATEMENT 26
OBJECTS 2
PERFORM STATEMENT 7, 34, 59
PROGRAMMING (PERFORM FILES) 59, 78
PROMPT FOR <ATB NAME> 52
REMOVE FILE STATEMENT 67
REMOVE STATEMENT 25
REQUIREMENT LISTS 45
RESET STATEMENT 63
SAVE STATEMENT 27
SELECTION CRITERIA 45
SEQ (SEQUENCE) ATTRIBUTE 24, 58
SET <ATB NAME> (CHANGING ATB VALUES) 51
SET DATA TYPE 10, 39
SETS 2
SHOW AND PRINT STATEMENT 67
SHOW <ATB NAME> 51
SHOW FILE STATEMENT 67
SHOW FORMAT CALLED <FORMAT NAME> 66
SHOW FORMATS STATEMENT 66
SHOW SETS STATEMENT 66
SHOW SIZE STATEMENT 65
SHOW STATEMENT 13
SHOW TEMPLATE CALLED <FILE NAME> 8, 67
SHOW TEMPLATES STATEMENT 7, 67
SORT STATEMENT 21
SORTING CONVENTIONS FOR DATA TYPES 21
STOP STATEMENT 14, 69
SYMBOL DATA TYPE 39
TEMPLATES--GETTING STARTED 2, 7
TEMPLATES--DATA TYPES 39
TEXT DATA TYPE 39
TIME DATA TYPE 39
YEAR FUNCTION 51
%--GETTING TO UNIX WHILE IN ISIS 67