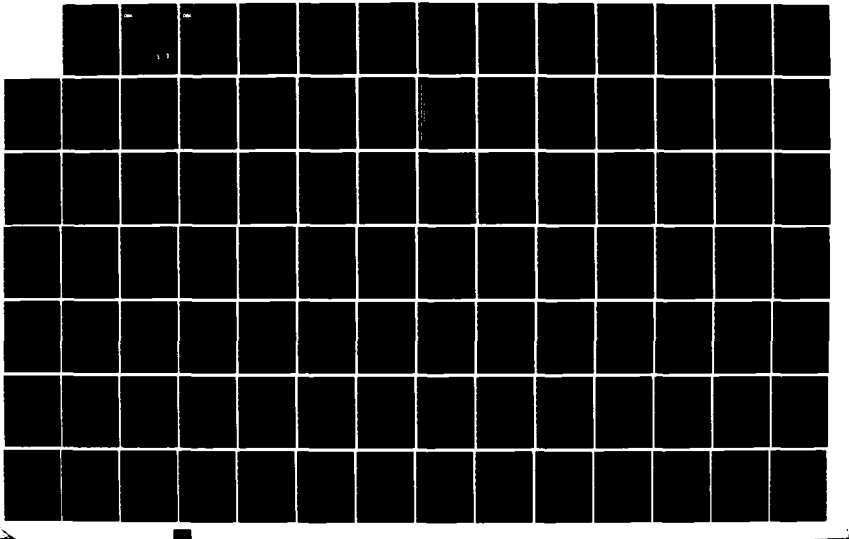
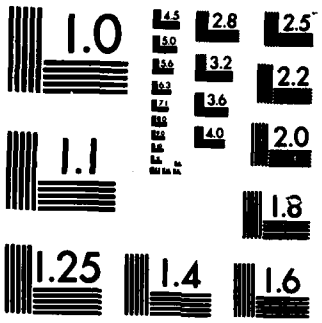
AD-A150 422 ALIAS (ACQUISITION AND LOGISTICS INFORMATION AND ANALYSIS SYSTEM) MAINTEN.. (U) DECISION-SCIENCE APPLICATIONS INC ARLINGTON VA M S CAREY ET AL. 1/7

UNCLASSIFIED 31 OCT 84 DSA-593-VOL-1 N00014-82-C-0813 F/G 15/5 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(13)

DSA DECISION-SCIENCE APPLICATIONS

DSA Report #593

October 31, 1984

AD-A150 422

ALIAS MAINTENANCE AND EXPANSION GUIDE

VOLUME I

DTIC ELECTE FEB 21 1985 S D E

Submitted to:

Scientific Officer
Naval Center for Acquisition Research
NAVMAT 08
Washington, D.C. 20360

Attention: Dr. Thomas C. Varley

This document has been approved for public release and sale; its distribution is unlimited.

85 02 20 002

Suite 1000, 1901 North Moore Street, Arlington, Virginia 22209 (703) 243-2500

DSA DECISION-SCIENCE APPLICATIONS

DSA Report #593

October 31, 1984

ALIAS MAINTENANCE AND EXPANSION GUIDE

VOLUME I

M.S. CAREY
J.C. KRUPP
J.M. KABAT

Accession For
NTIS GRA&I [X]
DTIC TAB [ ]
Unannounced [ ]
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

Distribution unlimited per Dr. Thomas C. Varley, Navy Office for Acquisition Research, ATTN: NAVMAT-08, Washington, DC 20360

Submitted to:

Scientific Officer
Naval Center for Acquisition Research
NAVMAT 08
Washington, D.C. 20360

Attention: Dr. Thomas C. Varley

DTIC COPY INSPECTED

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Direct (Naval Sea Systems Command(SEAOOD)) Washington, D.C. 20362
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) DSA Report No. 593	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Decision-Science Applications, Inc. 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Navy Office for Acquisition Research
6c. ADDRESS (City, State and ZIP Code) 1901 North Moore Street, Suite 1000 Arlington, Virginia 22209	7b. ADDRESS (City, State and ZIP Code) NAVMAT 08 Washington D.C. 20362
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Office of Naval Research 8b. OFFICE SYMBOL (If applicable) ONR	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-82-C-0813
8c. ADDRESS (City, State and ZIP Code) Department of the Navy 800 North Quincy Street Arlington, Virginia 22217	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO.: N68342 PROJECT NO.: 98077Q TASK NO.: WORK UNIT NO.:
11. TITLE (Include Security Classification) ALIAS Maintenance and Expansion Guide, Volumes I and II	
12. PERSONAL AUTHOR(S) M. Carey, J. Krupp, J. Kabat	
13a. TYPE OF REPORT Technical 13b. TIME COVERED FROM 10/82 TO 8/84	14. DATE OF REPORT (Yr., Mo., Day) 31 October 1984 15. PAGE COUNT
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES		
FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
ALIAS, DATA BASE, DBMS, MODELS, STRUCTURED PROGRAMMING.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This documentation explains the structure of the Acquisition and Logistics Information and Analysis System (ALIAS). With this documentation, the experienced programmer should be able to easily maintain and expand the ALIAS system. In addition, the manuals explain all standards to which ALIAS extensions should conform. For the non-programmer these manuals describe the philosophy of ALIAS and its extent and limitations. This volume contains a table of keywords included.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☒ OTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Joseph C. Krupp	22b. TELEPHONE NUMBER (Include Area Code) 703-243-2500 22c. OFFICE SYMBOL DSA

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED

TABLE OF CONTENTS

VOLUME I

Section		Page
	LIST OF TABLES	iii
	LIST OF FIGURES	iv
1.0	INTRODUCTION	1-1
	1.1 Description of this Manual	1-1
	1.2 Purpose of ALIAS	1-4
	1.3 Overview of the ALIAS System Structure	1-7
	1.4 Overview of the Principles Guiding the ALIAS System Design and Implementation	1-27
2.0	GOALS, PRINCIPLES AND STANDARDS FOR ALIAS SOFTWARE	2-1
	2.1 System Goals and Requirements	2-1
	2.2 Design Principles and Standards	2-3
	2.3 Implementation Methods and Standards	2-7
3.0	ALIAS DATA STRUCTURES AND DATA FLOW	3-1
	3.1 Conceptual Data Structures and Their ALIAS Implementations and Usage	3-1
	3.2 Summary of Data Structure Usage and System Data Flow	3-24
4.0	SALIENT FEATURES OF THE HEWLETT-PACKARD 3000 COMPUTER	4-1
	4.1 HP 3000 and MPE Operating System Architecture and Highlights	4-1
	4.2 Methods of Coding With a Limited Address Space	4-6
	4.3 Fortran Compiler	4-7
	4.4 Use of Intrinsics	4-8
	4.5 Terminal Restrictions	4-9
5.0	SUPPORTING SOFTWARE PACKAGES	5-1
	5.1 The Relate Family of Software	5-1
	5.2 HP Draw/Decision Support Graphics	5-9

TABLE OF CONTENTS

VOLUME I (Continued)

Section		Page
6.0	THE ALIAS ENVIRONMENT	6-1
	6.1 Tools Available for Constructing an Environment	6-1
	6.2 Summary of File Structure and System Naming Conventions	6-3
	6.3 Description of User Environment	6-9
	6.4 Description of Development Environment	6-29
	6.5 Communications: Notices, Mail, Messages and Memos	6-57
	6.6 Maintaining and Expanding the Environment	6-58
	6.7 Narrative Description of ALIAS Software Development Methodology	6-61
7.0	ALIAS SECURITY	7-1
	7.1 ALIAS Resources Requiring Security Protection	7-1
	7.2 Threats to ALIAS Security	7-2
	7.3 Generic Security Tools Available	7-3
	7.4 Implemented ALIAS Security Protection Schemes	7-4
	7.5 Weaknesses of Current ALIAS Protection Schemes	7-8
	7.6 Procedures for Managing System Security	7-9
8.0	THE SYSTEM CORE	8-1
	8.1 The Data Base	8-1
	8.2 The Command System	8-3
	8.3 The Scenario System	8-113
	8.4 The Data Base Updating System	8-168
9.0	ADDING NEW MODULES TO ALIAS	9-1
	9.1 Developing the Module	9-1
	9.2 Hooking onto the System Core	9-14

LIST OF TABLES

Table Number		Page
3-1	Standard Files Used by ALIAS	3-9
3-2	Special Data Structure Usage by Major System Component	3-25
6-1	SEA90 Account File Groups by Category	6-4
6-2	General-Interest UDCs by Functional Category	6-12
6-3	General-Interest UDC Descriptions	6-13
6-4	Group Catalog Listing for SEA90 Account	6-30
6-5	Software Development UDCs by Functional Category	6-35
6-6	Software Development UDC Descriptions	6-36
6-7	Editor Macros and Related Procedures by Mode of Execution	6-45
6-8	Summary of Methods of Fortran Compilation by Circumstance	6-53
8-1	MNUR Routines	8-9
8-2	MNUR Include Files	8-13
8-3	Files and Relations Used by MNUR	8-17
8-4	Files Created by Each MNUG Run	8-26
8-5	Program MNUG Routines	8-27
8-6	MNUG Include Files	8-30
8-7	Command System Code Files	8-112
8-8	Scenario System Routines in User Service Modules	8-129
8-9	Include Files Used by the Scenario System	8-132
8-10	Scenario System Code Files	8-167
8-11	Fortran Routines Called by the DBU	8-180
8-12	Data Dictionary Relations and Their Usage by the DBU	8-184
8-13	DBU Global Variables	8-188
8-14	DBU Global Variables Definitions	8-191
8-15	Alphabetical List of DBU Screens	8-204
8-16	Annotated List of DBU Screens by Subsystem	8-207
8-17	VERIFYD Special Variables	8-356
9-1	Format of Parameter Menu Definition Section	9-22
9-2	Structure of Parameter Menu Definition Section	9-23
9-3	Format of Choice Menu Definition Section	9-25

LIST OF FIGURES

Figure Number		Page
1-1	Basic ALIAS System Structure	1-8
1-2	High-Level Examples of ALIAS Execution	1-9
1-3	ALIAS System Structure	1-11
1-4	Data Base Structure	1-14
1-5	Sample ALIAS Menus	1-16
1-6	Sample Map of ALIAS System Menus	1-17
1-7	Three Data Relations Containing Two Scenarios	1-20
1-8	Overall ALIAS System Structure	1-32
3-1	Three Sample Relations	3-2
3-2	Hierarchies and Networks	3-5
3-3	Sample Network	3-6
3-4a.	A Record	3-19
3-4b.	Common Block Implementation of a Record	3-19
3-4c.	Equivalenced Implementation of a Record	3-19
3-5	ALIAS Data Flow	3-26
6-1	Text of UDC1.DSA	6-19
6-2	ALIAS Subroutine Abstract	6-48
8-1	Structure of the ALIAS System Core	8-2
8-2	Overall Data Base Structure	8-2
8-3	ALIAS Common D System Components	8-7
8-4	Program MNUR Calling Tree Diagram	8-8
8-5	Program MNUG Calling Tree Diagram	8-25
8-6	UDCs Relevant to Command System/Core	8-42
8-7	Organization of Inner-Scenario Data Sharing	8-115
8-8	Scenlst and Scendsc Structure	8-118
8-9	Sample Scenlst and Scendsc Contents	8-119
8-10	Structure of the Filinfo.db Relation	8-120
8-11	Sample Contents of the Filinfo.db Relation	8-121
8-12	Structure of the Snusers.sysrw Relation	8-123
8-13	The /scenar/ Include File	8-124
8-14	Flow of SNSTRT Execution	8-125

LIST OF FIGURES (Continued)

Figure Number		Page
8-15	Scenario System - DBIF Interface Routines	8-127
8-16	SNPICK Calling Tree Diagram	8-134
8-17	SNMAKE Calling Tree Diagram	8-136
8-18	SNDEL Calling Tree Diagram	8-139
8-19	SNMDFY Calling Tree Diagram	8-141
8-20	Basic Structure of the DBU	8-171
8-21	DBU Initialization Code	8-213
8-22	Subroutine Curini	8-225
8-23	Screen GETFIL	8-227
8-24	Subroutine Curswp	8-228
8-25	Screen OPNFIL	8-231
8-26	Text of Data Screen Template	8-234
8-27	Text of CURR_NC_SKED Data Screen	8-237
8-28	Text of Comment Screen Template	8-250
8-29	Text of CURR_NC_SKED C Comment Screen	8-252
8-30	Text of Help Screen Template	8-257
8-31	Text of C_NC_SKED_HELP Help Screen	8-258
8-32	Text of Menu Screen Template	8-261
8-33	Text of SKED_MENU Menu Screen	8-263
8-34	CLASS_CHARS Data Screen	8-267
8-35	NC_JOB_TYPES Data Screen	8-273
8-36	RE_JOB_TYPES Data Screen	8-278
8-37	PROJ_NC_SKED Data Screen	8-285
8-38	Calcdate FORTRAN Procedure Abstract	8-291
8-39	PROC_MENU_CMD Subroutine Screen	8-294
8-40	PROC_DATA_CMDA Subroutine Screen	8-297
8-41	PROC_DATA_CMDB Subroutine Screen	8-301
8-42	PROC_COMT_CMD Subroutine Screen	8-316
8-43	PROC_HELP_CMD Subroutine Screen	8-322
8-44	PROC_REPT_CMD Subroutine Screen	8-323
8-45	COMT_INIT Subroutine Screen	8-328
8-46	COMT_UPDATE Subroutine Screen	8-331

Figure Number		Page
8-47	SET_MORE Subroutine Screen	8-333
8-48	INS_COMT_LINE Subroutine Screen	8-334
8-49	DEL_COMT_LINE Subroutine Screen	8-335
8-50	LEGALS_INIT Subroutine Screen	8-337
8-51	NEXT_LEGAL Subroutine Screen	8-339
8-52	PREV_LEGAL Subroutine Screen	8-340
8-53	CHECK_LEGAL Subroutine Screen	8-342
8-54	VERIFYJ Subroutine Screen	8-343
8-55	VERIFYU Subroutine Screen	8-350
8-56	VERIFYD Subroutine Screen	8-351
8-57	SECURITY_CHECK Subroutine Screen	8-363
8-58	SHOWPRIV Subroutine Screen	8-365
8-59	REPORT Subroutine Screen	8-368
8-60	FIELDHELP Subroutine Screen	8-371
8-61	SUSPEND Subroutine Screen	8-376
8-62	SET_INSPECT MODE Subroutine Screen	8-377
9-1	MALIAS.DBA	9-26
9-2	Dual-Version ALIAS Architecture	9-46

1.0 INTRODUCTION

1.1 DESCRIPTION OF THIS MANUAL

This ALIAS Maintenance and Expansion Guide describes how the Acquisition and Logistics Information and Analysis System (ALIAS) works, how to maintain it in an operational state, and how to expand and modify it.

1.1.1 Audience

The manual is intended to serve the needs of three groups: those responsible for maintaining ALIAS in an operational state from day to day, those who are tasked to expand the functions of ALIAS, and those who wish a deeper understanding of the operation of the system than is given in the ALIAS User's Guide.

It is presumed that readers of the manual are familiar with the HP 3000 computer system, with the FORTRAN programming language, with the use of relational data base management systems in general and the RELATE family of software in particular, and with the use of the BUILDER screen application programming language. There is some discussion of each of these subjects here, particularly in Sections 4 and 5, but it is meant to familiarize the reader with subtle or advanced features. It is possible to read and profit from many sections of the manual with little or no understanding of these subjects, particularly the discussion of principles and standards, but an attempt to perform maintenance or expansion tasks without a good grounding in the fundamentals is not recommended.

1.1.2 Organization

The manual is divided into several sections, with the content of the sections proceeding from the general to the specific. The remainder of this Introduction describes the purpose, overall structure, and design philosophy of ALIAS. Section 2 follows with a more detailed discussion of the design goals of ALIAS, of the principles and standards that were

developed from these goals, and of the implementation methods that have been used.

Section 3 describes the overall ALIAS data structure and data flow in somewhat more detail, though still at a general level. ALIAS is a highly 'data-driven' system and makes use of a rich array of data structures; some understanding of these is necessary to understanding of the following sections.

A brief discussion of salient features of the Hewlett-Packard (HP) 3000 computer system and the various software packages used by ALIAS follows in Sections 4 and 5. This discussion is not meant as an introduction to these subjects; a fair amount of familiarity is assumed.

The ALIAS operating environment is described in Section 6 in some detail, including both user support features and software development support features. The methods used to implement, expand, and maintain this environment are also described. System security occupies Section 7, including security requirements, resources available, and the procedures now in place.

This leads to a discussion of the heart of ALIAS, the System Core, in Section 8. Each piece of the core is discussed in detail.

Section 9 describes how to 'hang' new functions onto the System Core, including 'cookbook' procedures for the modification of the menus the command system displays.

Section 10 describes the routines in ALIAS' various utility and interface libraries.

Detailed programmer-level descriptions of each existing application and analytical module follow in the sections beginning with 11. These are meant to provide an understanding

of the structure and operation of the modules to those who wish to modify or upgrade them, and include discussions of interfaces and program logic.

A number of appendices support the main document. Of particular interest are a special table of contents which references sections according to the documentation standard (Appendix A) and a listing of ALIAS' most important host system dependencies (Appendix B). Appendix C, under separate cover, contains program listings.

1.1.3 Related Publications

It is recommended that those reading the ALIAS Maintenance and Expansion Guide as an introduction either to the conduct of ALIAS system maintenance or to the development of new ALIAS modules have already read the ALIAS User's Guide and made some use of the system. Many of the concepts underlying ALIAS are subtle and difficult to describe from a purely theoretical point of view: some operational experience is essential. Likewise, it is recommended that at least a cursory inspection of the ALIAS Data Base Reference Manual have been made, since the data base is in many ways the heart of the system. It will probably be necessary to consult that manual while reading this one.

In addition to these ALIAS-specific publications, the reader is likely to need all of the reference manuals for the RELATE family of DBMS software (RELATE, CREATE, GRAF, MENU, and BUILDER), and for the HP 3000 computer system (FORTRAN, MPE Commands, and System Intrinsics in particular).

1.1.4 Documentation Standard

This manual, and the other ALIAS Guides, pay attention to, but do not strictly follow the DoD Automated Data Systems Documentation Standards (7935.1-S, September 1977). The set of

manuals exceed the standard, and a reference to one or more sections in the manuals is made for each required section in the standard in Appendix B. DSA felt that strict adherence to the standard would result in manuals with an inappropriate organization and with insufficient information to support the Navy's needs. Readers who wish to read in the order of the standard may detach Appendix B and use it as a guide to appropriate sections.

Note that portions of the standard not provided for in this set of manuals include the Functional Description, (see the ALIAS System Decision Paper), Computer Operations Manual (not relevant), Test Plan, and Test Analysis Report. Strictly speaking, the Data Requirements Document is also not provided for, although in practice the ALIAS Data Base Reference Guide performs this function as well as that of a Data Base Specification. Note that there are in effect several System/Subsystem Specifications and Program Specifications, since ALIAS is a modular system.

1.2 PURPOSE OF ALIAS

ALIAS is an Analyst Information System (AIS) designed to support Navy acquisition management. At this time it focuses on overall ship acquisition program monitoring and on long-range shipbuilding program planning and feasibility analysis. Its primary users are NAVSEA 90 (Deputy Commander for Acquisition) and the Navy Shipbuilding Support Office (NAVSHIPSO).

ALIAS is a workbench and toolbox for program analysts, engineers, and managers. An ALIAS user may draw on an extensive data base describing acquisition programs, U.S. shipbuilding industry capacity and capability, and fleet requirements and statuses. He may employ one or more analytical 'modules' to make projections, analyze alternatives, or produce reports. ALIAS aims to take care of data retrieval, manipulation, and routine analysis tasks (which are quite time consuming for the unassisted human), thus enabling analysts to concentrate on the substantive issues at hand.

More specific functions which ALIAS supports, or for which ALIAS support is planned, include:

- Analysis of proposed shipbuilding program feasibility, both short and long-run, with an emphasis on resource requirements and availability
- Analysis of the schedule realism of proposed programs
- Monitoring of current shipbuilding programs for signs of trouble; risk analysis
- Production of monitoring reports and documents
- Monitoring and projection of capacity and capability trends for both the shipbuilding industry and its supporting industries
- Economic impact analysis, e.g., analysis of the regional economic impact of SCN spending
- Business planning analysis, i.e., the development of strategies which the Navy can use in its contractual and business relations with contractors, particularly shipyards and major weapon system manufacturers, to reduce program costs and risks
- Analysis of the lead times required for proposed programs and the adequacy of the lead times in proposed schedules
- Estimation of supporting industry capacity usage requirements for a large number of materials and components required in Navy shipbuilding, and identification of likely shortfalls
- Analysis of the shipyard labor requirements of alternative shipbuilding programs at various levels of detail (overall shipyard, by skill within shipyards, regionally, total shipbuilding industry, etc.)

- Estimation of yearly SCN appropriation requirements implied by alternative programs
- Analysis of the size and composition of the fleet which would result from alternative programs
- Estimation of shipyard facility capacity usage and identification of likely shortfalls
- Production of acquisition program summaries and descriptions at various levels of detail, ranging from single-page summaries of 15-year projections of SCN spending by ship class, to yard-by-yard, ship-by-ship schedules
- Automated generation of trial shipbuilding programs to support sensitivity analyses

The number of functions which ALIAS can serve is limited only by the capabilities of the hardware system it resides on, making it likely that this list of functions will grow as time goes on.

1.3 OVERVIEW OF THE ALIAS SYSTEM STRUCTURE

1.3.1 Basic Structure

As an Analyst Information System (AIS), ALIAS differs from Management Information Systems (MIS) in that analytical and modeling capabilities are present in addition to more traditional information retrieval and manipulation functions. Performance of many ALIAS functions demands a high degree of technical expertise on the part of the user in the areas of analysis that the functions support.

Because of the wide variety of users it serves and functions it performs, and in order to meet its design goals of usability, high control, flexibility, expandability, and maintainability, ALIAS is a multi-layered, modular system composed of many parts.

There are four fundamental parts: an integrated data base; a command system which interacts with the user and takes care of many supervisory functions; several libraries containing utility procedures and standard interfaces; and an unlimited number of applications modules which interface with both the command system and the data base (but not directly with each other). Figure 1-1 diagrams this structure as a pyramid: the data base, command system, and utilities form a base on which all other functions depend (the System Core); independent applications sit on top of this core.

Figure 1-2 shows an example of the flow of execution/use of the ALIAS system at a high level. In a typical feasibility analysis, an analyst will tell the computer he wishes to use ALIAS, which will cause the command system to present a list of optional actions. He chooses to modify the structure of program 'POM85b2'. Following this he will use the labor requirements outcome calculator to draw workload curves for various shipyards, the Data Base Updater to modify some ship schedules, the labor

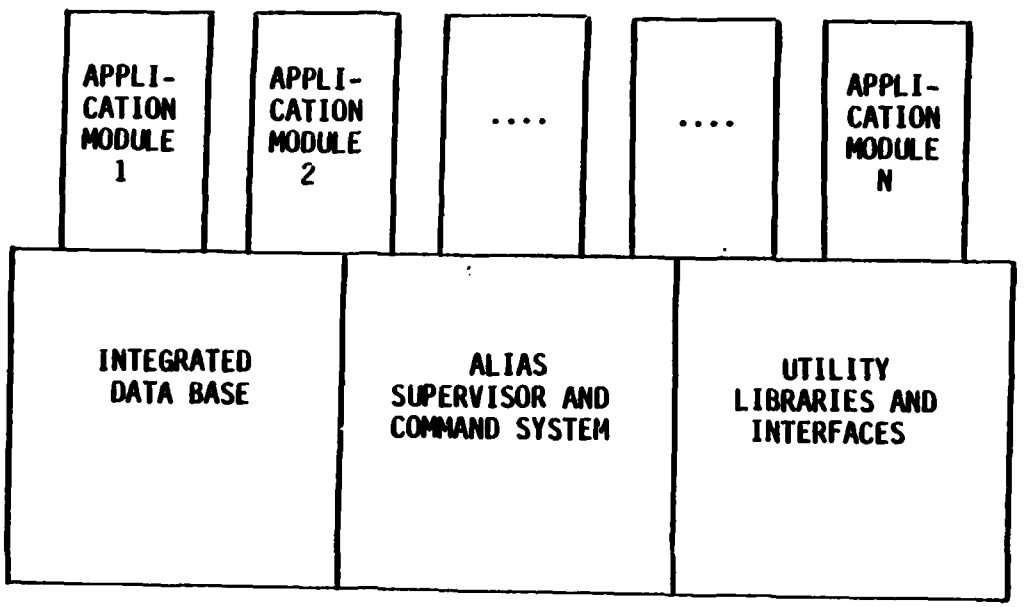


Figure 1-1. Basic Alias System Structure

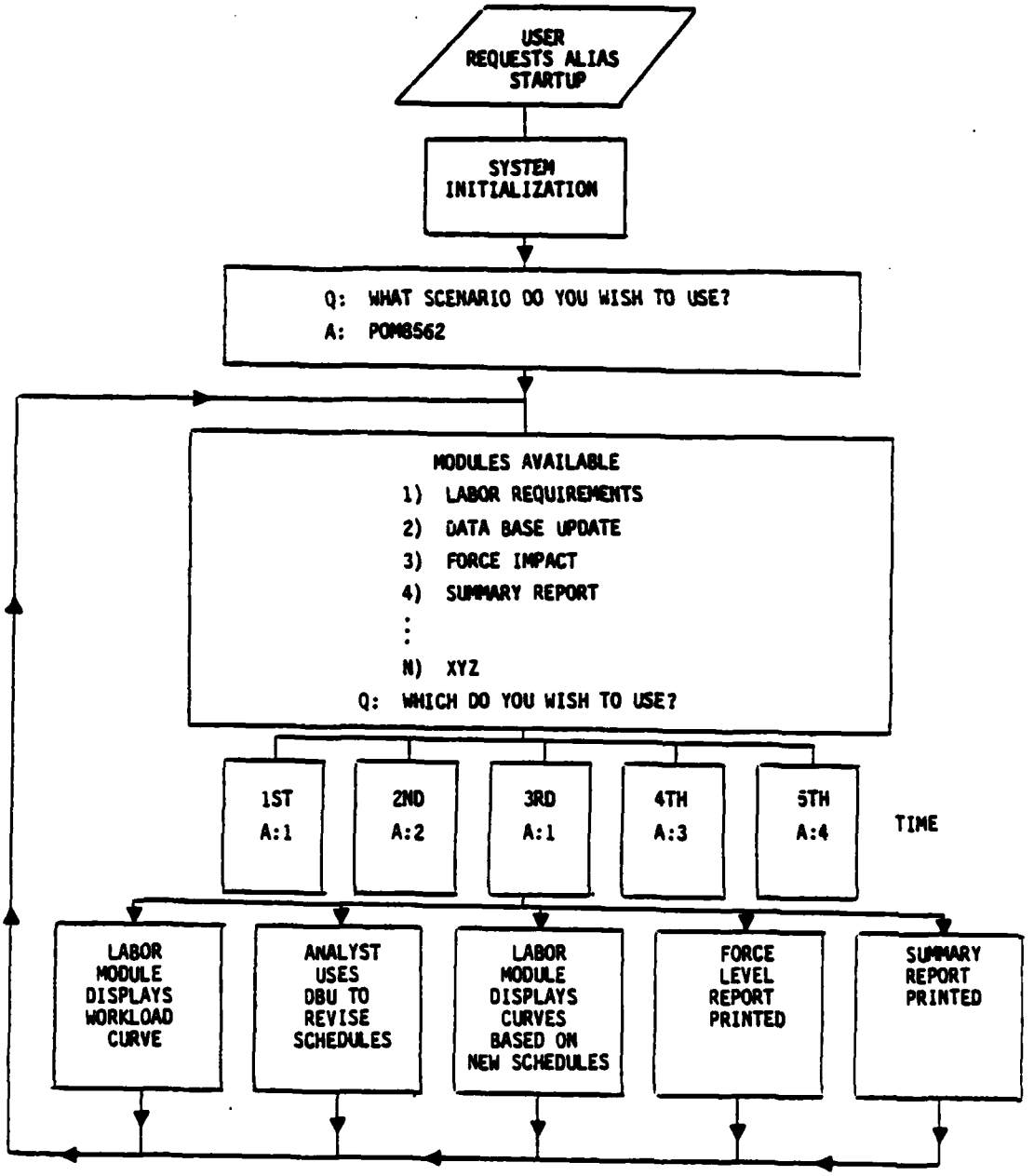


Figure 1-2. High-Level Examples of ALIAS Execution

requirements module again, will request a force impact analyis, and will ask that a summary report be generated. Use of a variety of analytical, reporting, and data control capabilities is typical.

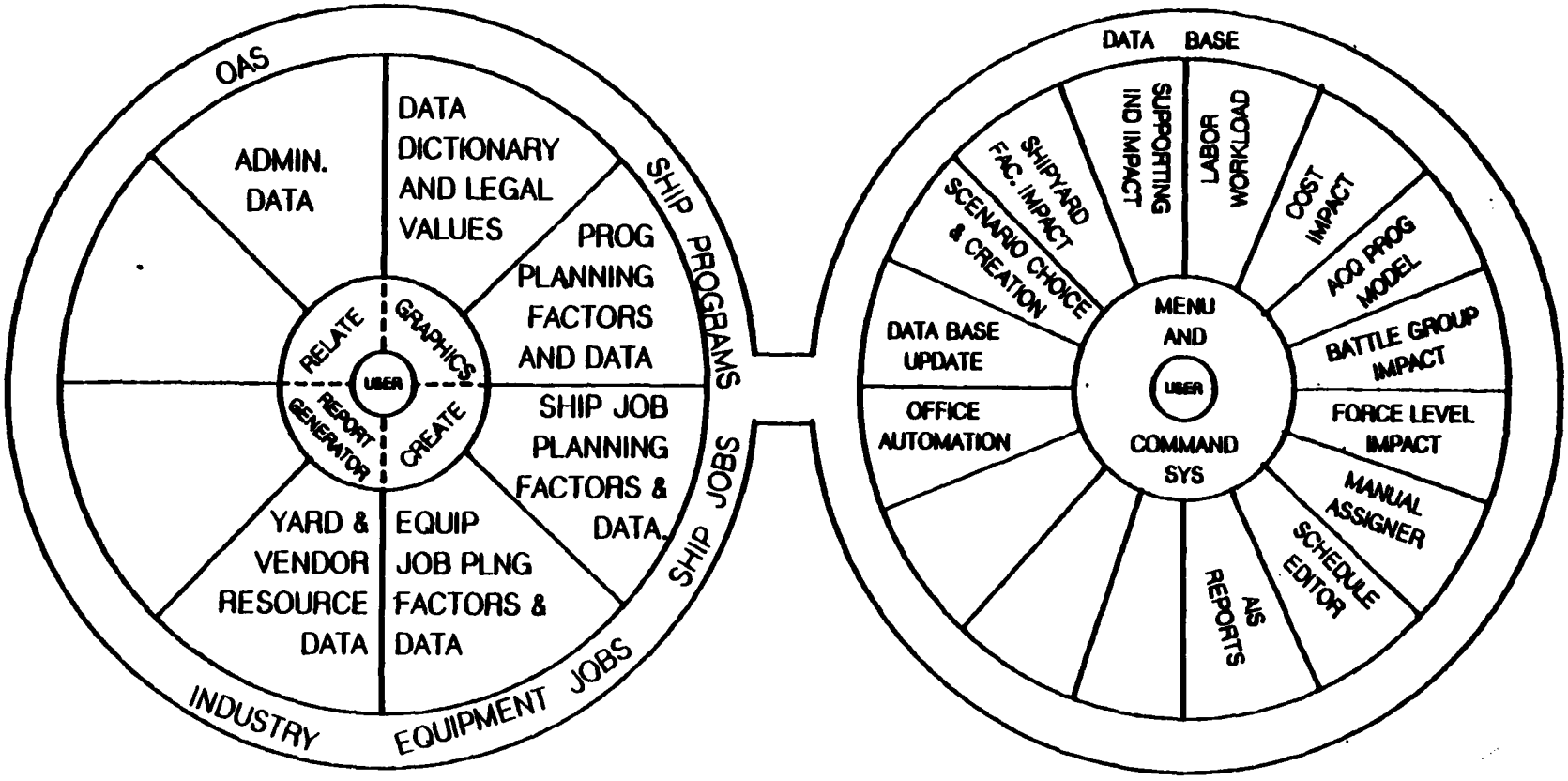
Figure 1-3 presents an alternative summary of the overall system structure. Here, the system is pictured in two major sections which lie 'inside' a box which is the host computer. The sections are the data base and the modular applications. Each section is a series of concentric rings, with the user 'sitting' in the middle of either of the rings. The ring around the user contains service software and interface software, which is what the user actually communicates with. In the case of the data base section, this software is the DBMS packages; in the case of the applications, it is the command system. The DBMS software in turn accesses various sections of the data base; the command system manages the execution of the various modules. Both the command system and the application modules depend on libraries of interface programs and utilities which occupy an intermediate ring. The application section's outermost ring is an extension of the data base outer ring, indicating that the modules communicate with the data base.

This representation of the system structure is more accurate in many respects than that of Figure 1-1. It shows more of the pieces and the layering of the system, and the analogy implicit in the positioning of the human user is more apt. It is in fact possible to use the resources of the data base 'directly' via the DBMS query and report generation software, or to use both the data base and the analytical capabilities of the system by asking the command system to execute appropriate modules. The modules are properly shown in separate slices of their pie, with communication between one and another possible only through the data base. The data base updating system acts as a direct interface to the data base itself, bypassing some of the interface rings between the user and the data base. Finally,

ACQUISITION AND LOGISTICS INFORMATION AND ANALYSIS SYSTEM

SOFTWARE ARCHITECTURE

(ALIAS)



DATA BASE MANAGEMENT SYSTEM AND DATA BASE GROUPS

COMMAND SYSTEM AND ANALYSIS MODULES

Figure 1-3. Alias System Structure

there are several utility and interface subsystems which the user sees little of, but which in fact lie at the heart of the system.

1.3.2 Introduction to the Underlying Software Structure

The implementation of ALIAS has been guided by a commitment to modularity of software, to use of well-defined interfaces between modules and observance of strict inter-module communications rules, to the building and use of libraries of utility routines, and to storage of as much system data as possible in the data base. Modularity and well-defined interfaces promote reliability and maintainability, and reduce implementation costs. In combination with isolation of hardware system dependent functions in utility routines, portability is also promoted. Maximum use of relations supervised by the DBMS for storage also promotes reliability and maintainability, since the DBMS allows easy, interactive inspection and modification of data values.

FORTRAN is the ALIAS language of choice. It was chosen over other candidates as the most portable language (it is very likely that ALIAS will be converted to run on new host hardware at some point). It is also one of the few languages suitably structured for support of very large software systems (for example, restrictions placed on the programmer's ability to separately compile and link routines by most versions of PASCAL make that language unsuitable for a project of ALIAS' size). Compilers conforming to the ANSI 1977 standard, provide the FORTRAN programmer with all of the basic mathematical and string-handling capabilities and with many conveniences. When used in combination with the sorting, searching, selection, and storage/retrieval capabilities of a modern relational DBMS, FORTRAN becomes a terse, powerful, and efficient programming language.

1.3.2.1 The Data Base

The data base is the heart of ALIAS. It holds almost all system input data, and many module outputs as well. It is fully relational in structure and is easy to modify and expand.

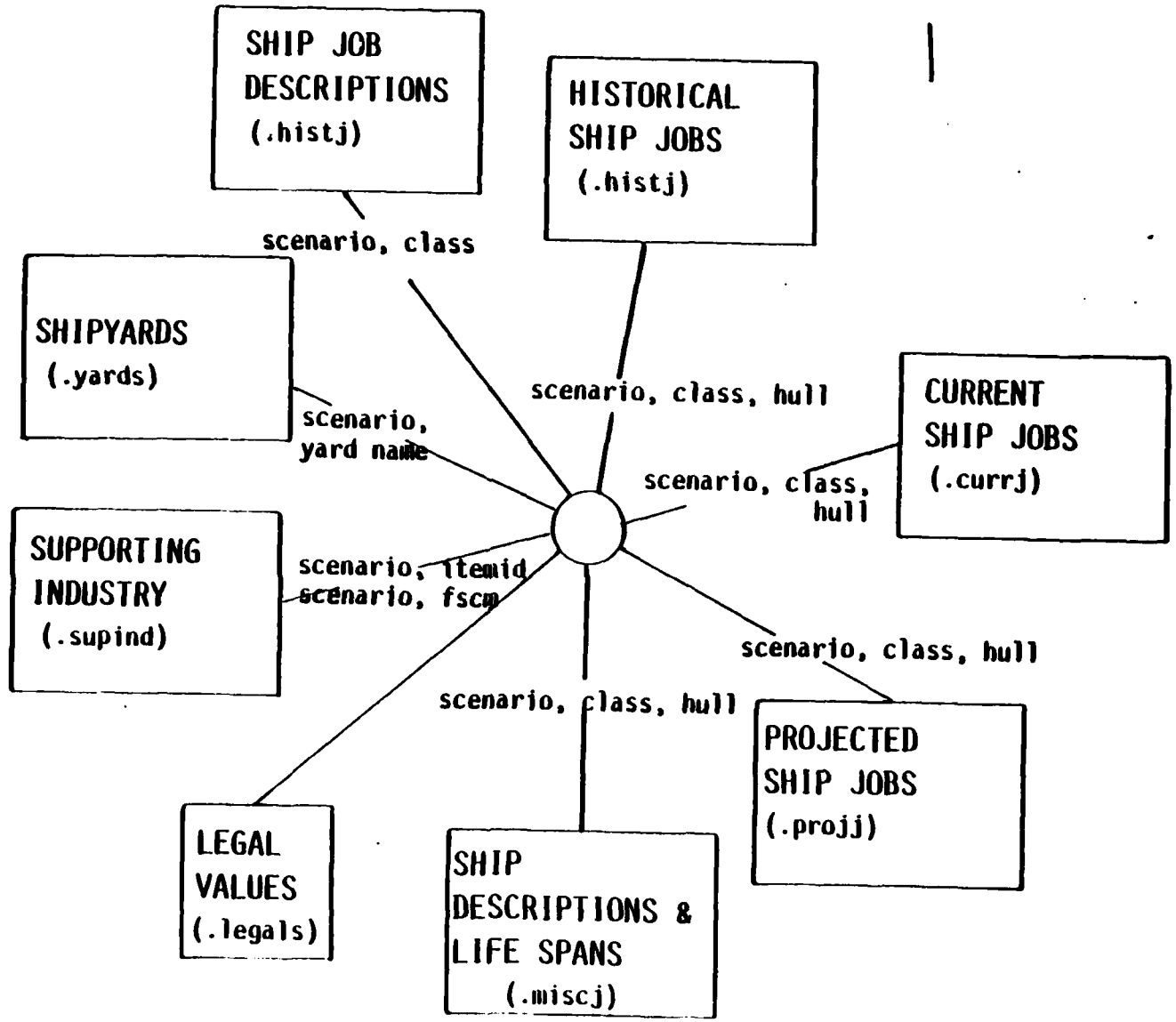
For convenience and manageability, the data base is divided into several sections of related files, as shown in Figure 1-4. Large boxes represent sections, each of which is implemented in a separate HP file group named after the section. The relations (files) which comprise each group are listed within the boxes. Labels on the lines drawn between the boxes and the central circle (the user) indicate the principal retrieval keys for the sections, though any field in a file may be used as a retrieval key.

As the diagram indicates, the data base is quite extensive; it now consists of more than 100 files, and is expected to grow steadily. Descriptions of the data base at the section, file, and field levels are contained in a custom data dictionary which forms a small data base of its own.

The ALIAS data base has an integrated design: that is, no data element appears in more than one place unless it must function as a key. This structure, and the fact that all modules use the data base for input and output, ensures that ALIAS system outputs are mutually consistent. An analyst can be confident that a cost estimate and a force level estimate made for the same projected shipbuilding program will in fact be using the same program structure as their basis.

1.3.2.2 Supporting Software Packages

ALIAS depends on the facilities of several supporting software packages. In addition to basic operating system utilities and the FORTRAN compiler, ALIAS makes heavy use of the RELATE relational data base management system to manage most data base operations. RELATE is a full-featured DBMS, and provides



OVERALL DATA BASE STRUCTURE

Figure 1-4. Data Base Structure

interactive, programmatic, and procedure file modes of execution. ALIAS makes use of all three. All programmatic use of RELATE is buffered through a special FORTRAN interface library to simplify any future conversions.

ALIAS also makes heavy use of the facilities provided by the BUILDER screen application generator, particularly for the Data Base Updating System (DBU). BUILDER is an interpreter which runs 'on top' of RELATE. It reads and processes standard ASCII files, which may contain either any RELATE command or any BUILDER command. Its principal purpose is the generation of fill-in-the-blank screens which can be used to display and update data base data. However, the BUILDER procedure language includes a full set of control structures including DO-WHILE and IF-THEN-ELSE, and also includes facilities for calling of user-generated FORTRAN subroutines and programs. This makes BUILDER a full-scale, though specialized, programming language.

Graphics support is provided both by the GRAF business graphics companion to RELATE, and by HP's Decision-Support Graphics (DSG) package.

1.3.2.3 The Menu System

ALIAS execution is supervised by a menu-oriented command and control system. The menu system performs security checking when a user first runs the system and at appropriate points thereafter, and takes care of much of the housework of executing modules. Its principal overt purpose is the display of a series of menus (organized in a hierarchy) containing lists of modules which the user may choose to execute, as well as lists of variables whose settings control the modules' execution. The user may move freely through the menus, executing modules and setting values at will. Figure 1-5 displays a short ALIAS session in which several menus were displayed and Figure 1-6 displays a "map" of the menus of a sample ALIAS configuration.

Figure 1-5. Sample ALIAS Menus

```
Standard options include:                                  Scenario is IMAGINATION
 ^   pop to previous menu              |  /   pop to top menu
 &   reprint with current values       |  B   build command file
 E   end command file building         |  U   use command  file
 ?   use help processor                |
-------------------------------------------------------------------------------
                    TOP LEVEL ALIAS COMMAND MENU
    1.  CUSTOMIZE USER ENVIRONMENT
    2.  CALL NON-ALIAS PROCESSORS
    3.  DATA BASE UPDATING SYSTEM
    4.  MANUAL ASSIGNMENT EDITOR
    5.  FORCE LEVEL REPORT GENERATOR
    6.  SCENARIO CHOICE/MAKEUP SYSTEM
  COMMAND: 4


Standard options include:                                  Scenario is IMAGINATION
 ^   pop to previous menu              |  /   pop to top menu
 &   reprint with current values       |  B   build command file
 E   end command file building         |  U   use command  file
 ?   use help processor                |
-------------------------------------------------------------------------------
                    MANUAL ASSIGNER SPECIFICATIONS
    1.  ASSIGNER INITIALIZATION PARAMETERS
    2.  EXECUTE THE ASSIGNER
  COMMAND: 1


Standard options include:                                  Scenario is IMAGINATION
 ^   pop to previous menu              |  /   pop to top menu
 &   reprint with current values       |  B   build command file
 E   end command file building         |  U   use command  file
 ?   use help processor                |
-------------------------------------------------------------------------------
          MANUAL ASSIGNER MODULE INITIALIZATION PARAMETERS
 1.  TIME UNIT                 = FISCYR       (FISCYR,CALYR,QTR,MONTH,WEEK,DAY)
 2.  STARTING DATE             =  1/ 1/1980   (MM/DD/YYYY)
 3.  ENDING DATE               =  9/30/1990   (MM/DD/YYYY)
 4.  CANDIDATE SHIP YARDS      = LIST         (ALL/LIST)
 5.  CANDIDATE SHIP CLASSES    = LIST         (ALL/LIST)
 6.  CANDIDATE JOB TYPES       = LIST         (ALL/LIST)
 7.  DISPLAY BASIS             = AWARD        (APPROP,AWARD,START,KEEL,LAUNCH,DE
 8.  ADJUST BASIS              = START        (APPROP,AWARD,START,KEEL,LAUNCH,DE
 9.  ADJUST MODE               = PROGRAM      (NONE,PROGRAM,COMPLX-GROUP)
10.  JOBS EPOCH OPTION         = PROJ         (ALL,CURR/PROJ,PROJ)
11.  SHIPCLASS SORT ORDER      = ALPHABETIC   (ALPHABETIC,INPUT ORDER)
12.  SHIPYARD SORT ORDER       = INPUT ORDER  (ALPHABETIC,INPUT ORDER)
13.  AUTO REFRESH              = OFF          (ON,OFF)
  COMMAND: 6=list


Currently list options include:                            Scenario is IMAGINATION
 ^   pop to previous menu              |  /   pop to top menu
 &   print screen with current values  |  A   all items are INCLUDED
 N   no items are INCLUDED             |  ?   help for this menu
 <   print previous list screen*       |  >   print following list screen*
 * NOTE   only effective when list cannot fit on one   output screen.

-------------------------------------------------------------------------------
          CHOOSE THE SET OF VALID JOBS WHICH MAY BE ASSIGNED

-------------------------------------------------------------------------------
    1. * CONV                          4. * REFUEL
    2. * NEWCON                        5. * REPAIR
    3. * REACT                         6. * SLEP


-------------------------------------------------------------------

  COMMAND: /


Standard options include:                                  Scenario is
 ^   pop to previous menu              |  /   pop to top menu
 &   reprint with current values       |  B   build command file
 E   end command file building         |  U   use command  file
 ?   use help processor                |
-------------------------------------------------------------------------------
                    TOP LEVEL ALIAS COMMAND MENU
    1.  CUSTOMIZE USER ENVIRONMENT
    2.  CALL NON-ALIAS PROCESSORS
    3.  DATA BASE UPDATING SYSTEM
    4.  MANUAL ASSIGNMENT EDITOR
    5.  FORCE LEVEL REPORT GENERATOR
    6.  SCENARIO CHOICE/MAKEUP SYSTEM
  COMMAND:
```

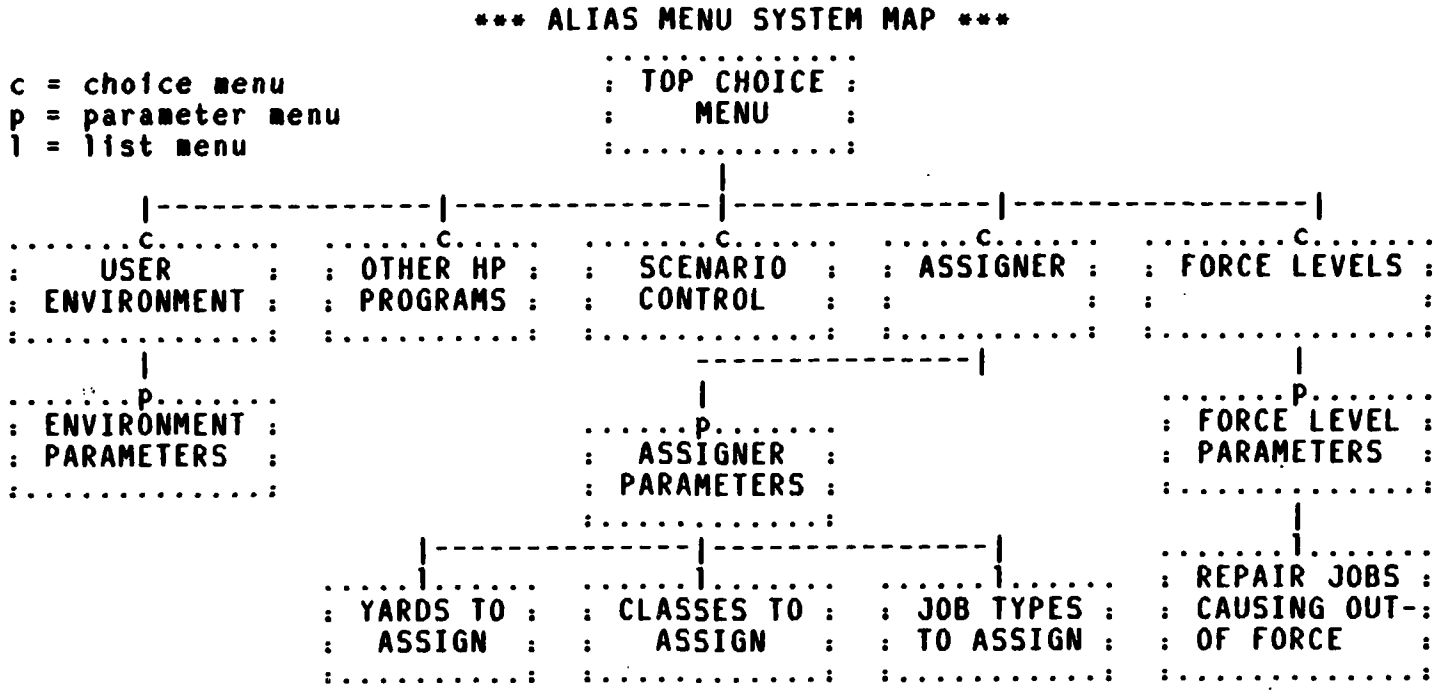


Figure 1-6. Sample Map of ALIAS System Menus

From a system developer's standpoint, the menu system is a principal reason why ALIAS is an expandable and easy-to-maintain system. Because the number of menus and menu options which can be displayed is effectively unlimited (only host hardware limitations bind), the menu system acts as a convenient hook on which an unlimited number of 'black box' modules may be hung.

Expanding ALIAS is relatively easy because the menu system has a two-program design: there is a preprocessor or translator, which reads an input file describing the menus a system creator wishes to have displayed; this produces a number of data files, relations, and FORTRAN source code. There is a run-time program which uses these data files and relations to display the menus, accepts the user's commands, and carries out his wishes. The preprocessor need only be run when a change to the displayed menus must be made.

An extensive help subsystem underlies the run-time system, providing descriptions of the effect of any menu option as well as a variety of other aids. The text of this help subsystem is fully data driven and may be changed by a system supervisor at any time.

1.3.2.4 The Scenario System

Every data base file has a field named 'scenario'. The contents of this field determine which scenario the data in each record belongs to. The existence of this convention allows the data base to be partitioned into several sections, each of which has the same fundamental structure. Each section may be thought of as containing the data for a separate study. For example, a scenario used for a study of a normal POM/EPA shipbuilding plan would contain one set of projected ship construction schedules, while a scenario for a mobilization study would contain a vastly different set of schedules. The schedules for both scenarios

(studies) would reside in the same file; the scenario system makes sure that analysts doing the POM study work only with the POM schedule data and not with that for the mobilization study.

The multi-scenario capability is a key method of implementing the ALIAS data-driven philosophy of modeling, since the isolation of one scenario from another allows any user to freely alter data base values (even descriptions of shipyard facilities, for example) without fear of affecting anyone else.

Users may create new scenarios at any time, and may specify in detail the source of data for a new scenario on a file-by-file or group-by-group basis. Scenarios may start empty, or data may be drawn from an existing scenario. Existing data may be drawn on in one of two ways. If the creator of a scenario expects to need to change the data in a particular file or group, he may specify that read/write access will be required for the data in that file/group. In this case, a full copy of the source scenario's data in that file/group is made, with the name of the new scenario placed in the scenario field of the copy. If no need to make changes is foreseen, the creator may specify read-only access. This option does not lead to a data copy; the users of the new scenario will use data in the source scenario directly (they may not change it, of course). The 'indirect access' option allows a user to depend on the 'main' scenario (or any other scenario) for much of his data; since the 'main' scenario is the one for which updates are performed daily, this ensures data currency with a minimal effort, while simultaneously protecting the integrity of the 'main' scenario.

Figure 1-7 illustrates three data base files containing two scenarios. The "*" lines indicate the implicit partitioning of the SCHED and SHIPDESC files which RELATE's indexing capability permits. Note the absence of data for scenario "POMEPA" in the YARDS files; if the creator of "POMEPA" so specified, the YARDS data belonging to "POM" may be referenced "indirectly" during "POMEPA" runs.

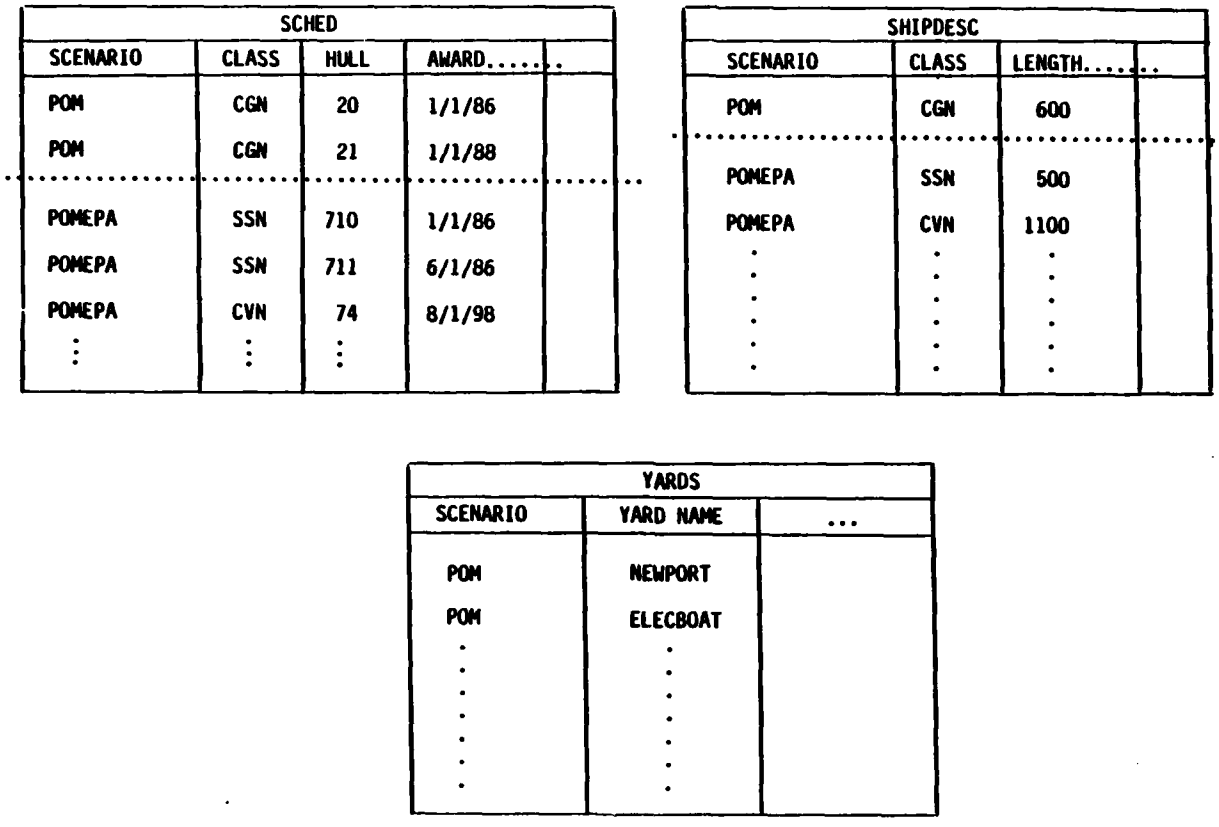


Figure 1-7. Three Data Relations Containing Two Scenarios

The software which implements this capability is diffused throughout the ALIAS system. The bulk of it is concerned with prompting the user for the scenario to use and with the creation of new scenarios. Its most important parts, however, are to be found in the data base interface utility library. The interface routines provide the programmer with tools to use in obtaining the proper scenario field key value for the current scenario for use in file searches and data retrieval. In addition, the DBU allows access only to data records which have the proper scenario name as their key field value.

When adding a new module or modifying the DBU, it is important to recognize the need to protect scenario integrity. All code conducting data base operations should be written in a manner which causes only data from the proper scenario to be retrieved.

1.3.2.5 The Data Base Updating System

Ensuring data base integrity is a particularly difficult task when relational DBMSs are used, since the DBMS itself enforces very few limitations on the data a user chooses to enter. It is easy to add data to one section of the data base and forget to add companion data to another section. In the ALIAS system, the primary means of ensuring integrity is the Data Base Update module (DBU), a series of screen-oriented procedures built with the BUILDER fourth-generation programming language. In addition to providing a very user-friendly 'window' on the data base, the DBU checks all requested DB changes against standards contained in the data dictionary before carrying the changes out.

1.3.2.6 Utilities and Interfaces

Two implementation techniques which grow out of the modular design of ALIAS, are the development and use of libraries of utility subroutines and procedures, and the development and use of strictly defined interfaces between parts of the system.

A utility-oriented philosophy of programming takes modern modular programming techniques to their logical conclusion. Courses and textbooks teaching programming, now routinely prescribe breaking complex programs into small, functionally oriented entities. By breaking a programming job into small pieces, implementation and debugging time requirements are shortened, and long-run maintainability is enhanced. Experience writing many programs shows that most require routines to handle various low-level tasks, such as error handling, vector arithmetic, sorting and searching. Careful specification and implementation of such routines allows them to serve the needs of many different programs; since each routine need only be written and debugged once, long-run development costs are further reduced. Placement of such routines in separate utility libraries makes them easily accessible.

More than one hundred FORTRAN subroutines now reside in the ALIAS general-purpose utility library. The utility concept has also been applied to development of the Data Base Updating System (DBU); tens of 'subroutine screens' provide services such as data checking and file fetching, vastly reducing the volume of data screen procedure code required.

Another function which the general-purpose utilities take care of is interfacing with the host computer operating system. Isolation of requests for operating system services in utility routines reduces the cost of maintaining and converting a software system, since system-dependent calls are not spread everywhere. For example, no two host computers respond to a request for the current date in exactly the same fashion. This information is needed in several places in ALIAS. To get it, ALIAS programs call the FORTRAN utility routine 'ddate', which in turn executes about ten lines of system-dependent code. If these ten lines were substituted everywhere there is a call to ddate, and the way in which the operating system responds to the date request changed, the maintenance fixes required would be

extensive. As it is, only a change to the ten lines would be needed. An ALIAS development standard is that ALL operating system calls shall be isolated in utility routines, and that the structure of these routines shall be such that convertibility is promoted.

In addition to the general-purpose utilities and associated interfaces to system-dependent functions, a special library of utilities which buffers all program calls for DBMS services has been written. The structure of the library routines is such that the DBMS services could be provided by any full-feature, cursor-oriented relational DBMS. A primary reason for the creation and use of the library is to allow conversion of ALIAS to use of a DBMS other than RELATE at reasonable cost. ALIAS code is ripe with requests for DBMS services; failure to buffer these requests through the interface would have made ALIAS completely dependent on the HP 3000 and RELATE.

The DBMS interface also handles a number of housekeeping functions which RELATE requires, such as cursor management and word alignment of command strings.

1.3.2.7 The Applications Modules

Applications modules are the parts of ALIAS that do the work that analysts want done; the rest of the system exists to support these and to provide services and conveniences for the user. Current applications include the assigner (ships-to-shipyards assignments entry), force level report generator, battle group report generator, and cost estimator. These are designed primarily to support long-run feasibility analyses of SCN POM/EPA plans.

Applications modules may take a variety of forms ranging from simple report generation procedures to large models. They may:

1) Cause execution of DBMS procedure files which produce reports or graphics,

2) Cause the BUILDER to execute procedure files which display data entry screens, or which perform functions which the BUILDER language does particularly well,

3) Be programs composed of one or a few subroutines written in FORTRAN, COBOL, PASCAL, or SPL,

4) Be large, separate programs which ALIAS will run and supervise as son processes (written in any HP language), running either in the foreground or in a background mode invisible to the user and allowing unhindered continuation of the ALIAS session,

5) Start up batch jobs which run one or more independent programs,

6) Be general purpose programs such as the HP graphics packages; ALIAS will temporarily hand over control to these packages.

This wide variety of application formats, supervised by a single consistent user interface, provides great flexibility to the user and the system developer alike. The number of ALIAS applications modules is now limited to 285, but this limit may be revised upward by a small revision to the WRUNP routine of the MNUG module of the menu system.

1.3.2.8 Actual Flow of System Operations

The modular architecture of ALIAS is carried through to the extent of implementing the system to run as several processes on the HP 3000, rather than as a single large process. This architecture sacrifices efficiency, but was required by the data memory limitations of the HP 3000 (64K bytes directly addressable per process).

When ALIAS is first run, it consists of two processes: the main ALIAS supervisory program (MNUR), which includes the run-time menu system, the scenario system, and various initialization servers; and a RELATE DBMS son process which performs services for MNUR.

No additional processes are started up until the user chooses a menu option which corresponds to an application module (many menu options merely cause a different menu to be displayed, or set data values; these functions are all handled by MNUR). The processing action taken depends on the nature of the application module. RELATE procedure files and small modules linked into the MNUR program are executed directly or by the RELATE service son process.

Larger applications are typically implemented as separate programs. MNUR executes these as separate son processes, handing off control and suspending its own operations until the module finishes and sends a reactivation signal. Before handing over control, MNUR writes a certain amount of control and security data into a communication segment, which the module then uses to initialize itself on startup. Modules such as these will typically need to start up their own RELATE service son process; multiprocessing on the HP is organized in a rigid hierarchy, so the process servicing MNUR is unfortunately not available for use by others.

The Data Base Updating System (and to some extent any module using the BUILDER) is a special case. The DBU is executed by running the BUILDER as a son process from MNUR, giving the DBU procedure file name as the primary BUILDER input file. Because of the large number of data files that the DBU needs to access, and the fact that a single RELATE son process can never handle more than about 25 files at once, the DBU starts up a large number of RELATE sons to conduct its DB servicing. Display of many DBU screens by the user can cause the number of such service processes to grow to 10 or more. Because of the overhead involved in starting these up, the DBU never terminates them, and never terminates itself either. Instead, when the user issues a 'Quit' command to the DBU, it places itself on hold and returns control to the main menu system. MNUR detects this status on the next request for the DBU and performs a 'reactivate' instead of a

'restart'. This architecture uses large amounts of virtual memory (for the service process data stacks) to gain improvements in performance.

1.4 OVERVIEW OF PRINCIPLES GUIDING THE ALIAS SYSTEM DESIGN AND IMPLEMENTATION

1.4.1 Software

The watchwords controlling the design, implementation, and expansion of ALIAS are usability, high control by the user of system operations, flexibility, expandability, consistency, and maintainability. To some degree, each of these goals grows out of the premise that ALIAS is both a tool and a workbench for Navy analysts, rather than a reference service or a traditional 'oracle-on-the-mount' model which provides the answer to one or a limited range of questions.

Tools should be easy to use, requiring a minimum of understanding of computers and software and a minimum of set-up time at each use. By analogy, one does not need to know wave mechanics to use the motor of a table saw, nor should one have to tighten the blade's retaining bolt for each cut; likewise, one should not have to understand the intricacies of computer data storage to use software, and should not have to respond to tens of 'initialization' questions each time a function is carried out. Tools that are not easy to use decrease throughput and distract the attention of analysts from the substantive issues they are interested in.

At the same time, tools should have enough adjustment 'knobs' to allow them to work with a variety of situations. Using the same analogy, a table saw must have a blade retaining bolt so that the appropriate blade can be mounted when a new material is to be used. ALIAS analytical modules must have enough adjustable initialization values to give the user high control of the assumptions each module makes as it works. These values, and the structure of the modules, must be sufficiently broad and general to allow a great variety of situations to be analyzed at varying levels of detail; this provides flexibility.

As time goes on, new situations and new questions will come up, and new knowledge will breed new techniques of analysis. Any software tool/workbench designed for analysts must therefore be expandable, so that it does not become obsolete. This expandability must take the form of ease of modification of existing functions and capabilities, and ease of addition of completely new ones.

The fundamental commodity of an analytical system is information; the information which is produced must be internally consistent or the system (and its users) loses credibility. It is important that all outputs of ALIAS be consistent with one another; for example, a current status report and a near-term capacity usage projection which both display or use current shipyard employment levels should use the same levels.

Finally, any tool should be easy to maintain. Oiling the table saw at each use reduces throughput; a saw which costs as much to fix as it did to buy is not economical. Software can and should be written in such a way that it is understandable to any competent technician, and it must be possible to move the software from one hardware system to another.

1.4.2 Modeling

1.4.2.1 Classic vs. Outcome Calculator Methodologies

A principal function of the ALIAS system is to evaluate the feasibility and efficiency of shipbuilding program plans. This requires the use of a large and diverse array of models: optimizing schedulers, cost estimation models, and industry capacity usage estimation models, to name a few.

The basic purpose of all of these is to allow the analyst to obtain an idea of the results of a shipbuilding program (i.e., an idea of what the world would look like were it implemented), and to identify or design the best possible program. To produce

this capability, classic modeling methodologies would identify all of the important factors, determine their interrelationships, identify criteria to be used at decision points and, from this information, design a single model which would, for example, accept a program plan as input and generate a number of outputs predicting the results.

The problem with this classic methodology is that for large or complex problems, it is costly, requires much research, is very risky in that there is no way to tell if the model will work as desired until the whole investment is made, is very time consuming, and often produces models which are rather inflexible and which are expensive to modify to accept new considerations. In spite of these drawbacks, the classic method is the only way to fully solve some problems.

There is an alternative for situations which are decision-intensive and for which some precision may be sacrificed. Known as 'outcome calculator support', this methodology emphasizes placement of a skilled human 'in the loop', who makes decisions and controls the operations of one or more supporting models. In the case of ALIAS, the shipbuilding program feasibility/efficiency problem has been broken down into a number of more specific questions, each of which is treated separately using rather simple models. The analyst may input a program plan, and may then require the system to estimate its funding requirements, whether any facility shortages will result in the shipyards, whether material and equipment order lead times are long enough, etc. Should the analyst not like the answer to any of these questions, he is free to change the plan and reevaluate. If a model says there will be a problem when the analyst knows there will not be (special circumstances, perhaps), he is free to ignore the model's judgment.

The human makes guesses and decisions, and the computer estimates and presents outcomes. This division of labor

corresponds nicely to the comparative advantages of human and computer; given information, the human is a much more efficient general-purpose decision-maker and can find his way to the solutions of many kinds of problems much faster. The computer is much more efficient at storing and retrieving large volumes of data, at performing large amounts of tedious calculation and manipulation of the data, and at presenting it in a variety of forms.

The outcome calculator methodology avoids all of the drawbacks of the classic integrated-model methodology. It can be implemented incrementally, reducing costs, risks, and lead times, often requires much less research and produces very flexible modeling systems because the amount of judgment being exercised by the computer is much less. Its drawback is that some precision is often lost, particularly since it is often necessary to ignore some interrelationships (that between shipyard facility capacity usage and cost levels, for example). Also, sensitivity studies become very difficult to conduct, since the human in the loop often will not react precisely the same way to the same situation.

Thus, the classic and the outcome-calculator methodologies are in a sense mirror images of one another: each is strong where the other is weak. The ALIAS philosophy is to take advantage of this complementarity and use both methods synergistically. The technology initially used to model a phenomenon should be of the outcome-calculator variety. This allows automated support to come on-line in a minimum time and with a minimal initial investment. Should the support provided prove adequate, nothing more need be done. If more precision or sophistication is needed, many valuable lessons will have been learned during implementation and use of the outcome calculator system, which can then be applied to the building of a classic integrated model. Much of the required data will be on-line, many relevant algorithms will be developed and thoroughly tested,

and key decision-points will be known. If the system data structure is flexible enough, the integrated model can be built 'on top' of the outcome calculator system, allowing the results produced by the integrated model to be massaged by the outcome calculators, and allowing the human-outcome calculator team to handle unusual situations as they crop up.

This synergistic choice of technologies is strongly recommended for future expansions of ALIAS capabilities. Figure 1-8, an adaptation of Figure 1-1, expands the ALIAS pyramid by dividing application modules into two groups. Outcome calculators, report generators, and data updaters sit on top of the system core, and 'integrated', 'classic models' sit on top of these. In fact, the integrated models are just another module as far as ALIAS is concerned; the analogy refers to a structuring of capabilities, not to the actual software structure.

For example, one 'integrated model' planned for ALIAS is an automated shipbuilding program generator. It is desired that this model be able to optimize some mixture of force level enhancements, cost containment, productivity enhancement, program risk minimization, etc., the mixture being determined by the analyst. Such a model would need to incorporate all of the considerations found in the feasibility outcome calculators, as well as many interrelationships. It would be very useful in conduct of sensitivity studies, since it would be possible to vary goals or assumptions about things like industry capacity levels and obtain a range of consistent results. However, its output will be a shipbuilding program plan very much like that produced manually by an analyst. It will be possible to inspect and, where special considerations appear, modify the plan in the standard fashion. Both precision and flexibility will be available.

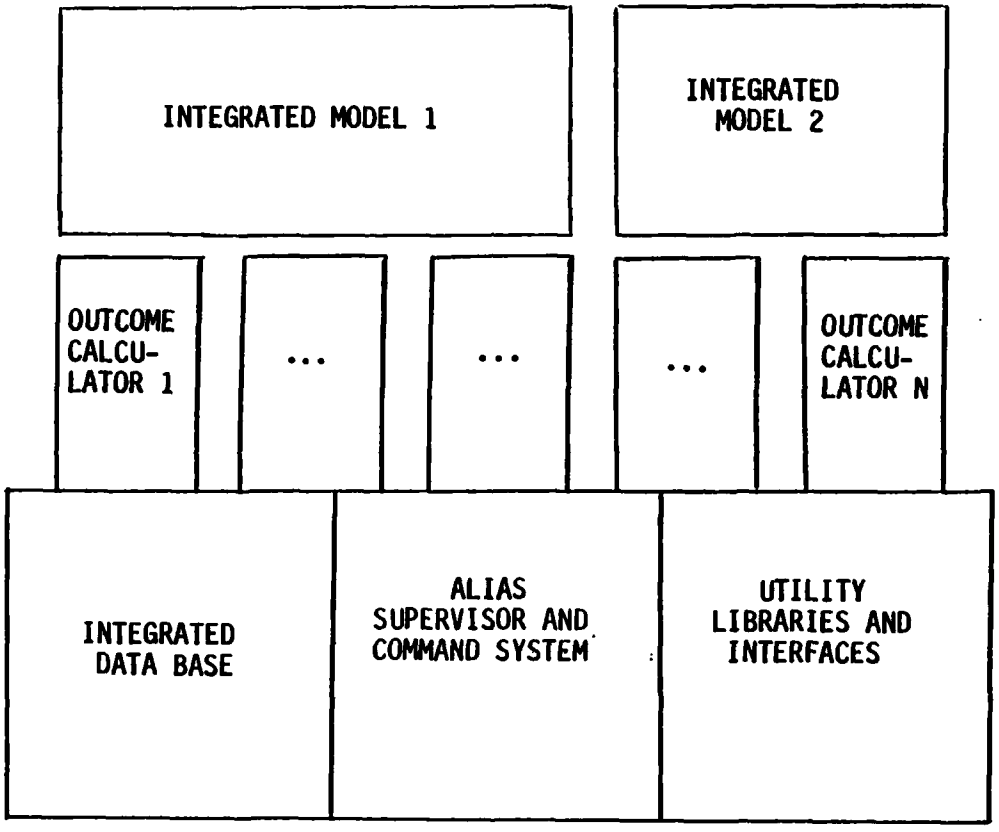


Figure 1-8. Overall ALIAS System Structure

1.4.2.2 Use of Data Driven Techniques

Another way to increase the flexibility of a modeling system is to embody as much of the models as possible in an accessible data structure, minimizing the amount of program code. This often allows unusual situations to be handled by changes to the data values, where a code-intensive system requires reprogramming.

As an example, suppose that an estimate of the yearly costs of constructing a series of destroyers at three shipyards is to be made. Suppose that one of these shipyards has never built destroyers before, and that, unusually, some one-time start up costs are to be financed on the contract of the second ship built at this yard. Suppose further that significant design changes are expected every four years, and that normal labor learning economies will thus receive a setback every four years.

A simplistic code-intensive system might accept as input only a single unit-cost estimate for the whole ship class, making it impossible to deal with the special yard and the learning. A more sophisticated one might take learning into account, but only continuously over the life of a program. Another might allow 'penalties' to be designated for specific ships.

A data-driven system would accept unit cost estimates by ship class and/or by class and shipyard, and would allow these to be overridden by individual base cost estimates for each individual ship if the user desired. It would accept labor learning rates by a similar general-overridden-by-specific keying scheme, and would allow the place of each ship on the learning curve to be specified, overriding automatic learning calculations. Handling the 'unusual' situation described above is then merely a matter of customizing the unit cost of the special third-yard ship, and the learning-curve position of the ships in each fourth year.

A data-driven system should minimize the difficulties of making such customizations. It should not be necessary to write a FORTRAN program to access and modify an obscure data file each time this sort of action is to be taken.

Data-driven models are most flexible and robust when their data structures closely reflect the structure of the phenomenon modeled, containing enough variables for a reasonably complete description.

2.0 GOALS, PRINCIPLES, AND STANDARDS FOR ALIAS SOFTWARE

This section describes in somewhat more detail the ideas and methods which played roles in the design and implementation of ALIAS. It also prescribes general practices that should be followed and standards that should be conformed to when new software is written for addition to ALIAS. The section repeats some material of the Introduction, but is somewhat more formal.

The relationship of each system design principle and standard to the system goals is described in the hope that personnel undertaking maintenance and expansion tasks will come to fully understand the ALIAS philosophy. It should not be thought that the principles and standards stated represent a final and complete set; these should be modified along with ALIAS itself, as new techniques and ideas appear.

2.1 SYSTEM GOALS AND REQUIREMENTS

2.1.1 Goals Related to System Operation

ALIAS should provide the best support possible to its users on a day-to-day basis. This basic goal may be translated into several subsidiary goals.

ALIAS must be as easy to use as possible. This does not mean that sophisticated technical knowledge should not be required to operate some application modules; it does mean that users should not be required to have a deep understanding of computers, file structures, data base management systems, and programming languages in order to carry out basic functions. They should not need to understand the whole system in order to use isolatable parts of it.

At the same time, users must retain a high degree of control of system operations, particularly application module operations. Simplification of ALIAS usage by 'hard-wiring' of standard assumptions into module logic is an unacceptable sacrifice.

In the same vein, ALIAS should be as flexible as possible. This goal is two-fold. First, it should be possible to combine system functions rather freely in order to produce custom reports and analyses. For example, combining the production of three separate reports into a single modular unit should be avoided in favor of a separate unit for each report. Second, each analytical module should permit the user to specify a wide variety of assumptions and control values so that it may serve for the analysis of a wide variety of situations and problems without continual reprogramming.

ALIAS should be a consistent system. It should not be necessary to enter a data item more than once, and outputs should agree with one another where they share common data.

2.1.2 Goals Related To System Maintenance

The long-run cost of the ALIAS system must be minimized. This basic goal may also be translated into subsidiary goals.

ALIAS must be maintainable at minimum cost.

ALIAS must be a changeable and expandable system. The Navy expects that the extent of support provided by ALIAS will expand as knowledge expands. It must be possible to revise or replace existing software as it becomes obsolete or inefficient, and it must be possible to add new software as appropriate. The system should be designed and implemented to minimize the cost of doing so.

ALIAS should be as portable as possible. It is likely that over a given amount of time, ALIAS software will have to migrate to several different host computers. Software should be designed and implemented to minimize the cost and disruption of conversion.

2.2 DESIGN PRINCIPLES AND STANDARDS

This section describes the design principles used to achieve the above goals in the existing ALIAS system. These principles should be considered as standards against which the designs of new system modules should be compared. Statements of these standards are in bold face for easy identification.

2.2.1 Modular Structure

ALIAS has a fully modular system structure. Each analytical function of the system is carried out by one or more separate software 'black boxes'. The modules are 'hung' on a System Core which provides command and control, interfacing, and security services. Strict communication rules forbid direct communication between modules--all module outputs go either direct to the user or into the data base. Interdependency is thus minimized, allowing modules to be unplugged and replaced with updated versions with a minimum of impact on the system.

Software additions to ALIAS should be designed and implemented using this modular philosophy. Available System Core services should be utilized where feasible in preference to the writing of additional software duplicating those services.

System modularity supports the design goals of high user control, flexibility, changeability and expandability, maintainability, and portability. High control is supported since users need execute modules only as desired, rather than in preset patterns. Flexibility is supported since functions may be combined at will. Changeability, expandability, and maintainability are supported by the ability to unplug and replace, or add 'black boxes' with relative ease. Modularity enhances portability by isolating functions, limiting the scope of portability problems.

2.2.2 Integrated Data Base

ALIAS is structured around an integrated data base, which holds almost all module input data, as well as many module outputs. The data base provides a consistent and stable data structure and inter-module interface for the system as a whole. Its integrated structure allows data on many subjects to be combined at will.

Software additions to ALIAS should make use of this integrated data base, expanding its scope where necessary. Under no circumstances should separate data bases duplicating in whole or in part the data of the integrated data base be created.

The use of an integrated data base supports the design goals of high control, flexibility, consistency, changeability, expandability, and maintainability. Direct access to system data by users is facilitated, improving their control over system operations and their ability to produce ad hoc reports and studies. Since all modules draw on the same source for data, reports using the same data will be consistent with one another. Lack of consistency is a major problem with outputs from systems using separate data bases for each function. Usability is promoted since the burden of data entry and updating is minimized. If properly implemented, an integrated data base can also promote changeability and expandability.

2.2.3 Outcome Calculator and Data Driven Modeling Methodologies

The foundation of ALIAS modeling capabilities has been laid using the outcome calculator and data-driven methodologies described in Section 1. When initiating a capability to model a particular phenomenon, decision-oriented portions of the problem should be left to the discretion of the user/analyst as much as possible. The structure of models should lie in the structure and use of the data base as much as possible, as opposed to the structure of executable program code. This maximizes flexibility

and reduces risk, and does not preclude the development of more integrated, code-intensive models as they become feasible and necessary.

Designers of new modeling capabilities for ALIAS should adopt this outcome calculator methodology by default, bypassing it in favor of classic integrated model construction only when the methodology does not promise to meet model requirements.

The outcome calculator/data driven methodology supports the system design goals of high user control, flexibility, and changeability. Many decisions are by design left to the analyst, rather than being obscured in the operations of the software. Likewise, model decisions and default operating methods may be overridden by modification of data base values, allowing examination of unusual situations. Changeability is promoted since outcome calculators typically have much less code and a more open structure than classic integrated models, reducing reprogramming costs.

2.2.4 Scenario Concept

The ALIAS data base is effectively partitioned into separate sections, each holding the data for one scenario or study. Data may be shared between scenarios, and the composition of scenario data may be altered at will. Users are limited to the use of one scenario at a time (to which they typically have exclusive access), allowing the update deadlocks often experienced with multiuser data bases to be avoided. Thus, no special provisions need be taken by a module developer to allow use of his module on a variety of studies.

Module developers must, however, make sure that any procedures they develop conform to scenario security procedures so that users of their modules, in fact, have access only to the data they are supposed to.

The scenario system supports the design goals of usability, flexibility, and consistency. Users are freed from almost all of the data management burden of maintaining and protecting data for multiple studies on-line. The ability to switch between studies at will, and to share and combine study data, greatly enhances system flexibility. The security and protection mechanisms of the scenario system help ensure consistent outputs.

2.2.5 Consistent User Interface

The ALIAS user interface has been made as consistent as possible. Principally, this involves using similar formats for presentation of menus where possible, and ensuring that command codes and words cause similar functions to be performed regardless of when they are given.

One problem with modular systems is that each module will need to obtain at least some command and control information from the user, directly or indirectly; some modules will spend much of their time 'talking' with the user. If the developer of each module designs and implements his own menus and command retrieval scheme, users will rapidly become confused about what to do and when to do it. User energy is wasted in determination of the 'context' they are currently in, and the proper actions to take in that context.

To avoid this problem, ALIAS has only four basic menu formats. All of these display reminders of at least a few of the command codes which may be given. Further, common functions such as 'Help!' or 'Show me the last menu again' are always commanded by the same codes. Once a user has mastered the basics of ALIAS, the mechanics of operating most parts of the system have also been learned.

Developers of new modules should make use of the menu-processing services provided by the System Core. If

for some reason this is not possible, command code conventions and menu formats should be observed.

A consistent user interface supports the design goal of usability, minimizing the amount of specialized knowledge necessary to operate the system, and thus maximizing the amount of attention which can be given to substantive tasks.

2.3 IMPLEMENTATION METHODS AND STANDARDS

This section describes methods used to achieve system goals and to conform to design principles. Many of these methods are stated as standards to which extensions of ALIAS are expected to conform; such statements are set off in the text for emphasis and easy reference.

2.3.1 Integrated System

Rather than conforming to the modularity principle via implementation of each module as a completely separate program, ALIAS is organized as an integrated system consisting of a System Core on which application modules are hung. ALIAS appears to the user to be a single program. The System Core provides user interface services (menu display and command decoding) and scenario control/security enforcement services. The Core as implemented, allows modules to be written in any HP programming or procedure language.

Standards:

---All ALIAS modules shall execute under the control of the System Core, preserving the integrated nature of ALIAS.

---No modifications shall be made to the Core which will force future modules to be written using a single programming or procedure language.

These standards promote usability, expandability, and maintainability. The appearance of ALIAS as a single system relieves the user of the need to know precisely how the system is

structured (e.g., the need to remember the names of a large number of programs in order to run each one), and also helps to ensure a consistent user interface. A well-defined core providing many services reduces the costs of expansion, and the uniformity of module interfacing promotes maintainability.

2.3.2 Regeneratable Menu-Oriented Command and Control System

The System Core is built around a special menu-oriented command and control system which displays choices to the user, accepts his commands, and executes modules. The command system consists of two FORTRAN programs and several data files and relations. The first program, the menu generator, reads an ASCII input file in which the ALIAS system creator has placed descriptions of the menus ALIAS is to display and of the actions that user choices are to invoke. The generator fills the data files and relations with data which the second program, the run-time command system, uses to display the menus. The number of menus that the system may display is basically unlimited (the limit is now set at 200, but this can be increased), and the generator may be run as often as desired.

There are three basic types of menus available. One, called a choice menu, contains numbered options. Specifying one of these numbers to the command system will either cause a module to execute, or will cause another menu in the hierarchy to be displayed. The second, called a parameter menu, displays current settings of program control data values. The third, called a list menu, displays a list of entities on which a module may operate, and an on/off status for each entity (see Section 8.3 for a full description of the nature of each menu).

Standards:

---All ALIAS modules shall be executed in response to user choice of an appropriate numbered option on a menu system choice menu. Appropriate additions to the displayed menus should be made when a new module is implemented.

---The use of independent command and control menus within modules is to be avoided. Where a module is composed of several independently executing parts, each part should have a separate option number on the choice menu.

---Modules should be structured to obtain user-adjustable initialization data and control data from parameter menu and list menu settings. The practice of prompting the user for settings at the beginning of module execution is to be avoided. The practice of assigning 'standard' values to such variables, in effect making their adjustment impossible for the user, is to be avoided. Appropriate parameter and list menus should be added when a new module is implemented.

These standards, and the existence of the configurable menu system, promote usability, flexibility, expandability, and maintainability. The consistent and user-friendly interface provided by the menus promotes usability; the ability to set module control parameters once and then not have to worry about them again until a change must be made promotes flexibility at no cost in usability. The preprocessor/run-time architecture of the menu system, allowing the rapid reconfiguration of displayed menus, is a primary reason why ALIAS is an expandable system. The substantial command processing services provided by the menu system promote maintainability by reducing the total amount of system code devoted to such processing.

2.3.3 Actual Implementation Along Functional, Modular Lines

In conformance with the modular architecture of ALIAS as a whole, most system applications are in fact implemented as separate, modular program units. The only code pertaining to specific modules to be found in the System Core itself is call statements and/or process activation statements. No data structures are shared by modules except for the central integrated data base, and changing of data base values by the modules is strictly limited.

When appropriate, modules are further broken down into separate functional units, each separately implemented. For

example, the Force Impact module is composed of a force level report generator and a battle group report generator.

As noted in Section 1, an ALIAS module may take many forms, ranging from a simple procedure file containing DBMS commands to a set of subroutines to a completely separate program run by ALIAS as a son process. Regardless of form, every effort should be made to separate the storage of the modules, particularly source code, in order to facilitate management of the system. Routines which support more than one module should be placed in an ALIAS utility library.

Standards:

---System modules shall be implemented and stored on the system as separate units, regardless of whether they actually run as separate programs or are linked into the System Core.

---Modules shall be broken down into lowest-common-denominator functional units, with the user having control over the decision to execute a functional unit or not where feasible.

The major benefits of modularity were mentioned as part of the discussion of Design Principles. Use of modular architecture in implementation as well as design, further promotes maintainability and portability by making the system easier to manage.

2.3.4 Use of Relational DBMS Technology and Data Dictionary

The ALIAS data base is implemented and managed by a fully relational data base management system. Relational DBMSs may be distinguished from the older network and hierarchical types by the fact that they allow the representation of more data relationships, support dynamic queries by the specification of intuitively appealing logical conditions, and may be implemented and changed at any time by nonprogramming users without resort to complex unload-regenerate-reload procedures.

The data base is described in a comprehensive data dictionary which is implemented as a small data base within the larger one. The dictionary is used to provide on-line help and to support data base integrity checking.

Standards:

---The ALIAS integrated data base should be implemented and expanded using a relational DBMS.

---Changes to the data base structure should be recorded in the data dictionary. Such changes should be made by or under the supervision of a Data Base Administrator.

---New modules should conform to the field value conventions contained in the legal data values reference library. This library should be expanded as appropriate.

The relational DBMS contributes significantly to the flexibility, expandability, and maintainability of ALIAS. Flexible partitioning of the data base into multiple scenarios, some of which share the same data, depends on relational technology. Relational systems allow the user to make exceptionally sophisticated ad hoc queries and report generation requests with a minimum of commands. The ability to structure the data base according to the data involved, rather than according to the needs of the DBMS, enhances this query capability and makes application modules easier to design and implement. The fact that one part of the structure of the data base can be expanded and changed without affecting any other part makes expansion and maintenance much less costly.

2.3.5 User-Friendly, Semi-Automated Data Updating System

The ALIAS Data Base Updating System (DBU) presents the user with a variety of fill-in-the-blank screens which will display the current contents of the data base and allow changes to be made. The system provides several methods for moving from screen to screen, for displaying data, and for making changes. In addition to these overt functions, it does extensive checking to ensure that the integrity of the data base is maintained before

it carries out any update request. As part of this checking, the DBU ensures that the data for any legal update is placed in all appropriate data base relations, thus ensuring that the problems caused by partial updates are avoided.

The DBU is implemented using the BUILDER procedure language.

Standards:

---All changes or extensions of the ALIAS data base shall be accompanied by appropriate changes to the DBU.

---DBU changes should always take into account scenario security and data base integrity requirements.

---Appropriate data dictionary changes should be made to ensure that the DBU operates properly.

The DBU is a key part of the ALIAS system, playing a role in support of all of its goals and design principles. Its screen-oriented, user-friendly nature and extensive error checking supports ease of use by minimizing the amount of knowledge required and by providing effective visual cues. It helps give the analyst high control of system operations by allowing data base searches and changes to be precisely targeted. It is vital to output consistency, since it makes sure that the key structure of the data base (which allows data from many files to be joined together) is never damaged by improper updates. It is implemented using the BUILDER interpretive programming language, making it easy to modify screens to reflect modifications in data base structure.

2.3.6 On-Line Help

At almost any point, an ALIAS user may obtain an extensive array of help by typing a '?'. Much of this help is provided by the menu system and the DBU. Each of these has a discrete help subsystem which explains system commands available, the purpose of ALIAS and/or the DBU as a whole, the purpose of the current menu or screen, the meaning of any menu option or screen field,

how to operate the system, etc. This help is all 'data driven'; that is, the text may be modified as appropriate. Menu system help text is part of the input to the menu generator, and is located and displayed automatically by the run-time menu system. DBU help is contained in special help screens. Modules which interact a great deal with the user, such as the assigner, have their own help subsystems.

Standard:

---On-line help shall be provided for each new ALIAS module, and shall be updated as appropriate when existing modules are changed. Appropriate menu system help text shall be provided at a minimum, and modules which require substantial user interaction are expected to have help subsytems incorporated. This help is to be data driven (the prthlp utility routine supports this requirement).

On-line help is an important part of making ALIAS a usable system. It reduces the burden of training novice users, and provides reminders for all users.

2.3.7 Programming Language and Use of Programming Standards

FORTRAN was selected as the primary ALIAS programming language for four reasons. First, programs written in FORTRAN are more portable than those written in any other language, particularly when written to conform to the ANSI 1977 Standard. Life cycle costs of a program likely to be converted to new host hardware one or more times are substantially reduced if FORTRAN is used. Second, FORTRAN is one of the few languages suitable for writing large systems. Many languages place restrictions on the modularity of code which can be achieved, for example by restricting the programmer's ability to separately compile subroutines for later linking (PASCAL often has such a restriction). Third, FORTRAN is terse as higher level languages go, requiring far fewer statements on the average to accomplish the same task than, e.g., COBOL (COBOL's advantages in the areas of data loading and sorting are negated by the use of RELATE in the

16

case of ALIAS). Studies have shown that programmer productivity in lines of code per day is relatively constant; thus, systems written in terse languages cost less to implement. Fourth, FORTRAN is fairly efficient in terms of execution speed.

The ALIAS environment provides excellent support for the FORTRAN programmer; it has large libraries of utility subroutines, which include fairly full implementations of a stack data type, a date data type, and several memory management subsystems. Many procedures which ease the burden of source code editing, compilation, linking, and debugging are available.

Standard:

---Implementers of new modules should choose FORTRAN as their high level programming language unless special circumstances override the above considerations.

Programmers writing in FORTRAN are required to observe the following standards; should programs be written in another language, it is expected that the spirit if not the letter of these standards will be observed.

2.3.7.1 Modular Coding

Standard:

---Programs are expected to have a modular structure following functional lines. Each subroutine should perform one and only one major function. Such a structure is generally characterized by one or more levels of 'executive' routines which supervise the operations of 'workhorse' routines.

Modular program structuring and coding reduces both implementation and maintenance costs. Debugging time is reduced since testing done along functional lines will point to errors in specific routines; most debugging time is spent searching for the source of an error. Maintenance costs are reduced because the time required for a maintenance programmer to understand a given

program is reduced. Given an understanding of a program's function and basic algorithms, an understanding of its operations follows rapidly. Programs structured according to the whim of the author are much harder to understand.

2.3.7.2 Small Subroutines

Standard:

---Programmers shall make every effort to keep subroutines smaller than seventy-five lines of executable source code (no limit on documentation or data declarations).

Few routines fully in conformance with the modularity standard will require more than seventy-five executable lines. A routine grown beyond this length generally should be organized into an executive and several workhorses.

2.3.7.3 Extensive Documentation in Each Routine

Standard:

---Each subroutine shall contain two types of documentation:

1) An abstract or description at the top, including author's name, description of purpose and method, and descriptions of data structures used, including the purpose of each important local variable. Particular attention should be paid to a clear statement of purpose and a complete description of method, including underlying concepts.

2) In-line documentation describing the operations of the routine at each major step.

Full documentation, especially of purpose, method, and data structures is crucial to the maintainability of the code.

2.3.7.4 Separate File Residence for Global Data Structures

Standard:

---The code defining FORTRAN common blocks, record data types used in more than one routine, and specialized global data types such as the date data type shall be maintained in source files separate from any subroutine making use of the data structures. This separate source code shall be inserted in appropriate subroutines just prior to compilation by an 'include' preprocessor (see Section 6.4.4).

If source code for global data structures is placed in each routine using them, and the structures must change, the editing requirement can become enormous. Separate maintenance of a single copy of the data structure source code removes this risk, as well as the risk of incompatible declarations in different routines due to typing errors, etc. Reliability and maintainability are enhanced.

2.3.7.5 Debugging Support

Standard:

---Debugging statements should be permanently placed at appropriate points in each subroutine, activatable by setting one of a set of controlling global logical variables to .true. (global array lprnts). It is suggested that such statements be the first and last executable statements of routines which primarily process arguments, and that they report on the incoming and outgoing argument values.

Permanent placement of debugging prints allows errors to be traced down by maintenance personnel with a minimum of fuss; this is particularly helpful when no symbolic debugging package is available. It almost always speeds up debugging at the time of initial implementation as well.

2.3.7.6 Explicit Typing of All Variables and Arrays

Standard:

---All local and global variables shall be explicitly typed. That is, each variable shall be explicitly declared to be integer, real, character, etc. FORTRAN's default typing of variables according to name should not be used.

Explicit typing makes code reading easier for maintenance personnel, and reduces the number of errors caused by improper declarations (such errors can be subtle and escape notice for a long time).

2.3.7.7 Minimal Use of Secondary Entry Points

Standard:

---The use of the secondary subroutine entry point feature of FORTRAN should be avoided. Use of secondary entries to initialize local variables or to allow processing of variable numbers of input arguments is acceptable.

Secondary entry points increase the number of paths through the code and degrade program modularity.

2.3.7.8 Formatting of Code for Readability

Standard:

---Source code should be formatted for readability. Indentation of loops and conditional processing sections and the setting off of major sections by several blank lines is recommended.

The job of the maintenance programmer is eased if the general flow of execution of routines is indicated by their formatting.

2.3.7.9 Error Handling

ALIAS divides errors into two classes. User errors, such as entry of an invalid command code, cause a message to be displayed but does not terminate the ALIAS session. At most they may terminate processing by a module, but this occurs only rarely and for unrecoverable errors. System errors, which occur on detection of hardware or software failures, terminates system processing with a message naming the routine in which the error and the nature of the problem was detected. These messages are meant for maintenance personnel and will seem cryptic to ordinary users. System errors will terminate module processing at a minimum, and may terminate the ALIAS session. Examples of causes of system errors are insufficient virtual memory to load data, absence of required data values in the data base (bad DB integrity), and overflow of module internal table limits (e.g., too many ships in one yard when running the assigner, etc.). System errors should receive the immediate attention of maintenance personnel.

Standard:

---Ordinary mistakes by the user, such as entry of an invalid command or data value, should result in an informative message and redisplay of the original prompt. Unrecoverable errors, such as overflow of hardware/software capacity, should result in display of a message giving the name of the routine in which the error was detected and the nature of the problem, followed by a system abort.

Extensive checking for user errors, with informative messages and retry capability rather than aborts, facilitate ease of system use. Extensive checking for software/hardware errors, such as array overflows, improves system reliability.

2.3.8 Use and Expansion of Utility Libraries

Standard:

---Software authors shall make themselves aware of the contents of ALIAS utility libraries, shall make use of library routines in preference to writing new code duplicating functions already to be found in them, and shall add new functions to the libraries as appropriate.

The habitual use of utility systems when writing traditional programs or BUILDER procedures (libraries exist for each type of author) reduces development and conversion costs by reducing the volume of code which must be written and increasing its modularity, and improves reliability since utilities are (must be) thoroughly debugged. Utility-oriented programming is a programmer attitude more than anything else; programmers should make themselves aware of the functions already available in the form of utilities, and should be on the lookout for new candidates for the libraries as they write.

2.3.9 Use of Interfaces to Buffer System Dependencies

Standard:

---The use of hardware-system-dependent features shall be minimized, and calls to those which are used shall be buffered by interfaces designed to maximize system portability.

ALIAS places a significant burden on any hardware host, and therefore must usually make use of many operating-system-dependent services. Also, ALIAS relies heavily on the services of a relational data base management system. The logical organization and formats of calls to such functions are sufficiently varied between operating systems and DBMSs that conversion from use of one to another is a job of great magnitude if direct requests for the services are spread throughout the main ALIAS programs and procedures. If, however, calls for the services are funneled through libraries of interface routines, the conversion becomes quite manageable, since only the interface library need be changed. Attention to functionally-oriented design of the libraries (in contrast to designs which implicitly duplicate system-dependent features) further eases the task of conversion.

ALIAS calls to operating-system-dependent features are all buffered through routines in the utility libraries. For example, use of HP extra data segments for fast-access data storage is buffered through the ...mem system (inimem, finmem, putmem, getmem), which on a fully virtual system could be easily converted to use a large array for storage purposes.

Calls to the DBMS are buffered through the DBIF interface library, designed to function with any cursor-oriented relational DBMS.

2.3.10 Use of DBMS for Data Storage

Standard:

---ALIAS system data should be stored in the ALIAS integrated data base whenever possible.

Use of non-data base files for storage of ALIAS data, particularly input data, should be considered only for data that is never directly accessed by users, and (at this writing) for

textually oriented data (RELATE has few facilities for storage and manipulation of textual data). Full use of an integrated data base provides a consistent interface for all modules, allows maximum accessibility of data, and makes system maintenance easier.

2.3.11 Attention to Multi-User and Multi-Processing Considerations

Standard:

---Authors of new or modified ALIAS software shall take care not to implement features which are unreliable in a multi-user system or which could cause such a system to fail, and shall make every effort to use the multi-processing capabilities of the host computer to improve response time for the user.

ALIAS is a multi-user system; careful attention must be paid to the intricacies of providing reliable multi-user services. For example, no user should ever be given exclusive access to a common file or relation; protection against data base update deadlocks and errors caused by simultaneous update attempts must be provided. In most cases, this means careful attention to scenario security.

ALIAS is an intense user of computer services; on a busy computer, response to user requests can lag. When possible, users should be able to choose to have functions performed 'in the background' by a batch job or concurrent process, freeing their terminal for other operations.

2.3.12 Use of Automated Documentation Production Support

Standard:

---System developers shall take care to maintain, use, and where appropriate extend the automated documentation production services that are now part of the ALIAS system.

Complete and current documentation is important to the efficient maintenance and operation of ALIAS. The system now has some on-line program documentation and some automated support for program documentation production. A small data base describes program subroutines in existence, and a processor exists to automatically plot calling tree diagrams from FORTRAN compilation listings. These facilities should be kept up to date, used regularly for updates of paper documentation, and extended where possible.

2.3.13 Maintenance of Production and Development Systems

Standard:

---Changes and additions to ALIAS shall be tested on the development version of the system before being moved onto the public, production version.

The ALIAS environment now supports a limited dual-system capability. Test versions of the main menu system and system modules may be generated and used by giving the DEVELOP command during a session, while ordinary users may simultaneously run the main system. Both systems use the single, integrated data base, however, so changes to it must be made and tested with particular care and timeliness.

2.3.14 Mnemonic Naming

Standard:

---ALIAS system units (files, relations, udcs, programs, subroutines, screens, and variables) shall be named mnemonically following their purpose or function. Random naming and names without informative content are to be avoided. All fortran names shall be six characters or less (for ANSI standard compatibility), and all data file and relation names shall be seven characters or less.

Informative naming is crucial in a system of the size of ALIAS; management of such a system is difficult enough without the need to constantly consult references to find the purpose of 'subl', 'varl00', common block 'quincy', etc. The limits on name

lengths often make mnemonic naming difficult, but are necessary for portability and because of hardware-imposed constraints. See Section 6.2.2 for a full list of naming conventions.

2.3.15 Use of Templates

Templates are skeletons or frameworks which already have in place some of the information necessary for a particular purpose. For example, the ALIAS subroutine template abs.template contains the basic structure for a FORTRAN subroutine, including the documentation abstract. The programmer need only fill in the blanks in the abstract with a description of the routine he is about to write. The existence of a "form" to be filled in saves time and promotes standardization.

Standard:

---Templates or standardized "forms" containing the basic outline of commonly created procedures shall be maintained in a special library on the ALIAS host, and shall be used as the starting point for creation of such procedures.

Templates are a tool which lessens the burden of conforming to the documentation and modularity standards.

3.0 ALIAS DATA STRUCTURES AND DATA FLOW

This section discusses the data structures used by ALIAS. 'Data structure' is a term used to refer to a method of organizing data for storage and use. In fact, all data on computers is ultimately organized as streams of bits (binary 1/0's). Higher-level methods of organization, superimposed on this basic method, provide more intuitively appealing and useful tools for the programmer and user.

Section 3.1 discusses each major ALIAS structure in turn, while Section 3.2 summarizes their usage in the system and the general flow of system data.

3.1 CONCEPTUAL DATA STRUCTURES AND THEIR ALIAS IMPLEMENTATIONS AND USAGE

3.1.1 Relations

3.1.1.1 Definition

'Relation' is the formal term used in relational data base theory for a grouping of related data elements. In practical terms, a relation may be thought of as a two-dimensional table of data, with rows and columns. Figure 3-1 displays three sample relations.

Somewhat more formally, a relation is a set of tuples, or records, of the same type. Note that there is no intrinsic ordering imposed on the tuples (though in any practical relational system, the user is allowed to impose the order of his choice). Tuples are rows of data, composed of any number of fields (columns or elements).

The definition is derived from the general mathematical definition of a relation, which, if the reader is interested, is as follows: for some positive number N, suppose there exist N sets of values named S_1, S_2, ..., and S_N. R is defined to be a relation on these N sets of values if R is a set of N-tuples

Figure 3-1. Three Sample Relations

SCENARIO	PROGRAM	CLASS	STARTHULL	ENDHULL
MAIN	CVN-67:2	CVN-67	71	75
MAIN	DDG-51	DDG-51	51	999

SCENARIO	CLASS	LENGTH	BEAM
MAIN	CVN-67	1092	128
MAIN	DDG-51	565	75

SCENARIO	CLASS	HULL	AWARD	DELIVERY
MAIN	CVN-67	75	1/1/1990	1/1/1997

(or rows of values), each of which has as its first element a member of the set S_1, its second element from the set S_2, and so on, so that the last element of each N-tuple is a member of S_N.

3.1.1.2 Implementation

ALIAS relations are created and used through the services of the RELATE data base management system. A new relation is created by giving a command which names it and which describes the attributes of each of its fields (columns) (name, data type, size, etc.). ALIAS relations may have an essentially unlimited number of fields (up to a few hundred) and tuples (records).

Each relation is stored on the HP computer as twin files: one file holds the structure of the relation (in its file label area) and its data records, while the other file holds indexing (ordering) information. RELATE places data in relations in no particular order, but allows ordered retrieval according to indexed field values. It does this by storing some data values, binary tree data, and pointers in the indexing file. Both file types may be manipulated by the RENAME and FCOPY MPE utilities without harm, but no attempt should be made to write data into them or to recreate them with any other non-RELATE program. It is possible to read the data files directly using a fortran program if great care is used, but this is seldom desirable or efficient. Note that if files are renamed or copied, BOTH data and indexing files must be processed.

Fields are ordered from 'left' to 'right' in the relations (and by default on output) according to the order in which they were described during the relation-creation step. The DBMS places two additional fields, which it uses internally, at the end of each tuple: a record number, and a 'flag' field which indicates whether or not a record has been deleted.

Relations may be actual or virtual. A virtual relation is one composed of a subset of the fields in an actual relation, or one composed of fields/tuples from two or more actual relations. The ability to create and use such virtual relations without creating actual files and copying data into them is the principal source of the great power of relational data management.

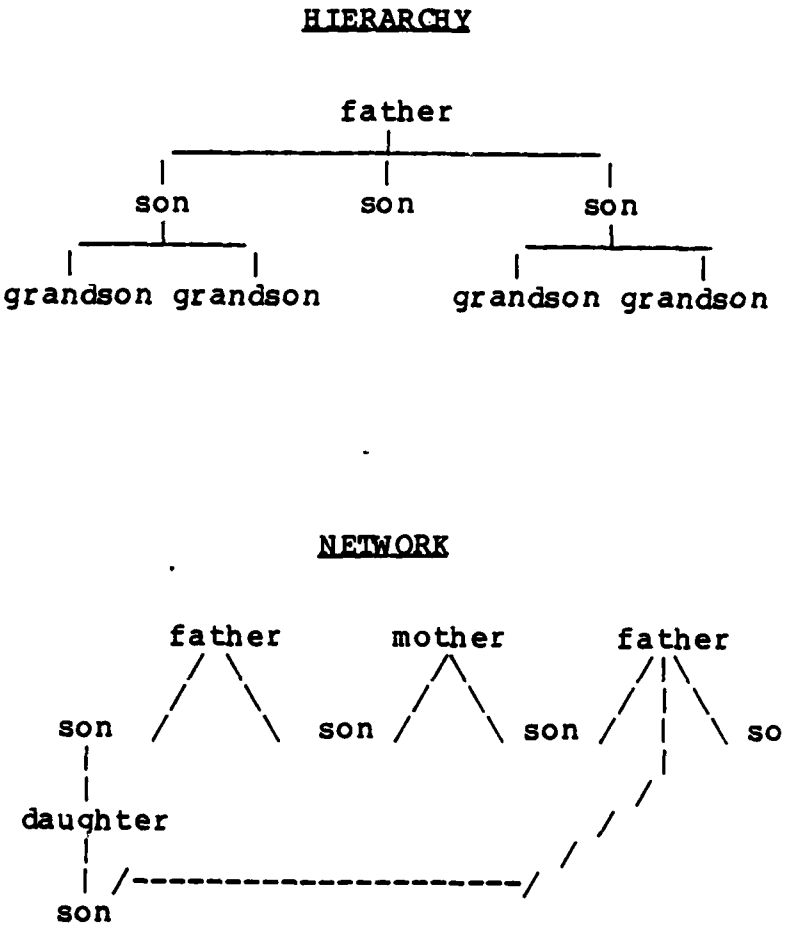
3.1.1.3 Usage

Relations are in general useful for data storage, since the sorting and searching functions provided by RELATE allow programs to accomplish tasks using much less code than would otherwise be required. Storing data in relations also makes the data directly accessible to users running RELATE interactively. If data were stored in binary files, special programs would have to be written to allow interactive user access. Ad hoc queries and report generation are supported by use of relations for storage. Most ALIAS system data is thus stored in the relations which are collectively referred to as the data base.

Relational data management supports implementation of a large variety of other data structures, such as hierarchies and networks (Figure 3-2 displays an hierarchy and a network). Much of the ALIAS data base is effectively implemented in a network form; the Data Base Updating system (DBU) makes sure that data in certain relations is matched by companion data in other relations. Figure 3-3 shows an example: records in the NCJODAT.PROJJ relation, which contains fields called CLASS and YARD, may not be added unless their values for the CLASS and YARD fields have matches in the SHDESC.MISCJ and YARDID.YARDS relations, respectively. There is thus something of a master-slave relationship, where SHDESC and YARDID are masters and NCJODAT is the slave: this is known as a network structure.

RELATE will not itself enforce the data update rules necessary to a functioning network (e.g. it will not delete all NCJODAT records for a given yard when that yard's name is deleted

Figure 3-2. Hierarchies and Networks



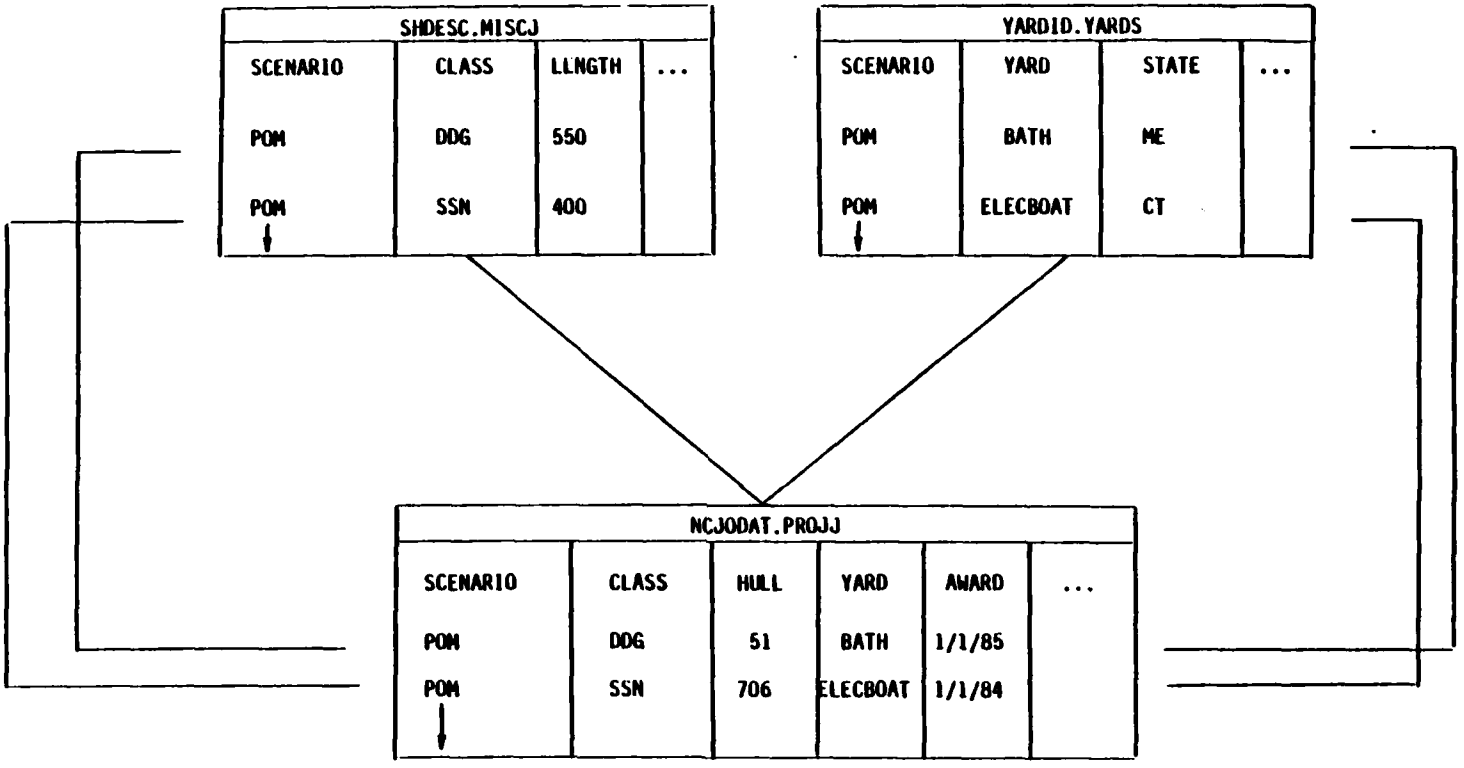


Figure 3-3. Sample Network

from YARDID); such enforcement must be explicitly coded into any modules which update the data base.

Another network-oriented structure which is so commonly used that it requires separate mention is the 'comment' facility of the data base. Every data relation has an associated relation which holds text comments associated with each tuple in the data relation. This companion to the main data structure allows users to make notes to act as reminders concerning the rationale of updates.

In addition to containing data base data, the ALIAS system also uses relations to hold some system data (such as security priveleges), to implement the data dictionary which describes the data base, and to contain some system documentation and information used to manage the environment.

3.1.2 Files

3.1.2.1 Definition

A file is a stream of bytes which may be referenced by name. In practical terms, it is a set of records of fixed or variable length which reside on mass storage (disk or tape). Unlike relations, the structure of each record is not necessarily constant: that is, the first byte of each record need not contain the same variable.

3.1.2.2 Implementation

On the Hewlett-Packard 3000, files are divided into types; when a file is created, its type and maximum size must be specified, and this specification may not later be changed. Major attributes of a file which determine its type on HP include whether it contains data in binary or ASCII form, the size of its records and whether they are of fixed or variable length, and the existence of subdivisions within the file to hold different types of data (such as the internal division of object code (USL) files

into directory and code areas).

A programmer working on the HP 3000 must be aware of the procedures for creation and use of a wide variety of files (program files, object code files, graphics storage files, etc.), but the ALIAS system itself makes direct programmatic usage of only two basic types of file at this time. These are ASCII files which may be processed by the editors (generally of fixed record length, although variable-length records are used in a few places), and binary files (again, usually with fixed-length records). These two types of file may be processed using ordinary fortran I/O.

3.1.2.3 Usage

ALIAS uses files to store permanent or working data which is not appropriate for storage in relations.

Table 3-1 lists the (non-relation) files now used by ALIAS. Certain modules use fixed-length-record ASCII files (standard editor files) for inputs, and optionally writes outputs in printable form into such files. The menu system uses variable-length-record ASCII files for storage of command files.

Binary files are used as temporary storage by modules (e.g. bufasn and cmnasn), for permanent storage of system data (e.g. relsnl), and to transfer data between one program and another (mnug output, mnur input files). Binary files may be accessed more efficiently by programs, but should never be used for data which users will wish to modify (unless custom programs are written to handle the modifications).

TABLE 3-1

STANDARD FILES USED BY ALIAS

GROUP	FILENAME	MODULE	STATUS	COMMENT
.CMDFIL	ALL	MNUR	PERM	MENU SYSTEM COMMAND FILES
.FMTFIL	ALL	FLRP,BGRP	PERM	REPORT FORMAT CONTROL FILES
.MNUFIL	MCMENU	MNUR	PERM	
	MHELP			
	MNUROOT			
	MYDESC			
	MPMENU			
	MPXREF			
	MTEXT			
.RPROCS	NEWHUL	ASGN	PERM	
.SCREENS	DBU	DBU	PERM	
.SYSRO	HLPUSN	ASGN		
	INIUSN	ASGN		
	MODHLP	VARIOUS		
	SYSHLP	MNUR		
	SYSMAP	MNUR		
.SYSRW	RELSNL	SNRO		
USER	BUFUSN	ASGN	TEMP	
	CMNASN	ASGN	TEMP	

3.1.3 List Memory

3.1.3.1 Definition

A list is a set of data items (each composed of one or more data elements). The set may have a fixed or variable number of members. (An array is an example of a fixed-length list of items). List memory is an ALIAS utility subsystem which provides storage and retrieval services for variable-length lists.

A common problem in programming is determination of the amount of storage to allocate for each set of data items a program will use. Often, the number of items is expected to vary from one run to the next (for example, the number of menus ALIAS can display is not fixed, but is determined by the system creator's input to program mnug). On one hand, each set should have enough storage for the maximum possible number of items; on the other hand, this can lead to creation of lots of very large arrays, all of which are never filled to capacity at one time.

List memory is a storage pool into which the programmer can dump data and receive a pointer in return; later, the data can be retrieved using the pointer. Small working arrays can be used to hold one or a few items from each data set. This allows much more efficient use of memory at a moderate cpu processing cost (for the storage-and-retrieval cycles).

3.1.3.2 Implementation

List memory is implemented as a library of subroutines to accomplish storage, retrieval, and release of used space, with special routines available for each data type. See Section 10 for descriptions of what each routine does. Note that inlist must be called before any other list memory routine.

3.1.3.3 Usage

List memory is not used by any ALIAS program. The data memory limitations of the HP mean that the list memory pool may be no larger than 64K bytes, and storage allocated to the pool is not available for usage by other program arrays. No applications have yet appeared which justified adding explicit paging of the pool to the list memory subsystem. The subsystem is operational for applications which require only a small storage pool, though, and may be used by linking in its object code segment.

3.1.4 Memory Manager

3.1.4.1 Definition

Until the advent of computer hardware and operating systems supporting virtual memory, one of the greatest challenges facing programmers of large systems such as ALIAS was fitting the necessary instructions and variables into a main memory of limited size. Demand-paged virtual memory made this problem transparent to the programmer: part of a program's instructions and working space are stored in main memory, and the rest in a 'swap' area on disk. The operating system automatically detects the need for data not in main memory, and brings the needed item in from disk. For various reasons, implementation of this capability at a level above that of hardware/operating system is often a very complex task.

The HP 3000 does not support fully virtual memory: it has an unusual architecture in which program instructions (code) are managed in a virtual fashion by the operating system, thus allowing the number of statements in a program to be unlimited. However, the variables the instructions act on may use no more than 64000 bytes of storage. A program which requires more than 64K of data storage must either be split into several programs, each requiring less, or its author must include explicit memory management procedures which store some of the required data on disk, and load it into memory as necessary.

The ALIAS memory management subsystem eases the burden of this latter method by allowing definition of any number of extra storage areas (each of any size up to 64K), and by accepting data for storage in these areas and retrieving it as necessary.

3.1.4.2 Implementation

The memory management subsystem is composed of four utility routines, maintained in the utlo library. They are inimem, putmem, getmem, and finmem. Inimem allocates a new storage area, putmem places data in it, getmem retrieves data, and finmem releases the area. See Section 10 for the format of arguments to the routines.

The MPE operating system offers a capability which makes it possible for this subsystem to operate quite efficiently. Instead of defining storage areas as files on disk, to be accessed by FORTRAN i/o, the areas are MPE extra data segments. An extra data segment is an area in system memory, managed in a virtual fashion, which may be as large as 64K bytes. Four operating system intrinsic subroutines, similar in purpose to the four utility routines, allow definition, use in storage/retrieval, and release of extra data segments. Since these segments are often stored in system semiconductor memory, data may be processed into and out of them at great speed.

Programs which use the memory extension subsystem are responsible for remembering the id code of each data area, and the exact location within each area that stored data has been placed in. Unlike list memory, the extension subsystem maintains no internal pointers; it merely buffers commands to MPE's extra data segment intrinsics.

3.1.4.3 Usage

Three parts of the ALIAS system are the principal users of the memory management subsystem at this time. The scenario

system stores scenario field key values for the user's current scenario for each data base relation in a segment; the assigner uses a segment for intermediate schedule record storage during its data base update processing stage; and the force level and battle group report generators use a segment to store the current page as output pages are constructed.

ALIAS implicitly makes use of many other extra data segments, since one such segment is used as a communication area with each RELATE son process. It is possible to specify the ID code of each such segment at process startup time to RELATE using the rdbinitx host language interface routine. It is VITAL that the id given for each such segment, and for each memory management storage segment, be unique. Use of the same segment id for multiple purposes will lead to irregular, mysterious aborts. Make sure to consult the xdseg.doc relation for a list of already-used segment id's before using the memory management subsystem to define new segments.

It is possible to make memory extension segments function very much like an array for data retrieval purposes by defining a fortran function which accepts an 'array element index' and uses this to calculate the proper retrieval location in the "array" (segment) (storage into the 'array' must still unfortunately be done by subroutine call). If in developing a module it becomes clear that memory usage must be reduced, and there are large arrays in the module, use of this strategy for storing the data in these arrays can make modification of module code relatively painless. See the scenario system code, particularly the snrlsn and snrlnm functions, for an example of this strategy.

3.1.5 Session-Global Data Structures Offerred by MPE

Multi-processing operating systems usually provide one or more means for processes belonging to the same user session to communicate with one another, and MPE is no exception. In addition to extra data segments (which may be accessed by more

than one process) and regular files, a few special file types for message sending and receiving are provided.

Extra data segments may also be used to communicate between processes which do not run simultaneously. That is, it is possible to run one program which defines and fills a segment, terminate that program, and then run another program which reads the segment. Once an extra data segment is defined, it remains available for the life of a user's job or session or until a process explicitly releases it.

Job Control Words (JCW) are similar to extra data segments in that they may be accessed by any process, and remain in place until log-off once defined. JCW are single-word (16-bit) entities which may be named and given an integer value (up to the 16-bit level of precision). Their names may be quite long, up to 255 characters. JCW may be defined, have their values set, and read either from within a program (using intrinsics) or interactively by the user (SETJCW and SHOWJCW commands). Their primary purpose within MPE is for use as variables in udc's and batch job stream files: they may be given as logical arguments to the MPE IF-THEN-ELSE construct, allowing conditional flow of execution of a job or udc. ALIAS makes some use of JCW in this context; JCW are used to store the mode switches which indicate whether a user wishes to run the production or development version of ALIAS.

JCW are also used for interprocess communication and session memory. BUILDER uses a JCW to store a code indicating a user's terminal type: this is why terminal type must be given to the BUILDER only once per session. The DBU uses JCW to communicate with its cursor management subsystem: settings of the SCREENSYS, NUMSWAP, CURSORNUM, and CURSORSWAP JCW tell this subsystem which cursor to swap in for DBU use.

3.1.6 Stacks

3.1.6.1 Definition

A stack is a data structure with the following property: data items may be placed into it only one at a time, and may be retrieved only one at a time in the order of their storage. To retrieve the next-to-last item stored, the last item must first be removed.

Stacks operate in a fashion quite similar to the tray holders found at the head of the line in many modern cafeterias. These holders have a shaft in which a spring-loaded platform moves up and down. The more clean trays placed on the platform, the lower it sinks, effectively hiding most of the trays inside the shaft. If the spring tension is just right, only one tray ever shows at the top of the stack. When a customer steps up and removes this tray, the platform moves up a little bit and presents another tray for the next customer; if the first customer puts the tray back, then the stack of trays sinks back down again. Removing the top tray is analogous to 'popping' an item off of a data stack; putting a tray on is analogous to 'pushing' an item onto a data stack.

Stacks are sometimes uniform in that they only handle one type of data: integers, real numbers, character strings, etc. Carrying the analogy further, a given stack can handle either oval trays or square trays only: the shape is designed into the shaft and spring-loaded platform.

3.1.6.2 Implementation

ALIAS has a fortran utility subsystem which implements an integer*4 (32-bit) stack data type. The routines involved are maintained in the utlr library, and all have names beginning with 'cs'; e.g cspsh and cspop push and pop data items onto/from the stack. The retrieval routines are typically implemented as

functions for ease of use in code. Note that csinit must be called before any other routine in the subsystem.

The subsystem is not a strict stack data type in that it allows certain unusual operations. The programmer may push or pop the top two items on the stack using a single call, and may read the top one or two items without popping them.

The subsystem allows use of only one stack, and only of the given data type. Extension to allow use of multiple stacks/data types would be fairly straightforward should this ever be desirable.

3.1.6.3 Usage

The stack data type is now used only by the System Core's menu display/command processing program. See Section 8.3 for a discussion of its role.

3.1.7 String Chains

3.1.7.1 Definition

A string chain is composed of one or more character variables which occupy contiguous space in storage, and which thus can be treated as a single variable for certain purposes. The necessity for this data type is somewhat unique to the HP 3000; it allows handling of strings longer than 255 characters (the variable/array element maximum on HP) and also as a memory manager of sorts. A string chain is managed as a storage pool; space in this pool may be efficiently allocated to varying numbers and sizes of strings.

3.1.7.2 Implementation

The string chain data type is implemented with several utility routines located in the utlr library; their names begin with 'chn', such as 'chnalo' ('chain allocate'). Note that chnini must be called before any other chain routine.

3.1.7.3 Usage

The chain data type is now used only by the Data Base Interface (DBIF) routines. It was implemented to solve the following problem: a service provided by the DBIF is final preparation and word-alignment of command strings (sent by DBIF callers) for passing to RELATE; a given DBIF routine may need to process up to three such strings, each of which may be up to 1500 characters long, and may need additional strings of the same size for buffers. However, the likelihood of all strings passed being so long is very small. For various reasons, the strings need to be placed in global storage. Placement of six 1500 byte strings in storage would use an unacceptable amount of very limited program data space.

Instead, the DBIF uses the chain utility to allocate sufficient storage in the global string pool for each string passed down, and for any buffers required. Use of a pool, just as with list memory, ensures that enough storage is available for the largest item without using up too much data memory.

3.1.8 Queues

3.1.8.1 Definition

A queue is a data type similar in some respects to a stack. It is 'linear' in nature, and data items may be stored and retrieved only one at a time. However, it is a First-In-First-Out (FIFO) type, as opposed to the Last-In-First-Out (LIFO) type of stacks.

Continuing the cafeteria analogy, a queue operates similarly to a cafeteria with a long narrow hallway between the food service area and the cashier. Servers 'push' people into this hallway with full trays; cashiers 'pop' them out at the other end. The narrow hallway prevents people from being served out of turn; a bend at the end of the hallway prevents the cashier from seeing any but the customer currently being served.

3.1.8.2 Implementation and Usage

A queue is now used only by the System Core menu display/ command retrieval system. The routines managing the queue are not written as utilities, but as an integral part of this system, and queues are thus not generally available as an ALIAS data type. The menu system uses the queue to allow typing of multiple commands on a line when in a list menu (multiple candidate numbers who statuses are to be flipped); the commands are stored in the queue and then processed in the order in which they appeared on the input line.

3.1.9 Records

3.1.9.1 Definition

A record may be composed of several data items, each of any type, but may also be treated as a single entity for certain purposes (usually storage, retrieval, and transfer). A tuple in a relation is a record: it is composed of fields, each of which may be accessed individually and may be character, numeric, etc. in nature; the tuple as a whole may also be operated on, however, for operations such as deletion. Figure 3-4a displays a record.

3.1.9.2 Implementation

In ALIAS programs, a record is an entity composed of one or more variables which are stored continuously. Because of the nature of fortran and computer adressing in general, this allows each variable to be accessed, and allows the entire storage area to be accessed as well.

Fortran provides three means of defining records. First, any common block is automatically a record, because variables and arrays in common blocks are always stored contiguously in the order of their appearance in the common statement. Second, variables not in a common block may be forced into contiguous storage by using the 'equivalence' statement to force their storage to be the same as appropriate elements of an array.

Figure 3-4a. A Record.

CLASS CHAR	HULL INTGR	JOBTYP CHAR	YARD CHAR	AWARD DATE
DDG___	55__	NEWCON	INGALLS	1/1/87

Figure 3-4b. Common Block Implementation of a Record.

```
common/recrd1/class, hull, jobtyp, yard, award
character class*8, jobtyp*6, yard*8
integer hull
integer*4 award
```

Figure 3-4c. Equivalenced Implementation of a Record.

```
integer recrd1(14)
character _______
integer hull
integer*4 award
equivalence (recrd1(1), class), (recrd1(5), hull)
1           (recrd1(6), jobtyp), (recrd1(9), yard),
2           (recrd1(13), award)
```

Figure 3-4b and 3-4c display the code required to implement the record in Figure 3-4a using each of these methods.

Third, records are effectively defined on read and write statements: data is transferred in the order in which variables appear in the statement. HP fortran allows such transfers to be made either to disk or to memory.

The common block method of record definition is the one most commonly used in ALIAS, for its ease and because it is most appropriate for the main use of records in ALIAS.

3.1.9.3 Usage

RELATE demands that data be transferred between a program and the data base in record form, and further that the records always be of type integer. Arguments to the DBIF routines which are the sources/targets of data transfers (these arguments are passed to the RELATE host language interface routines unchanged by the DBIF) are therefore almost always integer arrays which are equivalenced to the first variable in a common block; the variables in this block must appear in the same order as the field names passed to RELATE for storage/retrieval, and must be of the same type and size. RELATE uses the storage represented by the common block as a single entity, while the programmer may use each variable individually. Equivalencing to an integer array is done only to ensure that the record is word-aligned in storage, a condition which RELATE insists on (and which may not hold if the first variable in the common block is of type character).

The practice of implementing records by equivalencing variables directly to an integer array (without using a common block) should be avoided. The equivalence statements must name and locate each record variable in the array explicitly; changes to the record structure then necessitate many changes to the

record-defining equivalences. Use of common blocks accomplishes the same thing more easily and reliably.

Common blocks used as records are usually placed in their own include files, and are documented therein as being records rather than global storage (though of course they may function either way).

3.1.10 Parameter Data Structure

3.1.10.1 Definition

The System Core menus which display module control values, and which allow the user to change these values, are known as 'parameter menus', and the variables they display as 'parameters'. (These should not be confused with the variables defined within programs by the fortran 'parameter' statement.) ALIAS has a special data structure which contains the current values of these parameters during system execution, making them available to modules under certain conditions. The data structure is also used by the menu system to display the values, and to pass values to and from the relations in which the values are permanently stored.

3.1.10.2 Implementation

The parameter data structure is implemented in a record format to meet the requirements of RELATE transfers, as a storage pool to meet the needs of the menu system, and as individual variables to meet the needs of programmers. Three include files (see Section 6.4.4 for a discussion of include files) contain the structure: one contains statements defining common block /pvalue/, which contains a single 1000 word real array forming the storage pool; the second (pvdecl) contains type declaration statements which name and define the data type of each parameter (real, character, etc.); the third (pveqiv) equivalences each parameter to a location in the pvalue storage pool such that each menu's parameters are stored as a record within the pool.

Since the exact number and record location depends on the number and contents of the parameter menus, these three include files are written by the mnug menu generation program; they should NEVER be changed manually by programmers.

3.1.10.3 Usage

The menu display and command system loads the set of values for the current scenario from the parameter relations into the parameter data structure at system startup and each time a request to use a new scenario is made. Whenever a request for display of a parameter menu is made, the system calculates the storage location for each parameter on the menu in the pvalue array, retrieves the current data from this location, and converts it into displayable form. If the user changes a value, his update is placed into the proper pvalue location; the relation of the value is permanently stored and is updated (using the proper section of the pvalue array as the data record) when he exits the menu.

Since current parameter values are always present in this structure, the job of programmers wishing to use the values is made easier if they may extract them from the array rather than from the storage relations. Any module linked into the System Core, and any son-process module which calls the iniprc utility (and is started by the mrunp utility), may access the parameter data structure by simply including the three include files in its source code. Parameters may then be referred to as variables, using their names as specified in the mnug input file. Note that values in this structure should NEVER be changed, only referenced.

Since the parameter data structure will change each time the number and/or nature of menus displayed changes (i.e. each time mnug is run), it is necessary that routines including the structure be recompiled after each mnug run. Modules have not

been recompiled will often run, but the parameter names used as variables will be referencing improper parts of the pvalue storage pool. See Section 9 for a discussion of the procedures involved.

All ALIAS modules now use the parameter data structure. Typically, a module will include the structure into its initialization routine, and will execute code to transfer the parameter values into storage local to the module. This keeps the number of module routines which must be recompiled after each menu generation run down to one.

3.1.11 Date Data Type

3.1.11.1 Definition

ALIAS modules, and to a lesser extent the System Core, must do a good deal of processing of calendar dates. For example, the assigner's prime job is the display of ship construction schedules in a condensed form, and the generation of such schedules. To carry out this and other functions, it is necessary to be able to convert data from a form suitable for display (e.g. MM/DD/YYYY) to one suitable for calculations and for storage in the data base. Tasks like 'add one month to this date' or 'how many days between these two dates?' are not simple and require considerable code.

To support such operations, ALIAS has an explicitly defined date data type, complete with functions which allow operations to be carried out on the data type. Dates are effectively put on a par with real and integer variables. For those data types, various operations such as addition and truncation are supported by the compiler. The date data type implementation has support for all the operations which one commonly wishes to perform with dates.

The data type allows representation of any date between 1600 and 2300 A.D., roughly.

3.1.11.2 Implementation

Dates are stored in a defined bit pattern in an integer*4 variable. This pattern matches that used by RELATE when it stores dates in its 'D' format, which is used by convention throughout the ALIAS data base. The only ordinary fortran operations which should be performed on such a variable are assignments: that is, one integer*4 date may be set equal to another. All other operations are performed by utilities and statement functions. To access these utilities and functions, the programmer need only specify inclusion of the 'tddate.incl' include file in his subroutines; this contains all required declarations as well as source code for statement fuctions. Note that tddate must be included AFTER all other declarations in a subroutine, since it contains "EXTERNAL" statements.

Dates may also be stored in the form of three integer*2 variables (month, day, year), and as ten-character strings ('MM/DD/YYYY'). The data type utilities include means to convert between the three storage formats. See Section 10 for a listing of date processing routines and more detail on how they work.

3.1.11.3 Usage

The date data type is used throughout ALIAS; dates are never stored or processed using any other format or scheme. It is recommended that this convention be continued, for consistency and ease of system conversion.

3.2 SUMMARY OF DATA STRUCTURE USAGE AND SYSTEM DATA FLOW

Table 3-2 lists the usage of data structures mentioned above by major system components.

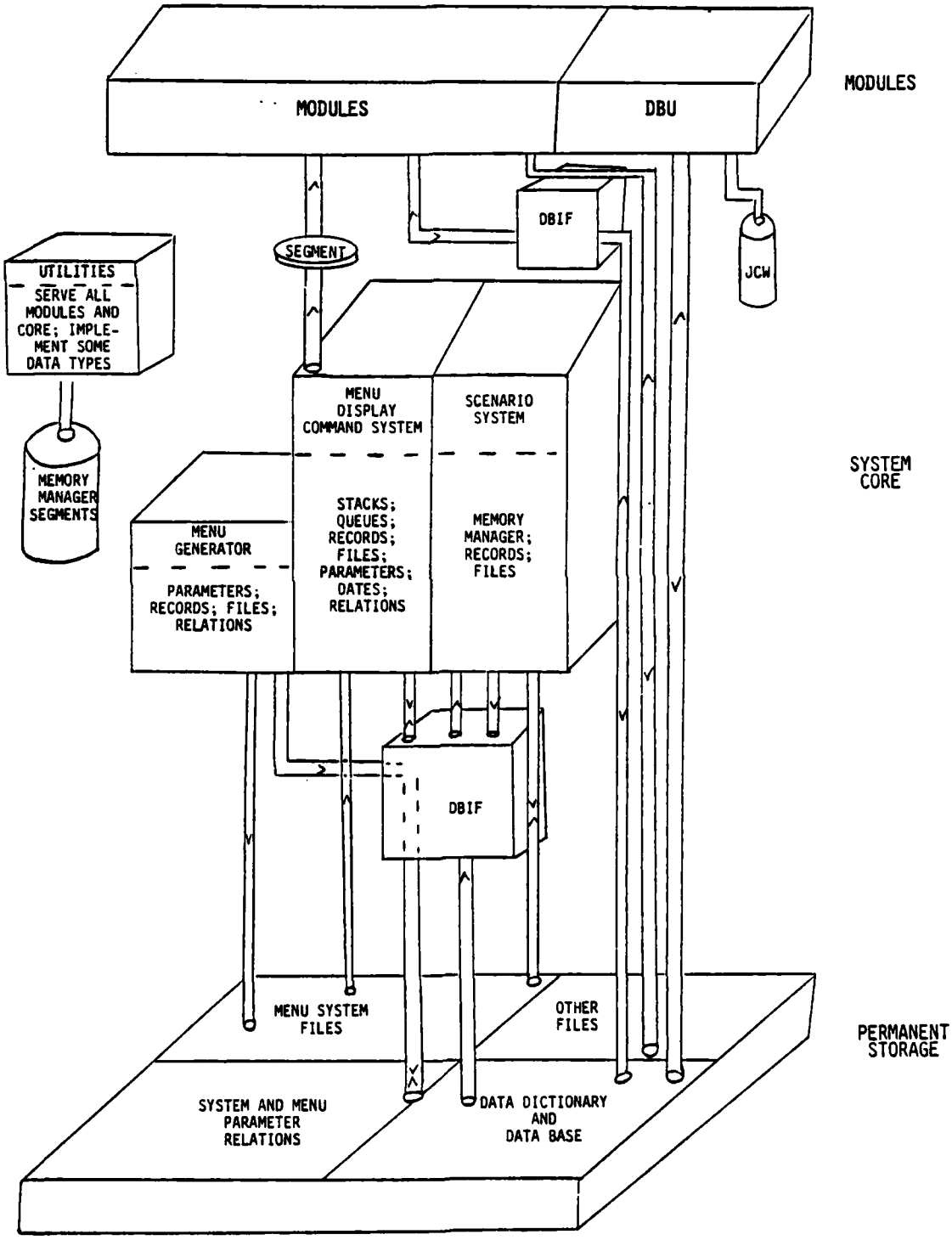
Figure 3-5 summarizes the flow of data through the system at a high level. Cylindrical figures indicate data storage;

TABLE 3-2

SPECIAL DATA STRUCTURE USAGE BY MAJOR SYSTEM COMPONENTS

STRUCTURE	COMPONENT				
	menu system	DBU	scenario	DBIF	app. modules
relations	x	x	x	x	x
files	x		x		x
list mem					
mem mgr			x	x	x
extra data segments	module comm. only	x			
JCW		x			mode control
stacks	x				
queues	x				
chains				x	
records	x		x	x	x
dates	x		x		x
parameters	x				x

Figure 3-5. ALIAS Data Flow



blocks indicate processors. Permanent storage is shown as a large pool on which the rest of the system draws. It is divided into four parts: data base relations and the data dictionary; relations used to hold system data and menu parameter values; files used to hold data describing the menus the System Core displays; and other files. Three smaller cylinders represent session Job Control Words, the extra data segment used to communicate Core data to modules, and segments used by the memory management subsystem. Processors represented include the System Core (partitioned into menu system and scenario system), the DBIF (shown twice for convenience of drawing only), the DBU, system utilities (all utilities, not just the memory manager), and the applications modules as a group.

The placement of figures is meant to be suggestive: all of the system rests on permanent storage data structures, and the modules (including the DBU) rest on top of the System Core. The DBIF and the system utilities serve both the Core and the applications modules.

Pipelines indicate major paths of data flow. Only the menu system uses its special files (and only mnug writes them, only mnur reads them); the DBIF buffers all access to relations except for that from the DBU; modules use both files and relations; the DBU is the primary user of Job Control Words; the modules receive data from the Core through a communications data segment.

4.0 SALIENT FEATURES OF THE HEWLETT-PACKARD 3000 COMPUTER

This section discusses some of the features of the ALIAS host computer. It is not meant either as an introduction to this computer or as an exhaustive guide to its operation and use; the reader is referred to the manuals published by Hewlett-Packard (HP) for that purpose. The features covered are those unusual ones that are of particular importance to ALIAS, and in certain cases those that are not well documented by HP. The discussion focuses not on how the features work, but rather on their impact on the ALIAS architecture and implementation. The intent is to support the understanding of how and why ALIAS works that this manual as a whole attempts to give.

4.1 HP 3000 AND MPE OPERATING SYSTEM ARCHITECTURE AND HIGHLIGHTS

The HP 3000 is a 16-bit minicomputer. It comes in various models, all of which are multi-user and multi-tasking, and which support the usual peripheral devices (disk, tape, printer, etc.). Its sole operating system, the Multi-Programming Executive (MPE), is interactively oriented and fairly flexible and friendly when compared to most mainframe operating systems. MPE and the 3000 include a number of advanced and powerful features; unfortunately, most of these features have limitations, not immediately apparent in HP's documentation, which limit their usefulness.

In the discussion which follows, the 3000 and its operating system will be implicitly compared with 32-bit minicomputers using the UNIX operating system or its close equivalent (DEC VMS, Prime PRIMOS, etc.). The 32 bit/UNIX architecture is the minimum level of support required to implement ALIAS in a natural and efficient fashion.

4.1.1 File System

The 3000's file system is organized into temporary and permanent systems. Each of these is further organized in a three-level hierarchy: at the base are individual files, which

users may create and purge with fair freedom and ease. Every file belongs to a storage 'group', which corresponds to a UNIX directory (ordinary users may not create and purge groups, however). Each group belongs to an account (user names also belong to accounts), each of which may have a fairly large number of groups. Any file may be referenced by giving all three level of its identification in the format FILE.GROUP.ACCOUNT; the group and account names will default to the "current" or "log-on" group and account if not specified. The current group and account may only be changed by logging on again: no change_directory command exists. Accounts may be created only by the System Manager; groups only by an Account Manager.

Although not nearly so flexible as a fully hierarchical file system, it has been possible to construct an adequate working environment for ALIAS by creating many groups in the SEA 90 account, each holding files for a particular purpose. Section 6 will discuss the ALIAS file system as implemented.

Every HP file has a number of unchangeable attributes attached to it, specified at the time of file creation, which define its type. Types range from fairly standard ascii and binary files to special-format library files which are internally partitioned into directory and data areas. Files must be explicitly created either by the user or a program before any data may be placed in them; when created, a maximum size for them must be specified. Most programs may only access a few types of files: for instance, the editors may not access binary files.

Temporary files may be saved as permanent files if desired; if they are not saved, they are purged as a user logs off. Temporary files are not as accessible as permanent files: a special utility must be run (LISTEQ) to get a listing of current temporary files.

a son process which is the job or interactive session. These 'main' processes may then create further son processes if they so desire. The process organization is rigidly hierarchical: a given process may communicate only with its father and its sons (though any process may communicate with MPE).

Sessions and batch jobs are virtually identical in terms of priveleges and operation: the major differences are that a spooled printer is always a job's standard list device, and that a fatal error during a job usually terminates the job. Many programs may be run in batch or interactive mode with no modification.

There are no interactive commands to support multi-processing. There is no way to run one program, 'detach' from it while it continues to run, and run another program simultaneously at the same terminal, as one can under UNIX. Programs may create processes and run them in the 'background', however.

Each process has its own data stack (thus large programs may be broken into separate smaller programs in order to circumvent data addressing limitations), and processes may share and communicate with one another through extra data segments.

4.1.4 Programming Support

The 3000 supports the fortran, cobol, basic, and pascal languages, and a 3000-specific language known as SPL (Systems Programming Language). Only fortran, cobol, and basic are resident on the current ALIAS host. Fortran and cobol programs are processed by the same linker; a calling standard is enforced which allows free calling of routines in either language by routines in the other, as long as attention is given to argument-passing conventions. The system also allows the construction of relocatable object code libraries (only one may be used during any given link), and segmented libraries which may be searched during program execution for any undefined

File aliasing is supported: the FILE command allows files (and devices) to be referenced by names of a user's choice.

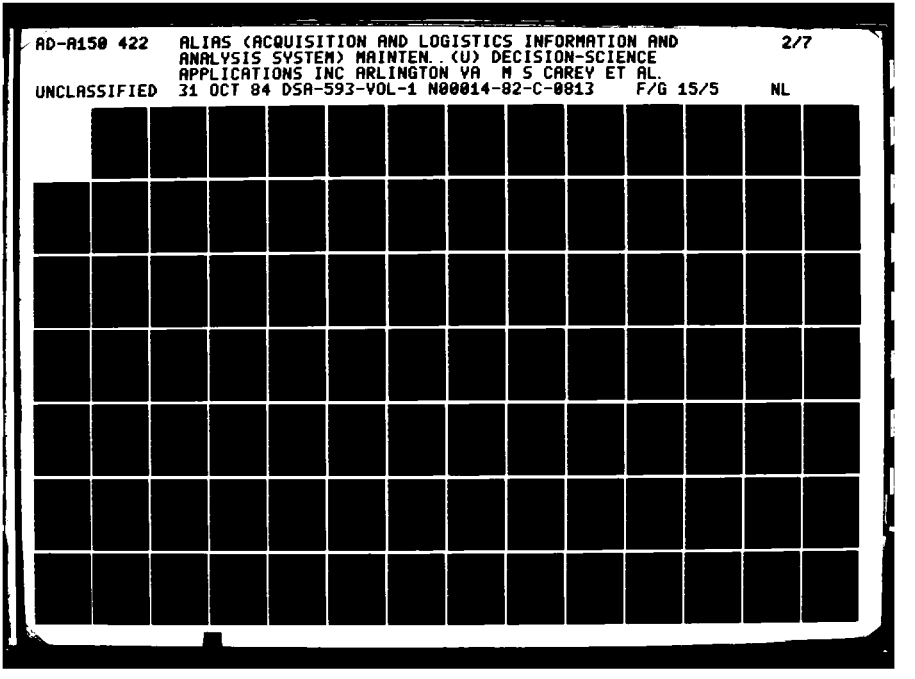
4.1.2 Addressing and Memory Management

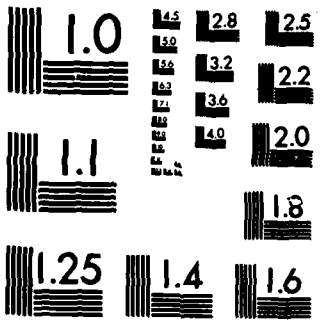
The 3000 is advertised as a virtual-memory machine by HP, but the meaning of 'virtual memory' as applied by them is not now commonly used in the computer industry. The 3000 has an unusual architecture in which program instructions are segmented and managed in a virtual fashion, while program data is not. No program may directly address more than 64K bytes (2^{16}) of data. Thus, programs may have as many lines of code as are desired (within quite broad limits), but are very limited in the amount of data they can operate on. Program overlaying is not necessary, but manual paging of data is.

The system as a whole is managed in a fully virtual fashion, using a segmented scheme. A 3000 may have up to several megabytes of semiconductor memory, as well as a swap space on disk of many more. When a program is run, MPE loads its first code segment from the program file into memory, and creates a main data segment (referred to as the "stack") which stores the variables and arrays which the program will directly address. Code segments may be no larger than 16K, while data segments may be no larger than 64K. As the program executes, additional code segments are loaded as necessary; the program may also create 'extra data segments', which it can use for temporary data storage much as it would a fast-access file. As memory fills up in a multiuser environment, segments not recently used are swapped out to disk. Note that code is fully sharable: that is, even if two users are running the same program, that program's code segments need by loaded only once.

4.1.3 Process Orientation

The 3000 has a fairly clean process-oriented architecture, but one which the user may not control interactively. MPE itself runs as a process; when a user or batch job logs on, MPE creates


AD-A150 422 ALIAS (ACQUISITION AND LOGISTICS INFORMATION AND 2/7
ANALYSIS SYSTEM) MAINTEN.. (U) DECISION-SCIENCE
APPLICATIONS INC ARLINGTON VA M S CAREY ET AL.
UNCLASSIFIED 31 OCT 84 DSA-593-VOL-1 N00014-82-C-0813 F/G 15/5 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

references. A utility processor called the SEGMENTER allows a programmer to make up a single object code file from several, thus allowing large systems to be separately compiled and then merged just prior to linking.

There is no symbolic debugger.

4.1.5 MPE Procedures

Most modern operating systems allow simple programs composed of operating system commands to be written and stored for later execution, and MPE is no exception. Known as user-defined commands (UDCs), these may use IF-THEN-ELSE constructs and simple 16-bit variables to control their flow of execution. They may accept arguments, and can substitute the arguments into the body of commands.

The principal weakness of UDC's is that they offer no facility for program i/o redirection. It is often desirable to have a program run from a UDC take its input from the body of the UDC rather than from the terminal or a file, thus allowing arguments passed to the UDC to be passed directly into the program. However, this is not possible on MPE.

4.1.6 Reading the HP Documentation

The documentation for the 3000 is designed for the non-programming user, and often fails to direct programmers to the proper manuals and sections. For example, the only available discussion of the 3000's fundamental architecture is to be found near the back of the MPE Intrinsics Manual. Programmers should pay particular attention to this manual, especially its later sections, to the final sections and appendices of the FORTRAN manual, and to the Segmenter manual. The System Manager and the System Utilities manuals are also useful. Note especially that many functions normally performed by operating system commands, such as file copies, are performed by utility programs on the 3000. Functions which appear not to be supported by the 3000 on

the basis of an inspection of the MPE Commands manual may be supported by a utility program.

4.2 METHODS OF COPING WITH A LIMITED ADDRESS SPACE

ALIAS as a whole requires much more than 64K of program data or 'stack space'; many ALIAS modules require more than this limit all by themselves. Two methods have been used to cope with the limitation, each of them unfortunately damaging to system performance.

First, most modules are prepared as separate programs, which are run as son processes by the ALIAS main program (System Core). This allows each module to enjoy a full 64K. Since the Core requires a good deal of data of its own, failure to do this would severly restrict the space available to each module.

Note also that the facilities of RELATE are accessed through one or more RELATE son processes, though the startup of these processes is handled implicitly by the host language interface routines in the RELATE segmented library segments. RELATE suffers from the same stack space limitations as ALIAS, and is seldom able to handle more than 25 simultaneously open data base files. Any module needing more files open at once must start additional RELATE processes to serve its needs (the DBIF now supports only a single service process per module, but could be extended to transparently use multiple processes if necessary).

The second method of coping is by manual paging of data as part of a program's execution. Large arrays of data may either be stored in a file or an extra data segment (see Section 3.1.4), with only a small subset maintained in an actual array in the main data stack at any time. Of course, the programmer is then responsible for detecting the need to swap data and for accomplishing the swap, a tedious and error-prone procedure. Use of the memory management utilities reduces the magnitude of the task somewhat.

In spite of these limitations, good modular design both of modules as a whole and of their internal data structures allows fairly demanding tasks to be accomplished.

4.3 FORTRAN COMPILER

The HP 3000 fortran compiler supports a greatly extended version of the ANSI '66 standard language. It supports character variables (no longer than 255 characters), substring operations (no concatenation), internal reads and writes (no encode/decode), bit-level operations, and run-time format statements. It does not support IF-THEN-ELSE or file OPENs or CLOSEs, and has no include facility. It has both 16-bit and 32-bit integers, and allows character and numeric variables to share the same storage. The 'parameter' statement is supported.

It is expected that ALIAS will eventually be converted to run using a compiler conforming to the 1977 ANSI fortran standard, and every attempt has been made to make ALIAS code conform as closely to that standard as possible. Major conventions include limiting ALL names (variables, program, etc) to no more than six characters and use of the filopn and filcls utility routines to handle all file opens and closes (these routines call MPE intrinsics to accomplish dynamic file opens and closes; their arguments follow those accepted by '77 standard open and close statements in content, allowing such statements to replace the intrinsics when conversion occurs).

Areas of unavoidable incompatibility include the syntax for substring operations (square brackets instead of parentheses, and second argument giving length relative to first argument rather than last character), the use of 's' rather than 'a' in format statements, the syntax of parameter statements, and the fact that character and numeric variables are often mixed in common blocks. This last is unavoidable in the HP implementation because of the need to provide RELATE with records for data transfer.

The compiler has certain unusual features which are under the control of the programmer through the placement of 'meta-commands' in the source program. These commands, whose basic syntax is '$CONTROL option,option,option' are used to specify the size of the table to be allocated for i/o unit numbers ($CONTROL FILE=1-99 is recommended for placement at the top of every program), to name the segment into which each subroutine will be compiled, to select one of two global-variable addressing schemes, and to specify the precision with which actual and formal subroutine arguments must match one another. See Section 9 of the HP FORTRAN manual for a full discussion of these commands.

The global variable referencing option requires particular mention: unless extended addressing is requested at compile time, no program may have more than 255 common block variables (extended addressing allows 255 labeled blocks of 255 variables each). All ALIAS routines should be compiled with the extended addressing option, which need appear only at the top of each source code file.

4.4 USE OF INTRINSICS

"Intrinsics" is the name which HP gives to operating system utility subroutines available on the 3000 for use by the programmer. ALIAS makes use of many intrinsics for such tasks as son process creation, extra data segment management, file management, programmatic execution of MPE commands, and retrieval of the current time and date. These routines are directly callable from FORTRAN programs (with a few practical exceptions) as long as they are declared using the 'SYSTEM INTRINSIC' statement, and as long as argument type conventions as specified in the Instrinsics manual are observed. All calls to these routines are now and should continue to be isolated in utility routines in ALIAS, to foster conversion to another host. See the MPE Intrinsics manual for a list of the routines available.

4.5 TERMINAL RESTRICTIONS

The ALIAS host computer assumes that all users are logged on with HP video terminals unless told otherwise on the log-on command. Those with non-HP video terminals should log on with the following syntax: HELLO username.account;TERM=4.

Note that much of HP's software, such as its screen handler, word processor, and graphics package, will work ONLY with HP block-mode/graphics terminals. ALIAS, however, has been designed to be completely terminal-independent except for graphics. Should a capability to handle terminals in addition to those listed on the user environment parameter menu be desired, the control code data for these must be added to the setccl.utlr routine and to the builder terminal configuration file (bldrterm.sysrw).

5.0 SUPPORTING SOFTWARE PACKAGES

This section discusses the software packages which ALIAS depends on for support in special areas. There are only two basic packages: the RELATE family of DBMS software, produced by Computer Resources, Inc. (CRI) of Santa Clara, CA., and HP DRAW/Decison Support Graphics, produced by Hewlett-Packard.

As with the previous section, this discussion is meant to complement, and not substitute for, the manuals which are provided by CRI and HP. The following subsections will describe each package in broad terms, and will point out features of particular interest to programmers or which are poorly documented.

5.1 THE RELATE FAMILY OF SOFTWARE

Software provided by CRI provides data base management, graphics, report generation, and screen handling support to ALIAS. This software is implemented as two separate programs and a segmented library of executable code. One program, RELATE.PUB.RELATE, contains the basic DBMS as well as the graphics and report generator. The other, BUILDER.BUILDER.RELATE, is an interpreter which accepts files containing a specialized programming language oriented toward displaying data on formatted screens and toward updating of data bases using screen inputs from the user.

Due to the method of installation of the RELATE family on the PMS 392 HP 3000, any account whose users wish to run RELATE or BUILDER must have a copy of the CRI segmented library stored as SL.PUB. The file SL.PUB.SEA90 contains the CRI library plus additional special routines.

5.1.1 The DBMS

The RELATE data base management system is a fully relational DBMS with many desirable features. In addition to the usual data base creation, modification, and query capabilities,

it has internal security, transaction processing, logging and recovery, and permanent view creation capabilities. All of its commands may be given either interactively, programmatically, or in the form of procedure files. It does not offer an integrated data dictionary at this time.

RELATE is fundamentally a non-procedural system when used interactively: that is, commands (and command arguments) may be given in any order. It does offer a more procedural menu-oriented interactive interface which will be discussed in Section 5.1.4.

5.1.1.1 Storage Method

RELATE uses up to two files to store a single data base relation. One file contains the data records, the second indexing information. The indexing file, which is created only when an index is created, is given the name of the data file with a 'Q' concatenated onto the end. A binary tree indexing method is used. Each relation may have up to eight indexes, each containing up to eight fields.

Unlike many DBMSs, RELATE does not divide files into separate data bases. Any RELATE file may be used with any other RELATE file. Further, RELATE can use data bases created by HP's IMAGE network DBMS, and can also create and use ordinary MPE files.

In addition to direct use of data relations for querying and printing of output, RELATE also supports the construction, use, and retention of 'virtual' relations using its SELECT and CREATE VIEW facilities. A virtual relation is one which is a subset of an actual relation, or one which is composed of data combined from two or more relations. Stored virtual relations are known as 'views', and represent the only way that more than one SELECT may be referenced at one time. Note that HP memory limitations make it impossible to use more than about three views

at a time. RELATE does not function well with data bases crossing account boundaries. This is not now a problem for ALIAS.

5.1.1.2 Programmatic Usage of RELATE

This section discusses programmatic usage of RELATE in some detail, due to the paucity of documentation provided by CRI. Although of use to any programmer, much of the discussion is not directly relevant to most ALIAS module development, since modules should use the facilities of the DBIF (covered in Section 10) for programmatic data base access.

When RELATE is used interactively, the RELATE program is run as the user's main process. When used programmatically, RELATE must be run as a son process of the using program. If this were not done, the RELATE routines would have to be linked into the user's program, and would take up much of the available data stack space.

Programmatic usage of RELATE is accomplished through the facilities provided by a set of subroutines referred to by CRI as the 'Host Language Interface' (HLI). These routines reside in a segmented library (which must be in the file SL.PUB.SEA90), and are callable by programs written in any HP-supported language. The routines take care of creation of and communcation with the RELATE son process.

All arguments to the HLI routines are expected to be integer arrays, even arguments which are fundamentally of type character, such as field names and RELATE commands. This is because the routines expect the argument data to be word-aligned (that is, the first byte of each argument must correspond to the first byte of a two-byte word in memory; this condition often does not hold for variables of type character). Fortran programmers must therefore equivalence their character arguments to integer arrays, and give the arrays as arguments to the HLI

routines. The data must be word aligned because it is passed to the RELATE son process through an extra data segment, and extra data segments will accept only word-aligned data for transfer.

Each HLI routine expects its first argument to be a 50-word integer array, mysteriously called a 'cursor'. Such cursors have no relation to the white marker on a video terminal which indicates the current character. However, a cursor is a pointer of sorts; it contains information about previous commands. A programmer may define and use multiple cursors, each for a different relation or set of relations; the information in a given cursor will instruct the HLI concerning which relation or relations should be used as part of the response to a call to a given HLI routine.

The cursor also acts as a global data storage area for the HLI routines; routines in a segmented library may not have any global variables, and thus must use some other means of storing the identity of the RELATE son process and of the communications data segment. By storing such data in cursors and forcing the user to supply a cursor with each HLI call, global storage is effectively made available to the HLI routines.

There are four basic types of HLI routine. The first is responsible for initialization. Before a given 50-word array may be used with other HLI routines, it must be turned into a functioning cursor by giving it as the argument in an rdbinit or rdbinitx call. These routines will fill the cursor with appropriate data values, and will cause appropriate data structures to be created in the RELATE son process. The first call to rdbinit will also cause the RELATE son process to be created. Rdbinitx may be used to create multiple son processes and/or to specify process and communications segment ids.

The second type of HLI routine allows any interactively executable RELATE command to be given. The 'relate' HLI routine

accepts a word-aligned character string which is the text of the command, such as "OPEN FILE JUNK" or "PRINT", and causes the given action to be taken.

The third type of routine provides a record-level interface to data bases, in contrast to the set-level interface available interactively. Interactive commands are oriented towards actions like 'delete all records with a class name of DDG'. Record-level commands find, read, update, delete, or add one record at a time, using the concept of a 'current record', in a fashion similar in many respects to direct-access fortran file i/o. Unlike fortran i/o, a relation's records may be processed in an order determined by any of its indexes, and it is possible to jump directly to the first match on any value of the current index (i.e. sequential searching is unnecessary). This record-level interface is the part of the HLI of primary interest to programmers, although the 'relate' routine must be used for tasks like opening files, setting indexes, etc.

The fourth type of routine provides services, such as information about a cursor and error checking and reporting.

The HLI is quite easy to use, as long as the following facts and concepts are understood:

---For many purposes, each cursor should be thought of as a separate invocation of RELATE. A file must be explicitly opened on a cursor before it may be referenced, regardless of whether it is open on other cursors. Operations on one cursor do not affect the status of other cursors. Thus, it is possible to maintain a 'current record' on many different files at once, or even two different current records on the same file at the same time. This capability is very useful for search and cross-reference procedures.

In spite of these facts, it is in general true that cursors are using the same RELATE son process. Thus, opening of too may files and/or initialization of too many cursors can overflow the data stack of the son process and cause a system abort, just as such actions can when RELATE is run interactively. Programmers will

also notice that opening of a file already opened on one cursor on a second cursor takes very little time.

Cursors refer to different RELATE sons only if the rdbinitx routine is used to start up multiple sons.

---EVERY argument to the HLI routines MUST be in the form of an integer variable or array. Use equivalence statements to place character arguments in the same storage as an integer array.

---The operations of the record-level interface routines implicitly depend on what the current index is on a file or selection. Reads will return records in the order of the index. If no index is set, records will be returned in their order in the data relation.

---The searching power of the record-level interface is lodged in the 'rdbpoint' routine. The syntax for use of this routine from fortran is "call rdbpoint(cursor,keyval,words, break,found_flag)". This routine expects its 'keyval' argument to contain target values for the current index. It will attempt to find the first record which has field values exactly matching the values provided in 'key'. For example, if the current file is SHDESC.MISCJ (ship descriptions), and the current index orders the file by the values of the fields SCENARIO,CLASS, and the 'key' integer array contains the value "IMAGINATION SSN-688 ", rdbpoint will look for a record with a scenario value of "IMAGINATION" and a class name of "SSN-688".

Note that the 'keyval' array must contain exactly as many words as the index (unless a subset of the index is specified using the "words" argument). Each field value within the array must be in the order of the index fields and exactly as long as each index field. If these conditions are not met the point will always fail.

Thus, it is only possible to use fields contained in existing indexes to search for matches (temporary indexes can be created through use of BY clauses in a SELECT command, but this can be very time-consuming for files with large amounts of data). However, the 'words' argument makes it possible to use a subset of an index as a search criterion. For example, if 'words' were 6 in the above example, the first record with a scenario value of 'IMAGINATION' would have been found, regardless of its class value.

If there are multiple records matching a 'key' value, a point will always find the first one. Since the point is done based on the current index, subsequent matches may be read sequentially following the first one. The

HLI can be made to inform a program when these subsequent matches run out: by setting the 'break' argument (similar in function to the BREAK keyword of the report generator) to be the same as the 'words' argument, rdbread calls will return an end-of-file when all matches have been read.

---Reads will return the record just pointed to, or the one following the one just read. Updates and deletes will act on the record just pointed to or read.

---The 'relate' HLI routine must be used to open files, set indexes, etc., using ordinary interactive syntax.

---The HLI is fairly slow for individual record operations in comparison to fortran i/o. A simple record read can be expected to take 25-30 milliseconds, a point 250 milliseconds. The time required for adds, updates, and deletes will depend on the number of indexes affected: index files must be updated as main data files are. Where many indexes are involved, update processing times of 500 milliseconds per record is not uncommon. All of these figures were obtained running benchmarks which were the single user on an HP 3000 Series 44.

A good deal of this time can be attributed to the poor performance of HP multi-processing and interprocess communication intrinsics.

Use the DBIF utility system whenever possible in preference to direct reliance on HLI routines.

5.1.2 RELATE Graphics

CRI offers a business-graphics companion to RELATE called GRAF. This package produces line graphs, bar charts, and pie charts, as well as custom figures and annotations to the charts using character, line, and figure-drawing commands. It currently works only with HP graphics terminals and plotters, but does support full-color plotting.

The package is used by some ALIAS application modules, and is available for ad hoc production of graphics as part of ad hoc queries. It is delivered as an integral part of the RELATE program, appearing simply as additional RELATE commands which may be given interactively or programmatically.

The most attractive feature of GRAF is its direct usage of data relations, and the fact that the powerful sorting and data-combination/reformatting facilities of RELATE may be used to prepare the data to be graphed. The package is quite easy to use given a familiarity with RELATE and a desire for simple graphics. It is more difficult to use for complex and annotated charts, and lacks some desirable features for such charts.

5.1.3 Report Generator

CRI also offers a report generator to work in concert with RELATE, called CREATE. Like GRAF, it is delivered as an integral part of the RELATE program. It greatly extends the textual output formatting capabilities of the basic DBMS, allowing report and page headings and footings to be defined, subtotaling and totaling, grouping of data according to field values with group headings, footings, and totals, and custom column headings and column placement.

Also like GRAF, CREATE draws data through the DBMS, thus allowing sophisticated sorting, searching, and combination conditions to be specified as part of a report's definition.

CREATE is used for some ALIAS report-generating modules, and is available for support of ad-hoc queries.

5.1.4 Menu Interface

A menu-oriented interactive version of RELATE is available which supports occasional users of RELATE and certain kinds of query and update activities. This interface offers many of the capabilities of the standard non-procedural version in a fashion which unfamiliar users may be better able to cope with. MENU is implemented as a BUILDER procedure file.

5.1.5 Screen Handler

CRI offers a program separate from RELATE, which uses RELATE's facilities, known as BUILDER. This package is a system developer's tool, designed especially for construction of data base updating systems featuring fill-in-the blank screens. It offers a number of capabilities in addition to this which make it a full-scale programming language in many respects.

BUILDER is an interpreter which reads and processes commands from a standard ascii file, produced using one of the HP editors. The file may contain any BUILDER command or any RELATE, CREATE, or GRAF command. Non-BUILDER commands are passed through the host language interface to a RELATE son process, which executes them.

One of BUILDER's most useful features is its association of screen 'blanks' or 'windows' with data base fields. When a data base record is read which has a field whose name matches the name of a screen window, the data from the record is automatically displayed in the window. The Data Base Updating System (DBU) is implemented using BUILDER.

5.2 HP DRAW/DECISION SUPPORT GRAPHICS

The HP business graphics package is used programmatically by the labor workload module for production of line graphs. It is used in preference to CRI's GRAF because of its superior capabilities in the areas of precise formatting and annotations. DSG capabilities are available via a set of fortran-callable subroutines resident in the system segmented library (SL.PUB.SYS).

6.0 THE ALIAS ENVIRONMENT

An environment is the sum of the file structure, procedures, and programs directly and indirectly available to those working with a given system. A good environment is one in which common functions may be quickly carried out with little mental effort and with a minimum of commands. Files should be well-organized; users should seldom 'get lost' in the file system or in programs; commands should be consistent, causing the same action to be performed regardless of when they are given. The necessity to be aware of one's context (i.e. which program is being run now) should be minimized, and different contexts should be clearly differentiated.

The ALIAS environment features organization of files by both purpose and owner, a library of general-purpose operating system procedures, many software development support procedures, several generally-accessible libraries of subroutines and procedures, and on-line documentation throughout. The programs which comprise ALIAS also incorporate support functions, and are internally consistent in terms of command codes. The casual user may accomplish a great deal with a minimal understanding of how the HP 3000 really works.

This Section divides the environment into two parts for purposes of discussion, one meant to serve all users ('user environment') and one meant to serve software development and maintenance personnel ('programmer environment'). In terms of actual implementation, the two are intertwined.

6.1 TOOLS AVAILABLE FOR CONSTRUCTING AN ENVIRONMENT

The quality of environment which can be built depends on the number and quality of tools provided by the host computer for use in its construction. There are several kinds of tools which are necessary or helpful:

1) FILE STRUCTURE. The extent to which files can be grouped according to logical criteria or pattern of usage is a prime determinant of the manageability of a large system. The HP 3000 has a file structure with limited hierarchical capabilities which has supported organization of files in a reasonable fashion.

2) OPERATING SYSTEM COMMANDS: The commands which may be given interactively to the operating system are fundamental building blocks of an interactive environment. Such commands perform functions such as file listing, file deletion, program running, and message/mail processing. The HP 3000 has a rather rudimentary set of operating system commands; it relies on utility programs for many functions, such as file copying, and offers little support for electronic mail.

3) OPERATING SYSTEM PROCEDURES: Most operating systems allow the creation and execution (by name) of procedures which consist of a series of operating system commands. This makes it possible to condense long and complex procedures requiring many precisely typed commands down to a single command. MPE does have OS procedure capabilities: any user may write 'User Defined Commands', place them in a file, and notify MPE that the file contains procedures; they then become executable. Argument passing and substitution is supported, but very little i/o redirection.

4) UTILITY PROGRAMS: The existence of system utility programs for tasks such as sorting, searching, and file comparison removes the need to write custom programs to accomplish the same ends. HP provides a small library of utility programs.

5) EDITOR MACROS: It is typical for most users, especially programmers, to spend much of their time using an editor. A capability to write, store, and use procedures consisting of editor commands offers the same general benefits as do operating system procedures. Both the HP standard editor and the TDP editor accept simple macros and have limited buffering capabilities.

6) MULTIPROCESSING: If a given user may run several processes (programs) at once, and/or may run batch jobs, an environment can be constructed which maximizes human throughput by minimizing the extent to which the terminal is tied up during program execution. MPE supports batch jobs, but multiprocessing may be conducted only by programs, and i/o redirection capabilities are very limited.

7) **DATA BASES AND DOCUMENTATION PROCESSORS:** Support for on-line help and documentation allows an environment to be made more usable and maintainable. The RELATE DBMS provides some on-line documentation support capabilities.

8) **SOFTWARE DEVELOPMENT SUPPORT:** The variety, sophistication, flexibility, and speed of programming language compilers, linkers, and debugging aids has a tremendous impact on programmer throughput. MPE has several fairly fast compilers which produce compatible object code, and a fairly fast linker which supports object code libraries. Debugging support, however, is poor.

6.2 SUMMARY OF FILE STRUCTURE AND SYSTEM NAMING CONVENTIONS

6.2.1 File Structure

Each file on the HP 3000 belongs to a 'group', which in turn belongs to an account. Accounts may have as many groups as desired (within broad limits). This facility has been used on the SEA90 (ALIAS) account as the primary file structuring vehicle. A large number of groups have been created, each containing files which are related by criteria of purpose or ownership. The groups may be conceptually lumped into a few categories: user groups (for files belonging to an individual user); data base groups (holding data base files); system groups (holding various public files); and development groups (of interest only to programmers and data base administrators). Table 6-1 illustrates this structure.

For those accustomed to hierarchical file systems with multiple levels of directories, note that the groups constitute only a single level of directories, directly owned by the account. The HP 3000 will not support a more complex path-oriented file structure.

The SEA90 account contained well over 500 files as of the beginning of 1984, and was growing steadily in size. It is crucial that files be properly placed in the file structure if

TABLE 6-1

SEA90 ACCOUNT FILE GROUPS BY CATEGORY

USER GROUPS	DATA BASE GROUPS	SYSTEM GROUPS	DEVELOPMENT GROUPS
al	currj	cmdfil	program development
ash	db	dsa	compile
bill	descj	fmtfil	incl
common	histj	mail	link
dba	legals	makmenu	listings
jack	miscj	mnufil	merge
jackie	projj	mnurel	obj
jerry	supind	pub	src
jim	yards	rprocs	database development
judy		screens	crdb
king		sysro	
linda		sysrw	
mark		util	
ron		doc	
tom		templat	
wolf		prog	
woo		macro	
ellie			
ranny			
charlie			
dick			
joe			
jfp			
alex			
bob			
hank			
jacky			
joey			
larry			
lena			
rosey			
steve			

ALIAS is to remain a manageable system. Careless placement of files makes their later location difficult, confuses other users, and makes the job of maintenance personnel much more tedious and time-consuming.

A screen-oriented file cataloging system (CAT) is available on the account to support documentation of the purpose and ownership of files.

6.2.2 Naming Conventions

Mnemonic naming is crucial to the successful management and maintenance of the ALIAS system. ALIAS is used, maintained, and added to by many organizations and individuals. Without mnemonic naming of all system units, it will be impossible for new maintenance and expansion personnel to find their way around the system. A number of naming standards and conventions have been developed to promote system maintainability.

6.2.2.1 File Naming Conventions

HP 3000 file names are composed of three extents: file.group.account. All ALIAS files have an account name of 'SEA90'. The following standards apply to the group and file extents:

GROUP NAME

---All files private to a single user shall have that user's group as group name.

---All files shared between users, but otherwise not maintained as part of the ALIAS system, shall have 'common' as their group name.

---All data base files shall have the most appropriate data base group as their group name.

---All data dictionary files shall be placed in the .db group.

---All legal-data-base-field-value library files shall be placed in the .legals group.

---All documentation files shall be placed in the .doc group.

---All template files shall be placed in the .template group.

---All BUILDER procedure input files shall be placed in the .screens group.

---All RELATE procedure files shall be placed in the .rprocs group.

---The .pub group shall be reserved for files required to be there by the HP system or supporting software packages (mainly the account SL and the RELATE security files); files related to these may be maintained there as well.

---ALIAS menu system command files are automatically placed in the .cmdfil group; no other files shall be placed there, and no manual purging of files in that group shall be done.

---All ALIAS menu system files and relations shall be placed in the .mnufil, .mnurel, and .makmenu groups; no other files shall be placed in these groups.

---Public versions of format control files (e.g. Force Level Report Generator input files) shall be placed in the .fmtfil group.

---The .mail group is reserved for mail files.

---ALIAS system files which ordinary users may not change shall be placed in .sysro; those which they may/must have write access to shall be placed in .sysrw.

---All data base relation creation procedure files shall be placed in the .crdb group.

---All debugged source code shall be placed in the .src group.

---All object code shall be placed in the .obj group.

---All ALIAS program files shall be placed in the .prog group.

---Compilation listings may be maintained on disk on a short-term basis only, and then only in the .listings group.

---All include files shall be placed in the .incl group.

---The .compile, .merge, and .link groups are reserved for batch compilation stream file templates, segmenter input files which make up linkable object code files, and batch link stream files, respectively. No other files shall be placed in these groups.

---All utility program source code shall be placed in the .util group.

---All executable system utilities, including programs and udc's, shall be placed in the .dsa group.

---New groups may be made up as appropriate. Careful attention should be paid to mnemonic naming and the granting of appropriate access priveleges.

---It is recommended that groups be created on a temporary basis for use in development of new modules and/or conduct of unusual projects. This prevents existing groups from being cluttered with transitory files, and reduces the likelihood that public files will be purged accidentally.

FILE NAMES

---All files shall be named mnemonically according to PURPOSE.

---All files should be documented using the CAT file cataloging system; this is especially true for public files.

---ALIAS system modules which are programs shall have a 'core' name of exactly four characters. The program file shall have this name, as well as dedicated segmenter input files and batch link stream files in the .merge and .link groups.

---Module source code files shall have the module name as their first four characters, and shall use the next three characters to indicate the first three characters of the name of the first routine in the file. Routines shall be stored alphabetically within these files, except that the program unit shall be at the top of the first file (e.g. program flrept is the first routine in flrpa.src). This scheme allows easy location of module routines in source code for maintenance work.

---No source code file shall have a name longer than seven characters. The compilation udc's append one character to the source file name to distinguish between initial source, post- include source, and listings, and thus will not work with source files which have 8-character names.

---No relation may have a name longer than seven characters. RELATE will enforce this.

6.2.2.2 Fortran Code Naming Conventions

Mnemonic naming of program source and object code units is also important. The following standards apply:

---No fortran entity (program, subroutine, function, variable, array, or labeled common block) shall have a name longer than six characters. This is vital to ALIAS portability; the '77 ANSI fortran Standard requires compilers to allow at least six characters, and several major compilers allow no more.

---Fortran subroutines and functions shall have the first two characters of their names dedicated to indication of the module or subsystem they belong to. For example, all scenario system routines begin with 'sn'.

---Segment names (the segment that a routine will belong to must be named before the first source statement of the routine using the $CONTROL SEGMENT=name compiler meta-command) may be up to 15 characters in length, but should indicate the module the routines in the segment belong to.

---All program unit names, including labeled common block names, shall be as mnemonic as possible.

---The names of labeled common blocks shall follow the names of the include files they reside in as much as possible.

6.2.2.3 Field Naming Conventions

Field names in data base files are restricted by RELATE to ten characters or less. Such names, as usual, should be as mnemonic as possible. Use of the underbar is encouraged as a separator indicating that a mnemonic is based on more than one word (e.g. HEIGHT_AWL for 'height above waterline'). When relations are created, each field should have a print width at least as wide as its field name so that users do not become confused.

6.3 DESCRIPTION OF USER ENVIRONMENT

The environment surrounding ordinary users of the ALIAS system is organized so that few of them will be particularly conscious of it. Also, they need do little to facilitate its maintenance except manage their personal files. System maintenance personnel, however, must have a thorough understanding of the system facilities that are in general use so that no actions are taken which interfere with day-to-day activities.

This section will concentrate on a description of the makeup of the user environment: those parts (e.g. utility programs) whose operation must be discussed in detail to ensure adequate maintenance are covered in Section 6.5.

6.3.1 File Structure

The ordinary ALIAS user will need to be consciously aware of only a small subset of the groups comprising the ALIAS file structure. Most files which users create and reference directly by name (rather than implicitly through a program) will reside in their own personal groups. System security is such that ordinary users may not change nor create files in other users' personal groups, so each user will generally work only with his own group.

Additional groups which might be accessed consciously include .mail, .fmtfil, .rprocs, all data base groups, and .common. To send mail to a given user, a file named after that user must be created in the mail group (minus the R/A appendage, e.g. Bob's mail must be placed in a file called bob.mail, not boba.mail or bobr.mail). Thus, most users can be expected to read or create .mail files from time to time.

Those using ALIAS modules which require format control files as input will reference files in the .fmtfil group, and those generating reports drawing on the data base may need to use .rprocs files. Those performing ad-hoc queries not supported by

the DBU will need to understand the file and group structure of the data base, and will use RELATE to open and read files in this structure.

Finally, the .common group is for files which any user wishes to make available to the ALIAS community, but which are not maintained by system maintenance personnel. Such files might include report generating procedures and format control files being made available for public inspection prior to inclusion in .rprocs or .fmtfil.

6.3.2 File Cataloging System

A utility is available to help all ALIAS users in keeping track of their personal files, and to support documentation of public files. Run by the CAT udc, it is a BUILDER procedure file which allows inspection and update of an account-wide file catalog data base. System security is enforced in that ordinary users may not inspect the catalogs of any but their own and public groups.

It is particularly important that maintenance personnel keep the catalog up to date as they make changes to the contents of public groups.

The CAT utility system is run by a udc in the main udc file (udcl.dsa); its data base is kept in fcatlog.sysrw and gcatlog.sysrw; its procedure file is stored in catsys.screens.

Whenever a new group is created, it is important that the name of this group be added to the cataloging system. CAT allows documentation both of groups and files within groups; a group's name must be entered before the names of files within that group can be.

6.3.3 User Defined Command Library

The main file storage and access facility available to users is lodged in their personal groups; the main procedural services available to them, when they are not running ALIAS, are implemented in the main ALIAS library of operating system procedures (udc's). This library is stored in file udcl.dsa.

Table 6-2 summarizes the procedures which might be of interest to non-programming users by functional category. Table 6-3 describes what each procedure does in more detail.

The only public udc not in this library is 'NOTICE'; it is maintained in udc2.dsa. All public udc's should be placed in udcl.dsa. The text of this udc file is given in Figure 6-1.

A number of the procedures described in Table 6-3 are implemented using utility programs. Compare, gather, type, and talk, each depend on one or more program files in .dsa (source code in .util). See Section 6.5 for detailed descriptions of each of these programs.

6.3.4 Inter-User Communication

Users may send messages and mail to one another and may write memos to themselves. A facility which lets account supervisors make a notice appear to all SEA90 users each time tney log on is also available.

Messages may be sent from one logged-on user to another, using either the MPE TELL command, or else the TALK udc. Talk uses the TELL command implicitly, but allows multiple message lines to be sent after giving the target user's name only once.

Mail may be 'sent' to any user by using one of the editors to create a file with the text of the mail in it, and storing this file in the .mail group under the target user's name. This name should omit the A/R character which is a suffix of all

TABLE 6-2

GENERAL-INTEREST UDC'S BY FUNCTIONAL CATEGORY

FILE MANAGEMENT AIDS

Directory Listing	Documentation	Move/Print/Compare
da	cat	compare
df		copy
dg		gather
tlistf		list
		move
		type

SUBSYSTEM EXECUTION AIDS

General	DBMS-Related	HP Packages
alias	relate	graph
	menu	hpdraw
		hpslate

DEVICE CONTROL

daisy
starlp
print

EDITING

ed
edn
tdp

COMMUNICATIONS

File Transfer	Info	Messages/Mail
prntloc	who	notice
link47	hmu	talk
store47		
restore47		

MISCELLANEOUS

p2
p3
p4
uinfo

TABLE 6-3

GENERAL-INTEREST UDC DESCRIPTIONS

FILE MANAGEMENT AIDS

Directory Listing

DA [destination=terminal,detail=2]
: Display All files. Lists the names of all files in the SEA 90 account to the specified destination (default is terminal). By default, the listing is in long form, which includes information such as file sizes and types.

DF [filename=@,detail=2]
: Display File characteristics. By default, lists the names and characteristics of all files in the log-on group in long form. Lists the characteristics of a single file if its name is given as the first argument.

DG groupname [,detail]
: Display Group files for group 'groupname'. By default, lists the names and characteristics of all the files in 'groupname' in long form.

TLISTF
: Lists the temporary files created during the current session and any file equations that are currently in force.

Documentation

CAT
: CATalog files. Runs the SEA90 screen-oriented file cataloging system, which allows updating, inspection, and printing of the file system documentation data base.

Move/Print/Compare

COMPARE filea,fileb
: Compares any two standard editor files (72 column width text) to see if they are identical. If they differ, 5 lines centered around the difference are printed.

TABLE 6-3

GENERAL-INTEREST UDC DESCRIPTIONS

COPY filea,fileb
: Makes a copy of filea named fileb. Fileb may not already exist.

GATHER filename [,newsize=2047]
: Identical in effect to the EDITOR GATHER ALL command or the TDP RESEQ command. Rewrites a standard editor file so that its line numbers are monotonically increasing. Faster than the editor commands, but tends to litter the disk with large files. Use DF filename to find the current size of the file in sectors, and then specify that number as the optional argument on this command.

LIST filename [,destination=terminal]
: Prints any text file to the terminal or other devices. To specify another device, first give that device's initialization udc (e.g. 'daisy') and then give that name preceeded by a '*' as the optional argument (e.g. LIST filename,*daisy).

MOVE filename,newname
: Similar in function to a rename, but operationally like a copy, move makes a copy of 'filename' into 'newname' and then purges 'filename'. Recommended for use in procedures which move files from group to group, since the rename command is not reliable on a multiple-disk system.

TYPE filename [,mode=regular,destination=terminal]
: Prints a standard editor file to the terminal. Similar in basic purpose to 'LIST', but offers a nicer output format for editor files. Options allow alternative output formats and a destination device other than the terminal (see 'LIST' for a discussion of device specification). The regular output format gives the editor line number up to two digits past the decimal point with each line. Specification of 'short' as the optional second argument causes simple sequential numbering of the output; short will also handle unnumbered editor files, i.e. those with a text-plus-line-number record length of less than 80 characters. 'Fulltype' as the second argument causes sequential numbering of output as well as the printing of the full editor line number. A 'print' option is

TABLE 6-3

GENERAL-INTEREST UDC DESCRIPTIONS

designed to format fortran source code for line printer output: this prints line numbers and inserts form feeds after each routine. 'Count' will count the number of lines in the file without printing any of them, and 'last' will print only the last twenty lines.

SUBSYSTEM EXECUTION AIDS

ALIAS

Runs the ALIAS system.

RELATE

Runs the RELATE Data Base Management System's standard command-oriented interface. This interface also provides access to the CREATE report generator and the GRAF graphics package.

MENU

Runs the menu-oriented version of the RELATE DBMS. Functions which can be performed from within MENU are more limited than if the standard RELATE interface is used, but MENU is much easier for the novice or occasional user.

GRAPH

Runs HP's Decision-Support Graphics package. The user must be logged on with an HP block-mode graphics terminal/

HPDRAW

Runs HP's figure/annotation-oriented graphics package.

HPSLATE

Runs HP's easy-to-use slide production graphics package.

DEVICE CONTROL

DAISY

Informs the system that the name '*daisy', when used in LIST or TYPE commands, means that output should go to the SEA 90 daisy wheel printer.

TABLE 6-3

GENERAL-INTEREST UDC DESCRIPTIONS

STARLP

Informs the system that the name '*lp', when used in LIST or TYPE commands, means that output should be sent to the PMS 392 line printer.

PRINT

Informs the system that the name '*print', when used as a file name (e.g. in LIST or TYPE), means that output should be sent to the SEA 90 dot matrix printer.

EDITING

ED filename

Runs the HP standard editor (EDITOR) with 'filename' as its TEXT file. This command causes the text of 'filename' to be brought in for editing automatically, and causes it to be saved automatically when the 'END' command is given. When run in this mode, the editor will not accept Text or Keep commands from the user.

EDN

Runs the HP standard editor in its normal mode. EDN is merely an abbreviation for 'EDITOR'.

TDP

Runs the Text-Document Processor editor, the preferred editor available on the PMS 392 system.

COMMUNICATIONS

File Transfer

PRNTLOC

Runs a special print program designed to drive printers slaved off non-HP terminals. A compilation listing print mode inserts formfeeds between routines.

LINK47

Runs the communications package which support file transfers between the HP 3000 file system and floppy disks on NAVSHIPSO's smart terminals.

TABLE 6-3

GENERAL-INTEREST UDC DESCRIPTIONS

STORE47
: Runs a special version of LINK47 which dumps a large number of files at once to a floppy disk.

RESTORE47
: Runs a special version of LINK47 which returns files dumped using STORE47 to the HP file system.

Information

WHO [users=all SEA 90]
: Lists the SEA 90 users currently logged on.

HMU
: How Many Users. Gives a quick idea of the load the system is under at present.

Messages/Mail

NOTICE
: Prints the notice on the current account bulletin board, any text in a file called 'MEMOS' in the user's current log-on group, and any mail sent to the user.

TALK
: Runs the SEA 90 message-sending program. This program may be used to send messages (not mail) to any logged-on user. The program will prompt for the target user's name, and will accept as many lines of message as are desired. The program will stop when a line containing only '//' is typed in.

MISCELLANEOUS

P2 command,file1,file2 [,rest=" "]
: P2 and its siblings P3 and P4 will perform any command on multiple files, reducing the amount of typing required for repetitive operations. For example, 'P2 PURGE,JUNK,YECH,COMMON' will cause the files junk.common and yech.common to be purged. Had the ',common' been left off, the files would have been assumed to be in the current log-on group. P3 and P4 differe only in that there are one or two more file arguments.

P3 command,file1,file2,file3 [,rest=" "]
: See p2.

TABLE 6-3

GENERAL-INTEREST UDC DESCRIPTIONS

P4 command,file1,file2,file3,file4 [,rest=" "]
See p2.

UINFO

Prints an annotated listing of the udc's in the main SEA90 udc file (udc1.dsa) to the terminal. The name, arguments, and purpose of each is described.

FIGURE 6-1. Text of UDC1.DSA

```
ALIAS group=relate,lib=p
FILE rdbecat.pub.sys=rdbecat.pub.!group
FILE  grecat.pub.sys= grecat.pub.!group
FILE bldrterm=bldrterm.sysrw.sea90
FILE FTN05=$Stdinx;Rec=-132
FILE FTN06=$Stdlist;Rec=-255;CCTL
FILE rdbin=empty.sysro
IF useversion=1 THEN
   ALTELL
   RUN  tmnur.prog.sea90;lib=!lib
ELSE
   RUN mnur.prog.sea90;lib=!lib
ENDIF
RESET rdbin
********
ALTELL
option list
comment   YOU ARE RUNNING THE DEVELOPMENT/TEST VERSION OF ALIAS
********
BATCH name,group=TEMPLAT,output=TMPBATCH
ZSUB !name.!group,!output
STREAM !output
********
CAT
FILE rdbecat.pub.sys=rdbecat.pub.relate
FILE bldrterm=bldrterm.sysrw
FILE bldrapp=catsys.screens
RUN builder.builder.relate;LIB=P
********
COL
option list
Comment..:123456789.123456789.123456789.123456789.123456789.123456789.12
********
COMPARE filea,fileb
FILE ftn01=!filea,old;acc=in
FILE ftn02=!fileb,old;acc=in
RUN  diffs.dsa
RESET ftn01
RESET ftn02
********
COPY filename,output
option list
FCOPY From=!filename;To=!output;New
********
CS
option list
RUN BUILDER.PUB.RELATE,COMPILE;LIB=P
********
DA dest=$stdlist,level=2
option list
LISTF @.@.SEA90,!level;!dest
```

FIGURE 6-1. Text of UDC1.DSA

```
********
DAISY
option list
FILE daisy;dev=49;Cctl
********
DATABASE901
FILE BLDRTERM=BLDRTERM.SYSRW
FILE BLDRAPP=DATABASE.COMMON
FILE RDBECAT.PUB.SYS=RDBECAT.PUB.RELATE
RUN BUILDER.PUB.RELATE;LIB=P
********
DEBUG
option list
SETJCW lprnton,1
Comment  WHAT DYNAMIC DEBUGGING SUPPORT THERE IS IS NOW ON
********
DEVELOP
option list
SETJCW useversion,1
Comment  ALIAS RUNS WILL NOW USE THE DEVELOPMENT/TEST VERSION
********
DF starname=@,level=2
LISTF !starname,!level
*******
DG group,level=2
LISTF @.!group,!level
********
ED filename
option list
FILE edttext=!filename
RUN  editor.pub.sys,basicentry
********
EDM macro=edmhelp
option list
ZSUB  !macro.macro,editin
FILE  fseg=editin,oldtemp;acc=in
RUN   editor.pub.sys;stdin=*fseg
RESET fseg
********
EDN
option list
RUN editor.pub.sys
********
EDS filename
option list
FILE edttext=!filename.src
RUN  editor.pub.sys,basicentry
********
FTN filename,dir=src,output=$newpass,size=2047
INCL !filename,!dir,!size
FTNLIST T!filename,!output,L!filename
```

FIGURE 6-1. Text of UDC1.DSA

```
********
FTNB filename,dir=src,odir=obj,size=2047
INCL !filename,!dir,!size
FTNLIST T!filename,!filename.!odir,L!filename
********
FTNLIST input,output=$newpass,listing=templist
option list
PURGE   !listing,Temp
BUILD   !listing;Rec=-133,1,f,ascii;Cctl;Disc=4095,32,1;Temp
FILE    flist=!listing,oldtemp;Temp
FORTRAN !input,!output,*flist
********
GATHER filename,size=2047
PURGE  G!filename,temp
PURGE  G!filename
BUILD  G!filename;Rec=-80,1,f,ascii;Disc=!size;Temp
FILE   ftn10=G!filename,oldtemp;acc=out
FILE   ftn09=!filename,old;acc=in
RUN    gather.dsa
SAVE   G!filename
PURGE  !filename
RENAME G!filename,!filename
********
GLUE modname
option list
PURGE !modname.obj
RUN segdvr.pub.sys;stdin=!modname.merge.sea90
SAVE !modname.obj
********
GRAPH
RUN GRAPH.PUB.SYS
***
HMU
SHOWJOB status
********
HPDRAW
RUN HPDRAW.PUB.SYS
*******
HPSLATE
RUN HPSLATE.PUB.SYS
********
INCL filename,dir=src,size=2047
option list
PURGE T!filename,temp
BUILD T!filename;Rec=-80,1,f,ascii;Disc=!size;Temp
FILE  ftn11=T!filename,oldtemp;acc=out
FILE  ftn09=!filename.!dir,old;acc=in
RUN   include.dsa
********
LIST filename,output=$stdlist
FCOPY From=!filename;To=!output
```


/31

FIGURE 6-1. Text of UDC1.DSA

```
********
LINK modname
option list
STREAM !modname.link.sea90
********
LINK47
option list
Comment  ABOUT TO RUN HP2647F FILE TRANSFER PROGRAM
Comment  IF YOU ARE NOT ON THAT TYPE TERMINAL, BREAK
Comment  AND ABORT IMMEDIATELY OR YOU WILL BE PLACED IN BLOCK MODE.
Comment
RUN LINK47.PUB.SEA90
********
MAKE modname
option list
STREAM J!modname.link.sea90
********
MENU group=relate
FILE rdbecat.pub.sys=rdbecat.pub.!group
FILE  grecat.pub.sys= grecat.pub.!group
FILE plotter;dev=50
FILE rdblist;dev=58;cctl
FILE BLDRTERM=BLDRTERM.SYSRW.sea90
FILE BLDRAPP=CRIMENU.PUB.!group
RUN BUILDER.PUB.!group,CREATOR;LIB=P;maxdata=31000
*****
MERGE musl,option=segmerg
option list
ZSUB   !option.templat,segin
PURGE  merge,Temp
FILE   fseg=segin,oldtemp
RUN    segdvr.pub.sys;stdin=*fseg
RESET  fseg
PURGE  !musl,Temp
RENAME merge,!musl,Temp
********
MOVE fromfile,tofile
SETJCW jcw 0
FCOPY From=!fromfile;To=!tofile;New
IF jcw=0 THEN
   PURGE !fromfile
   ENDIF
********
NORMAL
option list
SETJCW lprnton,0
Comment  DYNAMIC DEBUGGING SUPPORT NOW OFF
********
P2 command,file1,file2,rest=" "
Option list
!command !file1!rest
!command !file2!rest
```

FIGURE 6-1. Text of UDC1.DSA

```
********
P3 command,file1,file2,file3,rest=" "
option list
!command !file1!rest
!command !file2!rest
!command !file3!rest
********
P4 command,file1,file2,file3,file4,rest=" "
option list
!command !file1!rest
!command !file2!rest
!command !file3!rest
!command !file4!rest
********
PRINT
option list
FILE print;dev=58;cctl
Comment  MAKE SURE HP2934A IS TURNED ON,
Comment  OTHERWISE YOU'LL LOSE SPOOLER OWNERSHIP.
******
PRNTLOC
option list
FILE ftn06=$stdlist;acc=out;cctl
RUN prntloc.dsa
RESET ftn06
********
PRODUCTION
option list
SETJCW useversion,0
Comment  USE OF TEST VERSION OF ALIAS DISCONTINUED
********
PROG module,obj=.obj,prog=.prog,lib=utlr.obj
option list
PURGE !module!prog
PREP !module!obj,!module!prog;rl=!lib
SAVE !module.!prog
********
QCAT
RUN qcat.dsa
********
R
option list
RESUME
********
RELATE group=relate,lib=p
option list
Comment  MAKE SURE HP2934A IS TURNED ON BEFORE PRinting,
Comment  OTHERWISE YOU'LL LOSE SPOOLER OWNERSHIP.
FILE rdbecat.pub.sys=rdbecat.pub.!group
FILE rdbhelp.pub.sys=rdbhelp.pub.relate
```

FIGURE 6-1. Text of UDC1.DSA

```
FILE  grecat.pub.sys= grecat.pub.!group
FILE plotter;dev=50
FILE rdblist;dev=58;cctl
RUN   relate.pub.!group;lib=!lib;maxdata=31000
*******
RESTORE47
option list
Comment  ABOUT TO RUN HP2647F FILE TRANSFER PROGRAM
Comment  IF YOU ARE NOT ON THAT TYPE TERMINAL, BREAK
Comment  AND ABORT IMMEDIATELY OR YOU WILL BE PLACED IN BLOCK MODE.
Comment
RUN LINK47.PUB.SEA90,RESTORE
********
RPREP usl=$oldpass,prog=$newpass,rl=utlr.obj
option list
PREP !usl,!prog;Rl=!rl;Cap=ph,ds,ia,ba
********
RRUN prog=$oldpass,stack=3000,group=relate
option list
FILE rdbecat.pub.sys=rdbecat.pub.!group
FILE  grecat.pub.sys= grecat.pub.!group
RUN !proq;Stack=!stack;maxdata=31000;Lib=p
********
STARLP formal=LP
option list
FILE !formal;Dev=lp;Cctl
********
SCREEN appfile,account=relate
option list
FILE RDBECAT.PUB.SYS=RDBECAT.PUB.RELATE
FILE PLOTTER;DEV=50
FILE RDBLIST;DEV=58
FILE BLDRTERM=BLDRTERM.SYSRW.SEA90
FILE BLDRAPP=!appfile
RUN BUILDER.PUB.RELATE,CREATOR;LIB=P
********
SEG listfile=$stdlist
SEGMENTER !listfile
*******
STORE47
option list
Comment  ABOUT TO RUN HP2647F FILE TRANSFER PROGRAM
Comment  IF YOU ARE NOT ON THAT TYPE TERMINAL, BREAK
Comment  AND ABORT IMMEDIATELY OR YOU WILL BE PLACED IN BLOCK MODE.
Comment
RUN LINK47.PUB.SEA90,STORE
********
TALK
RUN msg.dsa
```

FIGURE 6-1. Text of UDC1.DSA

```
********
TDP
option list
RUN tdp.pub.sys
********
TLISTF
option list
RUN listeq2.pub.sys
********
TYPE filename,option=typfile,output=" "
FILE input=!filename;cctl
RUN  !option.dsa;stdin=*input;stdlist=!output
********
UPDATE option=REPLUNT
option list
ZSUB !option.templat,segin
FILE fseg=segin,oldtemp;acc=in
RUN  segdvr.pub.sys;stdin=*fseg
RESET fseg
********
WHO people=@.sea90
SHOWJOB job=!people
********
ZSUB template,outfile
PURGE !outfile,Temp
BUILD !outfile;Rec=-72,1,f,ascii;Temp
FILE  ftn09=!template,old;Acc=in
FILE  ftn10=!outfile,oldtemp;Acc=out
RUN   subst.dsa
********
UINFO
option list
Comment                UDC Command Directory
Comment
Comment  ALIAS                                          Runs alias (mnur)
Comment  BATCH name,grp=TEMPLAT,output=TMPBATCH           ZSUB name + STREAM
Comment  CAT                                            Run file cataloger
Comment  COL                                            Column numbers
Comment  COMPARE file1,file2                            Compares 2 ASCII files
Comment  COPY fromfile,tofile                           Copies files
Comment  DA dest=$stdlist,level=0                       Dir all groups
Comment  DAISY                                          Init Ron's Daisy
Comment  DATABASE901                                    Data Element Makeup
Comment  DEBUG                                          Turns on dynamic db
Comment  DEVELOP                                        Use test ALIAS
Comment  DF starname=@,level=2                          Dir current group
Comment  DG group,level=2                               Dir arbitrary group
Comment  ED filename                                    Edit a file
Comment  EDM macro                                      Run edit macro
Comment                              Read edmhelp.macro for assistance
```

FIGURE 6-1. Text of UDC1.DSA

```
Comment  EDN                                              Edit a New file
Comment  EDS filename                                     Edit filename.src
Comment  FTN filename,dir=src,out=$newpass,size=2047   FOR after INCL
Comment  FTNB filename,dir=src,odir=obj,size=2047 FORB after INCL
Comment  FTNLIST input,out=$newpass,list=templist Fortran temp list
Comment  GATHER file,size=2047                            Like EDITOR "G ALL"
Comment  GLUE modulename                                  segdvr;stdin=modnam
Comment  GRAPH                                            Runs DSG graph pkg
Comment  HMU                                              How Many Users (?)
Comment  HPDRAW                                           Runs HPDRAW
Comment  HPSLATE                                          Runs HPSLATE
Comment  INCL filename,dir=src,size=2047                  Performs includes
Comment  LINK modulename                                  batch prep modname
Comment  LINK47                                           2647F file transfer
Comment  LIST filename,output=$stdlist                    Display file
Comment  LPR usl=$oldpass,lib=utlr.obj                 Lib PrepRun
Comment  MAKE modname                                  GLUE then LINK
Comment  MENU                                             RELATE: menu version
Comment  MERGE filename,option=u                          Merges USL files
Comment                  Use option=S to merge segmented USL files
Comment  MOVE fromfile,tofile                             COPY plus Purge
Comment  NORMAL                                           No dynamic debug
Comment NOTICE file=notice.dsa                            Prints logon message
Comment  P2 command,file1,file2,rest=" "                  Does cmd for both
Comment  P3 command,file1,file2,file3,rest=" "    Like P2 for three
Comment  P4 command,file1,file2,file3,file4,rest=" "   Like P2 for 4
Comment  PRINT                                            Init Ron's HP2934A
Comment  PRNTLOC Print a file or listing on DSA printer
Comment  PRODUCTION                                       Use regular ALIAS
Comment  PROG file,dir=dsa,lib=utlr,size=1023,16 Make executable
Comment  QCAT                                             Quick print of catalog
Comment  RELATE                                           Call RELATE
Comment  RESTORE47                                        2647F disk to HP
Comment  RPREP usl=$oldpass,prog=$newpass,rl=utlr Prep for RELATE
Comment  RRUN  prog=$oldpass,stak=3000,sl=p        Run with RELATE
Comment  SEG listfile=$stdlist                            Call SEGMENTER
Comment  STARLP formal=LP                                 Set LP device
Comment  STORE47                                          HP to 2647F disk
Comment  TALK                                             Send messages
Comment  TDP                                              Run TDP.PUB.SYS
Comment  TLISTF                                           Listf for temp files
Comment  TYPE filename,opt=typfile,out=$stdlist   Type on consc e
Comment  UPDATE option=u   ("U2" for two USLs)    Update USLs via subs
Comment  WHO people=@.sea90                               SHOWJOB
Comment  ZSUB template,outfile                            Make arg substitutions
Comment  UINFO                                            This text
Comment
Comment  For the text of a particular UDC, type "HELP <udc>"
********
```

regular user names; for example, user JACK will have user names JACKR and JACKA, but his mail should be placed in JACK.MAIL. Mail is displayed to a user each time he logs on, and whenever he gives the NOTICE udc. A user may delete mail after having read it by purging the file from the .mail group.

Users may write memos to themselves by placing them in a file called memos in their personal group. The memos will be printed at log-on (or when the NOTICE command is given) when the user's log-on group is his personal group (as it usually should be).

A public notice, to be printed every time any SEA90 user logs on, may be placed in the file NOTICE.DSA.

6.3.5 Device Options

Seven major output devices are available to ALIAS system users at this time. One is the user's terminal; this is generally the default output device for udc's and programs. Printed output may be sent to either the system line printer (initialized by the 'starlp' udc and specified in commands as '*lp'), the SEA 90 daisy wheel printer (initialized by the 'daisy' udc and speicified in commands as '*daisy'), or to a printer slaved off the user's terminal. Slave printers echo text sent to the screen, and must be controlled locally at the terminal.

Graphic output may be sent either to the SEA 90 plotter (device 50), or to a plotter slaved to the user's terminal.

6.3.6 Operating Modes

Any user has the choice of running ALIAS either in production or development mode using the DEVELOP/PRODUCTION udc's to set mode switches. Users always start out in production mode, and ordinary users should in general run only in this mode. See Section 6.4.5 for a further discussion of the multi-mode feature.

6.3.7 On-Line Documentation and Help

There are four major sources of on-line documentation and help to the regular user. A primary source (and one which the user has a primary responsibility to keep current) is the CAT SEA90 account file cataloging system. Brief descriptions of the udc's in the public library are printed in response to giving of the UINFO udc. The data dictionary (contained in the .db group, and best accessed using the DBU module of ALIAS) describes the structure of the ALIAS data base. Finally, the ALIAS system provides a great deal of on-line help when it is being run; this help, however, is seldom concerned with the surrounding environment.

6.4 DESCRIPTION OF DEVELOPMENT ENVIRONMENT

The ALIAS system has many features to aid software developers. This section describes the features supporting each type of development activity (editing, compilation, etc.). It is meant to function as a reference guide. Section 6.6 contains a more narrative description of how to develop new ALIAS software.

6.4.1 File Structure

ALIAS software developers must have a more detailed understanding of the ALIAS file structure than ordinary users. Of the four categories defined in Table 6-1 (user, data base, system, and development groups), developers will make regular use of the development groups, and must understand the purpose of each system and data base group (so that files for new modules are placed properly). Table 6-4 (produced by the CAT on-line file/group cataloging system) describes the purpose of each system group.

The main system development groups in which developers will need to place files are .src, .obj, .prog, .incl, .rprocs, .screens, .sysrw, .doc, .merge, and .link. The groups used will depend on the form of the new module.

It is suggested that developers of new fortran programs have the system supervisor create a 'temporary' group for them in which they may place their working files. During development of a fortran program it is usually necessary to have many working files; such files should not be kept in the more public groups, such as .src, because they may interfere with the work of other development and maintenance personnel. An alternative to the temporary working group is to do all work in a personal group; this is feasible only when a single individual is involved.

Once developed, a new fortran module should have source code resident in .src (in one or more standard editor files, with routines appearing in alphabetical order according to the naming

TABLE 6-4

GROUP CATALOG LISTING FOR SEA 90 ACCOUNT

GROUPNAME	DESCRIPTION
CMDFIL	WAS CATALOGED 11/20/83 ; NOT NEEDED AFTER 8/16/86; ACCESS=DBA ALIAS menu system command files. These files are created automatica by ALIAS and should not be edited. The relation CFLIST.CMDFIL list information about all command files and is the real directory for t group.
COMPILE	WAS CATALOGED 11/20/83 ; NOT NEEDED AFTER 8/16/86; ACCESS=DBA A programmer's or system developer's group, COMPILE holds files which have batch (STREAM) job control language for executing compile compilations in batch mode. Slightly different in purpose than the batch UDC (FTNJ), these files can be useful when a large number of compiles need to be STREAMed; one can text them into the editor, globally substitute on the source file name, keep, and stream, and then repeat the last three steps; much less screen IO than FTNJ.
CRDB	WAS CATALOGED 11/20/83 ; NOT NEEDED AFTER 8/16/86; ACCESS=DBA CReate Data Base. CRDB holds RELATE execute procedure files which were used to create the ALIAS data base, and a set of ancillary data copying and file-copying procedures. New or changed data base files should be created using these procedures in the CRDB group, have dat copied into them from the old files in the host group, and then be written over the old files. This ensures that data is not lost duri a creation abort, and leaves a record of the data file creation for
DOC	WAS CATALOGED 2/10/84 ; NOT NEEDED AFTER 12/30/99; ACCESS=PUBR Contains on-line documentation support files and relations. Meant mainly for use by and support of programmers.
DSA	WAS CATALOGED 11/20/83 ; NOT NEEDED AFTER 8/16/86; ACCESS=PUBR DSA was the only group available in the SEA 90 account for software development when ALIAS development began. It now holds programs and data files which constitute improvements on the MPE operating system implemented by DSA. Utility processors like MSG and INCLUDE are there, as is UDC1, the main account udc file. It is a companion to the UTILS group, which has source code for the utility programs.
FMTFIL	WAS CATALOGED 11/25/83 ; NOT NEEDED AFTER 12/30/99; ACCESS=PUBW The ForMaT FILes group holds ALIAS format control/output control fil These files are used by ALIAS modules to guide their operations. For example, the contents and format of force level reprts and battle group reports are determined by the text of the format contro file which the module reads. Public files of this type should be stored in this group.
INCL	WAS CATALOGED 11/25/83 ; NOT NEEDED AFTER 8/21/86; ACCESS=DBA INCLude files. Include files hold fortran data structure source coc and are read in by the include pre-compilation processor which is executed by UDC's like FTNB, FTNJ, FTNLIST. These files should not be purged under any circumstances, and their contents should be changed only by knowlegable personnel. INCL is a companion to the SRC group.
LINK	WAS CATALOGED 11/25/83 ; NOT NEEDED AFTER 8/21/86; ACCESS=DBA

TABLE 6-4

GROUP CATALOG LISTING FOR SEA 90 ACCOUNT

GROUPNAME	DESCRIPTION
	Contains STREAM (batch) input files with JCL to PREPare or link each major ALIAS module. The names of the files in the group should be the same as those of the module program files (found in PROG). The LINK UDC takes a module name as an argument and looks for a file of that name in this group.
LISTINGS	WAS CATALOGED 11/25/83 ; NOT NEEDED AFTER 8/21/86; ACCESS=DBA Holds permanent file versions of FORTRAN compilation listings produced by the FTNJ batch compile UDC. This group is a primary offended when disk space gets short. All files except USLINIT and USLXREF may be deleted unless a programmer is actively using them.
MACRO	WAS CATALOGED 2/10/84 ; NOT NEEDED AFTER 12/30/99; ACCESS=PUBR Editor macros. That is, files containing editor commands which may be executed from within the TDP or HP editors at will. Note that most macros will work only with one of TDP or HP.
MAIL	WAS CATALOGED 11/25/83 ; NOT NEEDED AFTER 8/21/86; ACCESS=PUBW This group is used by the NOTICE mail/billboard logon UDC. Anyone wishing to send mail to someone else may create a file using one of the editors with the mail in it, and Keep the file under the target user's name (without the A/R suffix) in this group. The target user will receive the mail the next time he/she logs on.
MAKMENU	WAS CATALOGED 11/25/83 ; NOT NEEDED AFTER 8/21/86; ACCESS=DBA MAKe new MENU system. This is a testbed group, used when the ALIAS menu system is being regenerated. The menu system compiler program (MNUGEN) will operate only if the user is logged onto this group, and will write all of its files here so that the public ALIAS system is not disturbed. The new version may be tested in makmenuy to identify bugs before it is used to replace the working system. The group should be empty except when a generation/test is being conducte
MERGE	WAS CATALOGED 11/25/83 ; NOT NEEDED AFTER 8/21/86; ACCESS=DBA The merge should have one file for each major ALIAS fortran module. The files are segmented input files and should be Kept in an unnumbered format. The GLUE UDC takes a module name as an argument and looks for a file of the same name in this group, feeding the file into the segmenter to regenerate the main object code file for the module.
MNUFIL	WAS CATALOGED 11/25/83 ; NOT NEEDED AFTER 8/21/86; ACCESS=DBA MeNU FILes. ALIAS system files which are for use by menu system and which are read using standard fortran I/O. Companion group to MNUREL Most of these files are written by the menu system compiler MNUGEN. They should not be edited or deleted.
NUREL	WAS CATALOGED 11/25/83 ; NOT NEEDED AFTER 8/21/86· ACCESS=DBA MeNU RELations. ALIAS system files which are used by the menu system and which are accessed through RELATE. All of these files are created by the menu system compiler MNUGEN. Their contents is managed by the scenario system and should be changed with great care. Companion group to MNUFIL.

TABLE 6-4

GROUP CATALOG LISTING FOR SEA 90 ACCOUNT

GROUPNAME	DESCRIPTION
OBJ	WAS CATALOGED 11/25/83 ; NOT NEEDED AFTER 8/21/86; ACCESS=DBA OBJect code. The compiled or binary versions of all ALIAS system fortran programs and modules reside in this group. A system development group. Object code files with the exception of UTILIB may be purged in case of a severe disk storage, but this will sever inhibit the progress of development work and will make quick fixex bugs in the system impossible.
PROG	WAS CATALOGED 11/25/83 ; NOT NEEDED AFTER 8/21/86; ACCESS=DBA ALIAS system PROGrams. The executable versions of the main ALIAS program (mnurun) and of all modules reside here. Purging of these files will make the system inoperative.
PUB	WAS CATALOGED 11/25/83 ; NOT NEEDED AFTER 12/30/99; ACCESS=DBA Every account on an HP system should have a PUB group. This group is used for system file storage by software packages. RELATE security files are the principal occupants of the group at this time. The account library of segmented object code (file SL) must also reside in the PUB group because of the limitations of the PREPare command.
PROCS	WAS CATALOGED 11/25/83 ; NOT NEEDED AFTER 8/21/86; ACCESS=DBA Relate PROCedure fileS. These are RELATE EXECUTE-type files, i.e. files containing RELATE commands for processing. The group is meant to hold procedure files used by the ALIAS system. Public procedure files which are not part of the system should be stored in the common group, or a separate group should be created.
SCREENS	WAS CATALOGED 11/25/83 ; NOT NEEDED AFTER 4/29/00; ACCESS=DBA Contains procedure files which are read by the CRI builder to produce data entry and command screen systems, such as this account cataloging system. Many of these procedures call subroutines which reside in the account SL (SL.PUB); the source code for these routines is kept in SLPROC.SRC.
SRC	WAS CATALOGED 11/25/83 ; NOT NEEDED AFTER 8/21/86; ACCESS=DBA SouRCe code files. This group contains all fortran source code that forms program which are part of the ALIAS system. Source code is the English-like version of programs, as compared to the binary object code or executable program versions. Naming conventions are used to distinguish files containing code for one module from those with code for another. File names beginning with 'UTIL' hold utility subroutines which are used by many modules. Never purge source code files unless they are definitely redundant.
SYSRO	WAS CATALOGED 11/25/83 ; NOT NEEDED AFTER 8/21/86; ACCESS=DBA SYStem Read-Only files. This group holds files and relations used by the ALIAS system for which normal users have read priveleges but not change priveleges. Any system file whose contents need never be modified by a normal user should be placed in thsis secured group
SYSRW	WAS CATALOGED 11/25/83 ; NOT NEEDED AFTER 8/21/86; ACCESS=DBA

TABLE 6-4

GROUP CATALOG LISTING FOR SEA 90 ACCOUNT

GROUPNAME	DESCRIPTION
	SYStem Read/Write files. This groups holds ALIAS system and SEA 90 account files which ordinary users will need to both read and change The data base for this file cataloging system is stored in this grou. for instance. However, changes to these files will almost always be guided by programs; users should never work with the files without such programmatic supervision.
TEMPLAT	WAS CATALOGED 2/10/84 ; NOT NEEDED AFTER 12/30/99; ACCESS=PUBR An environment-support group which contains template or 'straw-man' files for various purposes. Most are input templates for th ZSUB environment utility processor.
UTIL	WAS CATALOGED 11/26/83 ; NOT NEEDED AFTER 8/22/86; ACCESS=DBA Source code for operating system utility programs written by DSA to supplement the capabilities of the MPE operating system. A companion to the DSA group, holds object code versions of the programs. File purging in this group should be done with great care.

conventions; see Section 6.2.2), object code in .obj, and program file in .prog. Global data structure source code (e.g. common blocks) should be placed in include files in .incl. Segmenter (object code editor) and link procedure files should be written and placed in .merge and .link. Documentation in .doc should be updated to cover the new module. Non-relation input files should reside in .sysrw (SYStem: public Read/Write access) or .sysro (SYStem: public Read-Only access). Throughout development, compilation listings should be purged promptly from .listings.

Developers of RELATE procedure (EXECUTE) files and BUILDER procedure files should place their finished products in .rprocs and .screens, respectively.

6.4.2 Development-Oriented User-Defined Commands

Table 6-5 lists the operating system procedures which support ALIAS software development, and Table 6-6 describes each one. These procedures are in addition to those listed in Table 6-2, most of which will also be of great use to software developers.

The majority of these procedures are further discussed in the sections below; the exception is 'screen', which executes the BUILDER using the file whose name is given as an argument to 'screen' as its input.

6.4.2.1 How Development UDC's Work

Most of the development-support procedures either take a file as an argument on the udc command line, and perform actions such as compilation or execution of that file, or they require a module name on tne command line, and perform actions such as program generation. There are three exceptions to this general rule, however, which merit special discussion: the ZSUB argument-substitution facility, editor macros, and the BATCH execution facility.

TABLE 6-5

SOFTWARE DEVELOPMENT UDC'S BY FUNCTIONAL CATEGORY

TEXT/CODE EDITING
col
edm
FORTRAN COMPILATION
ftn
ftnb
ftnlist
incl
OBJECT CODE EDITING
glue
make
merge
seg
update
EXECUTABLE PROGRAM PREPARATION
link
make
prog
rprep
EXECUTION
batch
rrun
screen

TABLE 6-6

SOFTWARE DEVELOPMENT UDC DESCRIPTIONS

TEXT/CODE EDITING

COL
: Prints a line of column numbers on the screen, e.g. Comment 90123456789012345678901234567890...etc. This is particularly useful for alignment of fortran format statements.

EDM macroname
: Runs the HP standard editor with terminal input redirected to a 'macro' file produced by the ZSUB facility. This allows certain functions, such as file concatenation, to be taken care of in a semi-automated fashion with only a few commands without the necessity of writing special-purpose programs. Current macros include EDMHELP (prints some help text regarding this facility), EXTRACT (extracts a text file section by line number), GRAB (extracts a subroutine by name), JMERGE (merges text files), and TMERGE (a different text merge).

FORTRAN COMPILATION

BATCH
: See under EXECUTION below.

FTN srcfile [,srcgroup=src,objcode=$newpass,incltempsiz=2047]
: Runs the include preprocessor on srcfile and compiles the result, by default placing the object code into the system work file $newpass. Of some use during debugging, but generally FTNB is preferred. Note that very large source files may cause the include preprocessor's output file to overflow; in such cases, specify a larger size than 2047.

FTNB srcfile [,srcgroup=src,objcode=srcfile.obj,inclsiz=2047]
: Similar to FTN, but places object code in a file with the same name as srcfile in the .obj group. This is the main command to use for interactive fortran compilation. Compilation listing is placed in a temporary file named Lsrcfile.

FTNLIST source [,objcode=$newpass,listing=templist]
: A 'workhorse' udc seldom called directly, ftnlist and incl in combination do the work for ftnb and ftn.

TABLE 6-6

SOFTWARE DEVELOPMENT UDC DESCRIPTIONS

Builds a listing file (named templist by default) and compiles the source code (into $newpass by default).

INCL srcfile [,srcgroup=src,incltempsiz=2047]
Also a 'workhorse' udc, incl implements a fortran include capability for the ALIAS software developer. A file named Tsrcfile is built (default size 2047) and the include.dsa program is run. Include.dsa reads srcfile and writes its code into Tsrcfile, locating any files named on statements of the form '%include filename' in the code and substituting their text in place of the %include line.

OBJECT CODE EDITING

GLUE modulename
Makes up a new main object code file for a module from its constituent object code (i.e. from the output of compilations of appropriate source code files). Most modules use code from several source files; a Segmenter run must therefore be made to combine (copy) the separate compilation outputs into a single linkable file. Glue runs the Segmenter interactively but with a procedure file containing all the necessary commands as its input. These procedure files are maintained in the .merge group, one per module, each named after its module.

Glue is much faster and less tedious than manual construction of the object file, provided a proper procedure file has been created.

MAKE modulename
Similar to GLUE, except that the object code file merge is done in batch mode, and a PREP (link) is done immediately following the merge. Make provides a fast and easy single command, background method of generating an executable image for a module.

MERGE objfile [,mode=segmergea]
A more dynamic means of combining object code than provided by GLUE, Merge uses the ZSUB facility to prompt for the names of object code files and segments, and then copies all of the specified segments into a temporary file named objfile. Three

TABLE 6-6

SOFTWARE DEVELOPMENT UDC DESCRIPTIONS

ZSUB templates allow two different operating modes for merge: the default, segmergea, prompts for the names of source object code files only, and copies all segments named "seg'" into objfile (which it creates); the alternative, segmerge, prompts for both file names and segment names.

SEG

An abbreviation for "SEGMENTER", Seg runs the segmenter in interactive mode.

UPDATE mode=replunt

Similar to MERGE in some respects, Update allows fast replacement of object code modules in an already existing object file by preparing a Segmenter command file and feeding the Segmenter with it. A common practice during debugging, after the initial MERGE is done to produce a linkable object, is to concentrate on only one routine at a time, modifying and recompiling that routine until it executes properly. Update allows the deletion and replacement of the code for such a routine in the linkable object without the necessity of a complete MERGE each time. Update uses ZSUB to prompt for the main USL file name, the file with the new object code in it (auxusl file in segmenter syntax), and the names of routines to be copied over in the USL file with versions in the auxusl file. An alternative operating mode, replunt2, allows code to be replaced in two file (i.e. two USL files may be specified).

EXECUTABLE PROGRAM PREPARATION

BATCH

See under EXECUTION below.

LINK modulename

Streams a batch job which PREPares an executable program file (into .prog) from the object code file (in .obj) named modulename. This is the easiest way of linking a program, but is dependent on the existence of an appropriate job control file (named modulename) in the .link group.

TABLE 6-6

SOFTWARE DEVELOPMENT UDC DESCRIPTIONS

MAKE modulename
Performs both a GLUE and a LINK in batch mode, allowing single-step background generation of program files. See the discussion above under MAKE.

PROG modulename
Does a LINK interactively (as opposed to through a batch job); PREPares an object code file named modulename.obj into a program file in modulename.prog, specifying utlr.obj as the Relocatable Library. Does not depend on existence of a stream file in the .link group.

RPREP [objfile=$oldpass,progfile=$newpass,rl=utlr.obj]
Does an interactive PREPare, allowing free specification of object file, program file, and Relocatable Library. Note that the program file will be a temporary unless it already exists as a permanent file, in which case the permanent file will be overwritten.

EXECUTION

BATCH template
Provides a general-purpose batch job execution facility; ZSUB template files from which stream (job control) files are constructed may be written to accomplish several different kinds of task, placed in the .templat group, and then executed using this UDC. Current templates support batch compilation and execution. Ftndsk, ftndsy, ftnlp, ftnull, and ftnxref all prompt for source file and group names, and the name of the executing user (so that a 'SUCCESSFUL COMPLETION' message may be sent); they send compilation listings to disk (in .listings), daisy wheel printer, line printer, black hole, and to line printer with cross references, respecitvely. The Prepjob template does a link in a manner similar to rprep, but in batch mode.

RRUN [progname=$oldpass]
Runs a program with the proper RUN statement argments to allow programmatic use of RELATE (initial stack space and the SL to use must be specified).

TABLE 6-6

SOFTWARE DEVELOPMENT UDC DESCRIPTIONS

SCREEN procfile
Runs the BUILDER in 'creator' mode with procfile as the application file.

6.4.2.1.1 ZSUB

One of the major roadblocks to construction of a useful environment on the HP 3000 is the lack of sufficient i/o redirection capability in the udc facility. It is not possible to execute a program from a udc and have the program take its input from the text of the udc rather than from the terminal, something which is often very useful.

Suppose, for example, that one wishes to extract a given subroutine from a large file of source code. The normal way to do this is to enter the editor, Text in the source file, Find the first line of the routine via F 'routine_name', find the end of the routine via FN ' END', and keep the lines between beginning and end via K output_file. This process is the same each time, the only difference being the name of the input file, the routine name, and the name of the output file. Ideally, one would like to give a command such as 'GRAB mabort,utlrm.src, temp', meaning 'get routine mabort from file utlrm.src and put it in file temp'.

This cannot be done on the HP 3000, because there is no way to pass the arguments on the GRAB command line into the editor, along with the necessary editor commands. The editor commands alone, however, may be passed in from a macro (an editor command file) by directing the editor to take its input from a file instead of the terminal. And this file can be constructed 'on the fly' by a fortran program which gets the editor commands from a template or skeletal file, prompts for the GRAB arguments, and puts them in the proper place relative to the editor commands. This is the ZSUB facility. Any udc which prompts for arguments before the main processor executed by the udc begins work is very likely to be using the ZSUB facility.

ZSUB is not as convenient or efficient as more conventional i/o redirection; the text of procedures is spread between udc file and template file, an extra processor must run, and

arguments are passed at the prompt, not on the command line. However, ZSUB provides much of the effective capability of i/o redirection, allowing many tedious tasks to be accomplished in a fairly automated fashion.

6.4.2.1.2 Templates

An ALIAS 'template' is typically a file which holds an incomplete version of something. The file contains special characters at the points where it is incomplete, and is usually processed by a program or procedure which finds the special characters and prompts for the values to replace them with. A 'finished' version is then written, which usually becomes input to yet another processor. For example, the generality of the ZSUB template format allows it to prepare command input files for programs as diverse as the editors, the segmenter, and the BATCH stream facility.

A sample ZSUB template is:

```
%0000Give source file name and routine name:
%ARG 2
T %1
FQ "%2"; FQ *(1)
K %2(*/"      END")
E
%END
```

This is the template for the GRAB editor macro, discussed above. The first line defines the substitution recognition character (here %), four diagnostic switches (only the first is of general interest: 1000 instead of 0000 causes echoing of the processed text to the terminal), and the text of the prompt for user input. %ARG 2 sets off a section to be processed when two arguments are input (here only two arguments are expected, so this is the only section, but the facility will handle variable numbers of arguments). The text of the section consists of editor commands and markers where the user's input is to substituted.

Other kinds of templates will be mentioned below.

6.4.2.1.3 Editor Macros

Strictly speaking, an editor macro is a command file or stored command that can be executed on command by an editor. Macros allows repetitive functions which require many editor commands to be performed with a single command. The ALIAS environment includes several different types of procedures, all of which function as macros do: these will be discussed in section 6.4.3.

6.4.2.1.4 BATCH

The BATCH udc uses the ZSUB facility to construct STREAM (batch job control) files for execution. It takes one argument, the name of the zsub template file to use; zsub then prompts for any additional arguments. At present, six BATCH templates (ftnask, ftnlp, ftndsy, ftnull, ftnxref, and prepjob) provide a means for easy batch fortran compilation/listing generation and program generation. Use of the batch procedures typically improves programmer productivity by freeing the terminal for other work while compilation and linking is in progress. The facilities of the .compile stream files and the GLUE/LINK/MAKE udcs are useful in similar fashion.

6.4.3 Source Code Creation and Maintenance

6.4.3.1 Editing

Source code creation/maintenance is an editor-intensive task. Unfortunately, there is no full-screen editor available on the host machine except HPWORD (word processor), which requires a special terminal. The best line-oriented editor available is the Text-Document Processor (TDP), which allows limited full-screen editing on any block-mode capable HP terminal (SCREEN command), permits limited simultaneous editing of two files (QuickText command), and has a global search capability (Find command). The other major editor is HP's standard EDITOR, which is a subset of TDP useful under some circumstances.

When creating new files, both of these editors will default the file characteristics to be fixed-length, 80 byte records, with text occupying the first 72 bytes and line numbers the last 8 bytes. These defaults are quite convenient for Fortran programming, but must be changed (to an 80-character text line length) when Builder procedure files are being created.

The principal support procedures available for source code maintenance are editor macros. These provide services which reduce the number of steps required to perform certain activities, which reduce mental fatigue by allowing performance of some functions with functionally oriented commands, and which aid the documentation effort.

These macros are either in files executed by the tdp/editor USE command (note that EDITOR and TDP do not always use the same macro syntax), are text files which may be included in new source via a Text or Join command, or are in special template files which must be executed using the EDM udc.

Table 6-7 lists the macros available by mode of execution, and briefly describes the purpose of each. The first group, 'Join' types, are technically not macros at all, but rather are template files which may be used by Texting or Joining them into a current editor work file. The second group are true macros, being collections of editor commands that are executable from the editor at the user's bidding. The third, editor procedures, are powerful but more difficult to construct macros written in a traditional programming language. Finally, the template files filled in by ZSUB as part of edm processing, and then executed by the editor, serve to extract subroutines or sections from large editor files, and merge individual files into the larger files in which source code is typically stored in .src. These last must be executed from MPE using the EDM udc; they cannot be executed while in the editor. They must have arguments substituted in their own text; because HP does not support passing of arguments

TABLE 6-7

EDITOR MACROS AND RELATED PROCEDURES BY MODE OF EXECUTION

EXECUTABLE FROM EDITOR/TDP BY TEXT/JOIN COMMAND

ABS.TEMPLAT
This file contains the skeleton of a fortran subroutine, including documentation header or abstract, RETURN, and END statements, all properly placed and formatted according to ALIAS standards. When writing new routines, programmers should pull this file into their work file, fill in the documentation in the abstract, and then write in their code. Use of abs.templat as a starting point also allow use of the runabs TDP macro, discussed below.

EXECUTABLE FROM EDITOR/TDP WITH THE "USE" COMMAND

EDMSEGL.MACRO (EDITOR ONLY)
Prepares a list of the files in the current log-on group, one per line, each left-justified. Very useful when a procedure or job file must be made up to perform the same operation on all files in a group (such as copying them into a new group).

ERR.MACRO (EDITOR ONLY)
Does a global Find on Error and Warning statements in fortran compilation listings, and prints out any found.

RCVFTN.MACRO (EDITOR ONLY)
Sets up the editor to receive fortran source code from DSA Superbrains.

RCVBLDR.MACRO (EDITOR ONLY)
Sets up the editor to receive builder procedure code (80-column line) from DSA Superbrains.

RUNABS (TDP ONLY)
Expects tdp work file into which file abs.templat has been joined, with line pointer at top of file. Walks user through the documentation abstract, stopping at each place where text needs to be filled in and placing user either in modify or add mode, as appropriate. Recommended for use in combination with abs.templat when new source code is being written.

TABLE 6-7

EDITOR MACROS AND RELATED PROCEDURES BY MODE OF EXECUTION

TAB
: Substitutes three spaces for every occurrence of "{" in a file, allowing implicit tabbing without use of the TAB key (which often does not work properly).

TDPBLDR (TDP ONLY)
: Sets up TDP to work with BUILDER procedure files. Mainly, sets line length to 80.

TDPFTN (TDP ONLY)
: Sets up TDP to work with fortran source code.

EXECUTABLE FROM EDITOR WITH THE PROCEDURE COMMAND

CNVTAB
: Complex editing function may be carried out using the HP editors by writing a fortran (or pascal, cobol, or basic) procedure consisting of one or more subroutines, the main one of which accepts arguments in a special format from the editor, by placing these routines into a Segmented Library, and by invoking them via the Procedure command. Cnvtab converts tab characters as sent by DSA Superbrains to the proper number of spaces, in case text has been transferred without first using the RCVFTN or RCVBLDR macros.

EXECUTABLE FROM MPE USING THE EDM UDC

EDM EDMHELP
: Prints a help file describing the operation of the EDM udc.

EDM EXTRACT
: Extracts a section of a text file by line numbers and keeps that section into a specified new file.

EDM GRAB
: Extracts a fortran subroutine by name (the routine must follow the ABS.TEMPLAT format strictly!) and places it into a specified new file.

EDM JMERGE
: Merges several editor files into one using the join command.

EDM TMERGE
: Merges several editor files into one by texting the first and joining the rest.

directly into the editor from a udc, it is necessary that the ZSUB facility prompt the user for arguments and then write an executable macro, using a template file as a model.

Each of these macros performs a function that is time-consuming or tedious if done manually; the effort to learn to use them will be rewarded by their provision of opportunities to take breaks of a few moments during editing while the computer performs the routine work.

6.4.3.2 Documentation

Fortran source code should be documented in two places as it is created. First, each routine should have the standard documentation header, filled in with such information as routine purpose, author, and method. Second, each routine should be listed in the entries.doc data base.

The template file in abs.templat and the runabs.macro editor macro support entry of the first sort of documentation. The template contains the skeleton of a documentation header, with a "!" at each point where text should be inserted. By Texting this file into an empty work file each time a new routine is to be created, the programmer can ensure that his documentation will appear in the proper format. After this step, the command "USE runabs.macro" (from TDP only) will run a macro which stops at each line in the template where there is a "!" and prompts for insertion or modification of text as appropriate.

Figure 6-2 shows both the empty subroutine abstract template and an example of one which has been filled in. Note especially that both formal parameters (subroutine arguments) and common blocks should be described as "in", "out", or "io". The descriptive information for common blocks should describe how they are used in the routine, not so much their contents. "Type" should be a keyword or keywords encapsulating the routine's role. Special attention should be given to describing the purpose of

Figure 6-2. ALIAS Subroutine Abstract

EMPTY

```
C     ! ******************************************************
$CONTROL segment=seg'
      SUBROUTINE  !
C*                        *** FORMAL PARAMETER DECLARATIONS ***
C*                                                 *** ABSTRACT ***
C#PURPOSE   !
C#AUDIT HISTORY
C         !name           !d-mmm-yy  AUTHOR
C#TYPE    !
C#FORMAL PARAMETERS
C!        !         !
C#COMMON BLOCKS
C!        !         !
C#CALLER  !
C#METHOD
C     !
C#LOCAL VARIABLES
C         !         !
C##
C*                      *** INCLUDES and LOCAL DECLARATIONS ***
%include lprnts
      equivalence (lprnt,lprnts(1))
C     ---LOCAL VARIABLES
      integer
      integer*4
      character
      logical
      real
C     ---FUNCTIONS
      integer
      integer*4
      character
      logical
C     ---DATA
      data
C*ENDDEC                                  *** END DECLARATIONS ***
C*    ---incoming debug report
      if (lprnt) write(ioutp,1000)  !
C*    ---error processing
900   continue
C*    ---outgoing debug report
999   continue
      if (lprnt) write(ioutp,1001)  !
9999  RETURN
```

Figure 6-2. ALIAS Subroutine Abstract

```
C*                                          *** FORMAT STATEMENTS ***
C*
4567890123456789012345678901234567890123456789012345678901234567890123456789
9012
1000  format(1h ,)
1001  format(1h ,)
      END
```

SAMPLE ROUTINE BASED ON ABSTRACT

```
C     GETMEM ************************************************
$CONTROL check=2,segment=utlr
      SUBROUTINE getmem(id,length,source,start)
C*                        *** FORMAL PARAMETER DECLARATIONS ***
      integer id,length,start
      logical source(1)
C*                                                 *** ABSTRACT ***
C#PURPOSE   Swaps data from extended memory into source array.
C#AUDIT HISTORY
C        MSCarey          11-aug-83   AUTHOR
C#FORMAL PARAMETERS
Cin      id        operating system id code for area
Cin      length    number of *2 words to swap
Cin      source    target array for words
Cin      start     starting position in extended mem to grab from
C#COMMON BLOCKS
C        none
C#CALLER various
C#METHOD
C     Calls dmovin.
C#LOCAL VARIABLES
C        locid     local version of 'id' arg, required for equiv
C                  to logical data type
C##
C*                     *** INCLUDES and LOCAL DECLARATIONS ***
%include lprnts
      equivalence (lprnt,lprnts(26))
      system intrinsic dmovin
      integer locid
      logical lid
      equivalence (locid,lid)
C*ENDDEC                                    *** END DECLARATIONS ***
      if (lprnt) write(ioutp,1000)length,id,start
      locid = id
      istart = start-1
      call dmovin(lid,istart,length,source)
      if (.cc.) 100,999,300
100   call mabort("+getmem: bad id code or length value+")
300   call mabort("+getmem: bounds violation; bad start+")
999   RETURN
1000  format(1h ,"getmem: moving ",i6," words into ",i6,
     1 " starting at word ",i6," of memory.")
```

local variables whose meaning is not immediately obvious in the code.

To ease the burden of adding to the entries.doc for the often large number of subroutines which comprise a new fortran module, a program is available. RRUN ADDSUB.DSA will execute this program, which prompts for each required field and makes appropriate entries in the documentation data base.

Documentation for RELATE and BUILDER procedure files must be entered manually into screens.doc and rprocs.doc; this is usually not a problem since the number of files to be documented at any one time is almost always small.

The entries.doc and subkeys.doc relations contain the main information for fortran subroutines; the iounits, xdseg, and lprnts relations list fortran io unit numbers, extra data segment id numbers, and lprnt diagnostic array elements that are already in use. The screens.doc and rprocs.doc relations document BUILDER and RELATE procedure files.

Software developers must also remember to document any files they create, including procedure files, using the CAT file cataloging system (relations fcatlog.sysrw and gcatlog.sysrw).

6.4.4 Compilation

This section and the one following apply only to development of new fortran subroutines or modules.

Fortran compilation is supported by a variety of procedures. It may be performed interactively or in a batch mode, may or may not generate listings, and may or may not require inclusion of global data structure source code prior to processing of the code by the compiler.

Note that the compiler takes no operating mode control arguments from its MPE run line, accepting only input and output file names from that source. Instead, specification of such things as i/o unit table size, routine segment names, and whether or not cross-reference listings are to be produced is made by inclusion of "meta commands" in the source code, of the form "$CONTROL command,command".

Four udc's support interactive compilation, though in practice only one (FTNB) is generally used directly. Udc FTN works like FTNB except that it places object code in the system temporary file $newpass instead of in the .obj group under the source file's name. Udc INCL, called by FTN and FTNB, executes a preprocessor which substitutes the text of include files named on '%include <filename>' lines into a temporary copy of the source code text, which is then passed on to the compiler. Udc FTNLIST actually calls the compiler, and also causes compilation listings to be placed into temporary files which are named L<source filename>.

The BATCH udc and files in the .templat group support batch compilation, with the include step performed and listings either discarded or sent to any of line printer, daisy wheel, or disk file in .listings. To use this capability, give the command BATCH <templatename>, where the template name may be any of 'ftnlp', 'ftndsy', 'ftndsk', 'ftnxref', or 'ftnull'.

Several files of batch job control language (stream files) are kept in the .compile group which provide capabilities identical to BATCH, but are designed especially to support the streaming of many compiles at one time. A problem with BATCH is that each invocation results in a good deal of terminal i/o, which can inhibit throughput. The files in .compile may be edited, with global substitutions being done on 'F' (source file name), 'G' (source file group), and '@' (user name). Following a Keep (to a temporary file) the Keep file may be streamed without

leaving the editor, and, in a typical case, another global substitution may be done on source file name. This method allows many compiles to be sent with virtually no screen i/o.

Table 6-8 indicates the appropriate method of compilation for a variety of circumstances.

6.4.4.1 The Include Facility

Most modern fortran compilers feature an include capability as an extension to the standard syntax, but HP fortran does not. It is therefore possible that the concept will be unfamiliar to some ALIAS programmers. An include facility looks for statements in a fortran program similar to "%include file1" (or perhaps "*call file1" for those familiar with CDC UPDATE), finds a disk file named, e.g., "file1", and substitutes the text in this file into the source code at the point of the include statement. The facility does not change the actual source file, but makes a temporary copy in which it does all expansion; this copy is then passed to the compiler for processing.

The include facility is used primarily for maintenance of source code which must appear in several routines or source files at the time of compilation; it allows only one copy of the given code to be maintained (thus ensuring consistency throughout), but still ensures that it appears properly at compile time. The primary example of such code in fortran is common block declarations.

As noted in Section 2, all ALIAS global data declarations MUST be maintained in the form of include files in the .incl group.

Since HP does not provide the include capability, a custom preprocessor was written. Udc INCL runs a fortran program (include.dsa) which takes the raw source as input and writes the source with the "%include <filename>" statements expanded into a

TABLE 6-8

SUMMARY OF METHODS OF FORTRAN COMPILATION BY CIRCUMSTANCE

Unless otherwise noted, all object code is placed in .obj

FOR BRAND-NEW SOURCE FILES USE:

FTNB srcfile,srcgroup
Interactive compilation is wise until all %include statements are debugged, since the include preprocessor aborts the first time it fails to find a file named on a %include line in the .incl group.

FOR ROUTINE RECOMPILATION WITH LISTING TO DISK USE:

BATCH FTNDSK
Batch compilation with listing placed in a file named after the source file in the .listings group.

FOR ROUTINE RECOMPILATION WITH NO LISTING USE:

BATCH FTNULL
Batch compilation with listing sent to null.

FOR ROUTINE RECOMPILATION WITH LISTING TO PRINTER USE:

BATCH FTNLP or
BATCH FTNDSY or
BATCH FTNXREF
Batch compilation with listing to system line printer, daisy wheel printer, or line printer with cross references included on listing.

FOR COMPILATION OF FILES WITH NO %INCLUDE STATEMENTS:

FORTRAN srcfile,objfile,listfile

temporary file. The temporary file is named Tsourcefilename (e.g. the source in file asgna.src would be expanded into temporary tasgna); the FTNLIST compilation udc specifies this temporary file to the compiler.

6.4.5 Program Generation

Program generation (generally known as linking) is a two-step process on the HP 3000. First, a single USL (object code) file must be made up which contains the object code for all routines that the program needs (except for routines in the Relocatable Library (UTLR.OBJ) or Segmented Library (SL.PUB)). This object code file is then processed by the linker (executed by the PREP MPE command); the name of the RL to use is specified as an argument to the PREP command.

Typically, object code for a program will be in several files, reflecting good source code modularity. The segmenter utility program is used to create merged object code files by copying code from one file to another. Entire segments of code may be moved or deleted (the default segment name is "seg'"; explicit naming of the segment a routine will be compiled into is encouraged via use of the $CONTROL SEGMENT=seg_name compiler meta-command). Alternatively, one routine at a time may be moved or deleted.

The segmenter is rather time consuming to use directly for most purposes because of the volume of commands it requires. However, it will accept command files, and several udcs have been created which take advantage of this capability. The GLUE udc accepts a module name as its argument, and looks for a file of that name in the .merge group which holds instructions for making up the linkable object code file for that module. MERGE performs a similar function, but allows interactive specification of the files/segments which are to be combined (through use of the ZSUB facility). The UPDATE udc also uses ZSUB to dynamically write a segmenter input file; UPDATE's files will update one or more

routines in an existing object code file with new code from another object code file.

Once the object code is ready, a program file may be generated using either the RPREP udc, the BATCH PREPJOB utility, or the LINK udc. RPREP accepts an object code file name and desired program file name and assumes that the utlr.obj library should be searched; it also includes arguments that cause the resulting program to be capable of process handling and data segment management, both necessary for use of RELATE programmatically. Note that the program file generated will be a temporary file; it must be made permanent using the SAVE command. BATCH PREPJOB takes similar action, but causes the work to be done in batch mode. The LINK udc takes a module name as its argument, and causes the given module to be re-linked in a batch job (a stream file named after the module must be present in the .link group).

The MAKE udc combines GLUE and LINK into a single batch job, regenerating a module in a single pass.

Note tnat ALIAS module program files should always be placed in the .prog group, while environment utility programs should be placed in the .dsa group.

6.4.6 Testing: Alternative Operating Modes

The ALIAS environment provides system testing support by allowing two separate versions of ALIAS to exist and be used simultaneously. After a new module has been developed and tested separately from the rest of the system to the maximum extent, it may be attached to ALIAS and receive a full scale test while production users are still isolated from it.

The development version of ALIAS is chosen by giving the DEVELOP udc command prior to the ALIAS (run) command. The choice

will remain in effect for the remainder of a session unless countermanded by the PRODUCTION command.

The development version consists of the main program TMNUR.PROG (as opposed to mnur.prog) and menu system files and relations in group .makmenu (as opposed to .mnufil and .mnurel). The regular data base relations and system files are otherwise used. Whenever a module is run, the system will attempt to execute a "T" version of it (e.g., TASGN.PROG rather than ASGN.PROG), falling back on the production version is no "T" version can be found. Constructing a test version of ALIAS therefore does not require duplicating the whole system.

LPRNTS are more easily set when the DEBUG command has been given; this causes ALIAS to prompt for lprnts settings during its initialization and during initialization of any fortran module.

6.4.7 Documentation and On-Line Help

There is a modest amount of system documentation on-line to assist software developers, all of it in the .doc group. A small data base (the entries.doc and subkeys.doc relations) contains information about every fortran routine, cross-referenced by keywords. Familiarity with this data base makes it possible to construct dynamic queries in RELATE to discover the names and purposes of routines which perform given functions (always consult this data base before writing a new utility).

Similar but less sophisticated data bases (screens.doc and rprocs.doc) list the BUILDER procedures and relate report generators on the system.

Of particular use are the files which record usage of key system resources. Lprnts.doc lists which lprnts diagnostic array locations are already in use by system programs (use of the same lprnt in different modules for different purposes is not wise). Iounits.doc similarly lists "ownership" of fortran io unit

numbers. Xdseg.doc tells which extra data segment id numbers are already in use--it is very important that a given data segment be used by only one module or process. Modnum.doc assigns a module number to each ALIAS module, which may be used to cross-reference into the sysusr.sysro relation in order to discover who has run priveleges for each module.

6.5 COMMUNICATIONS: NOTICES, MAIL, MESSAGES, AND MEMOS

The ALIAS environment supports three kinds of inter-user communication: a "bulletin board" which can be used to display messages to users when they log in (only high-privelege users may place messages in the bulletin board), mail which is automatically shown to the recipient when he logs in, and messages which may be sent from one logged-on user to another. Also, users may write notes to themselves to be displayed at log-in.

6.5.1 Notices

All notices must be placed in a file named NOTICE.DSA. The account logon udc (also called NOTICE, in udc2.dsa) runs a fortran program (source in pnotice.util) which searches for this file and prints it to the terminal if it is found.

6.5.2 Mail

The same fortran program searches for a file named after the user (minus his A/R suffix) in the .mail group, printing it to the terminal if it is found. The file should be a standard editor file with 72-character lines. Note that new high-privelege users (those with a single user name) must at this time have a hard-wired check in the pnotice program to make sure that the last character is not stripped from their name before the file search is conducted.

6.5.3 Messages

Messages may be sent between two logged on users with the MPE command TELL.

6.5.4 Memos

If a user creates a standard editor file called MEMOS in a given group, that file will be printed by the logon udc whenever he logs on with that group as his home group.

6.6 MAINTAINING AND EXPANDING THE ENVIRONMENT

6.6.1 Modifying the System File Structure

Changing the ALIAS file structure is synonymous with changing the list of HP file groups assigned to the SEA90 account. Groups no longer needed (e.g., those belonging to user names which have been removed) may be purged with the PURGEGROUP MPE command. This command can only be given by the account manager. It will purge the group and all its files, so care should be taken.

New groups (whether for personal storage for new users or to support system expansion) may be created by the account manager with the NEWGROUP command. The command should read:

```
NEWGROUP groupname;CAP=IA,BA,PH,DS;ACCESS=(security_spec)
```

The security specification will depend on the purpose of the group. Any combination of Read, eXecute, Lock, Append, Write, and Save may be assigned to any of the ACcount, Account Librarians, and/or Group Librarians. Personal groups should have (R,L:AC;R,W,X,S:AL;R,W,X,S:GL) as a specification, for example, with the user who "owns" the group being designated as the Group Librarian in the NEWUSER or ALTUSER command. Groups for development only should have (R,W,X,S:AL) to restrict access to Account Librarians only. Data base groups should have (R,W,X:AC;R,W,X,S:AL) so that user may read and change files in the group, while being unable to place new files in it.

An entry for a new group should always be made in the group section of the CAT file catalog data base as soon as the group is created.

6.6.2 Adding New UDC's

Most User Defined Commands maintained as part of the ALIAS system are in the file UDC1.DSA. New udcs of general interest should be placed in this file after being tested in a personal udc file by their developer. The procedures appear in alphabetical order in the file; be sure to place any new ones properly in sequence. Also, an entry for each udc appears at the end of the file in the UINFO udc; be sure to add an entry describing the new procedure.

Special purpose udcs for the account manager are in MGRUDC.PUB, while those for the Data Base Administrator (DBA) are in PPPUDC.DBA. The NOTICE udc, which implements the account bulletin board and mail capabilities, is in file udc2.dsa. Udc2 is a "system" udc, which new users will automatically be cataloged to.

All other ALIAS users must log off before the contents of udc1 or udc2 can be changed by an account librarian. The AL (DBA, generally) must then "unhook" from the udc by giving the SETCATALOG command with no files named, make the changes to the file, and reconnect to the file by giving the SETCATALOG command again with the names of all files he wishes to be cataloged to.

6.6.3 Device Additions

A number of changes must be made to the ALIAS system whenever a new device is added to the host computer that ALIAS should know about. The steps to be taken depend on whether the device is a terminal or a printer/plotter.

6.6.3.1 Terminals

First check to see if the terminal's type is mentioned on the User Environment Parameter menu. If so, and if the terminal is used only by dialing up via a general-purpose modem, no action can be taken. ALIAS cannot reliably identify the terminal since its only method of identification is by device line number, and any terminal could dial up through a general-purpose modem. The user is responsible for specifying the type of terminal he is using in the environment parameter menu whenever he logs on via modem.

If the terminal's type is already supported, and it is to be "hard-wired" into a specific cpu port, then it is possible to relieve the user of the burden of identifying his terminal type. Edit and recompile the settty routine in the recomp.src file, and replace the old object code in the utlr.obj library with the new. When editing, find the section in the routine which identifies the given type of terminal by line number, and add the new line number to the IF statement. Re-link ALIAS as the last step.

If the terminal is of a new type, several steps are required. First, edit MALIAS.DBA, placing a code for the new terminal on the options list of the terminal type line of the menu. Re-generate the menu system following the steps in Section 9.

Now add an entry to the BLDRTERM.SYSRO terminal definition file so that screen-oriented procedures can be run on the terminal. First, log on again (to flush your JCW table), run any BUILDER application using the SCREEN udc, and respond with "NEW" when prompted for terminal type. BUILDER's terminal driver modification subsystem will come up.

After completing entry of the new terminal specification, determine what ID number BUILDER assinged to the new terminal. This id number is the record number in the BLDRTERM file. Ask

for a list of existing terminal types and count the entries (left to right, top to bottom). The id number will be the numerical position of the new terminal's name on the list.

Edit the setccl routine in file recomp.src. Create a new section in this routine which recognizes the terminal code from the user environment parameter variable, and which sets the TERMTYPE job control word with the proper id number. Recompile this routine and replace the object code in utlr.obj.

Re-link the main ALIAS program (mnur).

6.6.3.2 Printers

A new printer must be given a code name on the user environment parameter menu, just as for a new terminal type. After modifying MALIAS and running mnug to regenerate the menu system, edit the lpset routine in file recomp.src. Add code to execute a file open call to the printer's device number when the user has specified the given code in his parameter menu. Recompile the routine and replace the object code in utlr.obj.

6.6.4 Utility Programs

This section gives the source code for all the fortran utility programs which service the environment, and the code for the BUILDER procedure which implements the CAT file cataloging system.

6.6.5 Templates

This section gives the text of every template file in the .templat group.

6.7 NARRATIVE DESCRIPTION OF ALIAS SOFTWARE DEVELOPMENT METHODOLOGY

The following sections describe step-by-step procedures for developing various kinds of ALIAS modules. They are meant as reminders and guides, and do not cover every eventuality. Read the rest of this manual before making use of them.

6.7.1 How to Develop a Fortran Program

1) Create the source code for all common blocks and other global data structures. Make sure to document each common block variable in the conventional format (see file uzrprv.incl as an example). Each common block should be in a separate file in the .incl group.

2) Create the subroutine source code. If you have access to a PC, do the creation on it using a screen editor and move the code onto the HP using a file transfer program. You may lose some code in the transfer, but the lack of a screen editor on the HP 3000 makes source code creation very time consuming.

   Make sure to place the standard subroutine abstract at the top of each routine, and fill it in completely. Pay special attention to purpose, method, and documentation of each important local variable. Don't use more than six characters for any variable name, and try to make all names mnemonic. The source of common blocks may be included at compile time by placing "%include filename" statements in the routine at appropriate points.

   If the source is being created on the HP, use the TDP editor. Give the command "JQ abs.templat" with the buffer empty, and follow it with the USE runabs.macro command. This will lead you through fill-in of the abstract.

   For programs of significant size, have the System Supervisor create a "temporary" group in which you place the source code. Keep each routine in a separate file. When you are through debugging, merge the files into a smaller number according to the rules given in Section 2 (no more than 500 lines per file, with the main routine at the top of the first file and all others in alphabetical order; file names to have the module name as their first four characters) and place the files in the .src group. Make sure to flush the individual object code files out of the .obj group after the source has been merged.

   Modules which use information from menu system parameter menus should have a separate initialization routine, maintained in the recomp.src file, which reads the /pvalue/ data structure for the parameter values and transfers the values into a module-specific common block. This isolates the module from the /pvalue/ data structure and reduces the number of routines which must be maintained in the recomp.src file.

3) Compile the routines. The first pass at each routine should be made interactively (using FTNB) to detect any mis-typed %include statements. Then use the BATCH FTNLP command to send a listing to the system line printer. Pick up the listings and fix the compiler bugs, then use FTNLP again until all routines are clean.

4) Prepare an object code merge procedure. Typically, object code from several files will have to be merged into a single file before your program can be linked. Identify which object code files are required: your own of course; usually dbif.obj; sometimes utlo.obj; perhaps others. Construct a SEGMENTER input file (along the lines of those in the .merge group) which creates a target object code file and then copies segments from the files you have identified into the target. If source code is merged into a smaller number of files when debugging is complete the contents of this file must be changed to reflect the new code storage structure. The procedure should have the core name of the module (e.g., ASGN as its file name).

5) Prepare a batch link procedure and a batch merge-and-link procedure, modeling them along the lines of those in the .link group. The procedure should have the core name of the module (e.g., ASGN) as its file name.

6) Run the merge procedure using the *GLUE* udc. Debug the procedure if necessary.

7) Try to link the program using the "RPREP objfil,exefil" udc. Any incompatibilities in subroutine arguments will be detected. Fix these before proceeding.

8) Create any necessary files or relations if you have not already done so. Edit the MALIAS.DBA menu definition file to add menus and options for your new module. Create a new test version of the System Core by following the steps in Section 9. This step can be delayed until debugging is complete if you do not need to add any parameters or lists to the menu system. If you do need parameters, add them now so that they will be avaialble during debugging.

9) If the program makes a call to the iniprc utility (which it should) you must run ALIAS and execute at least one fortran module so that the data segment which iniprc reads will be placed in your session memory. This step must be done only once for each time you log on to do debugging, since extra data segments remain in your session memory even after the program which requested them (here mnur) has terminated.

10) Make test runs of the program. Recompile using BATCH FTNLP or BATCH FTNDSK (listing to .listings) or BATCH FTNULL as necessary, re-merge with your GLUE procedure, and re-link with your LINK procedure (or use MAKE instead of GLUE+LINK). Note that you will be prompted for lprnts settings by iniprc if you have given the DEBUG udc command.

11) Add an entry point to the mrunp utility in the mrunit.src file to service your module. This entry point should have the name you gave for the module run choice menu option in MALIAS.DBA. Recompile mrunit and re-link mnur.

12) Try running the module from within ALIAS.

6.7.2 How to Develop a Builder Procedure

If the procedure is to be a part of the DBU, see Section 8. This Section is concerned only with stand-alone procedures.

1) Prepare the procedure file. As with fortran source, this is best done on a PC with a screen editor. If you must do it using TDP, give the command USE TDPBLDR.MACRO before entering any text. This will set the input line length to 80, allowing full-width layout sections to be defined.

   Every application is composed of some combination of data screens, menu screens, help screens, and comment screens. There is a conventional format for each of these types; a template for each type is contained in the files data.templat, menu.templat, help.templat, and comment.templat. These templates are designed as skeletons of DBU screens and as such contain a good deal of code. The code may be discarded, but use the layout sections as a guide.

   If you wish to use any subroutine screens from the DBU (this is encouraged) you must extract the text of these screens from the DBU files and include it in your own application file. Note that most DBU screens implicitly depend on the existence of the global variables defined as the first step in DBU execution. Inspect any screens you extract carefully to identify variables you will have to make global in your application. Some DBU screens also depend on the data dictionary files being open on pre-defined partitions.

2) Test the application using the SCREEN UDC.

3) Add an entry to either the mrunp or the mrunb routines in mrunit.src. An mrunp entry should be made if the application is a major one composed of many screens which is to be executed by a separate invocation of BUILDER (much as the DBU is executed). An mrunb entry should be made if the application is to be run by the BUILDER "service" son process that is part of the System Core. Applications run by this service process are provided with the name of the current user, the date, and the user's current scenario in the variables UZRNAM, TODAY, and SCENARIONM. Such applications should take care to close all files they open and purge all partitions they create before terminating so that the memory area of the service process does not become cluttered. They should terminate by a RETURN SCREEN, not by an EXIT.

4) Follow the steps in Section 9 to add a menu option for the application/module to the main menu system. Make sure the name given is the same as was used for the entry point in the mrunit routine.

5) Recompile mrunit and re-link mnur. Save the application in the .screens group.

6) Test the application from ALIAS.

6.7.3 How to Develop a Relate Procedure

1) Develop a RELATE EXECUTE file, i.e. a standard editor file filled with RELATE, CREATE, and/or GRAF commands. A good way to develop this file is to go through the procedure interactively, fixing bugs as they come up, and saving all the commands to a file with the LIST COMMANDS command when finished. Note that if CREATE is used a separate LIST COMMANDS from CREATE to the same file must follow the RELATE LIST COMMANDS. Try to keep all commands less than 70 characters in length; this will avoid problems with the editor's default line length later.

2) The file produced by LIST COMMANDS will not be in a very useful internal format. Run the HP standard editor with the EDN udc. Give the command JQ <file>, where file is the name you gave on the LIST COMMANDS lines. Keep this file, exit EDITOR, and run TDP to work with the resulting standard editor file.

3) Clean up the EXECUTE file, removing any lines that turned out to be in error during the interactive session. Make sure that any invocations of CREATE via the REPORT command now read REPORT:I.

4) Test the file from RELATE until it is satisfactory.

5) Most such files will need to include the name of the scenario the user is running as part of WHERE clauses. Hard-wire in the name of your test scenario for RELATE testing. Then replace this name with ' "{scenario}" ' everywhere. Before executing the file, ALIAS will search for instances of this string and replace it with the current scenario name.

6) Follow the steps in Section 9 to add a new choice menu option for execution of this file.

7) Add an entry point to the mrunrp routine in mrunit.src. The entry point must have the same name as the choice menu option you added to MALIAS. The entry point will cause the file to be executed by the System Core RELATE son process.

8) Move the EXECUTE file to the .rprocs group, recompile mrunit and re-link mnur, and test from ALIAS.

7.0 ALIAS SECURITY

This section will discuss the security requirements of ALIAS, the threats to ALIAS security, the generic security measures made available by the HP 3000 computer system and the RELATE DBMS, the specific security barriers and procedures which have been constructed using those generic tools, and procedures for changing the security environment. The remainder of the section is provided only for those responsible for ALIAS system management or with another valid need to know. See the MPE System Manager Manual for more information about the HP 3000's security capabilities and commands.

7.1 ALIAS RESOURCES REQUIRING SECURITY PROTECTION

7.1.1 The Data Base

ALIAS's data base is its key resource. Its contents must be protected against unauthorized inspection, and against wholesale destruction by purging of its files. The usability of the data base (its "integrity") can be lost if required data is deleted, or is not fully added, whether intentionally or unintentionally. It is possible for integrity to be lost without detection for a long period.

7.1.2 System Data

System data is that which is used by ALIAS programs and procedures for their operations, but which is typically not inspected or changed directly by users. System data resides in both standard MPE files and in RELATE files. ALIAS operations can be disrupted if this data is lost through improper purging of files or through improper modification of file contents.

7.1.3 System Outputs

System outputs, typically reports in one form or another, must be protected against unauthorized inspection. Only

authorized users should be able to cause the outputs to be generated.

7.1.4 System Operating Procedures

System operating procedures consist of the fortran programs, BUILDER procedures, and RELATE EXECUTE files which perform tasks for users. Only authorized users should be able to execute the procedures.

System operations can be disrupted if improper modifications are made to the procedures. Such modifications might lead to production of erroneous outputs, or open the way to breaches of security. Improper purging of the files which the procedures reside in will disrupt operations by forcing retrievals from tape backups.

7.2 THREATS TO ALIAS SECURITY

There are three major threats to ALIAS security, i.e., to the resources mentioned in the previous Section. The most dangerous but least likely are unauthorized users bent on extracting sensitive data or disrupting system operations. By their nature such users will cause maximum damage, but it is difficult for them to acquire the necessary access to do damage.

Similarly, authorized users might exceed their priveleges, perhaps inspecting data they should not, or purging or otherwise altering data in a way that causes damage. If bent on causing harm these users are very dangerous, but typically their motivation is exploration.

On balance, the greatest threat to security is that posed by authorized users who mistakenly cause damage. Accidental purging of files they have access to is an example of such damage. More difficult to retrieve is accidental destruction of data base integrity through improper updates, usually made with the best of intentions.

7.3 GENERIC SECURITY TOOLS AVAILABLE

This Section will discuss the basic tools available on the ALIAS host system for construction of a secure processing environment. The goal of security strategy design and implementation has been to combine these tools in a way that maximizes security while still permitting users to get work done.

The most powerful tool is the logon "gate" placed in the path of all prospective users. No part of ALIAS may be used or accessed (except printed outputs) by a person until he has supplied both a valid SEA90 account username and the password that goes with that username. Practically speaking, this means that no person may cause damage unless a valied username/password combination has been released to them.

Once logged on, a user's fundamental file-access and modification priveleges are determined by the privelege level assigned to the username by the SEA90 account manager. There are four such privelege levels: manager, account librarian (AL), group librarian (GL), and ordinary user.

There are six types of file access privelege for permanent HP files: read, execute (relevant only for programs), lock (keep other users from using a file while the given user is using it), write (change current file contents), append (add to the file), and save (add new permanent files to the account). Unfortunately, any user with write access also has "purge" access.

Files are in groups on the HP 3000; e.g., the file usrprv.sysro resides in the sysro group. It is possible to specify user access priveleges by group according to user privelege levels. Thus, ordinary users might be given only Read access to files in the sysro group, while account librarians might have all six access priveleges.

It is not possible to assign priveleges on a file-by-file basis, or on a user-by-user basis, except that a file can be made accessible to its creator only. However, passwords can be specified for groups as well as users, requiring a user wishing to access a file in a particular group to give the password for the file. Similarly, lockwords may be specified for individual files.

The RELATE DBMS can be made to impose additional restrictions on file access. Users can be given only programmatic or only interactive access to the DBMS itself, on the basis of user name. Permission to use files can be given or removed on a user-by-user and a file-by-file basis; users can be allowed to update relations but forbidden to delete data, etc.

Finally, special instructions and security checks can be placed in ALIAS fortran programs and BUILDER procedures. Users wishing to use ALIAS software must meet the requirements imposed by these procedures.

7.4 IMPLEMENTED ALIAS SECURITY PROTECTION SCHEMES

The main protection against unauthorized users and against authorized users exceeding their priveleges is lodged in the username/password logon gate. Once a person possesses a username/password with a given privelege level, that user will be able to take all actions consonant with the level. Thus, protection of ALIAS depends fundamentally on each user keeping their username and password a secret.

Each ALIAS user is assigned a privelege level. In practice, there are three such levels available. Day-to-day users are assigned the Group Librarian level, with librarian authority over the files in their personal group. The DBA and software development users are assigned Account Librarian status (organization of the file structure makes it necessary to give software developers high priveleges). Guest users who are to be

given only limited access to the system are assigned HP ordinary user status. Account Manager status is reserved for the single SEA90 account manager user.

Protecting ALIAS breaks down in practice into protecting three types of resource: files which users need not change, files which they will need to change, and processors.

7.4.1 Files Which Day-to-Day Users Need Not Change

Protection of these files is simple: they are placed in groups to which only account librarians have write/append/save access. For those which the day-to-day user need not ever see, such as fortran source or object code, all access can be shut off to those below the account librarian level. The full protection of MPE security can thus be brought to bear on these files, making them secure except against someone who has obtained an account librarian username/password.

7.4.2 Files Which Day-to-Day Users Will Need to Change

Protection of these files against unauthorized change by ordinary users is ultimately impossbile. The sophisticated programmer who possesses even a guest-user username/password can programmtically alter the contents of any file for which write access is not limited to account librarians. In practice, subtle alterations, the most damaging, can be made very difficult in many cases (seemingly major damage, such as purging of a file or rendering it unusable, is easily recovered by retrieving a copy from a backup tape, but subtle changes can go undiscovered).

The key to making a file difficult to change subtly is making it difficult to read; if someone cannot figure out how data is stored in the file, it is difficult to make unnoticed changes. Most data that ALIAS users need to change is stored in binary-type files which cannot be read by HP text editors. Unless a great deal of trouble is gone to, it is necessary to get access to the programs which manage these files in order to

change them. The files in question are managed either by RELATE or directly by ALIAS fortran programs. Control of access to these programs is therefore a fundamental tool.

7.4.3 Processors

The two most important processors to protect, RELATE and the ALIAS System Core, each have internal checking based on user names which makes restriction to authorized users possible. In addition, it is possible to secure individual ALIAS modules.

7.4.4 Details of the Security Scheme

Source and object code for ALIAS programs is in groups which limit all access to account librarians.

Data files and relations which ordinary users need only read are in groups which only account librarians have write access to.

Each user has a personal group, for which he is group librarian with full priveleges. Other day-to-day users have read and lock access to files in this group (lock required by RELATE for normal OPEN FILE execution), but no other access.

Ordinary files which day-to-day users need to change are in binary forms designed to be read by the programs which manage them. The programs protect against unintentional damage (except for file purging); subtle intentional damage would require the user to know the file format and write a program. Unintentional purging is unlikely since the files are isolated in their own groups.

It is necessary to give most users the ability to change data base files, but in order to preserve data integrity it is necessary that the changes only be made under the control of the DBU or another ALIAS module. This is enforced by forbidding day-to-day users use of interactive RELATE (via RELATE security),

but allowing programmatic use with full change priveleges for relevant data base files. Thus, they can run both ALIAS programs and BUILDER procedures (both of which use the programmatic interface), but are unable to make uncontrolled changes using the RELATE CHANGE, LET, ADD, and DELETE commands.

However, it is important to let users have interactive access to RELATE so they can make ad hoc queries and so they can make use of personal data bases. Each user therefore has two user names: one for running ALIAS which has data base write priveleges but no interactive RELATE access, and one for running RELATE interactively which has no write priveleges for ALIAS data base files but which does have full read priveleges (personal data bases are uncontrolled by RELATE security, but are protected by MPE security if placed in a personal group). The names are distinguished by an A/R suffix, e.g., John would have both a JOHNA and a JOHNR username.

ALIAS itself has some built-in security features. First, any user who tries to run ALIAS that is not on the access list contained in the sysusr.sysro relation will be met by an ALIAS abort. Second, data base write priveleges via ALIAS can be denied across the board by setting the ALTDB field in sysusr to 0 for the given user. Third, access can be granted to a subset of ALIAS modules by setting some of the M1-M50 fields in sysusr (the modnum.doc relation tells which module each of the 50 fields controls) to 0.

The DBU is a key part of ALIAS data base security. Individual users may be granted or denied DBU access to screens and relations (regardless of overall RELATE security settings), and may be restricted to one or a few of Read, Update, Modify, and Delete functions in any given screen. Most importantly, the DBU maintains data base integrity by making sure that any update

or addition takes place only if required companion data is present elsewhere in the data base, and by ensuring that any deletion also leads to deletion of subsidiary data (if any).

Finally, the ALIAS scenario system ensures that only one person can make use of a given scenario at a time, and that he sees and changes data only for the scenario he is using. This allows ALIAS to be multiuser without deadlocks and loss of data.

7.5 WEAKNESSES OF CURRENT ALIAS PROTECTION SCHEMES

One weakness of the security scheme is that any user with an "A" user name can make changes to data base relations by writing programs which use the RELATE programmatic interface. The advent of BUILDER has made it much easier to write such programs. On balance, however, this is not a great weakness since it presumes an active desire to do damage on the part of an authorized user.

It is very difficult to protect against a sophisticated authorized user with an active desire to do damage.

It is VERY IMPORTANT that usernames/passwords not be "leaked" to outsiders.

A more serious weakness results from the MENU version of RELATE. Technically, this version of RELATE is a programmatic and not an interactive interface to the data base. Thus, a full privelege "A" user will be able to make changes to ALIAS data base files using MENU. And MENU is very easy to use. There are only two solutions to this problem: make MENU inaccesible to ordinary users, or ask CRI to supply source code for MENU to which extra security checking would be added.

Another weakness of the system is that it is complex and subject to unintentional degradation by bugs in programs. Developers of modules are responsible for ensuring that their

products work ONLY with the data from the user's current scenario, for example. Failure to pay attention to this responsibility can result in modules which damage data base integrity. Care must be taken when adding DBU screens to add the necessary entries in the data dictionary to ensure proper integrity checking.

The final weakness grows out of the small number of privelege levels allowed by MPE security. In order for software developers to get work done, the Navy must have given them user names with very high priveleges. This is not a good idea in the long run.

7.6 PROCEDURES FOR MANAGING SYSTEM SECURITY

7.6.1 Adding New Users

Perform the following steps to add a new user to the system:

1) Choose a base user name (e.g., John) and verify that no user has that name by the LISTUSER command (you must be logged on as the account manager.

2) Create the user's personal group with the command
NEWGROUP basename;CAP=IA,BA,PH,DS;
ACCESS=(R,W,X,S:AL,GL;R,L:AC), all on one line.

3) Create the user names with the commands
NEWUSER basenameA;HOME=basename;CAP=IA,BA,ND,SF,PH,DS,GL
;PASS=password
NEWUSER basenameR;HOME=basename;CAP=IA,BA,ND,SF,PH,DS,GL
;PASS=password

Substitute "AL" for "GL" if the user is to be a software development/high privelege user, and only create one user name called "basename".

4) Log on as each user name to verify execution of the commands, and to connect the user to the system udcs. For each username, give the command SETCATALOG UDC1.DSA

5) Log back on as the DBA in the .pub group. Run RELATE and open the file sysusr.sysro. Add a record to this

file with the user's "A" name and proper settings for his overall data base read and write priveleges and for each module. To find out which modules each of the M1-M50 fields refers to, consult the RELATE file *modnum.doc*. Set the USERLEVL field value to 2 for day-to-day users, 3 for high-privelege users.

6) Give the user access to the data base using RELATE security commands. To do this, give the command SCREEN ADDUSR.DBA. This will run a BUILDER procedure which will draw on the list of files in FILINFO.DB to assign access priveleges to the user. The job can be done manually using RELATE ALLOW and PERMIT commands, but it is tedious.

7.6.2 Adding New Relations

When a new relation is added to the ALIAS data base (which can be done only by DBA) it must be secured. To do this, log on as DBA in the .PUB group and give the command SCREEN ADDFIL.DBA. This screen will take the list of users found in sysusr.sysro and assign security priveleges for the new file for each one.

7.6.3 Adding New Modules

To secure a new module (make it executable only by certain users) take the following steps when logged on as DBA:

1) Allocate an "M" field in the sysusr.sysro relation (one of M1 - M50). Do this by opening the *modnum.doc* file using RELATE, determining the lowest unused module number, and adding a record to modnum recording the fact that the number is now used by the module in question.

2) Open the sysusr.sysro relation and set each user's access to the new module by placing a 1 or 0 in the given M## field.

3) The menu system will ensure that only qualified users can run the module, as long as you record the module's number in the menu definition file MALIAS.DBA. Record the number on the line giving the module's internal name by placing a comma after the name followed by the number (e.g., ASGNE,2). If the number is omitted on the name line, it is assumed that anyone may run the module.

A similar method may be used to prevent unauthorized access to system menus.

Further protection can be provided by placing an additional check in the code of the module itself (this is difficult for

modules which are RELATE EXECUTE files). If the module is written in fortran, include the /uzrprv/ common block in your initialization routine, and check the <module number> location of the modprm logical array. If this element of the array is not .true., then the user has been denied run priveleges for the module in the sysusr.sysro file and processing should abort.

If the module is written in BUILDER, open the sysusr.sysro relation during screen initialization and check the proper field (M<module number>) in the record for the given user to ensure that it is set to 1. If it is not, EXIT.

7.6.4 Changing Passwords

To change a user's password, give the command ALTUSER username;PASS=newpassword. The command should be given twice for those users with both "A" and "R" names.

7.6.5 How to Add Independent File Groups

An "independent" group is one which is not the personal group of a given user. As shown in Figure 6-1, there are a large number of these. Their purpose is to hold system files, files belonging to a special project (e.g., a software development project), or files common to several users but not supported as part of ALIAS.

To create such a group, decide what access priveleges each level of users will have (basically, what can Account Librarians do and what can everyone else do?). Choose a name for the group that is a mnemonic as possible and that has not already been used (the REPORT MPE command when given by the account manager lists all SEA90 groups). Log on as the account manager, and type NEWGROUP name;CAP=IA,BA,PH,DS;ACCESS=([privs]:AC;[privs]:AL]) where [privs] is some or all of R,X,W,L,A,S.

7.6.6 How to Bootstrap RELATE Security

It may occasionally be necessary to re-create the RELATE security structure from scratch. The following steps were performed to put the current scheme into place:

1) Logged on as DBA in the .pub group, and with no other SEA90 users logged on, give the MPE commands PURGE RDBDD and PURGE RDBDDQ. These will purge the security information relation which RELATE consults every time a user requests an action.

2) Run RELATE. Give the CREATE DICTIONARY command to create a clean copy of RDBDD.

3) Give the command REORGANIZE RDBDD;RESERVE=10000. The default size for RDBDD, 4096 records, is unlikely to be large enough.

4) Give the command ALLOW ALL. This will set up an "open" security environment in which all users will be able to use RELATE freely on personal data bases.

5) Give the command DISALLOW SF-PERMANENT IN CURRJ, DB, DESCJ, HISTJ, LEGALS, MISCJ, PROJJ, SUPIND, YARDS, MAKMENU, MNUR, EL, SYSRO, SYSRW, SCREENS, RPROCS, PUB, MNUFIL, MAIL, FMTFIL, CMDFIL. Then give the command ALLOW SF-PERMANENT BY DBA IN [the same list of groups]. The two in combination will enforce the fact that only DBA should be creating new relations in any of these groups (and in fact no relations should be placed in the groups after SYSRW in the list).

6) For each day-to-day user, DENY IA-CI,BA-CI BY username A (e.g., DENY IA-CI,BA-CI BY JOHNA). This is best done by constructing an execute file using the editor or BUILDER. This will deny interactive access to RELATE by ordinary users in their ALIAS-capable persona.

7) For each data base file, DENY ALL ON filename. This will ensure that no one has any access unless specifically authorized.

8) For each data base file and each ordinary user name with the "R" suffix, PERMIT READ ON filename BY usernameR. This will allow interactive access by approved users.

9) For each data base file and each ordinary user name with the "A" suffix, PERMIT ALL ON filename BY usernameA. Though "A" users will not have interactive access (see step 6), they will be able to do anything programmatically, i.e. when guided by ALIAS software.

10) For each high-privelege user and each data base file, PERMIT ALL ON filename BY username. This will permit administrators and software developers full access.

8.0 THE SYSTEM CORE

The ALIAS System Core is a set of data structures and processors that provide services to users and programmers and that supervise and support application modules. Figure 8-1 diagrams the structure of the Core, which is composed of the data base, the command system, the scenario system, and the Data Base Updating (DBU) system. This Section will discuss the structure and operation of each part of the Core.

8.1 THE DATA BASE

Figure 8-2 diagrams the structure of the ALIAS data base. The data base as a whole is divided into data relations, the data dictionary, the legal field values reference library, and system support relations. Data relations are further divided into groups by subject. The data relations are a principal system asset and are the part of the data base that users are most aware of. However, the other parts are crucial to proper system operation.

The data dictionary is a set of 15 relations stored in the .db group. These describe data base fields and files and the DBU screens which service the files. They describe which fields are in each file, and which screens and report generators service each file. The proper way to modify the structure of the data base is to make changes or additions to the data dictionary and then use the BUILDER procedure stored in makfile.db to create or re-create the affected relations. This ensures that the data base structure and the data dictionary are always in agreement, and that the format and name of a given field are the same throughout the data base.

The DBU uses the data dictionary as the source for its on-line help capabilities, and to display lists of reports that can be printed from each screen (when the user chooses the "P" command option.

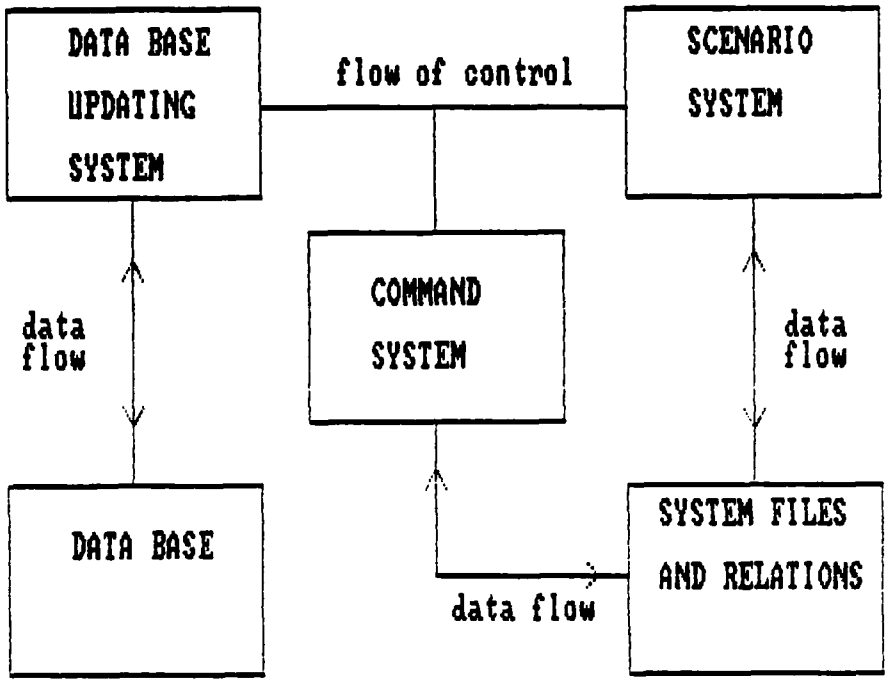


Figure 8-1. Structure of the ALIAS System Core

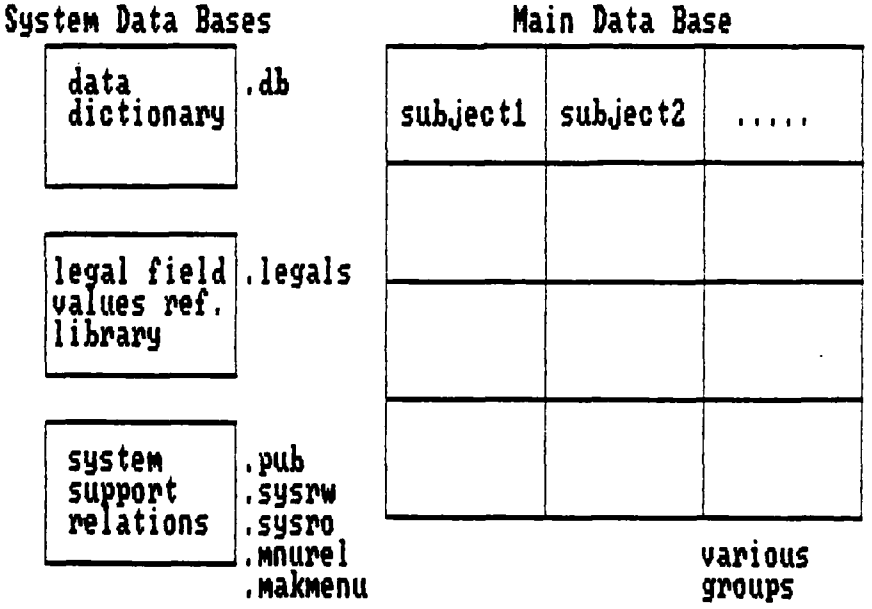


Figure 8-2. Overall Data Base Structure

The legal field values reference library is a set of relations in the .legals group. Each relation contains a list of legal (standard) values for a given data base field. For example, timunt.legals has records with the values DAYS, WEEKS, MONTHS, etc. By creating such a relation for a new field, making a proper notation in the fldinfo.db data dictionary file, and making a notation in the DBU screen(s) for the field's file(s), the DBU will ensure that no user enters a value for the field that is not on the legals list.

The purpose of legal values lists is to promote standardization of keywords for appropriate fields, so that programs and users making queries can rely on there being one and only one keyword used for each meaning of the field.

System support relations are those in which ALIAS operating information is stored. They include the relations in which command system parameter menu and list menu data are stored, the sysusr.sysro security data relation, the snusers.sysrw scenario usage tracking relation, and the cflist.sysrw list of command system command files. The contents of these files are managed by ALIAS itself and/or by the data base administrator.

The structure of the data relations was designed according to the rules of relational data base normalization, with a few exceptions made in the interests of improved performance.

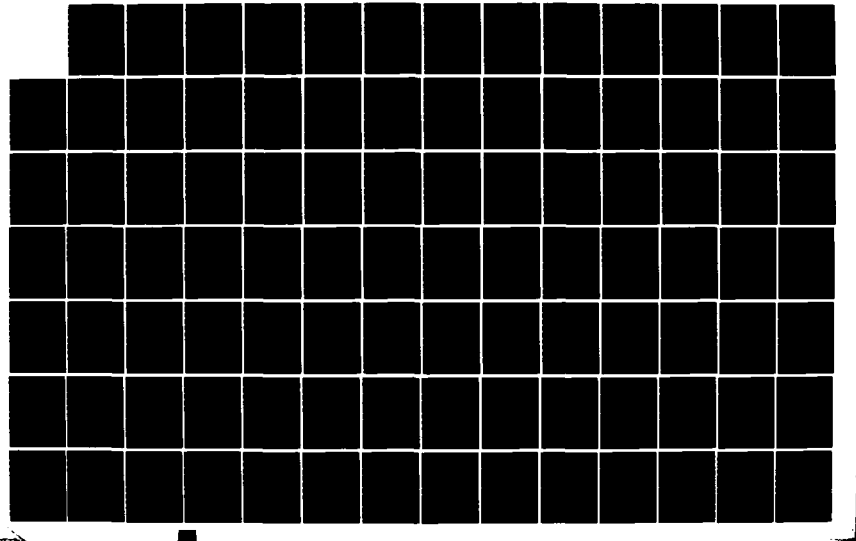
See the ALIAS Data Base Reference Guide for more information about the structure of the data base.

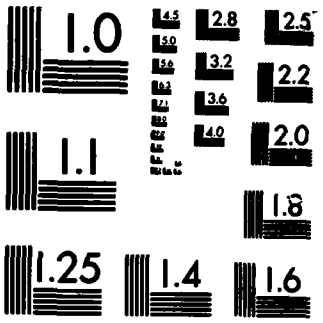
8.2 THE COMMAND SYSTEM

The command system is half of the heart of ALIAS, the other half being the data base. It manages system operations while being as user-friendly and programmer-friendly as possible.

AD-A150 422 ALIAS (ACQUISITION AND LOGISTICS INFORMATION AND ANALYSIS SYSTEM) MAINTEN..(U) DECISION-SCIENCE APPLICATIONS INC ARLINGTON VA M S CAREY ET AL. 3/7

UNCLASSIFIED 31 OCT 84 DSA-593-VOL-1 N00014-82-C-0813 F/G 15/5 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

The command system was designed to meet the following requirements:

1) ALIAS is composed of a number of modular processors, each of which executes separately from the others on command by the user. A user-friendly, menu-oriented way of informing the user of his options, accepting his choices, and executing the proper processors was required.

2) ALIAS minimizes the extent to which users must supply initialization values to modules, while still providing flexibility in terms of a wide variety of initialization values. A means of presenting the initialization options for each module to the user at the user's leisure, of allowing changes to the values, and of saving the changes from one session to the next was required.

3) System security and scenario security must be maintained during ALIAS sessions. A means of providing the current user privelege level and the current scenario name to application modules was required. A means of restricting application modules or menus to certain users was required.

4) It must be possible to add new modules to ALIAS without extensive reprogramming of the command system.

A dual-processor architecture for the command system was arrived at to meet the requirements. One program, called MNUG, reads an ASCII file which defines the menus that are to be displayed to users. This program writes out several files and relations and four files of fortran code (three include files and a subroutine). The code is compiled and merged with that of the second program, called MNUR, which is the program which is run when users give the "ALIAS" udc.

One part of MNUR's fundamental task is presentation of three kinds of menu to users: choice menus, where the choices imply either execution of a module or display of a different menu; parameter menus, where values of module initialization variables are displayed to the user and/or changed by the user; and list menus, which display lists of names which can be

assigned an "on/off" or boolean status by the user. The lists are also used for module initialization.

The other part of MNUR's fundamental task is module execution. The interface between MNUR and the modules is handled by the subroutine whose code is written by MNUG. When the user chooses a module-execution option, this routine is called and a module-specific subroutine call is made in turn. A number of utilities make it easy for developers to build the module-specific routine. An entry point may be added to the FORTRAN utility routine which runs a module as a son process, the one executes a file of RELATE commands, or the one which runs a BUILDER procedure.

This architecture makes the number and contents of the menus displayed to the user easy to change: just edit the menu definition file (in malias.dba) and run MNUG (actually, several additional file-management tasks are required---see Section 9 for details).

The mrunp module run utility routine will swap out System Core information (e.g., the current scenario name and system security status variables) to an extra data segment, which can be read into the standard common blocks in fortran modules via a single utility call. Also, MNUR can be instructed (through syntax in the menu definition file) to forbid a user from viewing a command system menu or executing a module if he does not have priveleges.

The module initialization values which appear in parameter menus are maintained in memory by MNUR, and are swapped to fortran modules by the same utility calls that swap security information. In addition, the current values for both parameter and list menus are always stored in system relations (created by MNUG), and are thus accessible to any module through RELATE queries.

Finally, MNUR provides a number of services, such as on-line help, creation and execution of stored commands consisting of a series of command system option choices, output device control (through a parameter setting in the User Environment Parameters menu), and dynamic lprnts setting (a choice option in the User Environment command menu).

Figure 8-3 diagrams the principal components of the command system.

8.2.1 Command System Data Structures and Flow of Execution

The command system uses ASCII files, binary files, relations, extra data segments, "regular" common blocks, "record" common blocks, a stack, and several pagers.

Paged storage is used extensively in order to conserve memory. The data and pointers relevant to system menus and command options are maintained in a set of disk files (in the .mnufil group), generally in a one-menu one-record fashion. A user request for a particular menu leads to a swap-in of the data for that menu from the disk files into "record" common blocks.

The total command system data structure is complex and very much intertwined with the flow of execution, making it necessary to discuss both at once.

8.2.1.1 Program MNUR

Figure 8-4 presents a calling tree of program MNUR (no utility routines are shown). Table 8-1 notes the purpose of each routine. Table 8-2 briefly describes the purpose of each of the principal include files. Table 8-3 describes each of the disk files and relations MNUR uses.

When executed, MNUR initializes itself by calling inimnu. Inimnu in turn calls initio and iniioc for basic i/o, csinit for the command stack utilities, rcinit for the DBIF utilities,

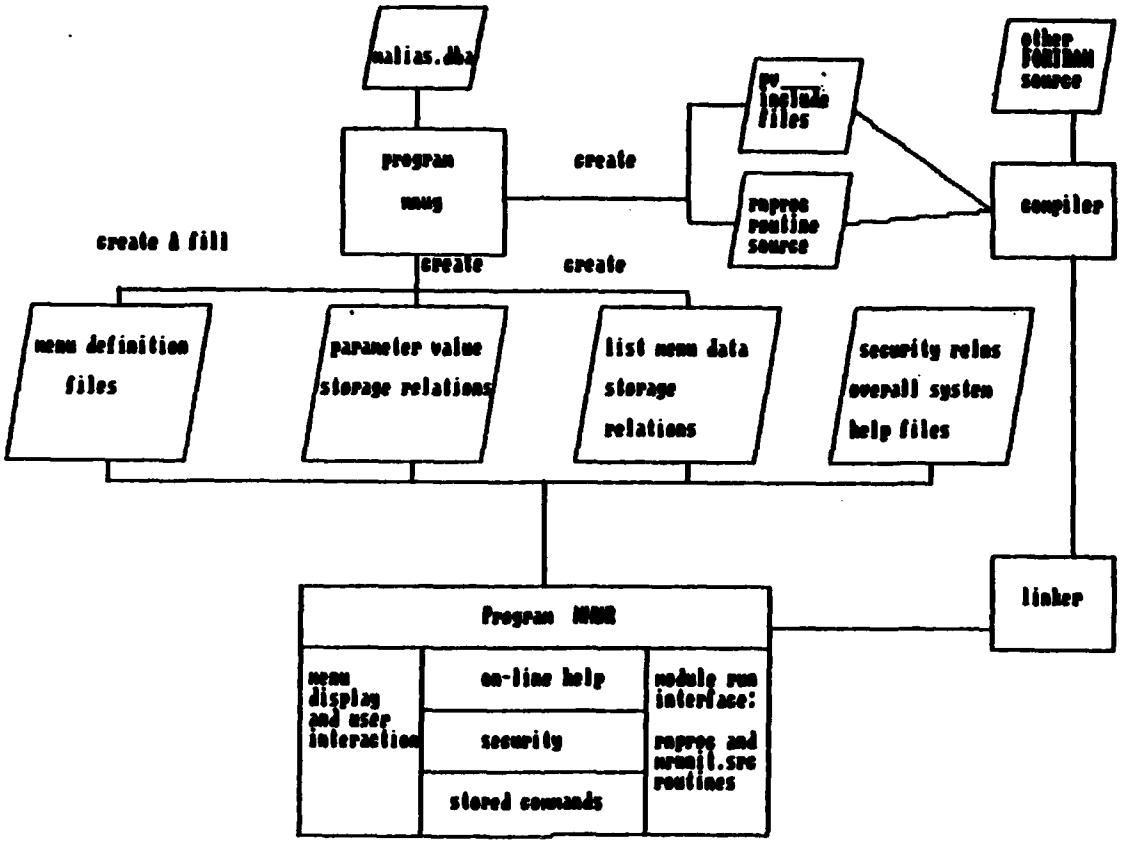


Figure 8-3. ALIAS Common D System Components

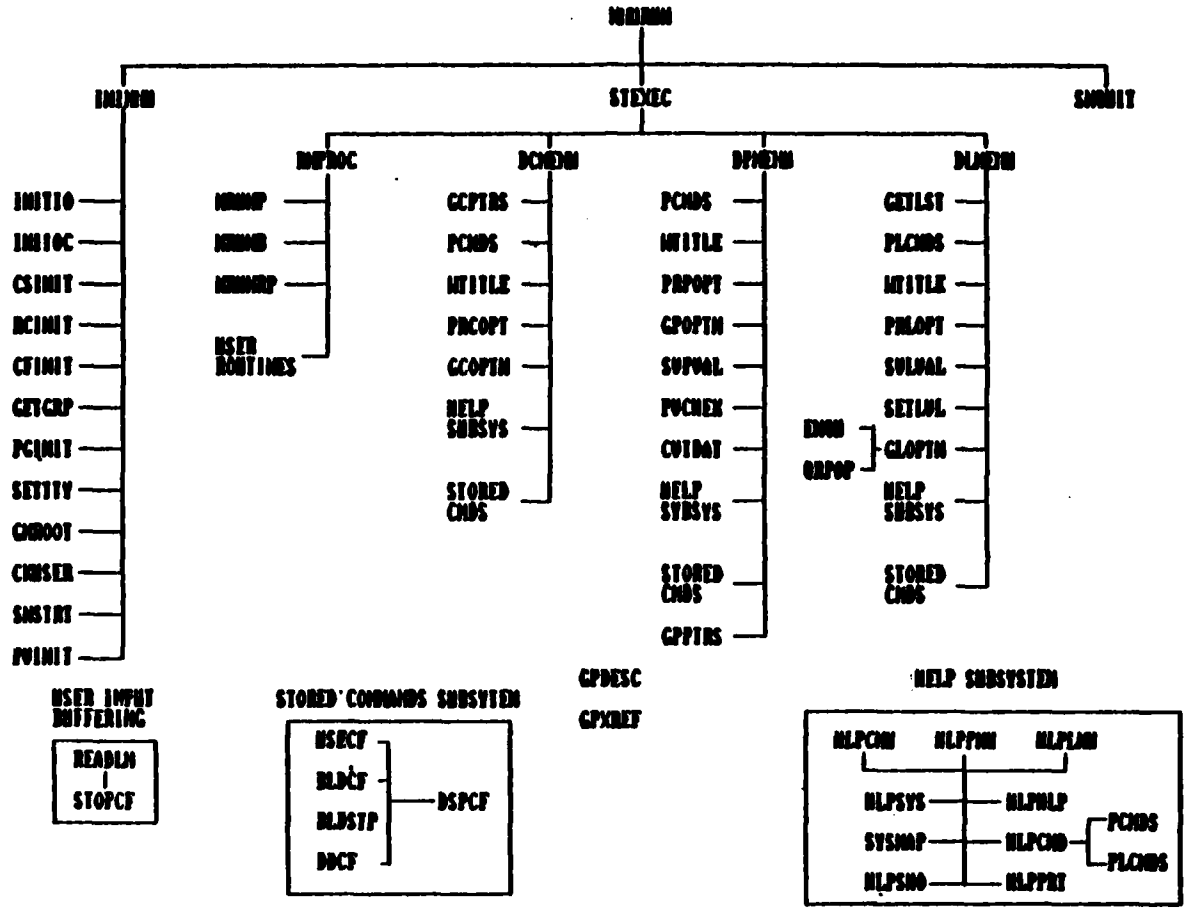


Figure 8-4. Program MNUR Calling Tree Diagram

TABLE 8-1

MNUR ROUTINES

ROUTINE	PURPOSE
BLDCF	BuiLD Command File. Prompts for required information, opens files, and sets flags which set the command file building process in motion.
BLDSTP	BuiLD SToP. Completes building of a command file by closing and saving the file and reseting flags.
CKUSER	Checks the user's security priveleges and records it in global arrays as part of initialization.
DCMENU	Display Choice MENU. Executive for display of choice menus and processing of user commands entered in choice menus.
DDCF	Delete Command Files. Utility for purging unwanted command files.
DLMENU	Display List MENU. Executive for display of list menus and processing of user commands there.
DPMENU	Display Parameter MENU. Executive for display of parameter menus and processing of user commands there.
DSPCF	DiSPlay Command Files. Utility used by the command file subsystem to print a list of the command files executable from the current menu.
ENQU	ENQUeue. Takes a command line entered in a list menu, breaks the line up into individual commands ("," separator), and places them in the command queue.
GCOPTN	Get Choice OPTioN. Retrieves user commands for choice menus and decodes them using the scrchr include file contents.
GCPTRS	Get Choice menu PoinTeRS. Retrieves information from the disk files when a new choice menu is to be displayed.
GETLST	GET LiST. Retrieves list menu candidates and statuses for display on a particular list menu.
GLOPTN	Get List OPTioN. Retrieves user commands for list menus and decodes them. Retrieval is done from the

TABLE 8-1

MNUR ROUTINES

ROUTINE	PURPOSE
	list menu command queue; if the queue is empty, gloptn prompts for a command line and reads it.
GMROOT	Retrieves pointers to the top choice menu as part of initialization.
GPDESC	Get Parameter DESCription. Retrieves a record describing the parameters on the parameter menu about to be displayed from the mpdesc disk file.
GPOPTN	Get Parameter OPTioN. Reads and decodes user commands given in parameter menus.
GPPTRS	Get Parameter menu PoinTeRS. Retrieves information needed to display the next parameter menu from the mpmenu disk file.
GPXREF	Get Parameter menu cross REFerence data. Retrieves information from the mpxref disk file for the next parameter menu. Mainly, field list for the storage relation.
HLPCMD	HeLP about standard system CoMmanDs. Prints some help to the terminal.
HLPCMU	HeLP for Choice MenUs. Help subsystem executive when subsystem invoked from a choice menu.
HLPHLP	HeLP about obtaining HeLP. Prints a description of how to use the help subsystem to the terminal.
HLPLMU	HeLP for List Menus. Help subsystem executive when subsystem is invoked from list menus.
HLPPMU	HeLP for Parameter Menus. Help subsystem executive when subsystem is invoked from parameter menus.
HLPPRT	HeLP PRinT. Help subsystem utility for printing help about a given menu or menu option. Accepts a pointer into the help file written by mnug based on the contents of the menu definition file and retrieves the help at that location.
HLPSHO	HeLP option SHOw. Prints the standard command options at the top of help subsystem menus.

TABLE 8-1

MNUR ROUTINES

ROUTINE	PURPOSE
HLPSYS	HeLP about the whole SYStem. Prints the overall system help (in file syshlp.sysro) a page at a time.
INIMNU	INItialize MNUr. Main initialization executive for MNUR. Calls utility subsystem initializers and opens required files and relations.
MNURUN	Main program unit for the System Core. Calls subsidiary executives.
MRUNP	Menu RUN Process. A command system utility, stored in mrunit.src, to which developers can add entry points for starting up modules (particularly son-process fortran modules).
PCMDS	Print major CoMmanD optionS at the top of choice and parameter menus (if user has Environment Parameter Menu switch set to do so.
PLCMDS	Print major List Menu CoMmanD optionS at the top of list menus.
PRCOPT	PRint Choice menu OPTions. Prints all the numbered choices for the current choice menu.
PRLOPT	PRint List menu OPTions. Prints all the candidates and their statuses for the current list menu page.
PRPOPT	PRint Parameter menu OPTions. Prints all the parameters for the current parameter menu.
PVCHEK	Parameter Value CHEcK. Checks to see that the user has entered a valid value when he specifies a change to a parameter.
PVINIT	Parameter Values data structure INITializer. Called by inimnu to fill the /pvalue/ common block with the current parameter values for all parameter menus.
QRPOP	list menu command Queue Retrieve-and-POP. Pops the next command off the command queue for processing.
RNPROC	RuN PROCedure. Source code for this routine is written by mnug to file rnproc. It is the interface

TABLE 8-1

MNUR ROUTINES

ROUTINE	PURPOSE
	between the command system and the routines which handle startup of each executable menu option (eacc module).
SETLVL	SET List menu VaLues. When the user chooses the "ALL" or "NONE" options in a list menu, or the "ALL" option in a parameter menu, this routine is called to set the logical status array elements corresponding to the list menu candidates to all true or all false.
SHOHLP	SHOw HeLP? Logical function which inspects the User Environment Parameter which permits turning the "standard help" (most used commands) at the top of menus on or off.
STEXEC	STack EXECutive. Main executive for MNUR operations. When a numbered choice menu option is chosen, or the user exits a parameter menu, this routine reads the command stack to determine what to do next (either display another menu or run a module).
SVLVAL	SaVe List menu VALues. Updates the relation in which list menu candidate statuses are stored when the user exits a list menu (or specifies the "ALL" parameter menu option).
SVPVAL	SaVe Parameter menu VALues. Updates the relation which stores values for the current parameter menu when the user exits that menu.
SYSMAP	print SYStem MAP. Prints the contents of the sysmap.sysro file, a page at a time.
USECF	USE Command File. Command file subsystem executive which redirects command input from the terminal to a command file of the user's choice.
WTITLE	Write TITLE. Writes the title line for any menu to the terminal.

TABLE 8-2

MNUR INCLUDE FILES

FILENAME	PURPOSE
CMENU	Holds a single choice menu description record (for the menu last displayed). ID information and pointers to text.
COMCFL	I/O unit numbers and status flags for the command file (stored commands) subsystem.
ENVIRN	Environment status information. Principally, name of the group(s) that system files and relations to be used are stored in (may be either .makmenu if user is running development version of system or .mnufil/.mnurel if running production version).
FIELDS	No common blocks, just data statements with the field list for /rcrd01/ (record buffer for reads from LCCREF relation).
FLD03	Field list for /rcrd03/ (CFLIST relation).
FLD07	Field list for SYSUSR security relation retrievals.
INCPAR	Fortran parameter statements giving the array dimensionings and string size limits for MNUR arrays and variables. Must be included before most other include files in routines.
INPUTL	Buffer holding last command line read from user or command file.
IOC	Main ALIAS fortran i/o unit number variables. ALIAS uses integer variables rather than fortran parameters to allow i/o redirection.
LPRNTS	Array of diagnostic print on/off switches.
LTYPAC	DBIF cursor index and field list for list type relation last opened. Allows relation to remain open during display of list menu, avoiding processing overheading of open-close by getlst followed by open-close by svlval.
LVAL	Data for the current list menu. Names of candidates (ALL are in memory, not just those displayed) and their boolean statuses, along with pointers to help text.

TABLE 8-2

MNUR INCLUDE FILES

FILENAME	PURPOSE
MROOT	Menu display system initialization data; mainly, pointer to root choice menu in file mcmenu.
PDESC	Holds description data for all parameters on the parameter menu displayed last. Pointers to text and data storage locations for each, and data type codes.
PMENU	Holds a record of descriptive data for the parameter menu displayed last. Where /pdesc/ holds data for each individual data, /pmenu/ has data pertaining to the whole menu.
PRMCRS	DBIF cursor indexes for the relations which are always kept open by MNUR. These are LCCREF, CFLIST, and SNUSERS.
PVALUE PVDECL PVEQIV	The common block, individual parameter variables, and equivalences of those variables to common block locations which for the parameter values data storage structure. Holds the current values for all parameters on all parameter menus. The menu system retrieves these values for display by index location in the *4 word-size block, while modules may retrieve the values by variable name. The variable names are the same as the parameter names given in the menu definition file.
PXREF	Holds the field list for the relation which stores the current parameter menu's values. The field list is required when the user has changed some values and the contents of the relation must be updated.
QUEUE	Storage for the command queue which is a feature of list menus.
RCRD01	Buffer for records retrieved from the LCCREF relation.
RCRD03	Buffer for records retrieved from the CFLIST relation.
SCENAR	Information about the current scenario. Proper scenario field value to use in queries on each open DBIF cursor, and write privelege statuses for each cursor.

TABLE 8-2

MNUR INCLUDE FILES

FILENAME	PURPOSE
SCRCHR	No common block, just fortran parameter statements defining all the valid command codes and a numerical index for each one.
SCREEN	Parameters defining the size screen MNUR expects.
SENPRM	No common block, just fortran parameter statements giving array dimensioning units and fortran unit numbers for the scenario system data structures and routines. Must be included before /scenar/.
STACK	Storage for the stack used by the command system.
TRNS03	Used to hold a copy of /rcrd03/. Required by the command file building routine, which saves a record describing a new routine here for addition to CFLIST when building completes successfully.
UZRPRV	System security information---user priveleges and id information.

ASNPRM	SYSRO	Assigner Parameter menu storage. Name specified in malias.dba input file, this relation created by mnug runs.
CFLIST	SYSRW	Relation listing the command system command files in the .cmdfil group and descriptive info about each.
ENVRN	MNUREL	User Environment Parameter Menu storage relation, created by mnug.
FLREPT	MNUREL	Force Level/Battle Group Report Generator parameter menu storage relation, as created by mnug.
LCCREF	MNUREL	List menu storage Cross Reference relation. Indicates which list type relation each list menus candidates and statuses are stored in. Created and filled by mnug.
MCMENU	MNUFIL	Choice menu definition records for the /cmenu/ common block.

TABLE 8-2

MNUR INCLUDE FILES

FILENAME	PURPOSE	
MHELP	MNUFIL	All help text read from the menu definition file by program mnug.
MNUROOT	MNUFIL	Pointers used to initialize the command system and force display of the root (top) command menu.
MPDESC	MNUFIL	Parameter menu parameter description records for the /pdesc/ common block.
MPMENU	MNUFIL	Parameter menu definition records for the /pmenu/ common block.
MPXREF	MNUFIL	Data records needed to define the relationship between parameters and their storage locations; one record per parameter menu; records are paged into the /pxref/ block.
MTEXT	MNUFIL	All menu title and option description text read from the menu input file.
SYSHLP	SYSRO	System level help. Editor-type file with text.
SYSMAP	SYSRO	System map. Editor-type file.
SYSUSR	SYSRO	User privelege relation. Searched during initialization by the ckuser routine to discover user priveleges; results initialize the /uzrprv/ block.

TABLE 8-3

FILES AND RELATIONS USED BY MNUR

FILENAME	GROUP	PURPOSE
VALCLS	MNUREL	List menu type relations. Created by mnug,
VALYDS	MNUREL	these store list menu candidates and
VLJTYP	MNUREL	statuses.
VLRJOB	MNUREL	

cfinit for tne stored commands subsystem, getgrp to dicover the user's identity and where he is executing from, pginit for the page printing utilities, settty to identify the user's terminal type, and ckuser to see if the user has priveleges. After the basic functions are initialized, inimnu opens all the files and relations that will be open throughout the user's session. As noted in Table 8-3, these include the files that MNUG stored menu definitions in, various files of help text, a relation listing stored commands and one which indicates which relation the data for each list menu is stored in, a file and a relation used by the scenario system, and a "black hole" for dumping undesired output.

Inimnu then reads the mnuroot.mnufil file to get a pointer identifying the first choice menu the system should display. It hands control to the scenario subsystem via a call to snstrt, which forces the user to choose the scenario he will use initially. Finally, values for all the parameter menus are loaded into the /pvalue/ common block (stored in files pvalue.incl, pvdecl.incl, and pveqiv.incl as written by MNUG) from their various storage relations.

After initialization, mnurun calls stexec, the main run-time supervisor. At this high level, the menu system can do three things: display a choice menu, display a parameter menu, and run a module. An initial design problem was caused by the fact that one of the options on a choice menu can be display of another choice menu. With a traditional program architecture, this would lead to the choice menu executive routine calling itself, an impossibility since FORTRAN does not support recursion. Instead, MNUR flow of control is managed by consulting a stack composed of two-entry cells. Each cell on the stack contains a code value and a pointer: the code indicates whether the system should display a choice menu (1), a parameter menu (2), or should run a module (3); the pointer is used to indicate which entity to display/run.

The gmroot initialization routine thus pushes a code of 1 and the pointer to the top choice menu onto the stack. The first call to stexec will therefore result in a call to the dcmenu choice menu management routine. Say a user chooses an option from the top menu that implies display of another choice menu: dcmenu pushes a code of 1 and the proper pointer on the stack and returns; stexec then just calls dcmenu again, but with the new pointer value. Say the user then wants to go back to the top menu: dcmenu pops the last cell off the stack and returns; stexec finds the commands directing display of the top choice menu at the top of the stack again, and does yet another call to dcmenu. A recursive system is thus effectively implemented.

Inside dcmenu, the first execution step is to retrieve information about the choice menu to be displayed. A record at the location indicated by the pointer taken from the stack is read from the mcmenu.mnufil file into the /cmenu/ common block by gcptrs. This record is in turn largely composed of pointers. Pointers into the mtext.mnufil file indicate the storage location of the text of the menu name, its title text, and the text for each option to display. The type code (for pushing onto the stack) of each option (1, 2, or 3) and a pointer value to the data for the option are also in /cmenu/.

Pcmds then prints the menu header of commonly used command codes, wtitle writes the menu title, and prcopt writes the text of each option (obtaining it using the pointers in /cmenu/).

Gcoptn prompts for the user's command and determines what it is. Choice menu commands are always either the number of one of the options displayed or else one of the system-wide command characters. The general approach that all MNUG command-processing routines use is as follows: strip leading and trailing blanks from the command; strip off the first character; attempt to find a match on the first character in the list of valid system command codes contained in the scrchr.incl include

file; if a match is found, return its command id numeral as a negative number to the calling routine (here dcmenu); if no match is found, see if the entire command is a number---if so, it is a choice of one of the menu options; return it as a positive number. If the command is neither a valid code nor a valid number, print a warning and prompt again.

Dcmenu takes the number returned and uses a computed goto statement to pass control to the appropriate section of the routine. If the number is positive, the appropriate stack operations are performed and dcmenu returns. If negative, either stack operations or calls to either the help subsystem or the stored command processing subsystem will result.

Dpmenu operates similarly to dcmenu in broad outline, but is somewhat more complicated because it displays dynamic data and accepts changes to that data. Its first act is to retrieve pointers from the /pmenu/ file via a call to gpptrs. /pmenu/ holds menu title and help text pointers, a pointer to the name of the relation where the values for the given menu are stored, and pointers to data for the first and last parameter on the menu in the mpdesc.mnufil file. Gpdesc is called to read pointers for each parameter into the /pdesc/ common block. These include the usual name and display-text pointers, data type code, the number of 32 bit words each parameter requires for its data value, and a pointer to the start of its data storage location in the /pvalue/ common block. Finally, gpxref retrieves the RELATE field list for the relation the parameter values are stored in and places it in the /pxref/ common block.

Since all parameter values are always in memory in the /pvalue/ array, all data required for display of the parameter menu is ready at this point. The menu is printed by prpopt, and commands retrieved from the user in the manner described above by gpoptn. Instead of passing back only a command code, however, gpoptn passes back a value as well (if the user has specified a

change to one of the displayed parameter's values). Pvchek and cvtdat ensure that the value supplied by the user is a valid one for the given parameter's data type, and perform any necessary data type conversions. When the user leaves the menu, svpval is called to update the value storage relation on disk, ensuring that the contents of the storage relations and the /pvalue/ array are always in agreement.

Dpmenu also is the "supervisor" of dlmenu. List menus may be displayed only by going through a parameter menu.

Dlmenu is again similar in broad structure to the other two routines. It need not make any calls for pointer retrieval; its menu name and pointers to title text and help text are all contained in the storage location in the /pvalue/ array which also contains the ALL/LIST value which is displayed for the "owner" parameter in the menu above. A call to getlst is made to retrieve the list menu contents from their storage relation. The menu is printed and commands are retrieved from the user.

Gloptn is similar to gpoptn and gcoptn, except that it will accept more than one command at a time. Two routines (enqu and qrpop) are used to manage a small queue, allowing the user to type in the numbers of several list members on a single line. The numbers (and any other commands) are fed one at a time to dlmenu for processing.

When the user leaves the list menu, svlval is called to update the contents of the storage relation on disk.

Rnproc, the module-execution overseer, operates much differently from the menu display routines. The code of rnproc is written by MNUG, and consists of nothing but calls to routines named in the menu definition file malias.dba, A computed goto at the top of the routine uses the pointer value on the command stack to determine which call statement to execute. Rnproc is

thus merely a reliable interface between the command system and whatever routine each module developer writes to supervise execution of his module.

A developer may always write a custom routine for calling by rnproc, but experience has shown that most such routines are quite similar. Time may usually be saved (and standardization promoted) by adding an entry point to one of the three utility routines in the mrunit.src file which were developed for this purpose. Mrunp is designed to run any programmatic module, whether it takes the form of a son process or a set of routines linked into the MNUR program. Mrunp will take care of the process handling, and will swap system data into the communication segment which is read by the iniprc utility (a good deal of coding for initialization can be saved by making a call to iniprc the first executable statement in your module). Mrunb will cause a BUILDER screen procedure file to be executed by the BUILDER service process which is started by the inibld call in inimnu. Mrunrp will execute a file of RELATE commands using the RELATE son process which serves the command system. In each case, the developer should add an entry point to the routine which then transfers control to the proper code section of the utility. See the documentation abstracts for these routines for more details.

When the user indicates a desire to terminate his ALIAS session, the command stack is flushed and stexec returns control to mnurun, which calls snquit to remove the notation in relation snusers.sysrw indicating that the user is using the current scenario. The program then stops.

8.2.1.2 The Stored Commands Subsystem

The stored commands subsystem appears as a black box on the calling tree diagram, callable by any of the menu executives. It provides the user with the capability to name and store any series of commands which he executes repeatedly, and then direct

the command system to execute the series by mentioning the name. The subsystem works by simple i/o redirection. When the user indicates that a given stored command is to be executed, the following occurs: the list of stored commands in cflist.sysrw is consulted to verify access, the file the command sequence is stored in is opened, the global standard input unit number is set to the unit number for that file, and the global standard output unit number is set to that for the black hole. Execution then continues "normally" until the end of the input file is encountered, when all unit numbers are reset to their standard values. The series of commands has been executed with minimum output to the screen.

To make up a new stored command, the user indicates a desire to do so, names it, and then just executes the command sequence to be saved. His input will be stored in the command file as it is entered (mistakes too!); the file will be closed and saved when he indicates that he is through building the file.

A key feature of the software architecture supporting the command file capability is the use of the readln terminal input utility by ALL MNUR ROUTINES. This routine takes care of resetting i/o unit numbers on end-of-command-file, and echos commands to command files being built.

8.2.1.3 The Help Subsystem

The command system offers a wide variety of on-line help; the display of this help is managed by the routines in the "help" black box shown on the calling tree. The text of help displayed comes from three sources: help about menus and menu options is stored in the mhelp.mnufil file as written by MNUG; it is retrieved and displayed on request using pointers. Help concerning standard commands is "hard-wired" into FORTRAN format statements. Overall system help and the system map are printed as they are found in the syshlp.sysro and sysmp.sysro files.

8.2.1.4 Program MNUG

A calling tree diagram for the command system generation program is displayed in Figure 8-5. This program writes the data structures and code required by MNUR to display run-time menus, taking its input from an ASCII file (stored in malias.dba by convention). Mnug must create the files and relations listed in Table 8-4, the text of the three include files pvalue.incl, pvdecl.incl, and pveqiv.incl, and the source code for the rnproc routine. Table 8-5 briefly describes the purpose of each routine, and Table 8-6 notes the purpose of each of the principal include files.

It expects its input file to be in three sections, one for each type of menu. The sections must appear in "reverse" order: first that for list menus, then parameter menus, then choice menus. An "END" card marks the conclusion of each section. Figure 9-1 gives an example of a MNUG input file.

Major sections are divides into subsections (e.g., one per menu), with these also separated by "END" cards. In the sample in Figure 9-1 the "END" cards are labeled for clarity, but MNUG does not interpret these labels: the meaning of a given "END" cards depends solely on its relative position in the file.

Where more than one piece of data is to be placed on a single line, the fields are usually expected to be delimted by commas. See Section 9 for a detailed description of MNUG syntax and usage.

8.2.1.4.1 Purpose of Input File Sections

The first major section does not actually define any list menus, but rather gives the names of each of the relations that list menu data will be stored in. Any given list menu displays two kinds of dynamic data: a list of the names on the list (referred to as "candidates"), and an associated yes/no status for each candidate. This data is used by modules for

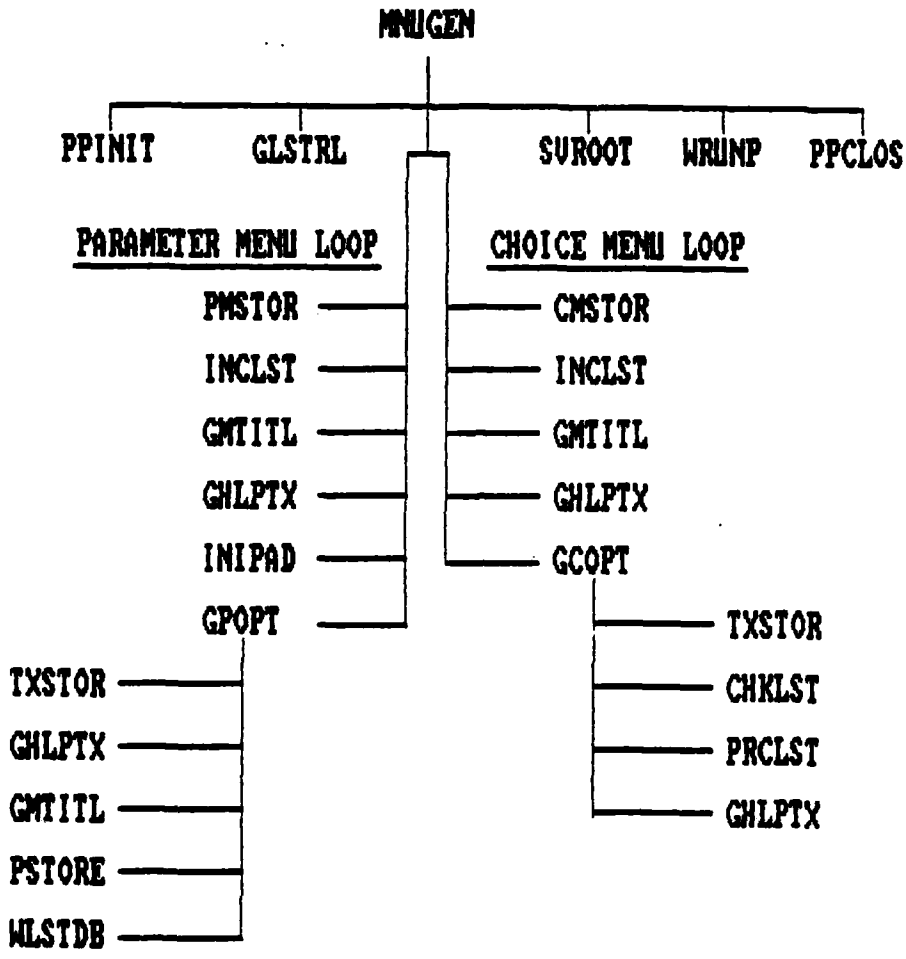


Figure 8-5. Program MNUG Calling Tree Diagram

TABLE 8-4

FILES CREATED BY EACH MNUG RUN

FILENAME	GROUP	PURPOSE
ENVRN	MAKMENU	User Environment Parameter Menu storage relation.
LCCREF	MAKMENU	List menu storage Cross Reference relation. Indicates which list type relation each list menus candidates and statuses are stored in.
MCMENU	MAKMENU	Choice menu definition records for the /cmenu/ common block.
MHELP	MAKMENU	All help text read from the menu definition file by program mnug.
MNUROOT	MAKMENU	Pointers used to initialize the command system and force display of the root (top) command menu.
MPDESC	MAKMENU	Parameter menu parameter description records for the /pdesc/ common block.
MPMENU	MAKMENU	Parameter menu definition records for the /pmenu/ common block.
MPXREF	MAKMENU	Data records needed to define the relationship between parameters and their storage locations; one record per parameter menu; records are paged into the /pxref/ block.
MTEXT	MAKMENU	All menu title and option description text read from the menu input file.
--- RELATIONS CREATED BY DIRECTION OF CURRENT MALIAS.DBA FILE ---		
ASNPRM		Assigner Parameter menu storage. Name specified in malias.dba input file, this relation created by mnug runs.
FLREPT	MAKMENU	Force Level/Battle Group Report Generator parameter menu storage relation, as created by mnug.
VALCLS VALYDS VLJTYP VLRJOB	MAKMENU MAKMENU MAKMENU MAKMENU	List menu type relations. Created by mnug, these store list menu candidates and statuses.

TABLE 8-5

PROGRAM MNUG ROUTINES

ROUTINE	PURPOSE
CHKLST	Given a name, searches for a match in a list and returns the location of the match. Used to ensure, e.g., that parameter menus named as options for a choice menu were defined in the parameter menu section.
CMSTOR	Writes a record of pointer information for a choice menu into the mcmenu file.
GCOPT	Reads and processes a subsection from the menu definition file which defines a single choice menu option. Such a subsection includes a line with the option type code and the option text to display, lines with the option-specific help, and a line with the option name. Pointers are set up and the name is cross-checked with lists and/or added to lists of command system names. The routine ensures that options of type 2 (parameter menu) already have had a parameter menu defined in the parameter menu section.
GHLPTX	Reads any help text subsection and stored it, returning a pointer and a length in numbers of lines.
GLSTRL	Reads the first section of the menu definition file, which consists of a list of names of relations in which list menu data will be stored. Creates each of the relations in the log-on group, and creates the LCCREF cross-referencing relation which tells mnur which relation each list menu's data is in (a single relation can serve more than one list menu).
GMTITL	Reads a menu title line from the menu definition file and returns it.
GPOPT	Reads and processes lines from the menu definition file forming the definition for a single parameter menu parameter. Such a definition includes the parameter code and name line, the line of text to display, and the help specific to that parameter. Pointers are set up, space is reserved in the /pvalue/ array, and the parameter name is added to the field list for the relation which will be created to hold the parameter menu's values. If the parameter is a list menu "gate", the name of the list menu and of the relation the list will be stored in are read, as well as the title of the list menu and

TABLE 8-5

PROGRAM MNUG ROUTINES

ROUTINE	PURPOSE
	its help text. More pointers are set up, and a record is added to LCCREF.
INCLST	Adds a name to a list and the location on the list to an array of pointers.
INIPAD	Initializes for gpopt: puts the scenario field on the field list for the next parameter menu storage relation and sets up some pointers.
MNUGEN	Main routine for program MNUG. An executive which calls others to do its work, it calls for initialization, for processing of each of the three major sections of the menu defintion file in turn, and for final writing out and saving of the data structures which will be needed by MNUR.
PMSTOR	Writes out everything having to do with the parameter menu just defined to make way for the next parameter menu. Writes pointer and field list records to the mpmenu, mpdesc, and mpxref files, and creates the storage relation for the menu's parameters.
PPCLOS	Closes and saves all files read and written by MNUG.
PPINIT	Supervises all MNUG initialization. Creates all the non-RELATE files which data will be written to, sets up some pointers, and prompts the user for the name of the menu definition (main input) file.
PRCLST	Adds a name to the list of module run subroutines to be called by rnproc, the routine which MNUG writes.
PSTORE	Takes care of details regarding storage of an individual parameter's data. Writes equivalence and declaration statements into the pveqiv and pvdecl files, and adds to the string which will be used to specify the structure of the storage relation when pmstor creates it.
SVPDSC	Writes a record to the mpdesc file.
SVPXRF	Writes a record to the mpxref file.
SVROOT	Saves the pointer to the "root" choice menu in the mroot file.

TABLE 8-5

PROGRAM MNUG ROUTINES

ROUTINE	PURPOSE
TXSTOR	Utility which stores any text line into the mtext file and returns a pointer to it.
WLSTDB	Writes a record into LCCREF indicating which list type relation a given list menu's data will be stored in.
WRUNP	Creates the rnproc subroutine which is the executive for all command system module run options.

TABLE 8-6

MNUG INCLUDE FILES

FILENAME	PURPOSE
CHLST	List of names of choice menus which have been defined.
CMENU	Holds a single choice menu description record (for the menu last displayed). ID information and pointers to text.
COLUSE	Keeps track of free space in each list type relation (each list menu takes up 1 column in a list type relation; current capacity is nine columns per relation).
COMCFL	I/O unit numbers and flags for the command file subsytem. This block must be initialized before the readln utility can be called.
ENVIRN	Environment status information, principally name of group system files and relations are to be stored in. Holds the output of the getgrp utility, which is used in filopn calls by ppinit.
FIELDS	Data statements with field list for /rcrd01/. Used when records are added to the LCCREF relation which is created.
INCPAR	Fortran parameter statements giving the array dimensionings and string size limits for MNUR arrays and variables. Must be included before most other include files in routines.
IOC	Main ALIAS fortran i/o unit number variables.
LISTYP	List of names of list type relations.
LMENU	Working buffer for storing list menu data as the menu is being defined.
MNUPRM	Fortran parameters, mainly defining maximum capacities.
MROOT	Menu display system initialization data record.
PARAMS	General-purpose fortran parameter statements.
PARLST	List of names of defined parameter menus.

TABLE 8-6

MNUG INCLUDE FILES

FILENAME	PURPOSE
PCREAT	Working buffer used during allocation of storage in the /pvalue/ buffer for a given parameter's value.
PDESC	Holds description data record for a the parameters on a single parameter menu (the one currently being defined when the parameter section of the input file is being read). Record is written to pdesc file when definition is complete.
PMENU	Holds description data record for a single parameter menu. Written to pmenu file when menu is fully defined.
PPINDX	Pointers to the next free space for parameter descriptions and parameter value storage.
PVALUE	Parameter value storage buffer; only parameter dimensioning size of buffer of interest to MNUG routines.
PXREF	Field list for relation which will store values for the parameter menu currently being defined. Written to mpxref file when definition is complete.
RCRD01	Buffer used to add data to the LCCREF relation.
READC	Text of the last line read from the input file.
RELNAM	List of names of all the relations which MNUG must create.
RPSUBS	List of names of all the module-execution entry points to be written into the rnproc routine source code.
TXTCNT	Working buffer for tracking the volume of help text read for a given item (e.g. the amount for a given menu).
UZRPRV	System security information---user priveleges and id information; required for its parameter dimensioning the maximum number of menus plus modules the system can handle from a security point of view.

initialization---an "on" status for a candidate indicates that processing for that named entity (e.g., a shipyard) should take place.

It seemed likely that some lists of candidates would be "popular" (used by more than one module/list menu), so storage for the dynamic list menu data was designed to minimize keeping of redundant lists of candidate names. Each TYPE of list (a list of ship classes, a list of shipyards, etc.) is stored in a single list type relation, regardless of the number of menus the list will appear on. A list type relation has eleven fields: scenario name, candidate name, and nine boolean status flag fields (C1-C9). Each menu has a dedicated status flag field, but all menus of the given type share the data in the candidate field. This arrangement currently limits the system to nine list menus of any given type, but it would be trivial to add capacity for more.

The list type relation section, then, gives the names of each list type relation, one name per line, and also the maximum number of characters in a candidate name (separated from each relation name by a comma).

The second major section defines parameter menus, and is composed of several kinds of nested repeating and non-repeating groups of data. Each menu to be defined has a separate subsection; there can be as many subsections a the configured limits of MNUG will allow. At the start of each subsection the menu's name, its displayed title, and the text of menu-level help are required. Following this are up to 15 subsections, one per parameter to be displayed. These give the parameter's name and data type code, the line of text which will describe it on the menu display, and the text of any parameter-specific help that will be available on-line. If the parameter is data type 6 it is a list menu "gate"---the user can cause a list menu to be displayed by setting its value to "LIST". The list menu must be

defined at this point: the name of the list type relation (must have been mentioned in major section 1), the name of the menu, the displayed title of the menu, and menu level help must be given.

The choice of names for menus and parameters is particularly important, since these will be used to name FORTRAN and RELATE entities. The name of a parameter menu will also be the name of the relation is which values for the menu's parameters will be stored. The names of the individual parameters will be the names of the fields within the relation (thus the values for a given parameter menu for a given scenario will be stored within a single record in the relation). The parameter names will also be the names of FORTRAN variables equivalenced to locations in the /pvalue/ run time parameter values storage buffer, making it possible for programmers to use the variables by name. Because of restrictions imposed by RELATE and in order to conform to the ANSI '77 FORTRAN standard, names of entities should never be longer than six characters.

The third and final major section defines choice menus. Similar in broad outline to the parameter menu section, it is simpler since choice menus do not display dynamically changing data. The section is composed of subsections, one per choice menu, each of which contains the name of the menu, its title, and its menu level help. This is followed by up to 15 subsections defining the numbered action choices the menu will display. Each choice is described by a code (1 for "display another choice menu", 2 for "display a parameter menu", and 3 for "run a processor") and the text which will be displayed for the option, followed by lines of option-level help text, followed by the name of the action-entity (the name of a choice or parameter menu defined elsewhere in the MNUG input file, or the name of a FORTRAN subroutine to execute).

8.2.1.4.2 MNUG Execution

MNUG is a fairly small program, but is complex in that it is manipulating a dense and complex data structure. Its main task is to create and fill the set of files and relations listed in Table 8-4 in such a way that the data is easily and rapidly retrievable by a MNUR which makes minimal use of directly addressable memory. Most of this data is retrieved in record form using pointers; MNUG must set up these pointers. In addition, MNUG does a limited amount of error checking on the information it reads from the menu definition file.

Program unit mnugen is the supervisor for MNUG execution, and contains much of the key logic of the processor. It first calls ppinit to create the necessary output file and to open the input file, and then glstrl which reads the first major section (list type relations) and creates the relations.

Mnugen then enters a loop for processing the parameter menu definition section. The data for the parameter menu last read is first stored (this step is skipped for the very first menu) by calls to svpxrf and svpdsc (to write /pxref/ and /pdesc/ records to the appropriate files), by a write to the pmenu file, by creation of the parameter value storage relation, and by writing of a record of default parameter values to this relation. The name of the menu is read and put on a list (inclst checks to ensure there are no duplicates), and the title and help text are read and stored in the mtext and mhelp files, with pointers placed in the /pmenu/ block (gmtitl and ghlptx do this). A subsidiary loop is entered in which gpopt is initialized by a call to inipad, and then is called to read the data for a parameter. The type and name of each parameter is read and stored and the name is added to the field list for the storage relation to be created for this menu. The descriptive text and help text is read and stored (pointers in /pdesc/). The proper number of *4 words is allocated in the /pvalue/ buffer for run-time parameter value storage; if the parameter is a list menu

"gate" the additional text for the list menu is read and stored, with pointers placed in the gate's area of the /pvalue/ array for writing to the parameter storage relation. Wlstdb is also called to write a record into the LCCREF list menu cross reference relation (when asked to display a list menu, MNUR queries LCCREF using the menu name as a key, receiving the name of the list type relation the menu's data is stored in and the name of the field within that relation that the yes/no status flags are stored in). Finally, pstore is called to write equivalences and type declaration statements into the pveqiv and pvdecl include files (so that programmers may access the proper location in /pvalue/ by name).

When all parameter menu data has been read, mnugen enters another loop which processes the choice menu section. The structure of this loop is similar to that for parameter menus: the data for the choice menu last defined is written to the appropriate files (here a /cmenu/ record is written to the mcmenu file), the name, title, and help text for the next menu are read and stored (pointers to the stored text in /cmenu/), and a loop is entered to read and process the data for each menu option. Gcopt performs this task, reading the option type code, the security code, the displayable text, and the help text. The texts are stored in mtext and mhelp as usual, with pointers placed in /cmenu/. Then the name of the action-entity (menu or subroutine) is read and placed on an appropriate list, with consistency checking where possible.

At the close of choice menu section processing all of the relevant relations and most of the relevant files are filled with the data necessary to display each menu. Svroot is called to write the pointer to the root choice menu to the mroot file for use in MNUR intialization, wrunp writes the text of the rnproc subroutine to the rnproc file, and ppclos closes and saves all files and relations.

8.2.1.4.3 MNUR Security Set-Up By MNUG

As noted in Section 7 of this manual, it is possible to restrict access to ALIAS command system menus and to ALIAS modules on a user-by-user basis. Security checking is done automatically by MNUR in the dcmenu routine: when a user picks any numbered choice menu option, his priveleges for that option are checked. The priveleges are stored in the modprv array of the /uzrprv/ block, which is initialized from the sysusr.sysro relation. This relation has one record per user name, and among others has fifty fields named M1-M50. Each of the fifty fields can be used to restrict access to a given menu or module: if the field value for a given tuple (i.e. username) is 0, the user does not have access; if 1 he does.

Part of the data structure set up by MNUG tells MNUR which choice options are secured, and if secured which of the fifty modprv array locations to consult for the user's privelege with regard to that option. The first line of each choice option subsection in the MNUG input file contains the option type code (1-3), a security code (0-50), and the text to display to describe the option. The security code is stored in the /cmenu/ block along with other pointer/code data. If zero, the given option is unsecured. Otherwise, the code is used by MNUR as the index of the element in the modprv array to consult.

Thus, to restrict access to a menu or module, a developer must BOTH place the proper index in the MNUG input file line for the given choice menu option, and must ensure that the settings in the referenced field in sysusr.sysro are correct for each user. Also, the field should be "reserved" by placement of a notation in the modprv.doc documentation relation so another developer does not use the same index/field by accident.

8.2.2 Command System Interfaces With Other Units

There are three kinds of unit which the command system interfaces with: modules, the scenario system, and i/o devices.

Since the primary purpose of the command system is to interface with modules, these are well defined and have been described in large measure above and in Section 9. The execution interface to a module of any kind is always via a subroutine call made in the rnproc subroutine (the one whose source code is written by MNUG). The called routine must always be linked into the MNUR program; it may either be a "working" routine which supervises the operation of the module and which perhaps calls other routines (i.e. the entire module is linked into the System Core), or it may be responsible only for creating a son process in which the module will execute.

Modules linked into the Core process may access the entire Core global data structure by including the proper common blocks in their routines. In particular, they may trivially access current parameter menu parameter values by including and referencing the pvalue-pveqiv-pvdecl data structure written by MNUG.

FORTRAN modules which execute as son processes may access a subset of the Core data structure if they make use of the data swap-out facility provided by the mrunp process creation routine and the complementary swap-in facilities of the iniprc module initialization utility. Mrunp can be instructed to write the contents of several key common blocks, among them /pvalue/ and /uzrprv/, into an extra data segment. The values are then read from the segment into the same common blocks in the data stack of the son process by a call to iniprc.

The scenario system is spread throughout the System Core, but a major portion of it takes the form of several user-executable modules linked into the Core process. These provide such services as scenario creation, modification, and listing, and interface with the command system in the normal manner. In addition, one of the Core initialization steps is a call to the

snstrt routine, which forces the user to choose a scenario to work with and which sets up the run-time scenario system data structure. On normal termination, the command system makes a call to the snquit routine, which removes the given user's name from the list of active users in the snusers.sysrw file.

One of the purposes of the scenario system is to ensure that no user changes data for any scenario but the one he is currently working with (and then only when he has change priveleges). Since command system parameter and list menu data are scenario-dependent, it may happen that a user is allowed to see the settings but not to change them. There is code in relevant places in command system routines which checks for change priveleges and bypasses the change attempt when changes are illegal. This avoids the abort which the DBIF does when it receives a request for an illegal change.

I/O device interfaces are handled in a somewhat kludgey fashion by the Core. Two classes of device are handled: terminals and printers.

The command system needs to know the user's terminal type in order to clear the screen before displaying each menu. During initialization it attempts to deduce terminal type from the line (host computer port) number the terminal is attached to, but an arbitrary decision must be made if the line is a dial-up modem. In either event, routine settty places the code name of the terminal type it has selected in the /pvalue/ memory location which corresponds to the terminal type option of the User Environment Parameter menu. The user may therefore override settty's choice by changing this parameter value.

Every time a screen clear is done (via a call to the sclear utility) the name stored in the /pvalue/ location is checked and the proper escape sequence retrieved (by a call to setccl). All logic, terminal type names, and escape sequences are hard-wired into the sclear, setccl, and settty routines.

Output device settings are handled in a somewhat similar manner: a call to the lpset utility leads to inspection of the "printer to use" parameter value on the User Environment Parameters menu, cross referencing of the printer name to a device number, and a file open call to this device. Lpset also returns the unit number the device is open on. Since most currently supported printers are spooled, a file close must be done before any output will result; this is done by a call to the lpsend utility.

As with terminal types, addition of any new output devices will require a change to the code of lpset.

8.2.3 Expanding the Command System

The command system was written to conform with ALIAS development standards, and therefore is fairly modifiable as FORTRAN programs go (with the caveat that, as the linchpin of ALIAS, particular care must be taken with the modifications). There are three ways in which the command system is particularly expandable.

The first and most obvious is that the actual menus displayed and modules executed by the command system are an almost totally data-driven quantity. It would be possible to use the basic programs MNUG and MNUR to implement several different systems, differing only in the contents of their menu definition files, their program file names, and the nature of their modules. It is thus possible to easily expand the command system as it appears to the user. The mechanics of doing this are the subject of Section 9 of this manual.

There are configured limits to the capacities of MNUG and MNUR, however. These are generally expressed in limits on the number of menus which can be defined, number of options on a menu, etc. The limits are defined by fortran parameter statements, making it possible to change them with a minimum of

editing and implement the changes by a simple global recompilation. It should be noted that some of the limits are interdependent, however (e.g. total system menus are now limited to 200, but only 50 of these could be secured using the 50 fields in sysusr.sysro), so that it may be necessary to change more than one parameter in some cases. Also, the motivation for many of the limits is the limited directly addressable memory afforded a program by the HP 3000. There is very littly stack space left in the Core process, so any significant increase in capacity might require that some of the current directly addressable Core data be paged (storage for the candidate/status lists for the current list menu is a good candidate for paging to an extra data segment).

Finally, because the top level executive of MNUR (stexec) is concerned only with the type of the next menu/module to display and with the pointer to the particular one desired, it is possible to add new types of menus and modules with a minimum of revision to existing code. To add the capability to work with a new type of menu to the system, define a type 4 stack code, write a routine to supervise handling of this menu, add a call to that routine to stexec, and add code to MNUG to read definitions for that menu type and produce appropriate data structures for your run-time routines. Alternatively, instead of RELATE and BUILDER modules being executed by entry points in the mrunrp and mrunb utilities, type 4 and 5 choice menu options could be added which would take care of the execution.

In making such changes, note that the stack-oriented architecture of the stexec executive does enforce display of menus in a strict hierarchical sequence.

8.2.4 Debugging Support

MNUG/MNUR have very limited internal debugging support in the form of lprnts, but do offer support for setting of MODULE lprnts at the start of their execution. The User Environment choice menu allows level 3 (DBA/development) users to set lprnt values dynamically (via a prompt); the contents of the lprnts array is part of the data that is swapped into a FORTRAN module's stack by the mrunp/iniprc utilties, thus making lprnt settings at the Core level effective in modules.

8.2.5 Code Files and Relevant UDCs

Figure 8-6 displays the User-Defined Commands which are relevant to Core/command system execution. The central udc is of course "ALIAS". Six file equations are given, and the USEVERSION Job Control Word is consulted to determine whether the production or the development/test version is to be run.

The first two file equations are required by RELATE, the third by BUILDER (which runs the DBU module). Record lengths are set for the FORTRAN terminal input and output unit numbers to override the configured defaults, which are not suitable on some of the PMS 392 HP 3000 lines. Also, Stdinx is specified instead of Stdin so that user input of a ":" as the first character in a response does not lead to an "END OF JOB" catastrophic abort. Finally the RELATE rdbin file specifier is set to an empty file (at one time RELATE would execute any rdbin file it found even on programmatic initialization).

The DEVELOP and PRODUCTION udcs set the USEVERSION jcw which is referenced by the ALIAS udc. The DEBUG and NORMAL udcs set the LPRNTON jcw, which is referenced by the inimnu and iniprc FORTRAN routines to determine if the user should be prompted for lprnts settings. This can be particularly useful from inimnu, since it allow lprnts sets at the very start of System Core operations.

FIGURE 8-6.

UDCs RELEVANT TO COMMAND SYSTEM/CORE

```
ALIAS group=relate,lib=p
FILE rdbecat.pub.sys=rdbecat.pub.!group
FILE  grecat.pub.sys= grecat.pub.!group
FILE bldrterm=bldrterm.sysrw.sea90
FILE FTN05=$Stdinx;Rec=-132
FILE FTN06=$Stdlist;Rec=-255;CCTL
FILE rdbin=empty.sysro
IF useversion=1 THEN
   ALTELL
   RUN  tmnur.prog.sea90;lib=!lib
ELSE
   RUN mnur.prog.sea90;lib=!lib
ENDIF
RESET rdbin
********
ALTELL
option list
comment   YOU ARE RUNNING THE DEVELOPMENT/TEST VERSION OF ALIAS
********
DEBUG
option list
SETJCW lprnton,1
Comment  WHAT DYNAMIC DEBUGGING SUPPORT THERE IS IS NOW ON
********
DEVELOP
option list
SETJCW useversion,1
Comment  ALIAS RUNS WILL NOW USE THE DEVELOPMENT/TEST VERSION
********
NORMAL
option list
SETJCW lprnton,0
Comment  DYNAMIC DEBUGGING SUPPORT NOW OFF
********
PRODUCTION
option list
SETJCW useversion,0
Comment  USE OF TEST VERSION OF ALIAS DISCONTINUED
********
```

Table 8-7 lists the disk files which hold code specific to the command system, and also those which are dedicated to Core linking. Note that the command system uses many utilities from the DBIF, UTLR and UTLO libraries (see Section 10). Routines appear in the source files in order as specified by the convention given in Section 2, with mnug... files holding MNUG routines and mnur... files MNUR routines. The mnug.obj and mnur.obj object code files hold the merged, ready-for-linking version of the object code. Mnug.merge and mnur.merge are Segmenter command files which specify how the several object code files produced by compilation are to be merged into mnug.obj and mnur.obj. Mnug.link, tmnug.link, mnur.link, and tmnur.link are batch JCL files which stream jobs to link (PREP) the merged object code into the tmnur.prog and mnug.prog program files (note that linking always produces a new development/test version, and that a production version of mnur.prog is to be produced by copying from a tested tmnur.prog.

8.2.6 Subroutine Abstracts

Documentation abstracts extracted from the routines which comprise MNUG and MNUR follow in alphabetical order, first those for MNUG followed by those for MNUR. Note that code and documentation for all ALIAS include files is given in Section 10.

```
C     MNUGEN*************************************************
$CONTROL SEGMENT=MNUGEN,FILE=1-99
      PROGRAM MNUGEN
C*                                                  *** ABSTRACT ***
C$PURPOSE  menu generator or preprocessor for command system
C  builds:
C     1) data structure describing menus for MNUR to display
C     2) special process executive subroutine 'runprocs'
C     3) creates storage relations for parameter/list menus
C     4) puts dummy data records into those relations
C$AUDIT HISTORY
C          MEMutchler      18 JAN 83  AUTHOR
C$TYPE     main program to mnugen
C$COMMON BLOCKS
Cin        incpar  global parameter statements
Cin        pxref   relation field list for each parameter menu
Cin        rpsubs  list of all special purpose subroutines
Cin        readc   holds file line number just read
Cin        cmenu   holds specifications of current choice menu
Cin        ioc     i/o file assignments
Cin        mroot   pointer into list memory at which top choice
C                  menu is specified
Cin        pmenu   holds specifications of current parameter menu
Cin        pdesc   holds descriptors for each parameter on a menu
Cin        ppindx  where last parameter value may be stored in
C                  pvalue
Cin        pvalue holds current values of each parameter on a menu
C$METHOD   intialize, read in parameter menus specifications,
C          save them, read in choice menus specifications,
C          make sure that all parameter menus that are
C          referenced had been specified, and save them.
C          Write all files that are needed by mnurun to disk.
C$LOCAL VARIABLES
C          text     a text string
C          line     a string of input
C          mname    current menu's name
C          cmnlist  list of all choice menu names REFERENCED
C          pmnlist  list of all parameter menu SPECIFIED
C          menucptr list of pointers into list memory where
C                   choice menu specs. are found
C          menupptr list of pointers into list memory where
C                   parameter menu specs. are found
C          nchmenus number of choice menus referenced, also
C                   max index into menucptr and cmnlist
C          lenname  non-blank length of a string
C          nomore   true iff nomore options to specify on a menu
C          eof      true if eof read form input file, an error
C$$
```

```
C     CHKLST*************************************************
$CONTROL CHECK=3,SEGMENT=MNUGEN
      SUBROUTINE CHKLST(STR,LSTR,LSIZE,MLSIZE,LIST,ELSIZE,
     1 PTRLST,IINDEX,FOUND)
C*                          *** FORMAL PARAMETER DECLARATIONS ***
      CHARACTER STR*(ELSIZE),LIST*(ELSIZE)(MLSIZE)
      INTEGER LSTR,LSIZE,MLSIZE,ELSIZE, IINDEX
      INTEGER*4 PTRLST(MLSIZE)
      LOGICAL  FOUND
C*                                                  *** ABSTRACT ***
C$PURPOSE CHECKS FOR MATCH IN 'LIST' OF NAMES AND LIST PTRLST OF MATCH
IN
C POINTERS TO LEADING DATA CELL IN LIST MEMORY.
C$AUDIT HISTORY
C         MEMutchler      18 JAN 83  AUTHOR
C$TYPE    mnugen utility
C$FORMAL PARAMETERS
CIN       STR       NAME TO GO ON LIST
CIN       LSTR      NUMBER OF CHARS IN STR
CIO       LSIZE     NAME-LIST SIZE.  WILL BE INCREMENTED IF STR NOT
C                   ALREADY ON THE LIST.
CIN       MLSIZE    MAX ALLOWED VALUE OF LSIZE
CIO       LIST      NAME-LIST
CIN       ELSIZE    MAX CHARS IN EACH ELEMENT OF 'LIST'
CIO       PTRLST    LIST OF MATCHING POINTERS
COUT      IINDEX    INDEX OF STR ON LIST
COUT      FOUND     TRUE IF STR WAS MATCHED IN 'LIST'
C$COMMON BLOCKS
Cin        readc    number of line just read from input file
C$METHOD
C  search 'list' for a match to 'str', index <= match's position
C  in 'list'.
C$LOCAL VARIABLES none
C$$
```

```
C     CMSTOR***********************************************
$CONTROL CHECK=3,SEGMENT=MNUGEN
      SUBROUTINE CMSTOR(IPTR)
C*                        *** FORMAL PARAMETER DECLARATIONS ***
      INTEGER*4 IPTR
C*                                                  *** ABSTRACT ***
C$PURPOSE  writes the /cmenu/ record describing the choice
C         menu just defined to the mcmenu storage file, i.e.
C         pages out.
C$AUDIT HISTORY
C          MEMutchler            18 JAN 83  AUTHOR
C          MEMutchler       16 FEB     TESTER  (program tstcmen)
C$TYPE     mnurun utility
C$FORMAL PARAMETERS
Cin        iptr    pointer into file dcmenu at which data belongs
C$COMMON BLOCKS
Cin        cmenu   data for file dcmenu
C$CALLER   dcmenu
C$METHOD   determine how many *4 words fill common/cmenu/ and
C          save them in file dcmenu
C$LOCAL VARIABLES
C          none
C$$
```

```
C     6COPT*************************************************
$CONTROL CHECK=3,SEGMENT=MNUGEN
      SUBROUTINE 6COPT(NOMORE)
C*                        *** FORMAL PARAMETER DECLARATIONS ***
      LOGICAL NOMORE
C*                                                 *** ABSTRACT ***
C$PURPOSE reads a single choice menu option specification
C$AUDIT HISTORY
C         MEMutchler      18 JAN 83  AUTHOR
C$TYPE    mnugen utility
C$FORMAL PARAMETERS
Cout      nomore  true only if 'END' as input
C$COMMON BLOCKS
Cin       incpar  global parameter statements
Cin       readc   holds iline, last line # reaad from input file
Cin       ioc     i/o file assignments
Cio       cmenu   holds specs. for current choice menu
C$CALLER  mnugen
C$METHOD
C  READS A CHOICE MENU OPTION SPECIFICATION
C  FORMAT IS <OPTION-TYPE>,<SECURITY CODE>,<DESCRIPTIVE TEXT><CR>
C            {<OPTION HELP TEXT><CR>}<END><CR><QUALIFIER>
C  WHERE <OPTION-TYPE> IS 1 FOR CHOICE_SUB-MENU, 2 FOR PARAM_SUB_MENU,
C        AND 3 FOR EXECUTABLE PROCESSOR (MODULE)
C  <SECURITY CODE> IS 0 IF ANYONE CAN CHOOSE THE OPTION, MODULE/MENU
C        NUMBER OTHERWISE (NUMBER IS COLUMN NUMBER IN SYSUSR.SYSRO)
C  <QUALIFIER> IS:
C     <SUB-MENU-NAME> FOR SUB-MENU OPTIONS
C     <RELATION-NAME> FOR PARAM-MENU OPTION
C     <SUBROUTINE-NAME> FOR SPECIAL PROCESS OPTIONS
C$LOCAL VARIABLES
C         err     error flag
C         eof     end of file read flag (cause abort)
C         found   an exact match was found in string
C         line    a character input string
C         text    a piece of 'line'
C         comma   ','
C$$
```

```
C     GHLPTX*************************************************
$CONTROL CHECK=3,SEGMENT=MNUGEN
      SUBROUTINE GHLPTX (IPTR,NLINES)
C*                        *** FORMAL PARAMETER DECLARATIONS ***
      INTEGER*4 IPTR
      INTEGER NLINES
C*                                                 *** ABSTRACT ***
C$PURPOSE read a menu's help text
C$AUDIT HISTORY
C          MEMutchler           17 JAN 83  AUTHOR
C          MEMutchler           16 FEB 83  TESTER (program tpmhelp)
C$TYPE     mnugen utility
C$FORMAL PARAMETERS
Cout       iptr     pointer to text storage in mhelp file
Cout       nlines   number of lines read and stored
C$COMMON BLOCKS
Cin        incpar   global parameter statements
Cio         txtcnt  holds records used in files dhtxt and ddtxt
Cin        ioc      i/o file assignments
Cin        readc    holds number of last line read from IN
C$CALLER   gcopt, gpopt
C$METHOD will read up to 'LHTXT' characters from IN.
C         reads help text until a
C         line = "END", and stores it at iptr in dhtxt
C$LOCAL VARIABLES
C         lenline  length of 'line'
C         lenstr   length of 'string'
C         line     one line of help text
C         string   all lines of help text
C         lfeed    aschii rep. of a line feed
C         begin    location in 'string' at which 'line'(j) should
C                  be stored
C         j        index as above
C$$
```

```
C     GLSTRL*************************************************
$CONTROL CHECK=3,SEGMENT=MNUGEN
      SUBROUTINE glstrl
C*
C$PURPOSE Creates the data base for list menus.
C$AUDIT HISTORY
C         MSCarey         15 DEC 83 AUTHOR
C$FORMAL PARAMETERS
C  none
C$COMMON BLOCKS
Cout      listyp  holds a list of list-menu type relations
Cin       ioc     io assignments
Cio       readc   holds count of records read from input file
Cout      relnam  table of relation names
C$CALLER  mnugen
C$METHOD
C  Reads the menu system generation input file for the names of
C  allowable list menu types (i.e. ship classes, yard names, etc.).
C  Saves these types in /listyp/ for later reference.  Creates one
C  relation for each type.  Creates a   cross reference relation
C  used by MNUR to map menu names into the proper list type relation.
C$LOCAL VARIABLES
C         line     receives an input line
C         temp     holds a list type before it is moved to ltyps
C         temp2    holds a delimited version of temp
C         junk     temporary
C         icrsor   relate cursor number
C         j        loop index
C         i        character position index in string
C         comand   input line with blanks removed
C         lencom   length of a string
C         dummy
C         comma    ,
C         fields   holds fieldlist for use in creating a list type rel
C         eof      end of file check
C         nchars   decoded from input line, number of characters requir
ed
C                  for candidate field in a given list type relation
C$$
```

```
C     GMTITL**************************************************
$CONTROL CHECK=3,SEGMENT=MNUGEN
      SUBROUTINE GMTITL (TITLE)
C*                        *** FORMAL PARAMETER DECLARATIONS ***
%INCLUDE INCPAR
      CHARACTER TITLE*LLINE
C*                                                 *** ABSTRACT ***
C$PURPOSE read a menu's title
C$AUDIT HISTORY
C          MEMutchler            17 JAN 83  AUTHOR
C          MEMutchler            17 FEB 83  TESTER  (program trmttl)
C$TYPE     mnugen utility
C$FORMAL PARAMETERS
Cout       title     title to be stored
C$COMMON BLOCKS
Cin        incpar  global parameter statements
Cin        ioc     i/o file assignments
Cin        readc   folds iline
C$CALLER   gpopt, gcopt
C$METHOD   read the title and store it
C$LOCAL VARIABLES
C          eof     true iff eof just read
C          lentitle length of title input
C          line    input line
C$$
```

```
C    GPOPT*************************************************
$CONTROL CHECK=3,SEGMENT=MNUGEN
      SUBROUTINE GPOPT(NOMORE,NOPTIO)
C*                        *** FORMAL PARAMETER DECLARATIONS ***
      LOGICAL NOMORE
      INTEGER NOPTIO
C*                                                *** ABSTRACT ***
C$PURPOSE reads a parameter menu's option's specification
C$AUDIT HISTORY
C         MEMutchler      18 JAN 83  AUTHOR
C$TYPE    mnugen utility
C$FORMAL PARAMETERS
Cout      nomore  true only if 'END' of parameter menu subsec found
Cin       noptio  number of previously specified options
C$COMMON BLOCKS
Cin       incpar  global parameter statements
Cin       readc   holds iline, last line # reaad from input file
Cin       ioc     i/o file assignments
Cio       lmenu   holds specs. for current list menu
Cio       pxref   holds relate filed list for all par. menus
Cin       ppindx  holds nxtpgi available for use
Cin       pdesc   holds descriptors of each parameter on a menu
C$CALLER  mnugen
C$METHOD  READS A PARAMETER MENU OPTION SPECIFICATION
C  FORMAT IS <TYPE>,<PARA_NAME>,(<LEN_CHAR_STRING>)<CR>
C            <DESCRIPTIVE TEXT><CR>{<OPTION HELP TEXT><CR>}<CR><END>
C  <TYPE> IS:
C     1(real), 2(integer), 3(character : list of allowable
C     scalors is delimted by ( ) in <DESCRIPTIVE TEXT> )
C      4(logical),5(date in mm/dd/yyyy format) ,
C     6(all/list,
C     note that data stored in /pvalue/ for type 6 options is as follo
ws.
C       (value, listmenuid, textptr,helpptr,helplength))
C$LOCAL VARIABLES
C         lpname  length of current parameter's name
C         pname   current parameter's name
C         nxtfld  next index in field name list to be used
C         plen    char rep of parameter's length in *1 words
C         eof     end of file read flag (cause abort)
C         delim   a delimiter '-'
C         line    a character input string
C         text    a piece of 'line'
C         comma   ','
C$$
```

```
C     INCLST*************************************************
$CONTROL CHECK=3,SEGMENT=MNUGEN
      SUBROUTINE INCLST(STR,LSTR,LSIZE,MLSIZE,LIST,ELSIZE,
     1 PTRLST,IINDEX,MATCH)
C*                      *** FORMAL PARAMETER DECLARATIONS ***
      CHARACTER STR*(ELSIZE) ,LIST*(ELSIZE)(MLSIZE)
      INTEGER IINDEX,LSTR,LSIZE,MLSIZE,ELSIZE
      INTEGER*4 PTRLST(MLSIZE)
      LOGICAL MATCH
C*                                              *** ABSTRACT ***
C*PURPOSE include string in a list
C*AUDIT HISTORY
C         MEMutchler      18 JAN 83  AUTHOR
C*TYPE    mnugen utility
C*FORMAL PARAMETERS
CIN   STR       NAME TO GO ON LIST
CIN   LSTR      NUMBER OF CHARS IN STR
CIO   LSIZE     NAME-LIST SIZE.  WILL BE INCREMENTED IF STR NOT
C               ALREADY ON THE LIST.
CIN   MLSIZE    MAX ALLOWED VALUE OF LSIZE
CIO   LIST      NAME-LIST
CIN   ELSIZE    MAX CHARS IN EACH ELEMENT OF 'LIST'
CIO   PTRLST    LIST OF MATCHING POINTERS
COUT  IINDEX    INDEX OF STR ON LIST
CIN   MATCH     .TRUE. IF MATCH IN LIST ALLOWED
C*COMMON BLOCKS
Cin       readc   # of line last read from input file
C*METHOD
C  CONSTRUCTS LIST 'LIST' OF NAMES AND LIST PTRLST OF MATCHING
C  POINTERS TO LEADING DATA CELL IN LIST MEMORY.
C*LOCAL VARIABLES
C         dummy   a dummy value
C*                    *** INCLUDES and LOCAL DECLARATIONS ***
C***
```

```
C     INIPAD**************************************************
$CONTROL CHECK=3,SEGMENT=MNUGEN
      SUBROUTINE INIPAD
C*                                                *** ABSTRACT ***
C*PURPOSE initialize for gpopt
C*AUDIT HISTORY
C          MEMutchler       19 JAN 83  AUTHOR
C*TYPE     mnugen utility
C*COMMON BLOCKS
Cio        pcreat  info for creation of parameter menu realtions
C*CALLER   mnugen
C*METHOD
C         Set up initial field list for creation of storage
C     relation (all have SCENARIO as first field) and initialize
C     variables used to "pad" each parameter's storage location
C     in /pvalue/ onto *4 word boundaries.
C*LOCAL VARIABLES
C**
```

```
C     PMSTOR***********************************************
$CONTROL CHECK=3,SEGMENT=MNUGEN
      SUBROUTINE PMSTOR (NMENUS)
C*                        *** FORMAL PARAMETER DECLARATIONS ***
      INTEGER*4 NMENUS
C*                                                  *** ABSTRACT ***
C$PURPOSE  stores parameter menu data structure in files
C$AUDIT HISTORY
C          MEMutchler            18 JAN 83  AUTHOR
C          MEMutchler            21 FEB 83  TESTER  (program tstpmen)
C$TYPE     mnurun utility
C$FORMAL PARAMETERS
Cin        nmenus  index into common pxref
C$COMMON BLOCKS
Cin        ioc    i/o assignments
Cin        pmenu   pmenu common block
Cin        incpar  global parameter statements
Cin        pcreat holds strctr to use to create storage relation
Cin        pxref   holds default values for parameter menu
C$CALLER   dpmenu
C$METHOD   determine how many *4 words fill common/pmenu/ and
C          put them into file dpmenu. Create the relation
C          needed for the current parameter menu, and fill
C          the default tuple with default values
C$LOCAL VARIABLES
Cin        iptr    pointer into file dpmenu at which pmenu data
C                  belongs
C          rname   delimited relation name
C          lenname length of relation name
C          icursor cursor to current relation (relate)
C          dummy   dummy variable
C          flist   field list for relation including
C                  scenario field name
C          len     len in characters of flist
C$$
```

```
C     PPCLOS**************************************************
$CONTROL CHECK=3,SEGMENT=MNUGEN
       SUBROUTINE ppclos
C*                        *** FORMAL PARAMETER DECLARATIONS ***
C*                                                 *** ABSTRACT ***
C$PURPOSE Closes and saves all mnugen output files
C$AUDIT HISTORY
C         MSCarey          02-feb-83  AUTHOR
C$FORMAL PARAMETERS
C         none
C$COMMON BLOCKS
Cio       ioc      input/output assignments
C$CALLER  mnugen
C$METHOD
C  many calls to filcls.
C$LOCAL VARIABLES
C         flag     true if filcls call successful
C$$
```

```
C     PPINIT**************************************************
$CONTROL CHECK=3,SEGMENT=MNUGEN
      SUBROUTINE ppinit
C*                      *** FORMAL PARAMETER DECLARATIONS ***
C*                                                *** ABSTRACT ***
C#PURPOSE Pre Processor INITialize.  Initializes mnugen.
C#AUDIT HISTORY
C         MSCarey           1-FEB-83  AUTHOR
C#FORMAL PARAMETERS
C         NONE
C#COMMON BLOCKS
Cio       ioc      file device assignments
Cout      ppindx   indexes for storage location of next parameter
Cout      rpsubs   number and list of callable special purpose subs
Cout      cmenu    choice menu pointer strucutre
Cout      pmenu    parameter menu pointer strctr
Cout      readc    keeps track of input file line just read
Cout      pxref    holds field lists for parameter menu relations
Cin       pdesc    describes parameters (here for system param maxprm)
Cout      comcfl   command file assignments
C#CALLER  mnugen
C#METHOD
C  Calls other initialization routines, sets global variables,
C  prompts for name of input file, and makes filopn calls.
C#LOCAL VARIABLES
C         file     name of input file as given by user
C         recrds   temporary residence of num data in char form
C         string   command string for passage to filopn
C         flag     true if filopn call successful
C##
```

```
C     PRCLST*********************************************
$CONTROL CHECK=3,SEGMENT=MNUGEN
      SUBROUTINE PRCLST (STR,LSTR,IINDEX)
C*                        *** FORMAL PARAMETER DECLARATIONS ***
      CHARACTER  STR*(LSTR)
      INTEGER LSTR
      INTEGER*4 IINDEX
C*                                                  *** ABSTRACT ***
C#PURPOSE  CONSTRUCTS LIST 'LIST' OF MODULE NAMES AND
C  LIST INDLST OF MATCHING INDEXES FOR COMPUTED GOTO IN SUBROUTINE
C  'RNPROC'
C#AUDIT HISTORY
C          MEMutchler       18 JAN 83  AUTHOR
C#TYPE     mnugen utility
C#FORMAL PARAMETERS
CIN        STR       NAME TO GO ON LIST
CIN        LSTR      NUMBER OF CHARS IN STR
COUT       IINDEX    INDEX FOR COMPUTED GOTO IN 'RNPROC'
C#COMMON BLOCKS
Cin        incpar  global parameter statements
Cin        reads   holds iline, input line# last read
Cio        rpsubs  list of subroutine names constructed
C#METHOD
C  look for the position in sublst for a match with str, if
C  there is none, add str to sublst, pass back position in
C  index.
C#LOCAL VARIABLES
C     NPROCS    NAME-LIST SIZE.  WILL BE INCREMENTED IF STR NOT
C                 ALREADY ON THE LIST.
C     SUBLST    NAME-LIST
C     NMSIZE     MAX CHARS IN EACH ELEMENT OF 'SUBLST'
C##
```

```
C     PSTORE*************************************************
$CONTROL CHECK=3,SEGMENT=MNUGEN
      SUBROUTINE PSTORE ( PNAME, LPNAME, IPGI )
C*                        *** FORMAL PARAMETER DECLARATIONS ***
%INCLUDE INCPAR
      CHARACTER PNAME*DIMNAME
      INTEGER LPNAME, IPGI,I,LENLINE
C*                                                  *** ABSTRACT ***
C#PURPOSE finish set up for a single parameter
C#AUDIT HISTORY
C         MEMutchler      18 JAN 83  AUTHOR
C#TYPE    mnugen utility
C#FORMAL PARAMETERS
Cin       pmname  parameter's name
Cin       lpname  length of parameter's name
Cin       ipgi    parameter's PGI
C#COMMON BLOCKS
Cin       incpar  global parameter statements
Cin       readc   holds iline, last line # reaad from input file
Cin       ioc     i/o file assignments
Cio       lmenu   holds specs. for current list menu
Cio       pxref   holds relate filed list for all par. menus
Cin       pcreat holds info to create parameter menu's relation
Cin       pdesc   holds descriptors of each parameter on a menu
C#CALLER  gpopt
C#METHOD
C  Writes an equivalence between parameter's position in pvalue
C  and its name.  If it will be used as an Integer*2 or logical
C  in applications programs, a padding variable is created and
C  equivalenced to the padding position in pvalue.  These padding
C  variables will not show up in the data base, instead all types
C  will be *4 and the left *2 part = 0.
C  The 'strctr' string needed to create the appropriate RELATE
C  relation will be generated, and a default scenario's values
C  will be generated for use in pmstor.
C#LOCAL VARIABLES
C         maxpad  maximum number of padding variables allowed
C         lreal   length of 'treal' string
C         lapha   length of 'alpha' string
C         ldouble length of 'double' string
C         zero    in*4 representation of zero
C         pdname  padding variable's name
C         blanks  string of ' '
C         pad     first 3 characters of padding variable names
C         lparan  '('
C         rparan  ')'
C         treal    for type = real in 'strctr'
C         double   for type = integer, date, logical in 'strctr'
C         alpha    for type = char or all/list in 'strctr'
C         delim   a delimiter '-'
C         line    a character input string
C         text    a piece of 'line'
C         comma   ','
C##
```

```
C     SVPDSC*************************************************
$CONTROL CHECK=3,SEGMENT=MNUGEN
      SUBROUTINE SVPDSC(IPTR)
      INTEGER*4 IPTR
C*                                                  *** ABSTRACT ***
C$PURPOSE write everything held in common/pdesc/
C         to disk file MPDESC
C$AUDIT HISTORY
C         MEMutchler            18 JAN 83  AUTHOR
C         MEMutchler            21 feb 83  TESTER  (program tstpmen)
C$TYPE    mnugen utilty
C$FORMAL PARAMETERS none
C$COMMON BLOCKS
Cin       ioc     i/o file assignments
C$CALLER  mnurun
C$METHOD  use same format as gpdesc
C$LOCAL VARIABLES
C         i       index counter
C$$
```

```
C     SVPXRF************************************************
$CONTROL CHECK=3,SEGMENT=MNUGEN
      SUBROUTINE SVPXRF(IPTR)
      INTEGER*4 IPTR
C*                                                  *** ABSTRACT ***
C#PURPOSE write everything held in common pxref
C         to disk file MPXREF
C#AUDIT HISTORY
C         MEMutchler            18 JAN 83  AUTHOR
C         MEMutchler            21 feb 83  TESTER  (program tstpmen)
C#TYPE    mnugen
C#FORMAL PARAMETERS none
C#COMMON BLOCKS
Cin       ioc     i/o file assignments
Cin       pxref   pxref common block
C#CALLER  mnurun
C#METHOD  write everything to file mpxref with same format
C         as it will be read in gpxref
C#LOCAL VARIABLES
C         i       loop counter
C##
```

```
C     SVROOT************************************************
$CONTROL CHECK=3,SEGMENT=MNUGEN
      SUBROUTINE SVROOT
C*                                                  *** ABSTRACT ***
C#PURPOSE write everything held in common/mroot/
C         to disk file MROOT
C#AUDIT HISTORY
C         MEMutchler            18 JAN 83  AUTHOR
C         MEMutchler            16 FEB 83  TESTER (program tmroot)
C#TYPE    mnurun
C#FORMAL PARAMETERS  none
C#COMMON BLOCKS
Cin       ioc     i/o file assignments
Cout      mroot   mroot common blocks
C#CALLER  mnurun
C#METHOD  write to mroot with same format as gmroot reads
C#LOCAL VARIABLES  none
C##
```

```
C     TXSTOR***********************************************
$CONTROL CHECK=3,SEGMENT=MNUGEN
      SUBROUTINE TXSTOR(DTXT,IPTR)
C*                        *** FORMAL PARAMETER DECLARATIONS ***
%INCLUDE INCPAR
       CHARACTER DTXT*LLINE
      INTEGER*4 IPTR
C*                                                *** ABSTRACT ***
C#PURPOSE  saves descriptive text at iptr in file ddtxt
C          and increment ndscln for next save
C#AUDIT HISTORY
C          MEMutchler            14 FEB 83  AUTHOR
C          MEMutchler            18 FEB 83  TESTER (program tpoption)
C#TYPE     mnurun utility
C#FORMAL PARAMETERS
Cout       iptr    pointer into file ddtxt at which data belongs
Cin        dtxt   descriptive text to be stored in ddtxt
C#COMMON BLOCKS
Cio        txtcnt  holds line record last used
Cin         ioc     i/o file assignments
Cin         incpar  global parameter statements
C#CALLER   dcmenu
C#METHOD   count and write
C#LOCAL VARIABLES
C          none
C##
```

```
C     WLSTDB*************************************************
$CONTROL CHECK=3,SEGMENT=MNUGEN
      SUBROUTINE wlstdb(menuid,type)
      character*8 menuid, type
C$PURPOSE writes a record into the lccref relation
C         so that queries for the contents of list "menuid"
C         will be directed to the proper "type" relation column.
C$AUDIT HISTORY
C         MSCarey           15 DEC 83 AUTHOR
C$FORMAL PARAMETERS
Cin       menuid  an identifier for a particular list menu
Cin       type    the list type relation which will hold values
C                  for "menuid".
C$COMMON BLOCKS
Clocal    rcrd01   holds a data record for passage to relate utils
Clocal    rcrd02   holds a data record for passage to relate utils
Cin       readc    holds input file record being processed (for aborts)
Cout      coluse   holds table of list type relation column usage
Cout      listyp   holds a list of list menu type relations
C
C$CALLER gpopt
C$METHOD
C         A record must be written to lccref in the MNUR data base
C     so that cross-referencing to the status of
C     candidates in a particular list menu can be accomplished
C     with knowledge only of scenario name and menuid.  The records
C     written to lccref map from a menuid to a type relation/column.
C     A search is conducted to
C     identify a free column in the appropriate list type relation.
C$LOCAL VARIABLES
C         dum       dummy
C         icrsor    relate cursor number
C         tindx     list type relation index in clused
C         i         loop index
C         column    index of free column in clused
C         len       string length
C         len2      string length
C         tmpstr    buffer for column
C$$
```

```
C     WRUNP*************************************************
$CONTROL CHECK=3,SEGMENT=MNUGEN
      SUBROUTINE WRUNP
C*                                                  *** ABSTRACT ***
C$PURPOSE write the rnproc module interface routine
C$AUDIT HISTORY
C          MEMutchler            17 JAN 83  AUTHOR
C          MEMutchler            15 FEB 83  TESTER (program twrunp)
C$TYPE     mnugen utility
C$FORMAL PARAMETERS      none
C$COMMON BLOCKS
Cin       ioc       i/o file assignments
Cin       rpsubs    holds list of subroutine names to be used
C$CALLER  mnugen
C$METHOD  write header, then computed goto, then call statements,
C         then footer.
C$LOCAL VARIABLES  none
C$$
```

```
C     MNURUN*****************************************
$CONTROL SEGMENT=MENU
      PROGRAM MNURUN
C*                                                *** ABSTRACT ***
C#PURPOSE main program for run time command system/System Core
C#AUDIT HISTORY
C         MEMutchler      18 JAN 83  AUTHOR
C#COMMON BLOCKS
Cin       mroot   holds pointer to top choice menu
C#METHOD
C  initialize, push bottom of stack marker, push top
C  command, call command handler 'stexec'
C#LOCAL VARIABLES
C         cmcnmd  *4 version of choice menu command
C         zero    *4 version of o
C##
```

```
C     BLDCF ************************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE bldcf(mnukey)
      character*8 mnukey
C*                                                *** ABSTRACT ***
C$PURPOSE BuiLD Command File
C$AUDIT HISTORY
C        MSCarey          27-feb-83  AUTHOR
C$FORMAL PARAMETERS
Cin      mnukey    name of current menu
C$COMMON BLOCKS
Cin      incpar    parameters
Cout     comcfl    command file info
Cio      ioc       io assignments
Cout     rcrd03    tuple for addition to CFLIST on call to bldstp
Cin      fields    field lists
C$CALLER various
C$METHOD
C  Offers to display existing files for this menu.
C  Prompts for name of new file. Checks to see that it doesn't exist.
C  Opens file ,sets switches, prompts for description of file, and
C  constructs tuple for later addition to CFLIST.
C$LOCAL VARIABLES
C        name      name of new command file
C        intkey    integer buffer for relate key value record
C        icursr    cursor number
C        jcursr    "
C        eof       true if eof on input unit
C        ok        true if file open was successful
C        notfnd    true if tuple in CFLIST not found
C        line      input line from terminal
C        filnam    name of new command file with group suffix
C        string    buffer for passage of command to filopn
C**
```

```
C     BLDSTP ************************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE bldstp
C*                                                *** ABSTRACT ***
C$PURPOSE command file BuiLDing SToP. Reset switches.
C$AUDIT HISTORY
C        MSCarey          27-feb-83  AUTHOR
C$FORMAL PARAMETERS
C        none
C$COMMON BLOCKS
Cio      comcfl   command file status info
Cio      ioc      io unit assignments
Cin      rcrd03   tuple constructed by bldcf for addition to CFLIST
Cin      fields   field lists
C$CALLER various
C$METHOD
C  Closes file being built, saves it, and resets switches.
C  If save is successful, adds tuple constructed by bldcf to
C  CFLIST.
C$LOCAL VARIABLES
C        flag     true if close successful
C        cursor   relate cursor
C$$
```

```
C     CKUSER ************************************************
$CONTROL segment=menu
      SUBROUTINE ckuser
C*                      *** FORMAL PARAMETER DECLARATIONS ***
C*                                                *** ABSTRACT ***
C$PURPOSE   Checks/sets the priveleges of the user.
C$AUDIT HISTORY
C        MSCarey         15-aug-83  AUTHOR
C$FORMAL PARAMETERS
C        none
C$COMMON BLOCKS
Cout     usrprv   privelege information for the user
C$CALLER inimnu
C$METHOD
C     Get the name of the user.  Open the sysusr.pub relation
C     and get to the tuple for this user.  If none, user is
C     unknown to the system and execution should be aborted.
C     If known, make sure execution is occurring from a legal
C     group (else abort).  The calc/read puts module running and
C     DB read/write priveleges into /usrprv/ automatically.
C$LOCAL VARIABLES
C$$
```

```
C     DCMENU*****************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE DCMENU
C*                                                  *** ABSTRACT ***
C$PURPOSE display choice menu and accept selections
C$AUDIT HISTORY
C          MEMutchler      18 JAN 83  AUTHOR
C$TYPE     display a type of menu
C$FORMAL PARAMETERS  none
C$COMMON BLOCKS
Cin        comfile command file information
Cin        cmenu   specifications for this choice menu
C$CALLER   stexec
C$METHOD
C  DISPLAYS MENU
C     display tickler command help at top if switch for it on
C     display menu title
C     for i =1 to noptios, display line# and option text
C  TOP OF STACK CONTAINS DISPLAY CHOICE MENU COMMAND
C  SECOND ITEM CONTAINS MENU POINTER
C  accept an input selection and push the appropriate command
C  on the command stack if a line # was selected else
C  process a standard option if it was selected else error.
C$LOCAL VARIABLES
C          formf   aschii representation of a form feed
C          ptr     pointer into listmemory at which menu
C                  specs. are
C          icmnd   int*4 'display choice menu' command
C          iopt    line # associated w/ this option
C$$
```

```
C     DDCF ***************************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE ddcf(mnukey)
      character*8 mnukey
C                                                  *** ABSTRACT ***
C#PURPOSE Display & Delete Command Files.  Allows inspection and
C         deletion of existing command files.
C#AUDIT HISTORY
C        MSCarey          27-feb-83  AUTHOR
C#FORMAL PARAMETERS
Cin      mnukey   name of current menu
C#COMMON BLOCKS
Cin      incpar   parameters
Cin      uzrprv   holds information about user privelege levels
Cout     rcrd03   data record for cflist relation
Cin      fields   field lists for all relations
C#CALLER d_menu
C#METHOD
C  Calls dspcf to display files for current menu, then prompts for
C  a deletion candidate or quit signal.  Checks to see that user is
C  file creator or has system administrator priveleges.  Purges file
C  and record in CFLIST.  Prompts for display or additional deletions.
C#LOCAL VARIABLES
C        uname    users name
C        icursr   RELATE cursor
C        jcursr   RELATE cursor
C        len      character string length
C        comand   buffer to hold monitor command
C        temp     input line buffer
C        line     input line buffer
C        filnam   name of command file to delete
C        filkey   buffer for passing index value to relate
C        notfnd   true if tuple not found
C        didit    true if MPE purge successful
C##
```

```
C     DLMENU*****************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE DLMENU (IPGI,TOPPOP)
C*                        *** FORMAL PARAMETER DECLARATIONS ***
      INTEGER   IPGI
      LOGICAL TOPPOP
C*                                                *** ABSTRACT ***
C#PURPOSE display list menu and accept numbers of list candidate
C         members whose status need be reversed
C#AUDIT HISTORY
C         MEMutchler      18 JAN 83  AUTHOR
C#TYPE    display a type of menu
C#FORMAL PARAMETERS
Cin       ipgi      pvalue(mpindx(ipgi)+1) = the menuid for this menu
C                   pvalue(mpindx(ipgi)+3) = the pointer into list
C                          memory where the menu title is
Cout      toppop    true if user specified menu exit with pop to top
C                   menu
C#COMMON BLOCKS
Cin        terminal crt screen dimension
Cin       ioc      i/o file assignments
Cin        pdesc    holds descriptors of all parameters on all menus
Cio        lvalue holds members names and most current status'
Cin        queue    holds list command queue and number in queue
C#CALLER  dpmenu
C#METHOD
C     displays list menu by:
C     display command tickler textxt
C     display menu title
C     for i =1 to noptios, display line#, status(*for included,
C     for not included), and the member's name, two to a line,
C     as many as possible to a page..
C  Accept an input selection and flip its status if a line # was
C     selected, else process a standard option if it was selected,
C     else error.
C#LOCAL VARIABLES
C         formf    aschii representation of a form feed
C         itptr    pointer into listmemory at which menu
C                  title is
C         ithpage number of page currently displayed
C         iopt     line # associated w/ this option
C         npages   number of crt screens needed to display this
C                  list menu
C         nlines   number of lines to display whole menu
C         noptvll number of options to be displayed on the
C                  very last line of the whole menu
C         noptll   number of options to be displayed on the
C                  last line of the current display page
C         seprat   char. line to be display around the list
C         minitem minimum line number displayed on current page
C         linpage maximum number of lines on a page for
C                  display of members and status
C         nlpage   number of lines on current page
C         nchoices number of members on current page
```

```
C       tptr    character representation of title pointer
C       len     non-blank character length of tptr
C**
```

```
C     DPMENU*****************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE DPMENU
C*                                                  *** ABSTRACT * *
C#PURPOSE display parameter menu and accept selections and new
C         values
C#AUDIT HISTORY
C         MEMutchler      18 JAN 83  AUTHOR
C#TYPE    display a type of menu
C#FORMAL PARAMETERS  none
C#COMMON BLOCKS
Cin       comfile command file information
Cin       pmenu   specifications for this parameter menu
Cin       pdesc   holds descriptors of all parameters on all menus
Cio       pvalue holds most current parameter values
Cin       incpar  global parameter statements
C#CALLER  stexec
C#METHOD
C  displays parameter menu
C     display command tickler help
C     display menu title
C     for i =1 to noptios, display line# and option text
C     ,the options current value, and text discribing the
C     allowable values for the parameter.
C  Top of stack contains display_param_menu command
C  second item contains menu pointer
C  Accept an input selection and value and update the curent
C  value for the parameter if a line # was selected, else
C  process a standard option if it was selected, else error.
C#LOCAL VARIABLES
C         formf   aschii representation of a form feed
C         err     true if input value is not on of theallowalble
C         iptr    pointer into listmemory at which menu
C                 specs. are
C         nchoice number of parameters on menu
C         ipgi    parameter global index into pdesc common
C                 for parameter selected
C         iopt    line # associated w/ this option
C         iindex  index into pvalue at which parmeter's value
C                 will be found
C         update  char. rep of value to be assigned to parameter
C         imonth  month of date entered
C         iday    day of date entered
C         iyear   year of date entered of form 1983
C##
```

```
C     DSPCF ************************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE dspcf(mnukey,cursor)
      integer cursor
      character*8 mnukey
C*                                              *** ABSTRACT ***
C#PURPOSE Displays all cataloged command files which may
C         begin execution at the current menu.
C#AUDIT HISTORY
C        MSCarey          26-feb-83  AUTHOR
C#FORMAL PARAMETERS
Cin      mnukey    the name of the current menu, used as a key
C                  for extracting records from the CFLIST relation.
Cin      cursor    id number of the cursor open on CFLIST.
C#COMMON BLOCKS
Cin      incpar    paramters
Cin      ioc       io unit numbers
Cout     rcrd03    data record for communication with CFLIST
Cin      fields    field lists, including that for CFLIST
C#CALLER  usecf,ddcf,bldcf
C#METHOD
C  Extracts records from CFLIST and writes to iout. Pages on command i
f
C  necessary.
C#LOCAL VARIABLES
C        mxcfos    maximum lines of command files on screen at once
C        page      number of page currently displayed
C        recs      counter for records written to screen
C        len       string length
C        runthr    number of tuples to be re-read to effect page-back
C        line      char buffer
C        input     char buffer
C        flag      general purpose flag
C        multpg    true if there are more than mxcfos records to displa
y
C        rdfstr    char function, converts RELATE real date format into
C                  character string MM/DD/YY
C##
```

```
C     ENQU*****************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE ENQU( ILINE, LLEN)
C*                        *** FORMAL PARAMETER DECLARATIONS ***
%INCLUDE INCPAR
       INTEGER LENLINE,LLEN
       CHARACTER LINE*LLINE,ILINE*(LLEN )
C*                                                *** ABSTRACT ***
C#PURPOSE Puts one line, length=lenline,  of list commands delimited
C         by commas as read from user into listqu of length=number.
C#AUDIT HISTORY
C         MEMutchler       17 Jan 83  AUTHOR
C         MEMutchler       8 FEB 83 tested program tsteq
C#TYPE    mnurun queue utility
C#FORMAL PARAMETERS
Cin       iline     input line from input file, may contain
C                  several commands
Cin       llen     length of input line in non-blank characters
C#COMMON BLOCKS
Cin       incpar   global parameter statements
Cio       queue    queue, and queue length
C#CALLER  gloptn
C#METHOD
C  Split up line into commands by keying on commas,
C  keeping track of the number of commands found and
C  thus those enqueued, fill a temporary queue with
C  commands so that temp(1) has the next command
C  to be performed.  Transfer the commands to listqu
C  so that listqu(number) holds the next command.
C#LOCAL VARIABLES
C         number   number of commands enqueued
C         i        index of begining of match in char. string
C         comma    ','
C         temp     temporary command queue
C##
```

```
C     GCOPTN*****************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE GCOPTN (IOPT)
C*                        *** FORMAL PARAMETER DECLARATIONS ***
      INTEGER IOPT
C*                                                  *** ABSTRACT ***
C#PURPOSE  retrieve character string from file IN.  If it is
C          an  positive integer value, iopt = integer value
C          If it is a standard command recognition character,
C          iopt is the negative of the value specified in
C          /scrchr/
C#AUDIT HISTORY
C          MEMutchler            18 JAN 83  AUTHOR
C          MEMutchler            8  FEB 83  TESTER(program tsgcop)
C#TYPE     mnurun utility
C#FORMAL PARAMETERS
Cout       iopt     integer command code
C#COMMON BLOCKS
Cin        incpar   global parameter statement
Cin        comfile  command file information
Cin        ioc      i/o file assignments
C#CALLER   dcmenu
C#METHOD   get a choice command from input file and convert
C          it to integer command code
C#LOCAL VARIABLES
C          lenline length of line in non-blank characters
C          line     command string
C          command  first character of command string
C          eof      true iff eof is read from input file
C##
```

```
C     GCPTRS*****************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE GCPTRS(IPTR)
C*                        *** FORMAL PARAMETER DECLARATIONS ***
      INTEGER*4 IPTR
C*                                                  *** ABSTRACT ***
C#PURPOSE  retrieve record describing a choice menu from the
C          IPTR location in the mcmenu file.
C#AUDIT HISTORY
C          MEMutchler            18 JAN 83  AUTHOR
C          MEMutchler       16 FEB     TESTER  (program tstcmen)
C#TYPE     mnurun utility
C#FORMAL PARAMETERS
Cin        iptr    pointer into list memory at which data begins
C#COMMON BLOCKS
Cout       cmenu   to be filled with data from list memory
C#CALLER   dcmenu
C#METHOD   determine how many *4 words fill common/cmenu/ and
C          retreive them from list memory
C#LOCAL VARIABLES
C          none
C##
```

```
C     GETLST ****************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE GETLST(ipgi)
C*    integer ipgi
C*
C#PURPOSE GET LiST. Pulls list candidates and statuses out
C         of the data base for the list menu just invoked.
C#AUDIT HISTORY
C         MSCarey          26-jan-83  AUTHOR
C#FORMAL PARAMETERS
Cin       ipgi    index in pvalue of info for this list menu
C#COMMON BLOCKS
Cin       scenar  holds info identifying the scenario in use.
Cin       pvalue holds values for parameter menu members.
Cout      lvalue  holds list candidates and statuses.
Cout      ltypac  holds access data for current list type relation
Cloc      rcrd01  holds record for lcrref relation
C#CALLER  dlmenu
C#METHOD
C  Retrieve the menuid from pvalue.  Use this and the scenario
C  ID to get the list type relation name holding candidates and
C  statuses, as well as the column name, from LCCREF.
C  Read the list
C  type relation and load its information into /lvalue/
C#LOCAL VARIABLES
C         key01    key value for lccref queries
C         key02    key value for lstcol queries
C         dest     destination record for ltype relation queries
C         cand     list candidate (equiv to dest, trans to lvalue)
C         stat     candidate status (equiv to dest, trans to lvalue)
C         trel     Type RELation name, delimited
C         icros    cursor number for lcrref
C         icolm    cursor number for lstcol
C         notfnd   true if failure to find cross-ref info
C         eof      true when all list type records read
C         i        loop index
C         fldnam   field name from colm02
C         lfldnm   non-blank length of fldnam
C        len      length of dts field list
C##
```

```
C     GLOPTN**************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE GLOPTN (IOPT)
C*                        *** FORMAL PARAMETER DECLARATIONS ***
      INTEGER IOPT
C*                                                  *** ABSTRACT ***
C#PURPOSE Retrieve character command.  If it is
C         a positive integer value, iopt = integer value
C         If it is a standard command, iopt is the negative of the
C         value found in /scrchr/
C#AUDIT HISTORY
C         MEMutchler            18 JAN 83  AUTHOR
C         MEMutchler            8  FEB 83  TESTER (program tstlop)
C#TYPE    mnurun utility
C#FORMAL PARAMETERS
Cout      iopt     integer command code
C#COMMON BLOCKS
Cin       incpar   global parameter statements
Cin       comfile  command file information
Cin       ioc      i/o file assignments
Cin       queue    list command queue
C#CALLER  dlmenu
C#METHOD  if command queue not empty
C            then get and pop next command off queue
C            else read in next command input line from file in
C                 enqu it, and get and pop command from queue
C         decode command string into integer command code
C#LOCAL VARIABLES
C         lenline length of a character string in non-blank chars
C         command first character of command string
C         line    character string
C         eof     true iff end of file read from IN
C##
```

```
C     GMROOT*****************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE GMROOT
C*                                                  *** ABSTRACT ***
C$PURPOSE Reads everything held in common/mroot/
C         from disk file MROOT
C$AUDIT HISTORY
C         MEMutchler            18 JAN 83  AUTHOR
C         MEMutchler       16 FEB     TESTER  (program tmroot )
C$TYPE    mnurun
C$FORMAL PARAMETERS  none
C$COMMON BLOCKS
Cin       ioc      i/o file assignments
Cout      mroot    mroot common blocks
C$CALLER  mnurun
C$METHOD  read record
C$LOCAL VARIABLES  none
C$$
```

```
C     GPDESC*****************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE GPDESC(IPTR)
      INTEGER*4 IPTR
C*                                                *** ABSTRACT ***
C$PURPOSE reads a /pdesc/ record from the mpdesc paging file
C$AUDIT HISTORY
C         MEMutchler           18 JAN 83  AUTHOR
C         MEMutchler           21 FEB 83  TESTER (program tstpmen)
C$TYPE    mnurun utilty
C$FORMAL PARAMETERS none
C$COMMON BLOCKS
Cin       ioc     i/o file assignments
C$CALLER  mnurun
C$METHOD  direct access read
C$LOCAL VARIABLES
C         i       index counter
C$$
```

```
C     GPOPTN*****************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE GPOPTN(IOPT,MINPGI,IPGI,VALUE)
C*                        *** FORMAL PARAMETER DECLARATIONS ***
%INCLUDE INCPAR
       CHARACTER VALUE*LLINE
      INTEGER IOPT, MINPGI, IPGI
C*                                                  *** ABSTRACT ***
C$PURPOSE   Retrieves a parameter menu command from the user
C     and decodes it.
C$AUDIT HISTORY
C          MEMutchler            18 JAN 83  AUTHOR
C          MEMutchler            7  FEB 83  TESTER (program tgpopt)
C$TYPE     mnurun utility
C$FORMAL PARAMETERS
Cin        minpgi  minimum parameter global index (PGI)for this menu
Cout       iopt    integer command code
Cout       ipgi    PGI for the parameter selected to be modified
Cout       value   parameter's modified value in chars.
C$COMMON BLOCKS
Cin        incpar  global parameter statements
Cin        ioc     i/o file assignments
Cin        comfile command file information
C$CALLER   dpmenu
C$METHOD
C          retrieve one command from file IN.  If it is
C          a positive integer value, iopt = integer value,
C          ipgi = minpgi + (iopt-1), and get param value from LINE.
C          If it is a standard options recognition character,
C          iopt is the negative of the corresponging /scrchr/ value
C          if the command was
C          to modify a parameter's value, the new value =
C          everything on the input line following an '=' or ','.
C$LOCAL VARIABLES
C          lenline length of a character string in non-blanks
C$$
```

```
C     GPPTRS*****************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE GPPTRS (IPTR)
C*                        *** FORMAL PARAMETER DECLARATIONS ***
      INTEGER*4 IPTR
C*                                                  *** ABSTRACT ***
C#PURPOSE  fills in parameter menu data strctr with data
C          record pointed to by IPTR
C#AUDIT HISTORY
C          MEMutchler            18 JAN 83  AUTHOR
C          MEMutchler            21 FEB 83  TESTER (program tstpmen)
C#TYPE     mnurun utility
C#FORMAL PARAMETERS
Cin        iptr     pointer file dpmenu at which pmenu data
C                   begins
C#COMMON BLOCKS
Cout       pmenu    pmenu common block
C#CALLER   dpmenu
C#METHOD   determine how many *4 words fill common/pmenu/ and
C          retreive them from the file
C#LOCAL VARIABLES none
C##
```

```
C     GPXREF*****************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE GPXREF(IPTR)
       INTEGER*4 IPTR
C*                                              *** ABSTRACT ***
C#PURPOSE Reads a record into common pxref
C         from a disk file MPXREF
C#AUDIT HISTORY
C         MEMutchler            18 JAN 83  AUTHOR
C         MEMutchler            21 FEB 83  TESTER (program tstpmen)
C#TYPE    mnurun
C#FORMAL PARAMETERS none
C#COMMON BLOCKS
Cin       ioc     i/o file assignments
Cin       pxref   pxref common block
C#CALLER  mnurun
C#METHOD  direct access read
C#LOCAL VARIABLES
C         i       loop counter
C##
```

```
C     HLPCMD*****************************************
$CONTROL SEGMENT=MENUH
      SUBROUTINE HLPCMD (LIST)
C*                        *** FORMAL PARAMETER DECLARATIONS ***
      LOGICAL LIST
C*                                                  *** ABSTRACT ***
C#PURPOSE display help for list, choice, and parameter menu's
C         standard command options
C#AUDIT HISTORY
C         MEMutchler      11-mar-83  AUTHOR
C#TYPE    mnurun utility
C#FORMAL PARAMETERS
Cin       list    true only if called by hlplmu
C#COMMON BLOCKS
Cin       ioc     i/o file assignments
C#CALLER  hlplmu, hlpcmu, hlppmu
C#METHOD  print help for appropriate menu
C#LOCAL VARIABLES
C##
```

```
C     HLPCMU*****************************************
$CONTROL SEGMENT=MENUH
      SUBROUTINE HLPCMU(REFRSH,TOPPOP)
C*                      *** FORMAL PARAMETER DECLARATIONS ***
       LOGICAL REFRSH, TOPPOP
C*                                                *** ABSTRACT ***
C#PURPOSE command decoder for help choices for a choice menu
C#AUDIT HISTORY
C          MEMutchler            8  MAR 83  AUTHOR
C          MEMutchler            8  MAR 83  TESTER
C#TYPE     mnurun utility
C#FORMAL PARAMETERS
Cout      refrsh    true if menu refresh should be done after return
Cout      toppop    true if pop out of help subsystem desired
C#COMMON BLOCKS
Cin        incpar   global parameter statements
Cin        ioc      i/o file assignments
Cin        inputl   holds last line input and its length
C#CALLER   dcmenu
C#METHOD   see if '?' followed by an allowable help option is in
C          lastln, if it is do the help command. ELSE display
C          the help options, read one in and process it.=
C#LOCAL VARIABLES
C          len      length of a character string in non-blanks
C##
```

```
C     HLPHLP*****************************************
$CONTROL SEGMENT=MENUH
      SUBROUTINE HLPHLP (LIST)
C*                          *** FORMAL PARAMETER DECLARATIONS ***
      LOGICAL LIST
C*                                                  *** ABSTRACT ***
C#PURPOSE display help about the help subsystem
C#AUDIT HISTORY
C          MEMutchler       11-MAR-83  AUTHOR
C#TYPE     mnurun utility
C#FORMAL PARAMETERS
Cin        list     true only if called from hlplmu
C#COMMON BLOCKS
Cin        ioc      i/o file assignments
Cin        incpar   global parameter statements
C#CALLER   hlpcmu,hlplmu,hlppmu
C#METHOD   display help options as seen from help menu and
C          expound on their effects
C#LOCAL VARIABLES
C          line     input line
C          lenlin   length of line in characters
C##
```

```
C     HLPLMU*****************************************
$CONTROL SEGMENT=MENUH
      SUBROUTINE HLPLMU(REFRSH,TOPPOP)
C*                        *** FORMAL PARAMETER DECLARATIONS ***
      LOGICAL REFRSH,TOPPOP
C*                                                  *** ABSTRACT ***
C#PURPOSE handle all possible help choices for a list menu
C#AUDIT HISTORY
C         MEMutchler.           8  MAR 83  AUTHOR
C         MEMutchler            8  MAR 83  TESTER
C#TYPE    mnurun utility
C#FORMAL PARAMETERS
Cout     refrsh   true if refresh in caller required
Cout     toppop   true if pop out of help subsystem
C#COMMON BLOCKS
Cin       incpar  global parameter statements
Cin       ioc     i/o file assignments
Cin       inputl  holds last line input and its length
C#CALLER  dlmenu
C#METHOD  see if '?' followed by an allowable help option is in
C         lastln, if it is do the help command. ELSE display
C         the help options, read one in and process it.=
C#LOCAL VARIABLES
C         len     length of a character string in non-blanks
C##
```

```
C     HLPPMU*****************************************
$CONTROL SEGMENT=MENUH
      SUBROUTINE HLPPMU (REFRSH,TOPPOP)
C*                    *** FORMAL PARAMETER DECLARATIONS ***
       LOGICAL REFRSH, TOPPOP
C*                                                *** ABSTRACT ***
C#PURPOSE handle all possible help choices for a parameter menu
C#AUDIT HISTORY
C         MEMutchler          8  MAR 83  AUTHOR
C         MEMutchler          8  MAR 83  TESTER
C#TYPE    mnurun utility
C#FORMAL PARAMETERS
Cout     refrsh   true if caller must refresh screen
Cout     toppop   true if pop out of help subsystem desired
C#COMMON BLOCKS
Cin       incpar  global parameter statements
Cin       ioc     i/o file assignments
Cin       inputl  holds last line input and its length
C#CALLER  dpmenu
C#METHOD  see if '?' followed by an allowable help option is in
C         lastln, if it is do the help command. ELSE display
C         the help options, read one in and process it.=
C#LOCAL VARIABLES
C         len     length of a character string in non-blanks
C##
```

```
C     HLPPRT*************************************************
$CONTROL SEGMENT=MENUH
      SUBROUTINE HLPPRT (IPTR,LEN)
C*                        *** FORMAL PARAMETER DECLARATIONS ***
      INTEGER*4 IPTR
       INTEGER LEN
C*                                                  *** ABSTRACT ***
C#PURPOSE  prints string of characters pointed to by IPTR
C          in the mhelp file on line of the menu display.
C          Length of the string is held at IPTR
C#AUDIT HISTORY
C         MEMutchler            18 JAN 83  AUTHOR
C         MEMutchler       16 FEB     TESTER  (program tpmhelp)
C#TYPE    mnurun utility
C#FORMAL PARAMETERS
Cin       iptr    pointer into file dhtxt  for help text
Cin       len     number of lines of help text
C#COMMON BLOCKS
Cin       incpar  global parameter statements
Cin       ioc     i/o file assignments
C#CALLER  dlmenu, dcmenu, dpmenu
C#METHOD  get text from list memory and display to iout
C#LOCAL VARIABLES
C         nlines  pointer into dhtxt for current line
C         text    help text string
C##
```

```
C     HLPSHO****************************************
$CONTROL SEGMENT=MENUH
      SUBROUTINE HLPSHO (LIST)
C*                          *** FORMAL PARAMETER DECLARATIONS ***
      LOGICAL LIST
C*                                                  *** ABSTRACT ***
C#PURPOSE display all help options on help menu
C#AUDIT HISTORY
C         MEMutchler       11-MAR-83  AUTHOR
C#TYPE    mnurun utility
C#FORMAL PARAMETERS
Cin       list.    true only if called from hlplmu
C#COMMON BLOCKS
Cin       ioc      i/o file assignments
Cin       comcfl   holds command file information
C#CALLER  hlpcmu, hlplmu, hlppmu
C#METHOD
C  print appropriate help option text to iout
C#LOCAL VARIABLES
C##
```

```
C     HLPSYS*****************************************
$CONTROL SEGMENT=MENUH
      SUBROUTINE HLPSYS
C*                      *** FORMAL PARAMETER DECLARATIONS ***
C*                                                *** ABSTRACT ***
C$PURPOSE explain runtime system and how to use it
C$AUDIT HISTORY
C          MEMutchler       11-mar-83  AUTHOR
C$TYPE     mnurun utility
C$FORMAL PARAMETERS      none
C$COMMON BLOCKS
Cin        ioc      i/o file assignments
C$CALLER   hlpcmu, hlplmu, hlppmu
C$METHOD
C  read help from syshlp file and print it
C$LOCAL VARIABLES
C$$
```

```
C     INIMNU****************************
$CONTROL SEGMENT=MENU
      SUBROUTINE inimnu
C*                                                  *** ABSTRACT ***
C$PURPOSE Initializes the run-time menu system.
C$AUDIT HISTORY
C         MSCarey          02-feb-83  AUTHOR
C$FORMAL PARAMETERS
C$COMMON BLOCKS
Cout      ioc      io device assignments
Cin       incpar   menu system parameters
C$CALLER  mnurun
C$METHOD
C  Opens files and calls subsidiary initialization routines.
C$LOCAL VARIABLES
C         flag     true if a filopn or filcls call was successful
C$$
```

```
C     PCMDS***********************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE PCMDS(MENU)
C*                                              *** ABSTRACT ***
c*                                    *** FORMAL PARAMETERS ***
      CHARACTER*8 MENU
C#PURPOSE display standard menu options to iout
C#AUDIT HISTORY
C          MEMutchler            17 JAN 83  AUTHOR
C          MEMutchler            8  FEB 83  TESTER (program tstan)
C#TYPE     mnurun utility
C#FORMAL PARAMETERS      none
C#COMMON BLOCKS
Cin        ioc      i/o file assignments
C#CALLER   dpmenu, dcmenu
C#METHOD   write text to file iout
C          prints string of text discribing all options
C          available to all menus, at top of menu display.
C#LOCAL VARIABLES    none
C##
```

```
C     PLCMDS*********************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE PLCMDS(MENU)
      CHARACTER*8 MENU
C*                                              *** ABSTRACT ***
C#PURPOSE prints string of text describing all options
C         available to the list menus, at top of list menu
C         display.
C#AUDIT HISTORY
C         MEMutchler          18 JAN 83  AUTHOR
C         MEMutchler          7  FEB 83  TESTER (program tlstan)
C#TYPE    mnurun utility
C#FORMAL PARAMETERS     none
C#COMMON BLOCkS
Cin       ioc     i/o file assignments
C#CALLER  dpmenu, dcmenu
C#METHOD  write text to iout
C#LOCAL VARIABLES none
C##
```

```
C     PRCOPT*****************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE PRCOPT (IOPT, IPTR)
C*                        *** FORMAL PARAMETER DECLARATIONS ***
      INTEGER*4 IPTR
      INTEGER IOPT
C*                                                  *** ABSTRACT ***
C#PURPOSE  Prints iopt and the string of characters at IPTR
C          on line of the menu display.
C#AUDIT HISTORY
C          MEMutchler      17 JAN 83  AUTHOR
C          MEMutchler      16 FEB     TESTER  (program tstcop)
C#TYPE     mnurun utility
C#FORMAL PARAMETERS
Cin        iopt     line number to be printed
Cin        iptr     ptr into list memory at which text to be displayed
C                   is found
C#COMMON BLOCKS
Cin        incpar   global parameter statements
C#CALLER   dcmenu
C#METHOD   Get text to be displayed fome list memory. Display it
C          and line number.
C#LOCAL VARIABLES
C          text     text string to be displayed
C          lentxt   length of 'text' in non-blank characters
C##
```

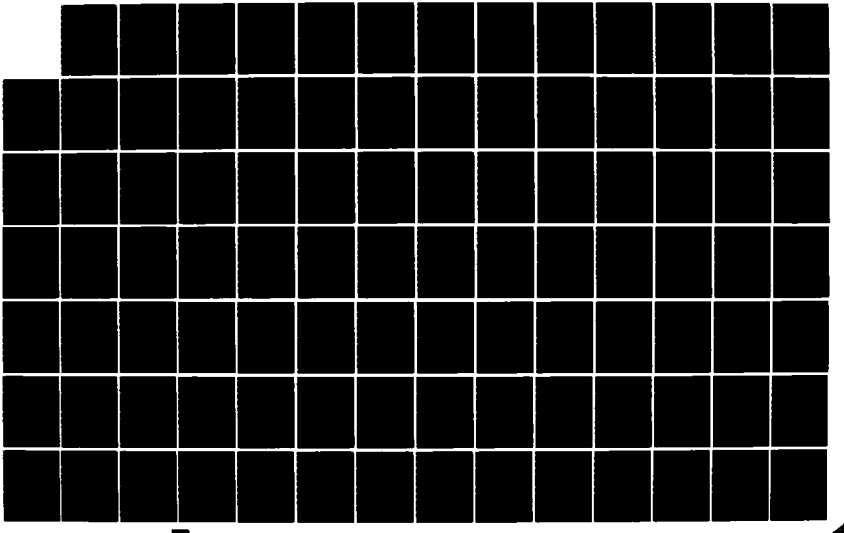
```
C     PRLOPT*************************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE PRLOPT (INUM,I,J)
C*                        *** FORMAL PARAMETER DECLARATIONS ***
      INTEGER INUM, I, J
C*                                                 *** ABSTRACT ***
C#PURPOSE Writes list menu candidates to screen.
C         writes the ith and jth list element and status
C         onto iout if inum = 2, else only ith element
C#AUDIT HISTORY
C         MEMutchler           18 JAN 83  AUTHOR
C         MEMutchler           7  FEB 83  TESTER (program tslopt)
C#TYPE    mnurun utility
C#FORMAL PARAMETERS  none
Cin       inum     number of option printed on this line
Cin       i        index into lvalue for leftmost option
Cin       j        index into lvalue for rightmost option
C#C0MMON BLOCKS
Cin       incpar   global parameter statements
Cin       ioc      i/o file assignments
Cin       lvalue   name and status of dlmenu members
C#CALLER  dlmenu
C#METHOD  according to number od options to be printed
C         on line, and their status' use an output format.
C#LOCAL VARIABLES   none
C##
```

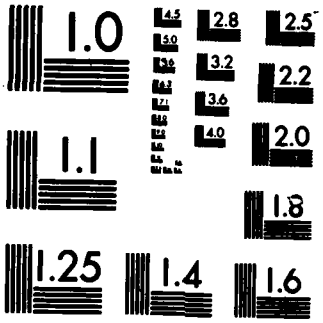
```
C     PRPOPT*************************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE PRPOPT ( IOPT, IPGI )
C*                      *** FORMAL PARAMETER DECLARATIONS ***
      INTEGER IOPT, IPGI
C*                                                *** ABSTRACT ***
C#PURPOSE display a parameter menu's option line
C#AUDIT HISTORY
C         MEMutchler            18 JAN 83  AUTHOR
C         MEMutchler            18 FEB 83  TESTER (program tpoption)
C#TYPE    mnurun utility
C#FORMAL PARAMETERS
Cin       iopt     line number to be displayed
Cin       ipgi     index into pdesc common for this parameter
C#COMMON BLOCKS
Cin       ioc      i/o file assignments
Cin       pmenu    parameter menu info
Cin       pdesc    holds descriptions of each parameter
Cin       pvalue parameter values are held here
C#CALLER  dpmenu
C#METHOD  splits text pointed to by mptxtp(ipgi) into DTEXT,
C         and OKTEXT where oktext is the part of the text
C         delimited by ( ).  Writes iopt, dtext, value
C         of the parameter as described by mptype(ipgi) and
C         mplen(ipgi) found at pvalue(mpindx(ipgi)), and oktext
C         onto output screen
C#LOCAL VARIABLES
C         maxok    max. length of allowable values text
C         lentxt   actual string length in nonblank chars.
C         i        index in string for begining of a match
C         j        index into pvalue for parameter's value
C         lparan   '('
C         rparan   ')'
C         im       month
C         iday     day
C         iyear    year   as in 1983
C         lenok    actual oktext length in nonblank chars.
C         ityp     integer type code of parameter
C         text     descriptive text for parameter in 'line'
C         oktext   allowable value text for parameter
C                  delimited by parens. in 'line'
C         line     text from list memory holding 'text'& 'oktext'
C         output   value for parameter for display purpose only
C##
```

```
C       PVCHEK*****************************************
$CONTROL SEGMENT=MENU
        SUBROUTINE PVCHEK(VALUE,LVALUE,IPGI,ERR)
        INTEGER IPGI, LVALUE
        LOGICAL ERR
%INCLUDE INCPAR
        CHARACTER VALUE*LLINE
C#PURPOSE checks string 'value' with length 'lvalue' to see if
C         it has an exact match in allowable parameter values
C         delimited by '( )' in the descriptive text as defined
C         during the parameters specification.
C#AUDIT HISTORY
C         MEMutchler      17-1-83    AUTHOR
C         MEMutchler      16 FEB     TESTER  (program tstcok)
C#TYPE    mnurun utility
C#FORMAL PARAMETERS
Cin       value   input value string to check
Cin       lvlaue  length of string in non-blank characters
Cin       ipgi    'parameter global index' of parameter to be modified
Cout      err     true iff no exact match is found
C#COMMON BLOCKS
Cin       incpar  global parameter statements
Cin        ioc    i/o file assignments
Cin       pdesc   holds info about parameters modifiable in mnurun
C#CALLER  DPMENU
C#METHOD  Get parameter's descriptive text from file ddtxt.
C         Get allowable values string from this text as
C         all characters following the left delimiter.
C         Get the postion in the allowable values string
C         of the first character of an exact match with
C         'value'.  If there was a match err = f else
C         err = true.
C#LOCAL VARIABLES
C         string  character string to be matched into
C         lenstr  length in non-blank characters of string
C         i       index into string at which match begin
C         line    rest of string following match
C##
```

AD-A150 422 ALIAS (ACQUISITION AND LOGISTICS INFORMATION AND ANALYSIS SYSTEM) MAINTEN..(U) DECISION-SCIENCE APPLICATIONS INC ARLINGTON VA M S CAREY ET AL. 4/7

UNCLASSIFIED 31 OCT 84 DSA-593-VOL-1 N00014-82-C-0813 F/G 15/5 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

```
C     PVINIT*************************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE pvinit
C*                                                  *** ABSTRACT ***
C$PURPOSE Fills the pvalue array with proper values.
C         Called once at start of MNUR execution and on change of
C         scenario.
C$AUDIT HISTORY
C         MSCarey          02-feb-83  AUTHOR
C$FORMAL PARAMETERS
C         none
C$COMMON BLOCKS
Cout      pvalue  holds all values for parameters displayed by menu.
Cin       pxref   holds fieldlists for each parameter relation
Cin       pdesc   holds parameter info
Cin       pmenu   parameter pointer record
Cin       scenar  holds name of scenario now in use
C$CALLER  inimnu
C$METHOD
C  Loop over all parameter menus.  For each, bring in a /pmenu/ record
C  using the pointer in pxref.  Then use pmenu to find (in /pdesc/) th
e
C  starting address in pvalues where the given menus parameters will b
e
C  stored.  Then open the proper parameter relation and do a read usin
g
C  the scenario key, sending the data directly into pvalues.
C        icursr   relate cursor
C        flag     true if relation read failed
C        temp     relation name
C        name     delimited relation name
C**
```

```
C     QRPOP*************************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE QRPOP (LINE, LENLINE )
C*                      *** FORMAL PARAMETER DECLARATIONS ***
%INCLUDE INCPAR
       INTEGER  LENLINE
       CHARACTER LINE*LLINE
C*                                              *** ABSTRACT ***
C$PURPOSE pop and read from list menu command queue
C$AUDIT HISTORY
C         MEMutchler       17 JAN 83  AUTHOR
C        MEMutchler        8  FEB 83   TESTER (program tstpop)
C$TYPE    listqu utility
C$FORMAL PARAMETERS
Cout      line    top command from queue
Cout      lenline length of 'line' in characters
C$COMMON BLOCKS
Cin       incpar  global parameter statements
Cio       queue   listqu and number in queue
C$CALLER  gloptn
C$METHOD  reads and pops the top item in listqu, length = number,
C         into line of length lenline.
C$LOCAL VARIABLES  none
C$$
```

```
C     SETLVL ************************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE SETLVL ( VALUE)
C*                          *** FORMAL PARAMETER DECLARATIONS ***
      LOGICAL VALUE
C*                                                  *** ABSTRACT ***
C$PURPOSE set status of 'number' list members to value
C        used for set of ALL candidates to "yes" or "no"
C$AUDIT HISTORY
C         MEMutchler            17 JAN 83  AUTHOR
C         MEMutchler            18  FEB 83  TESTER (program tsetl)
C$TYPE    mnurun utility
C$FORMAL PARAMETERS
Cin       value   set all status' to this
C$COMMON BLOCKS
Cio       lvalue  holds status array
C$METHOD  loop and assign
C$LOCAL VARIABLES
C         i       loop counter
C$$
```

```
C     LOGICAL FUNCTION SHOHLP ********************************
$CONTROL check=3,SEGMENT=MENU
      LOGICAL FUNCTION SHOHLP(IDUM)
C*                          *** FORMAL PARAMETER DECLARATIONS ***
      INTEGER IDUM
C*                                                  *** ABSTRACT ***
C$PURPOSE Inspects current parameter values to see if user
C      wants to have standard menu system command help shown.
C$AUDIT HISTORY
C          MSCarey          10-apr-84  AUTHOR
C$FORMAL PARAMETERS
Cin        idum     dummy
C$COMMON BLOCKS
Cin        pvalue   parameter values data structure
C$CALLER   menu system routines
C$METHOD
C  Looks at the PSMNUH parameter in the menu system data
C  structure.  If this parameter is true, returns true.
C$LOCAL VARIABLES
C          none
C$$
```

```
C     STEXEC*************************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE STEXEC
C*                                                  *** ABSTRACT ***
C$PURPOSE command handler, main MNUR stack executive routine
C$AUDIT HISTORY
C          MEMutchler      18 JAN 83  AUTHOR
C$TYPE     mnurun utility
C$FORMAL PARAMETERS  none
C$COMMON BLOCKS  none
C$CALLER  mnurun
C$METHOD  get next command off top of stack, call routine to process
C         it, untill no more commands on stack
C         MAIN LOOP
C           LOOK AT TOP OF STACK.  COMMAND TYPES ARE:
C           0 = STACK EMPTY (END SESSION)
C           1 = DISPLAY CHOICE MENU COMMAND ON STACK
C           2 = DISPLAY PARAMETER MENU COMMAND ON STACK
C           3 = EXECUTE THE RNPROC INTERFACE ROUTINE
C$LOCAL VARIABLES
C         ii       *4 version of command read
C         i        *2 version of command read
C$$
```

```
C     SVLVAL ************************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE svlval(ipgi)
C$PURPOSE SaVe List VALues.  Writes new statuses to list type
C         relation.
C$AUDIT HISTORY
C         MSCarey          26-JAN-83  AUTHOR
C$FORMAL PARAMETERS
Cin       ipgi     pointer to info for this list menu in pvalues
C$COMMON BLOCKS
Cin       lvalues holds candidate names and statuses for list.
Cin       ltpac    holds information identifying current list
C                  type relation. (List Type relation ACcess)
C$CALLER  dlmenu
C$METHOD
C  Assumes that the proper list type relation has already been
C  opened on the cursor in /ltpac/ and that the column fieldname
C  in the same block is correct.  Does
C  updates for every candidate.
C$LOCAL VARIABLES
C         dest     destination record for ltype relation queries
C         stat     candidate status portion of dest
C         cand     candidate name portion of dest
C         notfnd   true if failure to find a candidate
C         i        loop index
C         ecname   integer representation of CNAME(MAXLST)
C$$
```

```
C     SVPVAL***********************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE SVPVAL
C*                                                *** ABSTRACT ***
C$PURPOSE saves curent parameter menu's values to storage
C        relation
C$AUDIT HISTORY
C         MEMutchler           17 JAN 83  AUTHOR
C         MEMutchler           21 feb 83  TESTER  (program tsavvl )
C$TYPE    mnurun utility
C$FORMAL PARAMETERS none
C$COMMON BLOCKS
Cin       incpar  global parameter statements
Cin       pmenu   parameter menu info
Cin       pdesc   decsriptors for each parameter
Cin       pvalue parameter values
Cin       pxref   field list for each parameter menu
C$CALLER  dpmenu
C$METHOD  retrieve relation name, open
C         that relation, and update the tuple using
C         the field list and array of values.
C$LOCAL VARIABLES
C         rname   delimited relation name
C         string  undelimited realtion name
C         icursor cursor into relate for current relation
C         lenname length of name in non-blank chars.
C$$
```

```
C     SYSMAP**************************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE SYSMAP
C*                      *** FORMAL PARAMETER DECLARATIONS ***
C*                                                *** ABSTRACT ***
C$PURPOSE display a map showing flow of entire menu system
C$AUDIT HISTORY
C          MEMutchler      11-mar-83  AUTHOR
C$TYPE     mnurun utility
C$FORMAL PARAMETERS
C          none
C$COMMON BLOCKS
Cin        ioc       global io unit numbers
C$CALLER   hlpcmu, hlplmu, hlppmu
C$METHOD
C  Read from the sysmap file and print to screen, paged
C$LOCAL VARIABLES
C$$
```

```
C     USECF*****************************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE usecf(mnukey)
      character*8 mnukey
C*                                                  *** ABSTRACT ***
C$PURPOSE Redirects alias input stream from terminal to a
C         command file.
C$AUDIT HISTORY
C        MSCarey          02-feb-83  AUTHOR
C$FORMAL PARAMETERS
Cin      mnukey   name of current menu
C$COMMON BLOCKS
Cin       incpar  parameters
Cio      ioc     io unit assignments
Cio      comcfl  tracks command file information
Cio      rcrd03  field list and value destination for cflist relation
C$CALLER  various mnurun routines
C$METHOD
C  Discovers which menu the user is currently at.  Searches
C  the CFLIST relation for names of command files which may
C  begin execution at the given menu.  Displays these and
C  prompts the user to choose one or quit.  Then prompts for
C  display of menus on screen during execution.  Checks to see if
C  choice is in CFLIST; updates lastused field if it is.
C  Increments the command file nesting level by one and
C  checks against the limit; then opens the file and sets the
C  io switches properly.
C$LOCAL VARIABLES
C        len      length of a character variable
C        icursr   RELATE cursor number
C        jcursr   RELATE cursor number
C        temp     input buffer
C        file     name of command file
C        filnam   delimited name of command file
C        string   argument list to filopn
C        mesg     error message string
C        flag     error flag
C        notfnd   true if desired tuple not found
C        key      integer equiv buffer for relate index value
C**
```

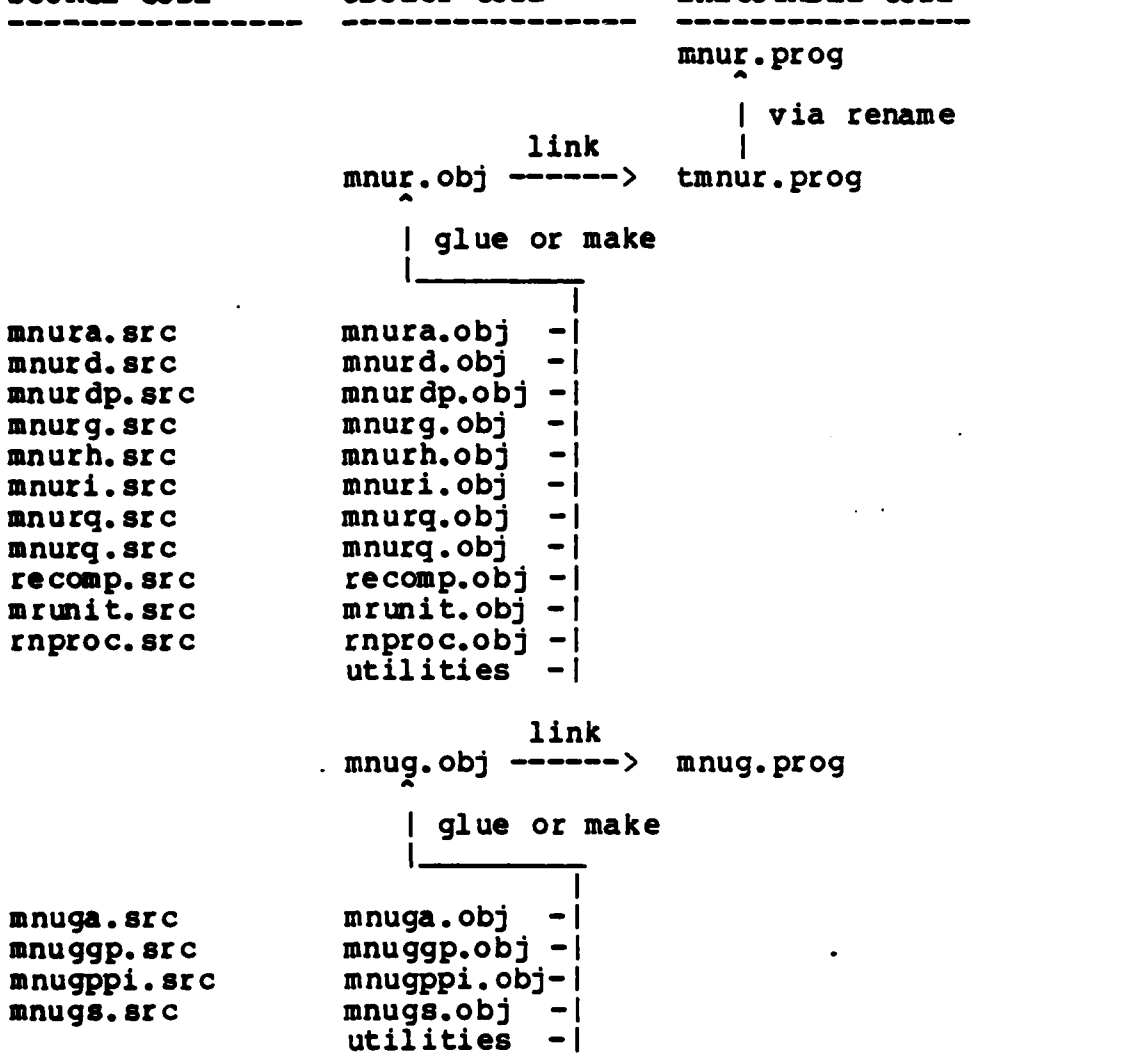
```
C     WTITLE**************************************
$CONTROL SEGMENT=MENU
      SUBROUTINE WTITLE (TITLE)
C*                        *** FORMAL PARAMETER DECLARATIONS ***
%INCLUDE INCPAR
      CHARACTER TITLE*LLINE
C*                                                  *** ABSTRACT ***
C$PURPOSE write menu title text to output screen
C$AUDIT HISTORY
C          MEMutchler           17 JAN 83  AUTHOR
C          MEMutchler           21 feb 83  TESTER  (program trmttl )
C$TYPE     mnurun
C$FORMAL PARAMETERS
Cin        title   title text to be displayed
C$COMMON BLOCKS
Cin        incpar  global parameter statements
Cin        ioc     i/o file assignments
Cin        terminal holds crt screen dimensions
C$CALLER   dlmenu, dpmenu, dcmenu
C$METHOD    prints string of characters in mtext file at IPTR record
C           centered on line of the menu display.
C           Lenght of the string is held at IPTR
C$LOCAL VARIABLES
C          lentxt  length of 'text'
C          blank   ' '
C$$
```

```
C     MRUNP ************************************************
$CONTROL segment=menu
      SUBROUTINE  mrunp
C*                                                  *** ABSTRACT ***
C#PURPOSE   Executive routine which performs actions for the
C     RNPROC menu system executive.  RNPROC makes subroutine
C     calls to run modules and perform functions represented
C     as menu options (other than display of another menu).
C     This routine has an entry point for all fortran options which
C     require startup of a son process, i.e. for all modules
C     which do not fit into the main menu system program,
C     and for BUILDER modules which require their own son process.
C#AUDIT HISTORY
C         MSCarey         13-dec-83  AUTHOR
C#FORMAL PARAMETERS
C         none
C#COMMON BLOCKS
Cin       pvalue    menu system parameter menu paramter value storage
Cin       lprnts    diagnostic switches
Cin       uzrprv    user privelege level information
Cin       senprm    scenario system paramters--holds dimensions for snrr
ef
Cin       snrref    map of actual scenario field keys for each DB file
C                   for the current scenario.
Cin       scenar    information on the current scenario
Cin       envirn    has develp flag indicating which version
C#CALLER rnproc
C#METHOD
C     There are two kinds of son-process module--those that require a
C     simple start-up, and those that also require that information
C     on the current system state be passed through an extra data
C     segment.  Each module/menu option has an entry point in this
C     routine; each entry sets procnm to the proper value,
C     and may perform special purpose code.  Entries then tranfer
C     control to a section which swaps out common blocks to a
C     communication segment and then starts the proc, or to a
C     section which just starts the proc.
C     The routine does lots of error checking on the intrinsic calls.
C     Note that at least one module, (Data Base Updater), returns
C     control to the menu system by suspending rather than
C     terminating itself.  This means that mrunit must keep track of
C     whether it needs to create a son, or merely reactivate it.
C
C     WHEN ADDING NEW MODULES:
C     make sure that a unique iproc (<= 100) value is set at the entry
C     point.  Make sure that pin(iproc) is reset to -1 UNLESS your
C     module suspends rather than terminates on user exit.  Make
C     sure to properly set mesg, talk, and restrt.
C     Set iflag=17 for all relate using sons, iflag=1 for others
C#LOCAL VARIABLES
C         procnm    name of the son process program file
C         prnamu    name of module for error messages
C         mesg      message written prior to creation if talk=.t.
C         talk      if true, mesg written before create call
C         iproc     must be set to a unique value for each module
```

```
C         modtyp    module type code, used to determine proper
C                   action to take when development version is
C                   being run: 1=program which will never have
C                   a 'T' version, e.g. TDP editor; 2=alias
C                   programmatic module which may have 'T' versions;
C                   for modules requiring a dedicated builder son,
C                   the developer is responsible for adding code
C                   to ensure that the 'T' version of the
C                   APPLICATION FILE is used when appropriate--
C                   see the entry point for the DBU for example.
C         pin       process identification numbers for each module
C                   In static storage so that non-terminating process
C                   id's are retained between calls to the routine.
C         restrt    if true AND activation of son process fails THEN
C                   routine attempts to re-create son process.  Allows
C                   trap of abort by son process which should only
C                   suspend, prevent user from having to restart system
C                   to restart module.
C##
```

TABLE 8-7

COMMAND SYSTEM CODE FILES



CODE MANIPULATION PROCEDURE SUPPORT FILES

OBJECT-CODE MERGE (for GLUE)	BATCH LINK (for LINK)	BATCH MERGE AND LINK (for MAKE)
mnur.merge	mnur.link	tmnur.link
mnug.merge	mnug.link	tmnug.link

8.3 THE SCENARIO SYSTEM

ALIAS is a multi-exercise, multi-user system. Data for more than one study may be maintained on-line simultaneously, even if the studies have to do with similar subjects, and several people may use the system simultaneously. Typical problems which arise in multi-exercise, multi-user systems are deadlocking and data loss. Two users might deadlock (a failure of ALIAS to continue execution, or, indeed, to do anything) if they both needed exclusive access to the same files simultaneously. One user could lock one file, and the other user another file, with each then attempting to gain access to the file the other had locked. Data loss occurs when the data for separate exercises is not fully separated: one user can unintentionally destroy another's work while saving his own.

If such problems are to be avoided, the data for each exercise must be stored separately. However, creation of a completely separate data base for each exercise and user would introduce a host of file-management and program-development problems. The current scenario system represents a compromise which attempts to provide the best of both worlds. Its rules are as follows:

1) Every data relation holding data which could differ between exercises, or into which any ordinary user will need to write data, must have "SCENARIO" as its first field.

2) Every user must choose a specific scenario to use when running ALIAS. He will be allowed to access only data belonging to that scenario.

3) No two users may use the same scenario simultaneously.

4) No program shall explicitly lock a system file (RELATE does dynamic file locking when performing tuple adds, updates, and deletes; this does not introduce any risk of deadlock).

The ALIAS data base, though consisting of only a single set of files, is therefore effectively partitioned into as many data bases as there are scenarios. This isolates the data for each

exercise while keeping file management difficulties to a minimum, since programs using the data base always know which relations to open. Since a given user always is the exclusive user of a scenario, no other user can interfere with his use of the data.

There is one major problem with this architecture. It is likely that most exercises will wish to use the most current versions of most of the data base data. But if each scenario has separate data storage, data entry/updating will have to be done for each scenario, not just once! This would be a terrible burden.

The system therefore does allow several scenarios/users to access the same data simultaneously---but only one scenario/user can have write access to the data for a given scenario at a time. Others have read access only.

This is implemented by maintaining a cross reference table which holds the scenario field key value for each relation for each scenario. Where a key field value is the the same as the name of the scenario it belongs to, the scenario has write access to the data in that relation. Otherwise, a user of the scenario has "indirect" read access to the data in the relation which belongs to the scenario whose name is the same as that of the key field value. Figure 8-7 displays the concept in tabular form.

Typically, it is the creator of a scenario who decides which relations will be read-only and which read-write for that scenario, and the decision is made for several groups or sets of relations at a time. However, a change capability allows revisions to be made.

8.3.1 Organization of the Scenario System

The scenario system is broadly distributed throughout ALIAS, consisting of six major data structures and four types of code. The data structures are:

FIGURE 8-7. Organization of Inter-Scenario Data Sharing

RELATION	SCENARIO FIELD KEY VALUES FOR SCENARIO:		
	MAIN	POM86-1	POM86-2
NCJODAT.PROJJ	MAIN	POM86-1	POM86-2
NCJOCOM.PROJJ	MAIN	POM86-1	POM86-2
NCJDAT.DESCJ	MAIN	POM86-1	POM86-1 *
SHDESC.MISCJ	MAIN	POM86-1	POM86-1 *
YARDID.YARDS	MAIN	MAIN *	MAIN *
NCJODAT.CURRJ	MAIN	MAIN *	MAIN *

* users of this scenario may not change data in this relation; they have only "indirect" access since the key field value is not the same as the scenario name

1) The scenlst.db and scendsc.db relations, which contain a list of currently existing scenarios and descriptive data for each.

2) The relsnl.sysrw file, a direct-access binary file which contains the tables of scenario key field values for each relation and each scenario. Each record holds a complete set of key field values for a single scenario.

3) The filinfo.db data dictionary relation, which contains important indexing information which associates each data base relation with a location in relsnl file records.

4) The snusers.sysrw relation, which contains a list of on-line ALIAS users and the scenarios each is working with.

5) An extra data segment in which the relsnl record for the in-use scenario is stored for fast access by Core and module routines.

6) The /scenar/ FORTRAN common block, which holds the name of the current scenario, the user's priveleges with regard to it, and the proper scenario field key value for the file open on each of the 20 DBIF cursors.

The types of code are:

1) Processors which allow users to manipulate the contents of the data structures just mentioned, that is, to list, create, modify, and delete scenarios. These take the form of several modular sets of FORTRAN routines which are executed from the ALIAS scenario choice menu. The majority of the code in these routines is concerned with presenting options to the user and with checking his responses for errors and consistency.

2) Integrity checking code within each ALIAS module. THE DEVELOPER OF EACH MODULE IS RESPONSIBLE FOR ENSURING THAT HIS CODE DOES NOT PRESENT THE USER WITH DATA BELONGING TO SCENARIOS OTHER THAT THE ONE BEING USED, AND FOR ENSURING THAT UPDATES ARE ONLY DONE TO DATA BELONGING THE THE SCENARIO IN USE. This is typically done either via SELECTs which make the proper scenario field value for the given relation a condition for retrievals, or by combinations of rtpcal/rbdpoint/RECORD POINT calls with SCENARIO as key, rtpnxt/rdbread/RECORD READ calls, and checking to ensure that the reads do not go past the end of data for the given scenario (and that updates and adds always have the proper scenario field value).

3) Integrity checking support and initialization utilities. These load the proper relsnl file record when a request for use of a given scenario is made, ensure that the user has access and that no one else is already using the scenario, provide a means for BUILDER applications to discover the proper scenario key value for a relation, and load the cursen array in the /scenar/ common block with the proper scenario field value each time a relation is opened through the DBIF routines. This last makes it much easier for programmers to ensure that their code enforces scenario security.

Also, DBIF routines which post changes to relations do last-ditch checking to ensure that the current user has write access to the requested relation for the current scenario. Aborts result if the access check fails.

8.3.2 Data Structures

The structures of the scenlst.db and scendsc.db relations are given in Figure 8-8, while sample contents are shown in Figure 8-9. Each scenlst record describes a single active scenario, giving its name, creator, creation data and last-used date, and read/write privelege limits as given by the creator. The rdallow and wrallow fields may have either a single user name as their value or the keyword "PUBLIC". The scendsc relation contains as many lines of text describing each scenario as creators chose to enter at creation time.

The relsnl file is a FORTRAN readable, binary file with fixed-length records 2413 bytes long. The format of a read statement for these records is:
READ(unit recnum) dlflag,scenam,(keyval(i),i=1,200)
where dlflag is a logical variable which is read as true when the record is for a deleted scenario, scenam is the name of the scenario the record belongs to, and the keyval are the scenario field key values which should be used in querying each relation.

In order to discover which element of the keyval array from the relsnl record holds the key value for a particular relation, the data dictionary relation filinfo.db must be consulted. The structure of this relation is given in Figure 8-10, with sample contents shown in Figure 8-11. The relation holds information for each data base file, and is used by other ALIAS subsystems and processors as well. Fields of interest to the scenario system are FILE, GROUP, SCENSET, and SCENUM. Data base relations are grouped into sets to make scenario creation and modification easier; the SCENSET field indicates which set each relation belongs to. The SCENUM field gives the index of the relsnl record element (keyval array location) which holds the proper scenario field key value to use when querying a given relation. Notice in Figure 8-11 that the STRT103 relation is listed as being in the "SYSTEM" set and that it has a value of 0 for scenum. "SYSTEM" and "LEGALS" relations are scenario

Figure 8-8. Scenlst and Scendsc Structure

```
FILE NAME              =SCENLST.DB.SEA90

                  T                C          L A U
                  Y PRINT          O          E D P   INT BEG   END
#   NAME          P  LEN   $  M SPECIAL      V D D SIZE WORD WORD

  1 SCENARIO      A  12          UPPER      3 Y Y  12B    0    5
  2 CREATOR       A   8          UPPER      3 Y Y   8B    6    9
  3 RDALLOW       A   8          UPPER      3 Y Y   8B   10   13
  4 WRALLOW       A   8          UPPER      3 Y Y   8B   14   17
  5 CREATED       D  10          M/D/C      3 Y Y   2W   18   19
  6 LASTUSED      D  10          M/D/C      3 Y Y   2W   20   21

PRINT LINE WIDTH = 67 CHARACTERS.


FILE NAME              =SCENDSC.DB.SEA90

                  T                C          L A U
                  Y PRINT          O          E D P   INT BEG   END
#   NAME          P  LEN   $  M SPECIAL      V D D SIZE WORD WORD

  1 SCENARIO      A  12          UPPER      3 Y Y  12B    0    5
  2 LNUM          I   4                     3 Y Y   1W    6    6
  3 DESCRIP       A  65                     3 Y Y  66B    7   39

PRINT LINE WIDTH = 89 CHARACTERS.
```

Figure 8-9. Sample Scenlst and Scendsc Contents

SCENLST.DB

SCENARIO	CREATOR	RDALLOW	WRALLOW	CREATED	LASTUSED
MAIN	DBA	PUBLIC	DBA	9/21/1983	8/15/1984
FIXIT	DBA	PUBLIC	PUBLIC	4/17/1984	8/21/1984
POM86	DBA	PUBLIC	DBA	7/05/1984	8/15/1984

SCENDSC.DB

SCENARIO	LNUM	DESCRIP
MAIN	1	THIS SCENARIO HOLD THE MAIN DATA FOR THE ALIAS SYSTEM
MAIN	2	IT IS THE ONLY SCENARIO WHICH IS MAINTAINED BY SYSTEM DATA ENTRY
FIXIT	1	COPY OF THE MAIN SCENARIO; TO BE USED FOR FIXING UP THE DATA
FIXIT	2	D FROM CSDS SO THAT SCENARIO MAIN IS AVAILABLE AS A BACKUP.
POM86	1	908 SHAPM SCHEDULES

Figure 8-10. Structure of the Filinfo.db Relation

FILE NAME =FILINFO.DB.SEA90

#	NAME	TYP	PRINT LEN	$	COM	SPECIAL	LEV	ADD	UPD	INT SIZE	BEG WORD	END WORD
1	FILE	A	8			UPPER	3	Y	Y	8B	0	3
2	GROUP	A	8			UPPER	3	Y	Y	8B	4	7
3	FAMILY	A	20			UPPER	3	Y	Y	20B	8	17
4	DOMAIN	A	10			UPPER	3	Y	Y	10B	18	22
5	CODERESP	A	8			UPPER	3	Y	Y	8B	23	26
6	RELNUM	I	6				3	Y	Y	1W	27	27
7	SCENSET	A	8			UPPER	3	Y	Y	8B	28	31
8	SCENUM	I	6				3	Y	Y	1W	32	32
9	SCRSET	A	8			UPPER	3	Y	Y	8B	33	36
10	SCRNUM	I	6				3	Y	Y	1W	37	37
11	SCRPROC	I	7				3	Y	Y	1W	38	38
12	SCRINDX	I	2				3	Y	Y	1W	39	39
13	QDESCRIP	A	65				3	Y	Y	66B	40	72
14	ENTRYBY	A	8			UPPER	3	Y	Y	8B	73	76
15	ENTRYDATE	D	10			M/D/C	3	Y	Y	2W	77	78

PRINT LINE WIDTH = 200 CHARACTERS.

Figure 8-11. Sample Contents of the Filinfo.db Relation

```
GROUP     FILE      FAMILY                    DOMAIN      CODERESP RELNUM
SCENSET   SCENUM SCRSET    SCRNUM SCRPROC SC
QDESCRIP
ENTRYBY ENTRYDATE

CURRJ     NCJOCOM  CURRENT JOBS              PUBLIC                    0
CURRJ          1 CURRJ          11      102  0
Comments regarding the construction of each current ship.
MARK       4/19/1984
CURRJ     NCJODAT  CURRENT JOBS              PUBLIC                    0
CURRJ          2 CURRJ          12      102  0
Milestone dates for the construction of each current ship.
MARK       4/19/1984
CURRJ     REJOCOM  CURRENT JOBS              PUBLIC                    0
CURRJ         26 CURRJ          39      102  0
Current repair job schedule comments.
MARK       4/19/1984
CURRJ     REJODAT  CURRENT JOBS              PUBLIC                    0
CURRJ         25 CURRJ          38      102  0
Current repair job schedules (including SLEP  refuel).
MARK       4/19/1984
DB        STRT103  SYSTEM                    PUBLIC                    0
SYSTEM         0                34      103  0
Dummy file used by DBU.
MARK       4/19/1984
DESCJ     NCJDAT   JOB DESCRIPTIONS          PUBLIC                    0
DESCJ         21 DESCJ          13      104  0
Time planning factors for each stage of a class's construction.
MARK       4/19/1984
DESCJ     NCJDCOM  JOB DESCRIPTIONS          PUBLIC                    0
DESCJ         22 DESCJ          14      104  0
Comments for the planning factors for construction jobs.
MARK       4/19/1984
HISTJ     NCJOCOM  HISTORICAL JOBS           PUBLIC                    0
HISTJ          3 HISTJ          15      102  0
Comments regarding the construction of each historical ship.
MARK       4/19/1984
HISTJ     NCJODAT  HISTORICAL JOBS           PUBLIC                    0
HISTJ          4 HISTJ          16      102  0
Milestone dates for the construction of each historical ship.
MARK       4/19/1984
HISTJ     REJOCOM  HISTORICAL JOBS           PUBLIC                    0
HISTJ         28 HISTJ          41      102  0
Historical repair job schedule comments.
MARK       4/19/1984
HISTJ     REJODAT  HISTORICAL JOBS           PUBLIC                    0
HISTJ         27 HISTJ          40      102  0
Historical repair job schedules (including SLEP
```

independent: they do not have a scenario field and therefore do not require any key values from the relsnl record.

Because of the large amount of memory required to store a relsnl record and the many demands on the restricted memory available to the Core, global storage for the relsnl record for the user's current scenario is in an extra data segment (see section 10.6 for the segment's ID). The segment also holds the names (file.group) of each of the scenario dependent relations in the form of a twin to the keyval array. This makes the scenario field key values available to other modules, including BUILDER modules (the DBU needs the key values, for example). Core FORTRAN routines may query this segment for key values by calls to the snrlnm and snrlsn functions (technically part of the DBIF). Snrlnm takes a file.group name as an argument, searches the list of relations in the data segment for a match, and returns the index of the corresponding key value in the segment. The value can be retrieved by a call to snrlsn with the index as an argument.

The snusers.sysrw relation (structure in Figure 8-12) holds a list of users currently running ALIAS, and the names of the scenarios each is working with. The date field is present to assist in cleaning out records left there after abnormal ALIAS terminations (a normal Quit or End from the command system causes the record for the given user to be removed). The relation is an important part of the means by which the rule that no two users may use the same scenario simultaneously is enforced.

The /scenar/ include file is pictured in Figure 8-13. The contents of the /scenar/ common block forms the primary interface between the scenario system and the code of the Core and FORTRAN modules. The block includes the name of the scenario currently in use, and three arrays indexed by a DBIF cursor index. An integer index number is returned by the DBIF in response for a request for a new path---this index is used to specify the given path in calls to DBIF tuple manipulation utilities. The path/file open routines will query the extra data segment with the current relsnl record for the proper scenario field key value to use in queries on the given path as part of their operations; they place undelimited and delimited versions of the key in the cursen and dlmsen arrays. They also compare the key value with the current scenario's name, and set the proper element of the wrtprv write privelege flag array accordingly (.true. only if the two match).

8.3.3 Initialization and Utility Processors

Four processors perform scenario system utility and interface functions.

8.3.3.1 SNSTRT

Figure 8-14 is a flow diagram for Core subroutine snstrt, which initializes the scenario system as part of overall ALIAS initialization. The routine has two primary functions: it

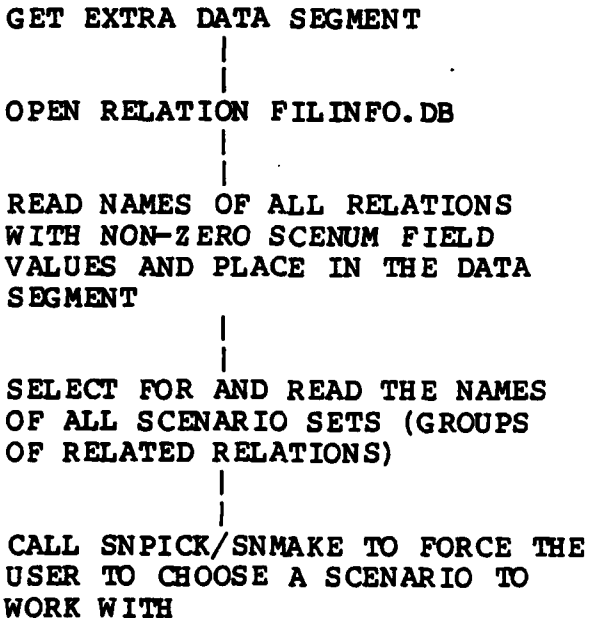
Figure 8-12. Structure of the Snusers.sysrw Relation

```
FILE NAME                =SNUSERS.SYSRW.SEA90

                  T              C              L A U
                  Y PRINT        O              E D P   INT BEG   END
#   NAME          P  LEN  $      M SPECIAL      V D D SIZE WORD WORD

  1 SCENARIONM    A   12            UPPER        3 Y Y  12B    0    5
  2 USERNAME      A    8            UPPER        3 Y Y   8B    6    9
  3 DATE          D   10            M/D/C        3 Y Y   2W   10   11

PRINT LINE WIDTH = 38 CHARACTERS.
```

Figure 8-13. The /scenar/ Include File

```
C  include file scenar
      parameter sncurs=20
      common /scenar/actsen,cursen(sncurs),dlmsen(sncurs),
     1               wrtprv(sncurs),snwovr
      character*12 actsen,cursen,dlmsen*14
      logical wrtprv,snwovr
      integer alinsen
      equivalence (alinsen,actsen)
C
C        current scenario data for application routines
C     sncurs   max number of relate cursors
C     actsen   name of current scenario
C     cursen   scenario key value for relation i
C     delsen   delimited version of cursen
C     wrtprv   true if user may write on the current cursor
C     snwovr   scenario write privelege override; allows
C              write on cursors regardless of scenario
C              status if true; used by scenario creator
C
```

FIGURE 8-14. Flow of SNSTRT Execution



creates and fills the extra data segment which is used to store the list of scenario-dependent relations and the relsnl record for the current scenario, and it forces the user to choose or create a scenario to work with.

8.3.3.2 SNQUIT

Core subroutine snquit flushes the current user's record from the snusers.sysrw relation as part of the Core's close-and-cleanup termination sequence. If this is not done, the scenario the user was working with will not be accessible to other users even after he logs off.

8.3.3.3 Utilities Located in the DBIF

The scenario system's primary purpose is the protection and management of the use of the data base, while the Data Base Interface library of FORTRAN routines provides the preferred means of programmatic use of the data base. An interface between the two subsystems was required: it takes the form of four scenario system routines located in the DBIF libraries.

Figure 8-15 illustrates the manner in which the routines function. When an open file/create path request is made by a call to one of the DBIF routines performing such functions, rvscen is called to retrieve the proper scenario key value for the given relation from the scenario system extra data segment and places it in the /scenar/ common block. It does this by searching the relation list in the data segment and then retrieving the proper key value using the snrlnm and snrlsn functions, as described in Section 8.3.2 above. If it cannot find the relation on the list the value is simply set to blanks, thus allowing the DBIF to be used to work with relations which are scenario-independent.

Rvscen also sets the proper write-privelege flag in the /scenar/ common block each time a cursor is opened. When calls are made to DBIF routines which add to or update scenario-dependent files, subroutine ckwprv is called as the last-ditch privelege check before the changes are posted. Ckwprv causes an abort if it finds that priveleges are lacking.

8.3.3.4 BUILDER Application Interface

Developers of BUILDER applications are responsible for ensuring that their processors enforce scenario security just as FORTRAN programmers are. In order to do so, the developer must be able to find out the proper scenario field key value for scenario-dependent data base files he wishes to use. This cannot be done by BUILDER code alone, since the information is held in the extra data segment maintained by the scenario system. The BUILDER-callable FORTRAN subroutine GETSCENV was created to provide this service (object code in sl.pub (required for FORTRAN routines to be called from BUILDER), source code in slproc.src).

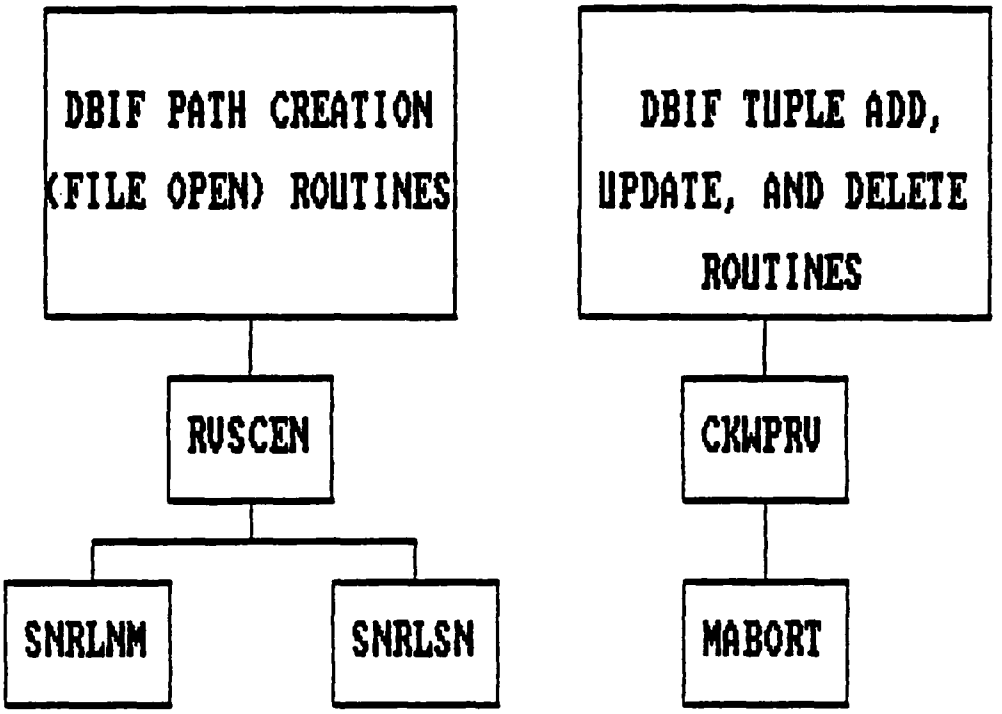


Figure 8-15. Scenario System - DBIF Interface Routines

GETSCENV depends on the existence of the scenario system extra data segment, and on its data being for the current scenario (this should cause no problem for any module executed by the System Core). It expects two arguments to be placed on its call procedure line: the name of the relation for which a key value is desired, and the name of the BUILDER variable into which the value should be placed. For example, if the current key value for ncjodat.projj was needed in application variable KEYVAL (must be data type alpha, length=12), then the call to getscenv should read "CALL PROCEDURE GETSCENV NCJODAT.PROJJ KEYVAL". Getscenv uses information provided in the call (see the BUILDER manual) to get the file name and variable name as given on the call line. It then searches the extra data segment using a method similar to the one described in Section 8.3.2 involving snrlnm-snrlsn calls, retrieves the key value, and places it in the storage area for the named variable.

8.3.4 User Service Processors

By displaying the command system's scenario choice menu, ALIAS users may choose to execute scenario system processors which will:

1) allow use of a different scenario
2) list or print the names and structures of existing scenarios
3) create new scenarios or delete existing ones
4) modify the structure of existing scenarios

These processors are composed of three types of FORTRAN subroutines: user interface routines whose primary purpose is presentation of options to the user and receipt and verification of responses; workhorse routines which perform substantial, specialized processing at the direction of one or more of the interface routines; and scenario system utilities which perform commonly required low-level data structure manipulations.

Table 8-8 contains an annotated listing of the routines which form the processor; Table 8-9 describes the include files which the routines require.

Substantial use is made of the prthlp utility from the UTLR library for presentation of options to users. This utiltity searches the modhlp.sysro file for a section whose name is an argument given to prthlp, and then prints the section to the device of the programmer's choice. Rather than hard-wiring the

TABLE 8-8

SCENARIO SYSTEM ROUTINES IN USER SERVICE MODULES

ROUTINE	PURPOSE
SNALTR	Modifies the structure of a scenario on a relation-by-relation basis. That is, the user can "re-create" sections of his scenario data base, drawing data from other scenarios. He can also change the data in a given relation from indirect to direct access.
SNALTS	Same as snaltr except changes are made at the relation set level---i.e. for a group of related relations.
SNCHEK	Streams a batch job which runs a BUILDER procedure to check the integrity of a newly created or modified scenario.
SNCLEN	Workhorse for SNDEL, the scenario deletion executive. When a scenario is deleted, all its data is deleted except for that being used "indirectly" by other scenarios. Ownership of the latter data is transferred to one of the indirect accessors, and the contents of the relsnl file is updated to reflect this. SNCLEN checks for such data and takes care of any reassignments.
SNCOMD	Command line parsing utility for scenario routines. Checks for a valid command character as the first character of a command and passes back an appropriate index if one is found.
SNCOMP	SceNario COMPose. Part of the SNMAKE module, sncomp queries the user for the name of the source scenario for each relation set.
SNCOPY	Also part of the SNMAKE module, SNCOPY acutally sets up a new scenario's data structure. Copies are made of source data which is to be direct access (read-write) in the new scenario, and the relsnl record is set up.
SNDEL	SceNario DELete. Flushes all data which belongs solely to a given scenario from the data base, and removes the name and description from the list of active scenarios.

TABLE 8-8

SCENARIO SYSTEM ROUTINES IN USER SERVICE MODULES

ROUTINE	PURPOSE
SNDSCR	Part of the SNMAKE module, sndscr prompts the creator of a new scenario for a name and description, and does some security checking.
SNGRLN	Utility which finds the record in the relsnl file for the NAME scenario and passes the record contents back in an array.
SNLIST	Prints a list of existing scenarios and their descriptions to terminal or printer.
SNLOAD	Utility which calls sngrln to get a relsnl record and then loads the record into the scenario system's extra data segment.
SNMAKE	SceNario MAKE. Executive for creation of new scenarios.
SNMDFY	SceNario MoDiFY. Executive for modification of structure of existing scenarios. Prompts for name of scenario to modify, checks security, and then calls either snaltr or snalts.
SNMKUP	Prints the composition of an existing scenario (essentially, the contents of the relsnl record) to screen or printer, either on a relation-by-relation basis or a set basis.
SNPICK	Lets user choose a (different) scenario to work with. Prompts for the name and does security checking.
SNPRNT	Interface between the command system and snlist---tells snlist to format the list of scenarios for a printer.
SNQUIT	Called by the command system on normal termination to take the user's name off the list of active on-line users.
SNSEEG	Prints the composition of a scenario by relation.
SNSEES	Prints the composition of a scenario by relation set.
SNSHOW	Interface between the command system and snlist---tells snlist to format the list of scenarios for the terminal screen.

TABLE 8-8

SCENARIO SYSTEM ROUTINES IN USER SERVICE MODULES

ROUTINE	PURPOSE
SNSTRT	Initialization routine for the scenario system. Creates and partially fills the extra data segment, and forces user to pick a scenario to work with by calling snpick.
SNUNSD	Checks to see if anyone is currently using a given scenario, and, optionally, makes a notation that the calling user now is using it.

TABLE 8-9

INCLUDE FILES USED BY THE SCENARIO SYSTEM

FILENAME	PURPOSE
FLD08	Data statement with field name list for scenlst relation. Rcrd08 must be included also where this is used.
IOC	System Core global i/o unit numbers.
LPRNTS	Boolean switches to control diagnostic output.
PRMCRS	Permanently open System Core cursor indexes. One is open on snusers.sysrw.
RCRD08	Data record for reads from scenlst relation.
SCENAR	Data for scenario currently in use: its name, and the key field values for any any relations opened through the DBIF.
SENPRM	Scenario system FORTRAN parameter statements (for dimensioning).
SNRREF	List of relation sets and data needed to access the scenario system extra data segment. Declarations for the snrlnm and snrlsn character functions found in the DBIF source libraries.
UZRPRV	Priveleges and name of the current user.

scenario system menus and help into FORTRAN format statements, these were stored in modhlp.sysro as editable text for presentation by prthlp.

8.3.4.1 SNPICK

SceNario PICK is the routine called during ALIAS initialization to obtain the name of the scenario to be used, and is also called when the user wishes to change scenarios in mid-session. Figure 8-16 presents a calling tree diagram for snpick. The routine first prints its menu to the terminal via a call to prthlp and prompts for a scenario name or command character. Sncomd is called to parse the response, with valid command character options implying calls to snlist for a print of existing scenarios to the screen, snmkup for a dump of the structure of a given scenario to the screen, or snmake if the user wishes to create a new scenario to work with. If a scenario name is given rather than a recognition character snpick makes sure the name is that of an existing scenario (by a point into the scenlst relation), calls snunsd to see if anyone else is using it already, calls snload to fetch the relsnl key value record and place it in the extra data segment, and finally calls pvinit to (re-)initialize the command system's /pvalues/ data stucture with parameter menu values for the given scenario. If any errors are encountered a message results and the user is prompted for another command.

8.3.4.2 SNSHOW and SNPRNT

SceNario SHOW and SceNario PRiNT are "dummy" routines which act as an interface between the command system and the general-purpose scenario system print/list utility snlist. Snshow calls snlist, specifying output of a list of existing scenarios to the terminal, while snprnt requests snlist to send the same list to the printed output device which the user has selected for the current scenario on the User Environment Parameters menu. Snprnt will also optionally print the structure of each scenario by calling snseeg.

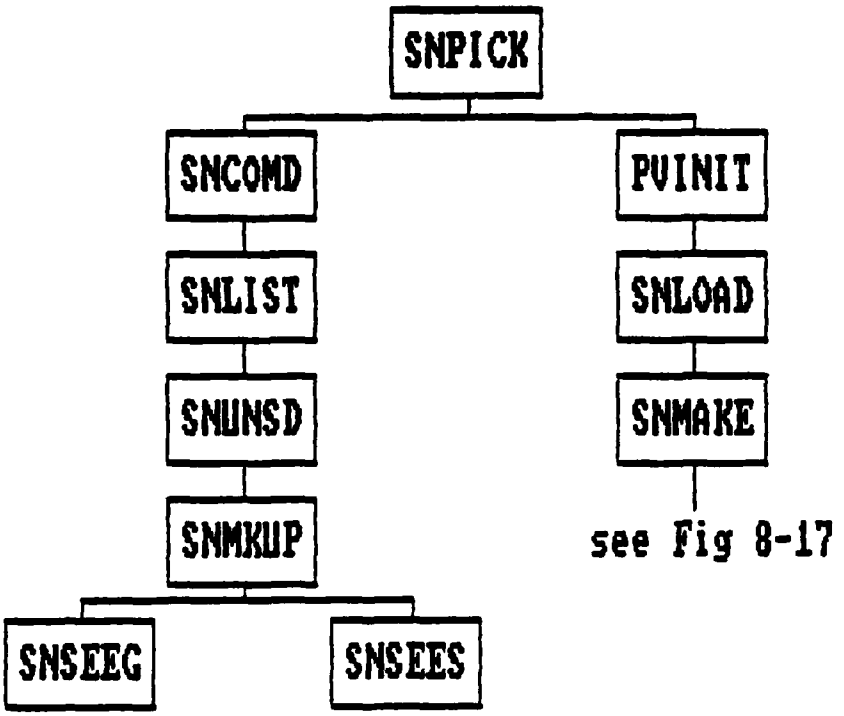


Figure 8-16. SNPICK Calling Tree Diagram

8.3.4.3 **SNMAKE**

SceNario MAKE creates new scenarios. Creation is a six-stage process:

1) Get the name and description of the new scenario from the user and place it in a buffer pending successful completion of the process

2) Get the name of the scenario to use as a data source for each set of data base relations, and whether the source data should be copied or used indirectly

3) Construct a relsnl record for the new scenario and store it

4) Copy data records with the source scenario name, giving the copies the new scenario's name, in appropriate relations; add the data identifying the new scenario to the scenlst and scendsc relations

5) Check the new scenario for internal consistency by streaming a batch job which will perform en masse the DBU's consistency checks

6) Make the new scenario the user's current working scenario, on the presumption that he wants to use it immediately

Figure 8-17 is a calling tree diagram for the creation module. Snmake, the executive, calls sndscr to obtain the new scenario name and description, sncomp to prompt for the data source for each relation set, sncopy to construct the new relsnl record and to make any copies of data base data that are required, and snchek to stream the consistency-checking batch job.

Defining the composition (source of initial data) of the new scenario is the most important part of the process. Relations in the data base are grouped into sets according to subject (these sets generally follow the division of the data relations among HP file groups, e.g. there is a yards set which resides in the .yards group, a job descriptions set, a projected new construction jobs set, etc.). Although a user creating a new scenario might wish to draw data from several existing scenarios,

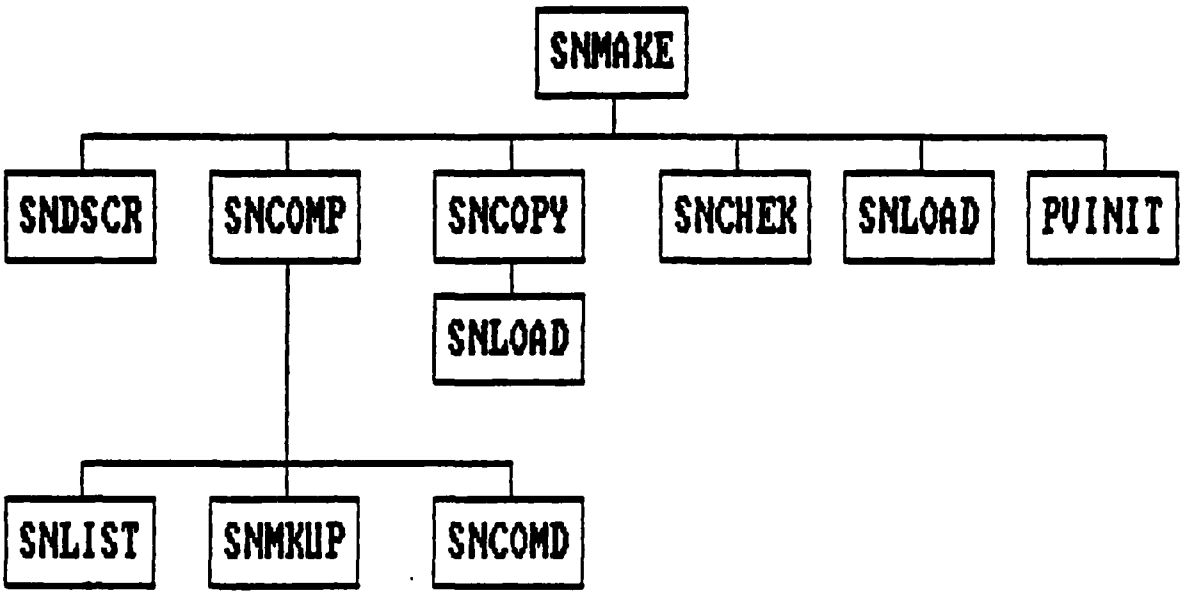


Figure 8-17. SNMAKE Calling Tree Diagram

generally he will want to do so in blocks corresponding to the sets rather than on a relation-by-relation basis.

The user has three options for the source of initial data for each set:

1) No data. The relations in a set can have no tuples with the new scenario's name as a scenario field value.

2) A copy of the data from another scenario, which users of the new scenario may modify as they please.

3) Access to the data from another scenario, but only via the "indirect" or "read-only" method. This is the most efficient option for sets whose data the user does not expect to need to modify.

The mechanics of the creation process require that the user be prompted for his choice for each set. Routine sncomp loops through the list of relation sets (read in by snstrt), asking the user for a source scenario name for each. If the name is the same as that of the new scenario, it is assumed that a clean slate is desired for that set. If the name of another existing scenario is given a second prompt asks whether the user will want to make changes to the data in the set appears, and the decision to make a copy of the source data or set up indirect access to it is made on the basis of the response.

Sncopy then constructs the data structure for the new scenario. The routine loops through all the scenario-dependent relations listed in filinfo.db. It identifies the scenario set each belongs to and looks up the initial data source decision entered during sncomp execution. If another scenario is listed as the source of data, the relsnl record for that scenario is retrieved and the proper scenario field key value for the given relation is extracted. If access to the data is to be indirect, this key is put in the relsnl record for the new scenario and execution passes to the next relation. Otherwise, the key is used to extract all the source scenario records from the relation, which are then added to the relation with the new

scenario's name in their scenario key field (i.e., the data is copied).

Since the user is able to combine data from several different scenarios to create a new one, there is no guarantee that the result will have good data integrity. For example, if the yards set from scenario A is combined with the projected jobs set from scenario B, there may be jobs assigned to yards for which A has no data. An integrity check is performed to identify such problems, which the scenario's creator must then rectify using the DBU.

The integrity check is done by brute force: every required and recommended relationship as defined in the filjoin.db data dictionary relation is checked for every relevant record in the data base. For example, every record in the ncjodat.projj relation (new construction projected jobs) is read, with a point into the yardid.yards relation to ensure that there is data for the yard performing the job, and a point into the shdesc.miscj relation to ensure that there is data for the ship class the job's ship belongs to. Missing data is noted on a printed report.

Production of the integrity-checking report can be very time consuming, since an enormous number of data base i/o operations are required. The report is therefore produced by a batch job, which is streamed programmatically by the snchek routine. The job runs a BUILDER procedure (in file integck.db) which contains the checking logic.

8.3.4.4 SNDEL

SceNario DELetion is accomplished by subroutine SNDEL and routines called by it (see Figure 8-18 for a calling tree diagram). Sndel operates in the usual fashion of scenario system routines which interface with the user, putting a menu on the screen (via a call to prthlp) and accepting either a command

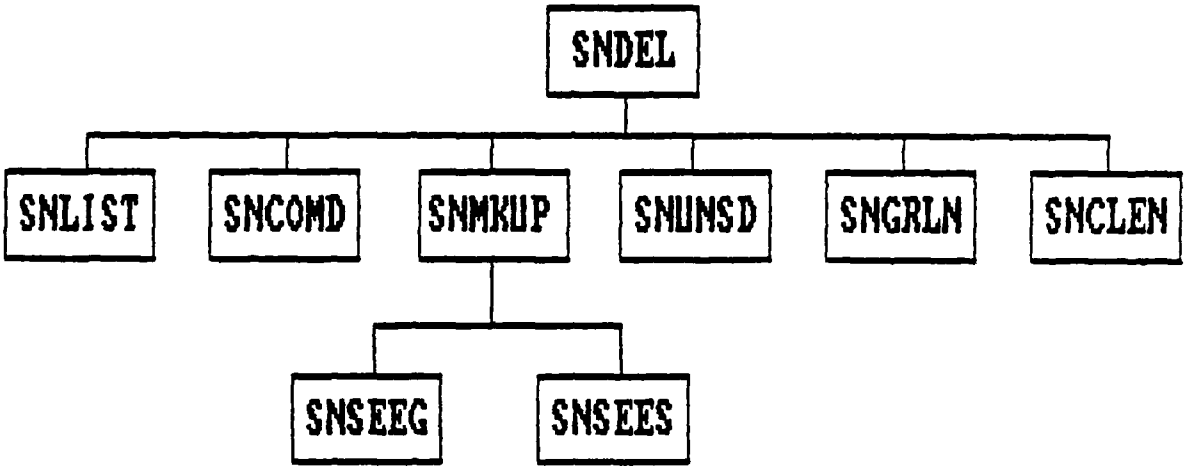


Figure 8-18. SNDEL Calling Tree Diagram

recognition character or the name of a scenario to delete. If the scenario named is not currently in use by anyone, and if the user is either the scenario's creator or the DBA, deletion commences. A RELATE 'DELETE FOR SCENARIO="scenarioname"' command is given for each scenario dependent relation which has "scenarioname" as its relsnl key value, except where another scenario is using the data "indirectly". In such cases a 'LET SCENARIO="indirect_user_name" FOR SCENARIO="scenarioname"' command is given, transferring ownership of the data. When the process is complete for all relations the relsnl record for the scenario is marked as deleted and the descriptive data is deleted from the scendsc.db and scenlst.db relations.

8.3.4.5 Modification of Composition

The snmdfy processor allows the creator of a scenario to modify its composition at a later data by bringing in data from other scenarios, or by changing data from read-only to read-write (via a copy). In contrast to creation, these changes can be made on either a relation-by-relation basis or for entire relation sets.

Figure 8-19 is a calling tree diagram for snmdfy. Snmdfy itself is primarily a user-interface and executive routine, calling snaltr to make changes on a relation-by-relation basis and snalts for changes at the set level. Snseeg and snsees list the composition of existing scenarios by relation or by set, respectively.

Snalts and snaltr work in a fashion similar to that of the creation processor, extracting the new source name (syntax for user input is "RELATION/SET=newsource"), checking for priveleges, prompting for read/write or read only status (i.e. copy the data or set up indirect access), and performing any copies, deletions, and relsnl record modifications required.

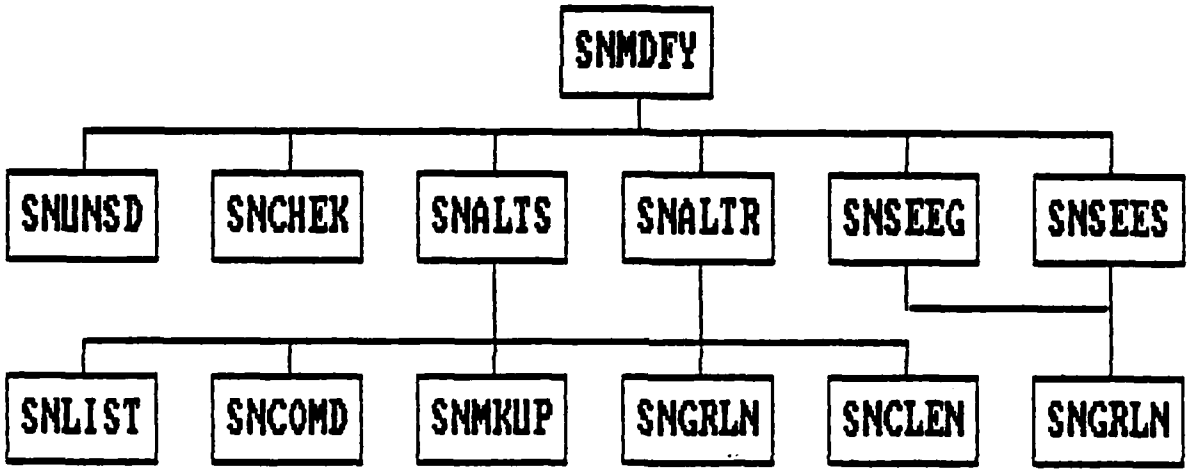


Figure 8-19. SNMDFY Calling Tree Diagram

When the user exits snmdfy snchek streams the integrity-checking job to ensure that no damage has been done by the modifications.

8.3.5 Interfaces With Other Units

The scenario system is in some sense nothing but an interface, and one which is used by every part of the ALIAS system. However, as mentioned in previous subsections, the manner of these interfaces is limited and well defined.

The scenario system "interfaces" with the data base structure by consulting the data dictionary, particularly filinfo.db, for lists of scenario-dependent relations and their corresponding key value storage locations in the relsnl.sysrw file. It also relies on the convention that the scenario field will be the first field in any relation.

Some security against improper data base modifications is offered by the rvscen/ckuser routines in the DBIF, which ensure that no writes are done to scenario data which a user has "indirect", read-only access to.

The job of writing scenario security into FORTRAN modules is made easier by the /scenar/ common block, which the DBIF places scenario key field values in when relations are opened. Securing of BUILDER modules is made easier by the getscenv FORTRAN procedure located in the SEA90 account Segemented Library, callable via "CALL PROCEDURE".

8.3.6 Files Which Are Part of the Scenario System

Data files and relations used by the scenario system are relsnl.sysrw (scenario structures), filinfo.db (list of scenario-dependent relations and indexes to relsnl record elements), scenlst.db and scendsc.db (basic identification and descriptive information for each scenario), snusers.db (on-line users and

scenarios they are working with), and modhlp.sysro (scenario system menu text).

Scenario system code files are listed in Table 8-10. Note that only the ckwprv routine in dbifa.src and the rvscen, snrlsn, and snrlnm routines in dbifrv.src belong to the scenario system. These routines are stored with the DBIF so that DBIF object code requires no external references except those found in UTLR.

Executable code for the scenario system's user modules is included in the System Core program file (mnur.prog) along with the Command System. They are examples of FORTRAN modules which require small enough data structures to be executed by CALLs within the Core rather than as son processes.

Since it is part of the Core, the scenario system is served by the same object code merge and link procedure support files as is the Command System.

8.3.7 Subroutine Abstracts

Documentation abstracts extracted from the scenario system routines follow in alphabetical order. Note that source and documentation for all include files is given in Section 10.5. Abstracts for the four routines which are stored as part of the DBIF (ckwprv, rvscen, snrlsn, snrlnm) are given in Section 10.2 along with other abstracts for the DBIF. A copy of the code for the BUILDER service procedure GETSCENV is given in Section 10.4 along with the other procedures stored in the account SL.

```
C     SNALTR ************************************************
$CONTROL segment=scen
      SUBROUTINE snaltr(scen,cursor,slcurs,changd)
C*                        *** FORMAL PARAMETER DECLARATIONS ***
      integer slcurs,cursor
      character*12 scen
      logical changd
C*                                                *** ABSTRACT ***
C*PURPOSE   SceNario ALTer by Relation.  Changes the data source
C     for a single relation for the given scenario.
C*AUDIT HISTORY
C        MSCarey        04-sep-83  AUTHOR
C*FORMAL PARAMETERS
Cin      scen     name of scenario to make alterations to
Cin      cursor   cursor open on filinfo
Cin      slcurs   cursor open on scenlst
Cout     changd   true if a change requiring a scenario integrity
C                 check was made
C*COMMON BLOCKS
Cin      snrref   need scenario field value cross ref data struc
Cin      ioc      units for terminal io
Cin      ____08   record data structure for scenlst retrievals
C*CALLER snmdfy
C*METHOD
C     We need to get a relation name to change, and a scenario
C     name to change its source to.  Then act according to one
C     of three cases:
C     1: same name is given as tgt as is current: do nothing
C     2: name of actsen for given scenario is given: rename snrlsn
C     3: name is different and not actsen: if src is now actsen,
C        delete its data and rename; else rename
C*LOCAL VARIABLES
C        reln     name of relation to make change for
C        sname    new source scenario for that relation
C        comp     buffer holding cross ref composition of a scenario
C        oldval   pre-change scenario field key value for reln
C        newval   key value for reln in new source scenario
C        set      scenario set reln belongs to
C        file,group split up version of reln name
C        grpcur   cursor open on filinfo by file,group
C**
```

```
C     SNALTS ************************************************
$CONTROL segment=scen
      SUBROUTINE snalts(scen,cursor,slcurs,changd)
C*                      *** FORMAL PARAMETER DECLARATIONS ***
      integer slcurs,cursor
      character*12 scen
      logical changd
C*                                              *** ABSTRACT ***
C$PURPOSE   SceNario ALTer by Set.  Changes the data source
C     for a group of relations for the given scenario.
C$AUDIT HISTORY
C         MSCarey         04-sep-83  AUTHOR
C$FORMAL PARAMETERS
Cin       scen      name of scenario to make alterations to
Cin       cursor    cursor open on filinfo
Cin       slcurs    cursor open on scenlst
Cout      changd    true if a change requiring a scenario integrity
C                   check was made
C$COMMON BLOCKS
Cin       uzrprv    is user allowed to make changes?
Cin       snrref    need scenario field value cross ref data struc
Cin       ioc       units for terminal io
Cin       ____08    record data structure for scenlst retrievals
C$CALLER snmdfy
C$METHOD
C     We need to get a group name to change, and a scenario
C     name to change its source to.  Then act according to one
C     of three cases:
C     1: same name is given as tgt as is current: do nothing
C     2: name of actsen for given scenario is given: rename snrlsn
C     3: name is different and not actsen: if src is now actsen,
C        delete its data and rename; else rename
C$LOCAL VARIABLES
C         reln      name of relation to make change for
C         sname     new source scenario for that relation
C         comp      buffer holding cross ref composition of a scenario
C         rname     delimted version of reln
C         oldval    pre-change scenario keyword for a given reln
C         newval    scenario keyword for a given reln in new src scen
C         new       list of all newval-s for all relns in scen family
C         lst___    newval and oldval for last reln processed
C         relst     list of relations in given DB family
C         scenst    name of relation family to make change to
C         snum      location in scen key value storage buffer of
C                   keyword value for given relation
C         loc       family of snum values for relations in scenst family
C##
```

```
C     SNCHEK ************************************************
$CONTROL segment=scen
      SUBROUTINE snchek(name)
C*                        *** FORMAL PARAMETER DECLARATIONS ***
      character*12 name
C*                                                *** ABSTRACT ***
C$PURPOSE   Sets up and streams a batch job to check the
C     internal consistency of the new scenario.
C$AUDIT HISTORY
C         MSCarey         28-aug-83  AUTHOR
C$FORMAL PARAMETERS
Cin       name      name of new scenario
C$COMMON BLOCKS
Cin       ioc       global io units
C$CALLER snmake
C$METHOD
C     Tell the user what's going on; check for existence of a
C     previous check file in group and make user rename it;
C     Create check file and put new scenario name into its
C     first record.  Batch the job, program snok.pub.
C$LOCAL VARIABLES
C         c         user responses
C$$
```

```
C     SNCLEN ************************************************
$CONTROL segment=scen
      SUBROUTINE snclen(sname,recnum,delsnl,delete)
%include senprm
%include snrref
C*                        *** FORMAL PARAMETER DECLARATIONS ***
      character sname*12,delsnl*12(snmxrl)
      integer recnum
      logical delete(snmxrl)
C*                                                *** ABSTRACT ***
C*PURPOSE   Cleans up the data base and the relsnl file
C     preparatory to a scenario deletion, making sure no
C     scenarios dependent on the scenario to be deleted get
C     their data lost.
C*AUDIT HISTORY
C         MSCarey         04-sep-83  AUTHOR
C*FORMAL PARAMETERS
Cin       sname     name of scenario to be deleted
Cin       recnum    record number sname's relsnl record
Cin       delsnl    snrlsn from relsnl record 'record'
Cout      delete    relations for which a DELETE FOR should be done
C*COMMON BLOCKS
C         none
C*CALLER sndel
C*METHOD
C     We must look through each record of relsnl except 'record'
C     for instances of 'sname'.
C     On finding one, set that relsnl cell to the scenario name
C     for its record; then do a LET SCENARIO=name FOR SCENARIO=
C     'sname' for that relation; remember that this has been done
C     and assign subsequent scenarios with a dependency in the same
C     column to the scenario the LET ... was done for; mark that
C     relation as one NOT requiring deletion by sndel.
C*LOCAL VARIABLES
C         buffer    surrogate snrlsn
C         newnam    holds name of new field value for column, if any
C**
```

```
C     SNCOMD ************************************************
$CONTROL segment=scen
      INTEGER FUNCTION sncomd(comand)
C*                        *** FORMAL PARAMETER DECLARATIONS ***
      character*72 comand
C*                                                *** ABSTRACT ***
C$PURPOSE   SceNario COManD parser.
C$AUDIT HISTORY
C         MSCarey         04-aug-83  AUTHOR
C$FORMAL PARAMETERS
Cin       comand   command to be parsed
C$COMMON BLOCKS
C         none
C$CALLER various scenario routines
C$METHOD
C     Check the first non-blank to see if it is one of the
C     recognized command characters.  If the first character is
C     not a recognition character, returns code saying scenario
C     name was given.
C$LOCAL VARIABLES
C$$
```

```
C     SNCOMP ************************************************
$CONTROL segment=scen
      SUBROUTINE sncomp(descur,grpcur,name,maxgrp,ngrps,grpsrc,
     1                  cpygrp,quit)
C*                     *** FORMAL PARAMETER DECLARATIONS ***
      integer descur,grpcur,maxgrp,ngrps
      logical quit,cpygrp(maxgrp)
      character*12 name,grpsrc(maxgrp)
C*                                              *** ABSTRACT ***
C$PURPOSE   Queries user for names of scenarios to serve as
C     data source for each relation group in the data base.
C$AUDIT HISTORY
C        MSCarey         27-aug-83  AUTHOR
C$FORMAL PARAMETERS
Cin      descur   cursor for scenlst.db
Cin      grpcur   cursor for filstat.db
Cin      name     name of new scenario
Cin      maxgrp   parameter snmxgp
Cout     ngrps    number of groups found
Cout     grpsrc   name of source scenario for relset(i)
Cout     cpygrp   true if data copy must be done for this relset
Cout     quit     true if user want to abort creation process
C$COMMON BLOCKS
Cin      snrref   supports access to data segment
Cin      uzrprv   can user access that source scenario
Cin      scenar   name of current scenario
Cin      rcrd08   scenlst buffer
Cin      ioc      terminal io units
C$CALLER snmake
C$METHOD
C     Get a list of the current data base groups.
C     For each relset, prompt the user for a source scenario name.
C     Give a list of current scenarios and the quit and help options
C     at each query; let the new scenario name be one option.
C     Check to make sure the name is legal and store
C     it for passage out.
C$LOCAL VARIABLES
C        sname    name of source scenario for set
C$$
```

```
C     SNCOPY ************************************************
$CONTROL segment=scen
      SUBROUTINE sncopy(grpcur,name,maxgrp,
     1                  ngrps,grpsrc,cpygrp,clean)
C*                     *** FORMAL PARAMETER DECLARATIONS ***
      integer grpcur,maxgrp,ngrps
      logical clean,cpygrp(maxgrp)
      character*12 grpsrc(maxgrp),name*12
C*                                              *** ABSTRACT ***
C$PURPOSE  Copies scenario data for each data base file and/or
C     sets up the snrlsn tuple for this scenario.
C$AUDIT HISTORY
C        MSCarey        28-aug-83  AUTHOR
C$FORMAL PARAMETERS
Cin      grpcur   cursor open on filstat
Cin      name     name of the new scenario
Cin      maxgrp   parameter snmxgp
Cin      ngrps    number of data base groups
Cin      grpsrc   scenario to take relset(i) data from
Cin      cpygrp   true if data in relset(i) is to be changeable
Cout     clean    true if no consistency check need be made
C$COMMON BLOCKS
Cin      senprm   scenario parameters
Cio      snrref   relation names and scenario field key values
C$CALLER snmake
C$METHOD
C     For each relset, there are three options.
C      (1) User wishes to start fresh.  No copy is required;
C          just set the proper cells of newsnr to new name.
C      (2) User wishes to use data from another scenario
C          read-only (i.e. no copy).   Set proper cells of
C          newsnr to grpsrc.
C      (3) User wants his own copy.  Set the proper cells, open
C          each relation and make the copy for each file.
C
C     The names of data base files and their cell numbers are
C     extracted from filstat.db on the fly.
C$LOCAL VARIABLES
C$$
```

```
C     SNDEL**************************************************
$CONTROL segment=scen
      SUBROUTINE sndel
C*                      *** FORMAL PARAMETER DECLARATIONS ***
C*                                                *** ABSTRACT ***
C$PURPOSE   SceNario DELetion command routine.
C$AUDIT HISTORY
C         MSCarey         04-aug-83  AUTHOR
C$FORMAL PARAMETERS
C         none
C$COMMON BLOCKS
Cin       uzrprv    user name
Cin       scenar    name of current scenario
Cin       snrref    access to extra data segment
Clocal    rcrd08    scenlst record buffer
C$CALLER menu system
C$METHOD
C     Present the user with a command menu.
C     On receipt of a name from the user, see that it is valid.
C     Check to see that the user is high-privelege or was creator.
C     For each relation for which the scenario key field equals the
C     name the user gave, do a DELETE FOR SCENARIO="name".
C     Mark the relsnl record as deleted.
C     Delete the scenario description information from scenlst,scendsc
C     Present the command menu again.
C$LOCAL VARIABLES
C         delsnl    buffer for relsnl record
C$$
```

```
C     SNDSCR ************************************************
$CONTROL segment=scen
      SUBROUTINE sndscr(descur,name,quit,lend,desc,ncomt,comt)
C*                      *** FORMAL PARAMETER DECLARATIONS ***
      integer descur,ncomt,lend,desc(lend)
      character name*12,comt*72(10)
      logical quit
C*                                                *** ABSTRACT ***
C*PURPOSE   Gets the name and description of a new
C     scenario, makes sure it is unique, and catalogs it.
C*AUDIT HISTORY
C        MSCarey        19-aug-83  AUTHOR
C*FORMAL PARAMETERS
Cin      descur   cursor opened on scenlst.db
Cout     name     name of new scenario
Cout     quit     true if user wishes to abort creation process
Cin      lend     length of description record
Cout     desc     description record
Cout     ncomt    number of comment lines
Cout     comt     comments for the scenario
C*COMMON BLOCKS
Cin      senprm   scenario system parameters
Cin      ioc      global io units
Cout     rcrd08   data transfer record for scenlst relation
C*CALLER snmake
C*METHOD
C     Prompt for a name or a request for a list of current
C     scenarios or an abort request.  Check the name for
C     uniqueness.  Prompt for the rest of the description data.
C     Write to the data base.
C*LOCAL VARIABLES
C        cdesc    one line of description
C**
```

```
C     SNGRLN ************************************************
$CONTROL segment=scen
      SUBROUTINE sngrln(scen,recnum,target,notfnd)
C*                        *** FORMAL PARAMETER DECLARATIONS ***
%include senprm
      integer recnum
      logical notfnd
      character*12 scen,target(snmxrl)
C*                                                *** ABSTRACT ***
C#PURPOSE   Finds record in relsnl file for scenario scen and loads
C     the snrlsn array with its contents
C#AUDIT HISTORY
C         MSCarey         04-sep-83  AUTHOR
C#FORMAL PARAMETERS
Cin       scen      name of scenario to search for
Cout      recnum    record number of found record (undef if notfnd)
Cout      notfnd    true if search unsuccessful
C#COMMON BLOCKS
Cin       senprm    scenario system parameters
C#CALLER various sn routines
C#METHOD
C     Assume the file is open on snrsun; read sequentially.
C#LOCAL VARIABLES
C         name      name of scenario owning record
C         del       true if record is for deleted scenario
C##
```

```
C     SNLIST ************************************************
$CONTROL segment=scen
      SUBROUTINE snlist(cursor,unit,pagmod)
C*                      *** FORMAL PARAMETER DECLARATIONS ***
      integer cursor,unit,pagmod
C*                                                *** ABSTRACT ***
C$PURPOSE   Prints a list of currently available scenarios
C     to the unit given.
C$AUDIT HISTORY
C         MSCarey         18-aug-83  AUTHOR
C$FORMAL PARAMETERS
Cin       cursor    cursor index for scenlst relation
Cin       unit      fortran unit to print list to
Cin       pagmod    1 for terminal output, 2 for printer
C$COMMON BLOCKS
Cout      rcrd08    record for communication with scenlst
C$CALLER snpick
C$METHOD
C     Rewind the cursor to ensure printing of whole list.
C     Retrieve tuples from relation and place them in the line
C     buffer; then send it down to the page printer.
C     Retrieve description tuples from scendsc for each scenario
C     and send them down.
C$LOCAL VARIABLES
C         header    heading to print at top of list
C$$
```

```
C     SNLOAD ************************************************
$CONTROL segment=scen
      SUBROUTINE snload(scen,recnum,notfnd)
C*                      *** FORMAL PARAMETER DECLARATIONS ***
%include senprm
      integer recnum
      logical notfnd
      character*12 scen
C*                                              *** ABSTRACT ***
C#PURPOSE   Finds record in relsnl file for scenario scen and loads
C     the snrlsn extra data segment with its contents.
C#AUDIT HISTORY
C         MSCarey         04-sep-83  AUTHOR
C#FORMAL PARAMETERS
Cin       scen      name of scenario to search for
Cout      recnum    record number of found record (undef if notfnd)
Cout      notfnd    true if search unsuccessful
C#COMMON BLOCKS
Cin       senprm    scenario system parameters
C#CALLER various sn routines
C#METHOD
C     Assume the file is open on snrsun; read sequentially.
C#LOCAL VARIABLES
C         target    working buffer for transfer of relsnl record
C                   to extra data segment
C##
```

```
C     SNMAKE ************************************************
$CONTROL segment=scen
      SUBROUTINE snmake
C*                      *** FORMAL PARAMETER DECLARATIONS ***
C*                                                *** ABSTRACT ***
C#PURPOSE   Supervises creation of a new scenario.
C#AUDIT HISTORY
C         MSCarey         19-aug-83  AUTHOR
C#FORMAL PARAMETERS
C         none
C#COMMON BLOCKS
Cin       senprm    scenario system parameters
Cin       ioc       global io units
Cio       uzrprv    user privelege status
Cio       snrrel    current scenario field keys for each relation
C#CALLER snstrt,rnproc
C#METHOD
C     Get the name, access, and description for the new
C     scenario, displaying current scenarios if desired.
C
C     Get the name of the scenario which the user wants to take
C     his data from for each group of relations and for system
C     parameters, and the usage-only vs. write mode setting
C     Set up the snrlsn tuple while doing this and save it for
C     writing out on successful completion.
C
C     Produce the new scenario by loading the proper snrlsn record
C     for each relation group for which copying is required, and
C     looping over the relations in the group to perform the copy.
C
C     On successful completion, write out the description tuples.
C
C     Perform consistency checking across the relation groups.
C#LOCAL VARIABLES
C         quit      true if user wants to abort scenario creation
C         name      name of new scenario
C         cpygrp    true if relset(i) should have a data copy done
C         grpsrc    name of scenario copy to be done from for relset(i)
C##
```

```
C     SNMDFY ************************************************
$CONTROL segment=scen
      SUBROUTINE snmdfy
C*                                                *** ABSTRACT ***
C#PURPOSE . Change the composition of a scenario.
C#AUDIT HISTORY
C        MSCarey        05-sep-83  AUTHOR
C#FORMAL PARAMETERS
C        none
C#COMMON BLOCKS
Cin      uzrprv   does user have priveleges
Cin      scenar   current scenario info
Cin      snrref   scenario key cross reference data structure
Cio      ____08   record buffer for scenlst retrievals
C#CALLER menu system
C#METHOD
C     Present a menu, allowing quit or spec of name of scenario or
C     request for list of current scenarios or refresh.
C     Check validity of name given.
C     Present a menu, allowing display by family or reln, mod by
C     group or reln, pop, or refresh.
C     Call appropriate routines.
C#LOCAL VARIABLES
C        sname    name of scenario to make change to
C##
```

```
C     SNMKUP ************************************************
$CONTROL segment=scen
      SUBROUTINE snmkup(name,error,slcurs)
C*                        *** FORMAL PARAMETER DECLARATIONS ***
      character*72 name
      integer slcurs
      logical error
C*                                                *** ABSTRACT ***
C*PURPOSE   SceNario MaKeUP.  Lists current contributors to
C     the given scenario at either the group level or relation
C     level.
C*AUDIT HISTORY
C        MSCarey         04-aug-83  AUTHOR
C*FORMAL PARAMETERS
Cin      name      name of scenario to list composition of.
Cout     error     true if scenario requested doesn't exist.
Cin      slcurs    cursor index for scenlst.db
C*COMMON BLOCKS
Cin      snrref    data segment access
Clocal   rcrd08    scenlst record buffer
C*CALLER various scenario
C*METHOD
C     Open the filinfo relation sorted on relset.
C     Give the user a choice between high-level or low-level list
C     Call the appropriate routines.
C*LOCAL VARIABLES
C        detail    true if user want structure by relation
C**
```

```
C     SNPICK ************************************************
$CONTROL segment=scen
      SUBROUTINE snpick
C*                      *** FORMAL PARAMETER DECLARATIONS ***
C*                                                *** ABSTRACT ***
C*PURPOSE   SceNario PICK.  Choose what scenario to work with.
C*AUDIT HISTORY
C        MSCarey         15-aug-83  AUTHOR
C*FORMAL PARAMETERS
C        none
C*COMMON BLOCKS
Cio      scenar   current scenario information
Cin      uzrprv   inspect user privs
Cio      snrref   set up extra data segment
Clocal   rcrd08   scenlst record buffer
C*CALLER runprocs or scenar
C*METHOD
C     Print a menu giving the command options here, which
C     are to choose a scenario, get help, list current scenarios
C     create a scenario, or quit and do nothing
C     On receiving a scenario name, look for it in the scenario
C     directory relation scenlst.db.  Then check to see if the
C     user may use the given scenario.  Set his priveleges, and
C     set up the snrlsn array of scenario keys by relation.
C
C*LOCAL VARIABLES
C        recnum   relsnl record number
C**
```

```
C     SNPRNT ************************************************
$CONTROL segment=scen
      SUBROUTINE snprnt
C*                      *** FORMAL PARAMETER DECLARATIONS ***
C*                                                *** ABSTRACT ***
C$PURPOSE   Prints list of current scenarios on printer.
C$AUDIT HISTORY
C         MSCarey         05-sep-83  AUTHOR
C$FORMAL PARAMETERS
C         none
C$COMMON BLOCKS
Cin       ioc       global io units
C$CALLER Menu system.
C$METHOD
C    Opens cursor on scenlst, get tgt device, and call snlist utility.
C$LOCAL VARIABLES
C         unit      logical unit to print to
C         paged     true if user wishes to be prompted for each page
C                   feed (appropriate for terminal print only)
C$$
```

```
C     SNQUIT ***********************************************
$CONTROL segment=scen
      SUBROUTINE snquit
C*                      *** FORMAL PARAMETER DECLARATIONS ***
C*                                                *** ABSTRACT ***
C$PURPOSE   Notes in snusers that caller is leaving alias and
C     is therefore finished with scen.
C$AUDIT HISTORY
C         MSCarey         04-aug-83  AUTHOR
C$FORMAL PARAMETERS
C$COMMON BLOCKS
Cin       uzrprv    user name
C$CALLER finmnu
C$METHOD
C     Open the snusers.mnurel file, find the record for this user,
C     and mark user as gone.
C$$
```

```
C     SNSEEG ************************************************
$CONTROL segment=scen
      SUBROUTINE snseeg(scen,set,fscurs,unit,all)
C*                        *** FORMAL PARAMETER DECLARATIONS ***
      logical all
      integer unit,fscurs
      character*12 scen,set
C*                                                *** ABSTRACT ***
C$PURPOSE   SceNario SEE Scenario makeup.  Lists names of
C     contributing scenarios for given scenario by relation.
C$AUDIT HISTORY
C        MSCarey        04-aug-83  AUTHOR
C$FORMAL PARAMETERS
Cin      scen     name of scenario to be listed
Cin      set      name of relset to list compositon for:
C                 undefined if all is true
Cin      fscurs   cursor open on filstat.db, index relset!file
Cin      unit     unit number to list to
Cin      all      true if listing for all groups is desired
C$COMMON BLOCKS
Cin      senprm   scenario parameters
Cin      snrref   data segment access
C$CALLER various scenario
C$METHOD
C     Assume that relsnl is open.
C     Get the snrlsn record for scen.
C     Print the header for the listing.
C     Point to a record in filstat
C     for a relset, and retrieve its relation name.
C     Find this relation in snrlnm, and get its scenario field
C     value for scen.  Use this as the source for the group.
C     Print out the group header line.
C     Then retrieve records until the group name changes, printing
C     them out in pairs.
C$LOCAL VARIABLES
C        sflist   relsnl record buffer
C$$
```

```
C     SNSEES ************************************************
$CONTROL segment=scen
      SUBROUTINE snsees(scen,fscurs,unit)
C*                      *** FORMAL PARAMETER DECLARATIONS ***
      integer unit,fscurs
      character*12 scen
C*                                              *** ABSTRACT ***
C$PURPOSE   SceNario SEE Scenario makeup.  Lists names of
C     contributing scenarios for given scenario by relset.
C$AUDIT HISTORY
C        MSCarey         04-aug-83  AUTHOR
C$FORMAL PARAMETERS
Cin      scen     name of scenario to be listed
Cin      fscurs   cursor open on filstat.db, index relset
Cin      unit     unit number to list to
C$COMMON BLOCKS
Cin      senprm   scenario parameters
C$CALLER various scenario system
C$METHOD
C     Assume that relsnl is open.
C     Get the snrlsn record for scen.
C     Print the header for the listing.
C     Point to a record in filstat
C     for each relset, and retrieve its relation name.
C     Find this relation in snrlnm, and get its scenario field
C     value for scen.  Use this as the source for the group.
C     Print out a line of info.
C$LOCAL VARIABLES
C        sflist   relsnl record buffer
C**
```

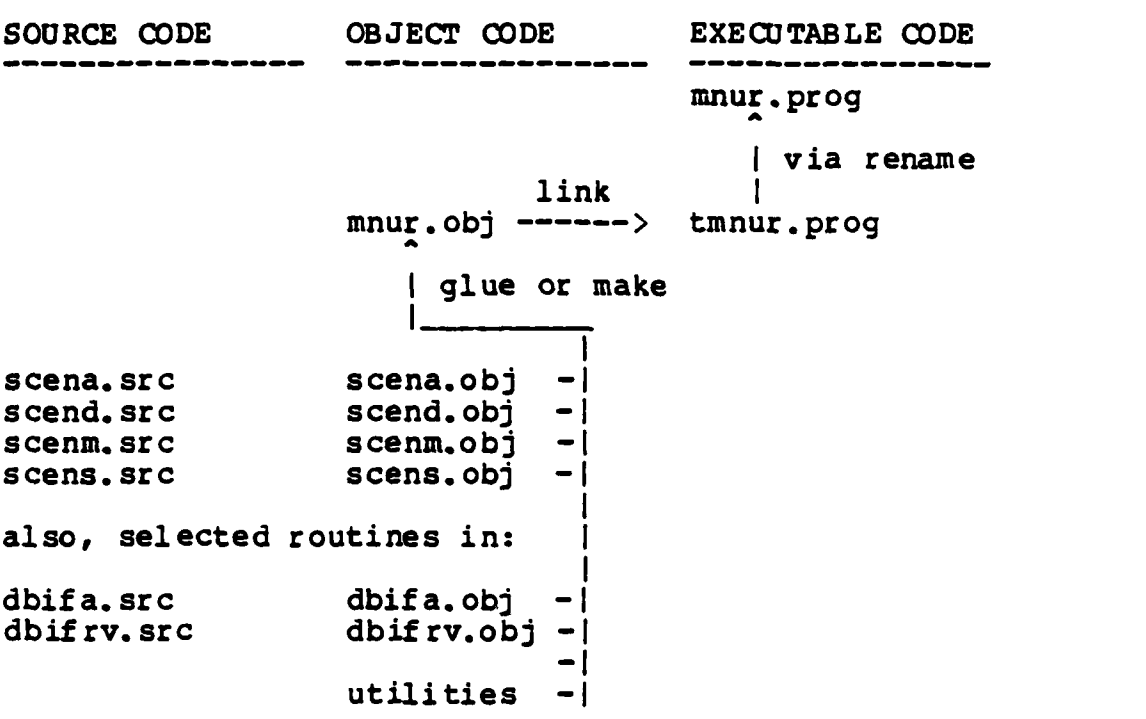
```
C     SNSHOW ************************************************
$CONTROL segment=scen
      SUBROUTINE snshow
C*                                                *** ABSTRACT ***
C$PURPOSE   Prints list of current scenarios to terminal.
C$AUDIT HISTORY
C         MSCarey         05-sep-83  AUTHOR
C$FORMAL PARAMETERS
C         none
C$COMMON BLOCKS
Cin       ioc       global io units
C$CALLER menu system
C$METHOD
C     Call snlist.
C$LOCAL VARIABLES
C         line      dummy user input buffer
C$$
```

```
C     SNSTRT ************************************************
$CONTROL segment=scen
      SUBROUTINE snstrt
C*                      *** FORMAL PARAMETER DECLARATIONS ***
C*                                                *** ABSTRACT ***
C$PURPOSE   SceNario STaRT.  Called by inimnu to force user
C     to choose a scenario to work with initially.
C$AUDIT HISTORY
C         MSCarey         12-aug-83  AUTHOR
C$FORMAL PARAMETERS
C         none
C$COMMON BLOCKS
C         ioc       global io units
C         senprm    scenario system paramters
C         snnref    scenario global reference info
C         scenar    current scenario info
C$CALLER inimnu
C$METHOD
C     Initialize actsen to "".  Call pick-scenario routine.
C     If this returns with actsen "", then user did not pick
C     one.  Ask him if he want to create a new one; if so,
C     call snmake.  Otherwise stop execution.
C     Also fills array of system relations.
C$LOCAL VARIABLES
C         iset      equiv to list of relation sets for RELATE io
C         relnum    relsnl record element for this relation
C$$
```

```
C     SNUNSD************************************************
$CONTROL segment=scen
      LOGICAL FUNCTION snunsd(caller,scen,useit,user)
C*                      *** FORMAL PARAMETER DECLARATIONS ***
      character scen*12,user*8,,caller*8
      logical useit
C*                                                *** ABSTRACT ***
C$PURPOSE   SceNario UNuSeD?  Makes sure scenario not already in
C     use by someone.
C$AUDIT HISTORY
C        MSCarey        04-sep-83  AUTHOR
C$FORMAL PARAMETERS
Cout     snunsd   true if scenario not in use at this point
Cin      scen     name of scenario to be checked
Cin      caller   name of user calling routine
Cin      useit    true if user wants to use the scenario
Cout     user     name of user already using scenario
C$COMMON BLOCKS
Cin      const    parametric constants
Cin      prmcrs   cursor open on snusers.sysrw
C$CALLER snpick
C$METHOD
C     Try to add a record stating that this user is using the
C     desired scenario to snusers.sysrw (open on snucrs).
C     Use rtpnew.  If it returns mode of 1, a unary violation
C     occurred, meaning scenario in use.  Point to in use
C     record to get user name.
C$LOCAL VARIABLES
C$$
```

TABLE 8-10

SCENARIO SYSTEM CODE FILES



CODE MANIPULATION PROCEDURE SUPPORT FILES

OBJECT-CODE MERGE (for GLUE)	BATCH LINK (for LINK)	BATCH MERGE AND LINK (for MAKE)
mnur.merge	mnur.link	tmnur.link

8.4 THE DATA BASE UPDATING SYSTEM

The Data Base Updating system (DBU) provides a general-purpose window on the ALIAS data base (DB). It was created to provide a controlled means of allowing users to enter, inspect, and change data. Interactive RELATE and MENU could have been used for data entry and updating, but such an approach would not have protected the integrity of the data base.

The DBU presents data entry screens sufficient to allow inspection and updating of the contents of every supported data base relation. It performs three kinds of data verification when the user requests an add, modify, update, or deletion in one of these screens: field range and type checking, legal value validation, and join/uniqueness verification.

"Range and type checking" refers to the standard verification provided by most updating systems. The DBU insists that numbers be entered in numeric fields, dates in proper format in date fields, etc. If a screen designer chooses, further validation can be performed (e.g., ensuring that the value of a numeric field is between 1 and 100).

"Legal value" fields are alphanumeric fields for which a short list of valid values can be defined, with the contents of the list managed by the Data Base Adminstrator (DBA). An example of such a field is a "TIME UNITS" field, which might have a legal values list of "DAYS", "WEEKS", "MONTHS", "QUARTERS", "YEARS". Forcing users to choose from a list of conventional values promotes standardization and system reliability.

"Join/uniqueness" verification aims to ensure that the contents of the data base as a whole are complete and internally consistent. The principal weakness of relational data bases is the independence of the data in each relation from that in other relations. This allows for great flexibility, but it also

demands that users be sophisticated and alert to the fact that an update to one relation may require updates to others as well.

The join/uniqueness verifications perform such tasks as ensuring that data describing a given yard is added to a scenario only once, and that no ship construction jobs for ships of a given class are entered before data describing the class is. The aim of the verifications is to ensure that at the completion of an update in any screen, the data base will be in a consistent state. A "consistent state" is one in which no ALIAS processor will find that data which it must have is missing (e.g. class level data about a ship of a class); it does not mean that all the data the user thinks should be there must in fact be there (he might forget to enter all the schedules for a program, for example---there is no way to detect that).

The join verifications effectively allow the construction and enforcement of hierarchical and network relationships among ALIAS data base relations while still permitting the fundamental structuring and management of the data base to be done using relational technology.

In addition to the basic requirements of fill-in-the-blank screens and integrity checking, the DBU also supports entry of extensive comments concerning the data that is entered, maintenance of update "track records" in appropriate relations, obtainment of printed outputs, enforcement of scenario security, and is in general as user-friendly as possible.

Most data relations have an associated comment relation in which text can be stored; the screens which serve the data relations have associated comment-entry screens which can easily be entered and exited.

Many data relations are designed to hold several instances of the same data item, distinguished from one another by data

date and entry date. This allows projections for things like construction schedules to be continually revised without losing the old projections, which can sometimes be of use. The "Update" facility of the DBU implements this capability.

This Section (8.4) is a particularly difficult one because the DBU's operations are complex and manyfold, stretching the capabilities of BUILDER and the HP 3000 to their limits. It is likely that the Section will have to be read more than once for full understanding. On the first reading it may be convenient to skip ahead and read Section 8.4.3.2, which describes the DBU's file management utility subsystem, as this motivates a significant part of the DBU's data structure and code design.

8.4.1 Structure of the DBU and Overall Flow of Execution

The DBU is implemented using the BUILDER screen application generator, which is part of the RELATE family of software, and a number of BUILDER-callable FORTRAN routines. This discussion will presume a moderate familiarity with BUILDER, but will attempt to illuminate important features which are not well described in CRI's BUILDER documentation.

8.4.1.1 The Processing Unit Structure

The basic structure of the DBU is pictured in Figure 8-20. DBU process creation and activation is supervised by the System Core, as with other ALIAS modules. The first request for the DBU causes a file equation of "FILE BLDRAPP=DBU.SCREENS" to be given and a BUILDER son process to created (which will open "BLDRAPP" as its application file). The DBU is unusual in that a "Quit" request from the user does not cause the DBU process to be terminated. Instead, DBU execution is suspended and the Core process is activated. Subsequent requests for the DBU will just cause a reactivation of the suspended DBU process, which is lightning fast in comparison with the initialization required when the process is created.

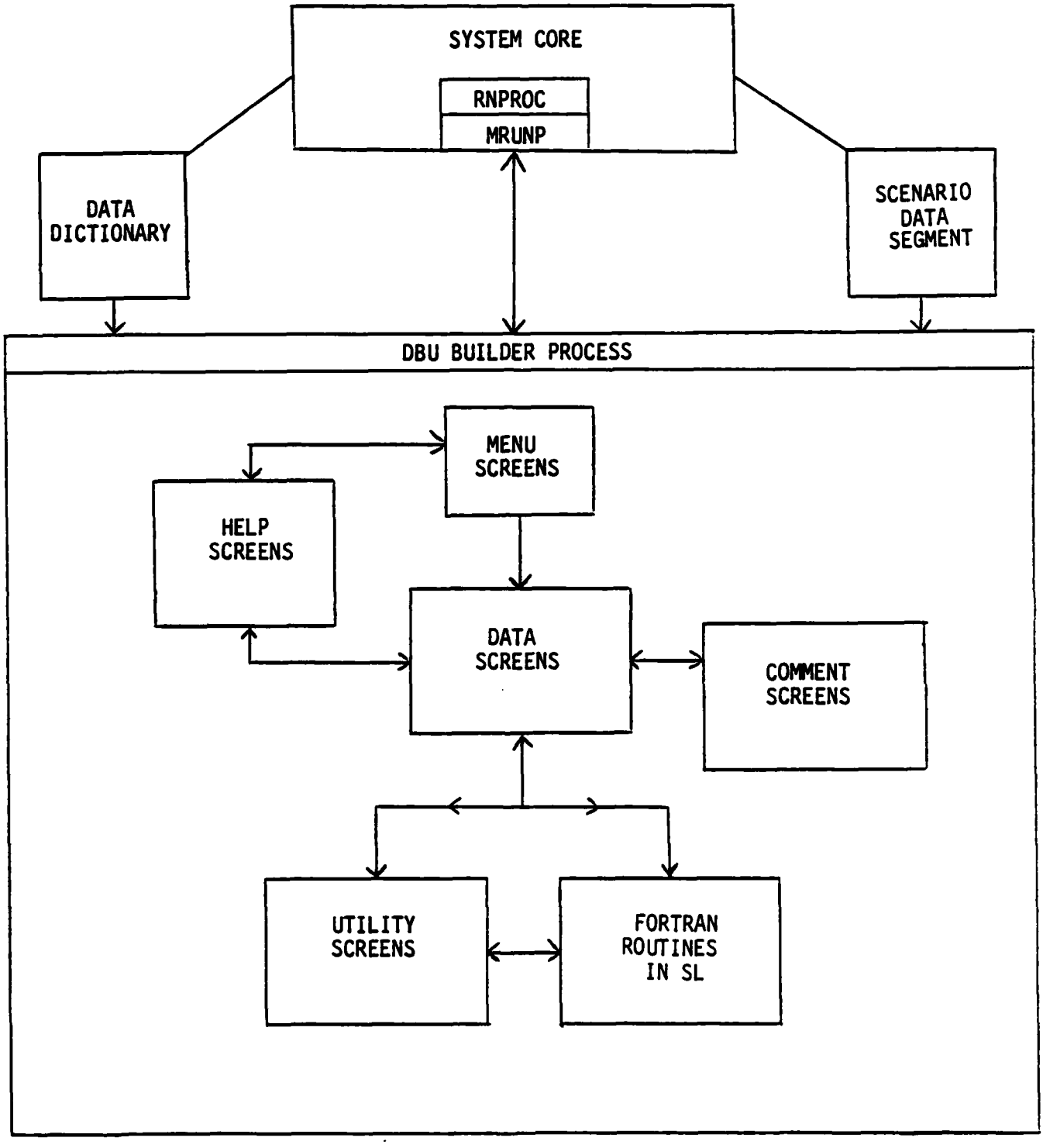


Figure 8-20. Basic Structure of the DBU

To the user the DBU appears to be composed of four kinds of screen, each type having a standardized format. There are menu screens whose function is to guide the novice or occasional user to the proper data screen; data screens which allow inspection and updating of the tuples in ALIAS data base relations; comment screens which are adjuncts of the data screens; and help screens which the confused user can consult to find out what to do.

The four major types of screens are supported by a number of utility or "subroutine" screens, most of which do only processing (i.e. they do not present a screen "layout"). These perform such functions as file management and command parsing. The utilities are in turn supported by a set of FORTRAN subroutines which can be executed by the BUILDER's CALL PROCEDURE command. The subroutines perform functions which are difficult or impossible to do with BUILDER facilities alone.

8.4.1.2 The Structure in Terms of Subsystems

Another useful way of looking at the DBU's structure is in terms of subsystems according to function. This section will list the functions the DBU performs and the subsystems that perform them.

1) DATA ENTRY/INSPECTION/UPDATE. The data screens are the DBU's primary user interface and the vehicle for its primary function.

2) DATA COMMENTING. The comment screens provide annotation capabilities that help users keep track of what lies behind the data in data relation tuples, or of facts which do not fit well into the established data structure.

3) HELP SUBSYSTEM. The help screens provide information to the user about every DBU feature and at every level. Each data screen has a companion help screen which explains its purpose and operation; there are screens which explain system commands, which cover DBU structure and overall operations, and a utility screen which provides help about any field found on a data or comment screen.

4) SCREEN SELECTION. A variety of means to choose the next screen to display are available to users. First, users may "follow" a hierarchical chain of menu screen options to arrive at any data screen. Menu screens present no data, only numbered options which attempt to describe ever more specifically the type of data the user will work with if he follows that "branch".

Second, a user may follow a predetermined "path forward" from some screens. If there is a logical next data screen from any given data screen, the "=" command will cause it to be displayed. Also, the user may always retrace a path back through the DBU screens he has already displayed, up to a limit of the last 8 previous screens (with the "^" command).

Third, experienced users who know the name of the screen they want to use (data or menu) may "jump" directly to that screen by naming it with an "=screen_name" command. The same concept works for help screens, but a "?help_screen_name" syntax is required.

Finally, special-purpose selection commands perform actions dependent on their context. The "C" command when given in a data screen brings up the comment screen associated with it; the "?" command with no screen name appended brings up the help screen the application designer thought most appropriate for the point the user is at; and the "/" command pops to the top of the current subsystem: to the MASTER menu screen from data and menu screens, to the last-accessed data or menu screen from a help screen, and to the companion data screen if given in a comment screen.

5) COMMAND PROCESSING. The BUILDER provides four means for obtaining data from the user: fill-in-the-blank data fields, action fields, prompts, and "ESCAPE" functions. The DBU uses all four, but mainly action fields and functions for obtaining user direction on what the DBU should do next. The convention is that action commands are used to request actions which can be thought of as affecting the whole screen, while function commands are used for actions relevant to <u>one field only</u>. In both cases single-character codes are used to indicate the command, occasionally with an argument appended (such as a screen name).

In most BUILDER applications, placement of a command code in an action field and pressing the return key causes BUILDER to execute a corresponding action section in the screen being displayed; giving a function code after pressing the escape key causes execution of a function section. Because of the large number of options available to the user in any DBU screen, this

strategy could result in each screen having many hundreds or even thousands of lines of code, in turn resulting in completely unmanageable application source code files and in a system whose conventions could not be changed in even a minor way.

To avoid this problem the command processing function was subroutinized to the maximum extent possible. Although function sections could not be removed completely, most function sections in DBU screens are one-liners which call a utility screen to perform the (usually standard) task being requested. Most screens have no action sections, but only the default "ENTER" section, which in turn calls a utility screen that acts as a command processor. These utilities parse the text in the "COMMAND" field on the screen and perform the task requested. There are six such command processing utilities, each appropriate for a different screen type or a different context.

This decision has resulted in a slower DBU, since command parsing is done interpretively, but a much more maintainable one.

6) DATA DICTIONARY. The DBU makes heavy use of the data dictionary for file management, data validation, and help presentation purposes. Nearly all data dictionary files are opened during the initialization phase of DBU operations.

7) FILE MANAGEMENT. HP RELATE is very slow at opening files; further, it can only have about 25 relations or 3 SELECTs open at once, or else the RELATE process runs out of memory. The DBU needs to have a great many files open at once just to get started (the data dictionary and the legal values reference library relations). Further, it is desirable to never need to close a data relation once it is opened, so that the open overhead can be avoided if the file is needed later.

These conditions would have made the DBU in its present form impossible to implement with only one RELATE son process serving the DBU's BUILDER process. However, it has been possible to have the DBU use multiple RELATE son processes through a series of "tricks".

A series of screens, routines, and file handling conventions were developed to implement this capability. Conventions which must be observed for file management to work are:

a) All data dictionary and legal values files are opened during initialization. Screen designers who wish to use the dictionary files directly should familiarize themselves with the partitions the files are open on.

b) All data relations are to be "opened" by a call to the GETFIL subroutine screen. The designer must put the name of the file in the USEFILE and USEGROUP variables before the call. The operations of this screen and its subsidiaries will be covered in a later section.

c) NEVER close any file, unless for some reason it was opened using the usual OPEN FILE syntax (perhaps on the REPORT partition for a report generator). In such cases ALWAYS close the file when finished with it.

8) LEGAL VALUES MANAGEMENT. As mentioned above, some data screen fields are designated as "legals" fields, i.e. only values from a (usually) short list will be permitted there. A set of utility screens verifies that the user has placed a legal value in such fields when a record add, update, or modify command is given, and also implements the + and - functions which let the user flip through the list of legal values for each such field.

9) DATA VALIDATION. Before any data base tuple is added or modified, subroutine screens are called which verify the uniqueness of the tuple's key field values and/or which check the data base for companion data which must be present. Likewise, tuple deletions are preceded by checks for subsidiary data which must be deleted if the given tuple is.

10) SECURITY. A utility screen can be called to verify the user's access to a screen and to obtain the user's access and modification priveleges for the underlying relation(s).

11) REPORT GENERATION. The P command, when given in any data screen, results in a call to a report-generation utility screen. This utility consults the data dictionary for a list of reports which may be executed from the given screen, presents the list, and runs the report of the user's choice (which may be either a RELATE EXECUTE file or a BUILDER procedure file).

12) DISPLAY/TUPLE RETRIEVAL MODE CAPABILITY. Most ALIAS data base relations keep "track records" of data superceded by updates; the latest data can be distinguished by its datadate field value. In a similar vein, schedule relations often hold both planned and actual schedules. The DBU's tuple retrieval facility (embodied in the "N" command) can be instructed to return all or only the latest data, and only actual, only planned, or all schedules, via the "L" command.

13) MISCELLANEOUS USER SERVICES. The DBU makes it possible for users to easily execute processors like text editors and MPE commands without returning to the System Core. Most such functions are implemented as utility screens called by a DBU command processing utility.

14) MISCELLANEOUS UTILITY SERVICES PROVIDED TO SCREENS. It made sense to utility-ize a number of functions required by many screens. These include output device control (consulting of the printer setting on the Command System's User Environment Parameter menu and giving the proper file equations), text paging for comment screens, and a tuple search facility for data screens.

8.4.1.3 Special Features and Conventions Used In the Implementation

8.4.1.3.1 Flow of Control Among Screens

BUILDER offers two methods of displaying a new screen, CALL SCREEN-RETURN SCREEN and SET SCREEN. Calls save the data structure for the current screen in its entirety (and make variables in this structure accessible to the called screen), while sets flush the current screen and replace it with the new one. Calls are nice because they do save the data structure, but they use more memory and force the developer to restrict the user's choice of screens at any time to those nearby on an hierarchical path. If too many screens get on the "stack" an overflow can result; if the user chooses screens in an order which brings him back to one that he never RETURNed from initialization problems might result (though screens may be called recursively if care is taken in constructing their data structure).

Sets are nice because they tend to keep the amount of memory used down, and because they allow any screen to be chosen at any time.

The DBU takes advantage of both of the facilities. Menu screens and data screens are ALWAYS reached by sets, while help, comment, and utility screens are always reached by calls and left by returns. This uses each facility to greatest advantage: when a user is through with a data or menu screen he typically no

longer needs its data structure; also, these are the screens which the user most needs to be able to jump between freely. On the other hand, help and comment screens are for reference---the user will usually want to return to his data or menu screen with its data structure intact. Utility screens always rely on the calling screen's data structure being accessible, making CALL the required means of executing them.

This is why different commands are used to enter help and comment screens than are used for menu and data screens. The = command leads to a set-oriented logic, while ? and C lead to a call-oriented one. Likewise, the only exit from help and comment screens is via a return, while menu and data screens are always left via a set.

The interested developer will note that the user does have freedom to jump among screens at will in the help system using the "?screen_name" facility. However, exit from the screens using the "^" will always lead back up through the screens called. The "/", which pops to the original primary screen (menu, data, comment), is implemented via checks during help initialization which cause an immediate return if the last command given was "/".

8.4.1.3.2 How the BUILDER Stores Variables

One of the most useful features offered by BUILDER is the ability to declare and manipulate variables with a variety of data types. It is often useful during debugging (and mandatory for FORTRAN procedure writing) to know how these are stored and retrieved. When the BUILDER processes layout and declaration sections, any variables found which are NOT declared as global are added to a linked list of all current BUILDER variables (which includes the built-in variables (e.g. $FOUND)). The linked list consists of a series of 19 word cells which contain pointers to the memory locations at which variable names and values are stored, and various other information. When a

variable is referenced in an application, BUILDER just runs through this linked list (starting at the most-recently-defined end) until it finds it. BUILDER permits more than one variable of the same name to be on the list, and (given the search algorithm) always uses the most recently defined one. Defining a variable as global therefore just inhibits the addition of another variable of the same name to the list.

Since BUILDER always looks through the whole list, variables which were defined on a screen "above" the current one (i.e. from which a call was made and not yet returned) are always accessible even if they have not been defined in the current screen. The sole purpose of the "GLOBAL" declaration keyword is to indicate that variables on the layout (which are local by default) have been defined in a previous screen.

The "GLOBAL" keyword does NOT cause a variable to remain on the linked list after the screen which defined it has been flushed by a return or set. The ONLY way to declare variables global for an application is to adopt an initialization strategy similar to that used in the DBU: the first screen in the application does nothing but declare variables, then calls the real initialization screen; this then is left by either a call or set, but the data-declaration screen is never returned to. Its variables are thus always on the linked list.

Note that ALL BUILDER variables are stored in ascii form, regardless of whether they are declared as numeric. When computations must be performed BUILDER does the necessary conversions. This is important to know when attempting to read or write to BUILDER's variable memory area from a FORTRAN routine.

8.4.1.3.3 CALL PROCEDURE

The DBU makes heavy use of BUILDER's subroutine-calling utility, mainly to accomplish tasks which require calls to system intrinsics. CALL PROCEDURE allows calling of FORTRAN (and other

language) routines whose object code is stored in the account Segmented Library (in file SL.PUB). Segmented libraries are the source for program unsatisfied external references at RUN time, and as such may contain only completely formatted object code segments. FORTRAN routines in these segments may not do FORTRAN i/o via read and write, and may not include COMMON or DATA statements. These limitations are unfortunate but surmountable through use of extra data segments for global storage and the fread/fwrite intrinsics for i/o.

The HP 3000 offers programmers a capability to dynamically link routines in segments in an SL, i.e. to decide AFTER run time that a given routine will be needed and to make it accessible. This capability underlies BUILDER's ability to call a FORTRAN routine it never heard of until the given application line was interpreted.

As noted in the BUILDER documentation, any FORTRAN routine called in this manner must be compiled with three formal parameters, which will contain the current builder partition (known as a cursor in the RELATE manual), which is a 50-word integer array, an array of pointers to such things as the first cell on the BUILDER variable linked list, and another array of pointers to the text of the CALL PROCEDURE line. These arguments make it possible to extract information from the executing application by looking at the contents of the CALL PROCEDURE line, or by reading the BUILDER's variable memory area directly. Data may also be written into this memory area.

Table 8-11 contains an annotated list of the routines now available in the account SL (source code in the slproc.src and sldate.src files). Interested programmers should consult Section 10.3 for a full discussion of the DBU utility routines now resident in the account SL, and the remainder of this Section for discussion of how each of the routines serves the various DBU subsystems.

TABLE 8-11

FORTRAN Routines Called By the DBU

ROUTINE	PURPOSE
ABTRNS BGTRNS DOTRNS	These three routines abort, begin, and commit RELATE transactions on ALL RELATE PROCESSES serving the DBU. The job must be done by these routines since a simple BEGIN TRANSACTION command given in the DBU will be effective only in the process serving the partition it was given on.
CALCDATE	Implements the schedule milestones recalculation facility invoked by ESC-R in the PROJ_NC_SKED screen.
CURINI CURSWP	These routines are the guts of the DBU file system. CURINI sets up the extra data segment in which partitions are stored. CURSWP inspects the numswap, cursornum, and cursorproc Job Control Word values and swaps the contents of the partition passed to it to the proper data segment location, then swapping the requested partition from the segment into the passed array, thus making it usable by BUILDER.
CDTODD DATEMK DCLRFY DDTOCD DDTOID IDTODD LMONTH MODCOR MRKDAY NWDATU NWIDAT	Date manipulation utilities identical in operation to the ones with the same names in UTLR.
DLTRIM DRTRIM	Same as routines ltrim and rtrim in the UTLR library. Integer functions which return the leftmost and rightmost non-blank character in a string.
DRUNED DRUNTDP	Now-obsolete routines which create and activate an EDITOR and a TDP son process, respectively.
DSAFETCH DSAGETC DSAPUT GETARR GETVAR PUTARR	Utilities for reading from and writing to the BUILDER memory map. Dsafetch places a word-aligned character string into an integer array, given a word address. Dsagetc performs a similar function using a byte address. Putvar and getvar will replace and retrieve the contents of a named BUILDER variable. Dsaput is

TABLE 8-11

FORTRAN Routines Called By the DBU

ROUTINE	PURPOSE
PUTVAR	a utility which writes to the address of the variable once it is found. Putarr and getarr are similar, but work with array variables. These routines make it easy to write FORTRAN procedures which work with the contents of BUILDER variables.
GETSCENV	A scenario system utility which takes a relation name as an argument on its CALL PROCEDURE line, searches the scenario system extra data segment for the proper scenario key field value to use in querying that relation when running the current scenario, and places this value into a variable also named on the CALL PROCEDURE line.
SPSUSP	Suspends the current BUILDER process and activates the father.

8.4.1.3.4 Partitions, Cursors, and RDBINITX

BUILDER "partitions" are exactly the same as RELATE Host Language Interface (HLI) "cursors", which in turn can be thought of in many cases as separate invocations of RELATE. This discussion will use the term partition exclusively in order to avoid confusion with the cursor on the display screen.

As implemented, a partition is a 50-word integer array supplied as an argument in any HLI call which tells RELATE information about what it is to work with. It contains data such as the process id of the RELATE son process being used, and the id of the data segment used in communicating data to and from that process. Programs will often use more than one partition, and BUILDER permits use of a number limited only by available memory. Multiple partitions allow multiple files to be searched simultaneously with no repetitive SET PATH or SELECT commands required.

New partitions are initialized by a call to the rdbinit HLI routine, which always uses the same RELATE son process. A currently undocumented variant of this routine, called rdbinitx, accepts additional arguments which allow the caller to specify process and data segment id's to be used. This allows the programmer to create multiple RELATE son processes to support large data base operations. There are restrictions on what can be done with multiple processes in comparison to a single one, but they are fairly minor.

BUILDER is designed to use only a single son process, but the DBU file management subsystem has extended its capabilities

through FORTRAN routines executed via CALL PROCEDURE which in turn use rdbinitx.

8.4.2 DBU Data Structures

The DBU uses five major data structures to support its operations, not counting the data base itself. These are the data dictionary, the snusers.sysrw, sysusr.sysro, and envrn.mnurel system relations, two extra data segments, BUILDER partitions, and BUILDER global and local variables.

8.4.2.1 The Data Dictionary

Table 8-12 lists the data dictionary's relations and decribes how each is used by the DBU. The dictionary covers six main subjects: data base files and file security for the DBU, fields within those files, DBU screens and screen security, what fields are in what files, what files each screen serves, and what reports are available from each screen.

Dictionary relations that are used are opened on separate partitions in the default RELATE son process of the DBU's BUILDER process. These are then immediately available for use by DBU screen logic. The most common uses of dictionary information are:

1) Retrievals of information required by the file management system to open/retrieve a relation from its data storage segment (from filinfo.db).

2) Retrieval of the legal values reference library relation in which a field's legal value list is stored (from dbflds.db and scrflds.db).

3) Retrieval of a user's priveleges for screens and relations (from filsec.db and scrsec.db).

4) Retrieval of help information about a field from the fldesc and dbflds relations, using the field name as key.

5) Retrieval of data base structure rules (join requirements) from the filjoin relation.

TABLE 8-12

DATA DICTIONARY RELATIONS AND THEIR USAGE BY THE DBU

FILE	USAGE
DBFLDS	Contains information about any field which is present in at least one supported data base relation, e.g. data type, units of measure values are in, etc. All information is used by the field help utility screen; the contents of the LEGLFILE field are used by the legals subsystem for legals fields.
FAMDESC	Unused by DBU. Description of each data base family.
FILDESC	Unused by DBU. Textual description of the purpose of each data base relation.
FILINDX	Unused by DBU. Lists the indexes which have been created on each data base relation.
FILINFO	Primary reference for data about individual relations; one filinfo tuple covers one relation. The data dictionary is concerned with the SCRSET, SCRNUM, and SCRPROC fields. SCRSET is used primarily to pick out legal value reference library files for opening during DBU initialization. SCRNUM is used to identify the location of the relation's dedicated partition in the file management system extra data segment (value must be unique!). SCRPROC gives the id of the RELATE process that partition should be opened on. SCRNUM is irrelevant for relations not managed by the file management subsystem, e.g. legals files.
FILJOIN	Used by the VERIFYJ and VERIFYD screens to obtain effective descriptions of the required data base structure for use in join tests and deletions, part of the DBU's data validation logic. See the descriptions of the logic of those screens for more details.
FILPRIV	Has a list of data base files, the users which may access them through the DBU, and the exact read/update priveleges of each such user.
FILSCRN	For each data relation, has the name of the screen which is the primary means of updating the relation. Used to construct certain dynamic error messages (mainly by the data validation subsystem) which point the user to the right screen to use in correcting his problems.

TABLE 8-12

DATA DICTIONARY RELATIONS AND THEIR USAGE BY THE DBU

FILE	USAGE
FLDESC	Textual description of data base fields. Same description applies no matter how many files the field is in. Used by the field help utility screen.
FLDFILE	Unused by DBU. Tells which fields are in each data relation.
FLDUSER	Unused by DBU. Tells which ALIAS modules use each field in each data relation. Useful for assessing the maintenance impact of any proposed changes.
REPTEX	Lists ALIAS report generators which take the form of RELATE EXECUTE files (such files should be in the .rprocs group.
SCRFLDS	Cross reference. Maps screen variable names to associated field names in data relations. Used by the field help utilities screen and other utility which need the mapping.
SCRPRIV	Security relation which tells which users can execute each DBU screen.
SREPTS	Lists which reports can be executed from which DBU screens, the type of each (i.e. BUILDER application file or RELATE execute file, and a description for display by the report execution utility screen.

6) Retrieval of lists of reports executable from a given screen from the reptex and srepts relations.

8.4.2.2 Snusers, Sysusr, and Envrn

These relations are consulted during initialization to obtain the name of the scenario the user is now working with, his basic priveleges, and the code name for the printing output device of his choice. This information could be swapped in using a call to the iniprc utility if the DBU were a FORTRAN module, but use of BUILDER for its implementation makes it necessary to consult the storage relations for the data.

Snusers and envrn are kept open permanently, and are consulted each time the user pops back up into the Command System and then returns to the DBU, since he may have changed scenarios or output devices.

8.4.2.3 Extra Data Segments

The DBU creates and maintains one extra data segment, and references another, both via calls to FORTRAN routines in the SL.

It references the scenario system's storage segment via subroutine getscenv to discover the proper scenario key field value for a relation to be queried.

The file management subsystem requires storage for a large number of partitions (50 word integer arrays), and its requirements are such that the storage cannot be done in a BUILDER variable. Since routines in the SL may not have global storage, an extra data segment is used to hold the partitions. See Section 8.4.3.2 for more details.

8.4.2.4 Partitions

The DBU creates and uses a large number of named BUILDER partitions. One file-one partition is a DBU convention, so each partition can be thought of as the window on a particular file.

The partitions can be divided into two groups: support partitions which always have the same file open on them, and utility partitions whose open file will vary. The support partitions are for the data dictionary, while the utility partitions include those for data screen relations (MAIN, MAIN2, and MAIN3), those for join and deletion validation (CHECK1 - CHECK4), LEGALS for the legal values library relations, COMMENT for comment screens relations, and REPORT to support report generator files. See Section 8.4.3.2 for more discussion of partitions.

8.4.2.5 Global and Local Variables

BUILDER variables are the most important part of the DBU's data structure. This section will concentrate on variables which are global to the entire application; where significant, those local to a particular screen are mentioned as part of the discussion of that screen's logic.

A few of the global variables are declared as such because their values are actually to be stable and used by many or all screens (scenarionm is a good example). However, most are working storage or buffers for the utility screens or the standard data/comment/menu/help screen logics. By making them global, the necessity of declaring them again and again in each screen which requires them is avoided, improving both maintainability and efficiency.

All global DBU variables are declared in screen START, the first one in the dbusubr.screens source file. Table 8-13 lists these variables in alphabetical order, along with their characteristics and the subsystem they serve. Table 8-14 is an annotated listing by subsystem, giving the purpose and usage of each variable.

TABLE 8-13. DBU GLOBAL VARIABLES

VARIABLE	DATA TYPE MODIFIERS	SUBSYSTEM
ABORT_MODFY	NUMERIC	VALIDATION
ADD	LENGTH=1	SECURITY
ADDOK	LENGTH=1;UPPER	VALIDATION
AKSHUN	LENGTH=80	CMD PROC
ALLOK	NUMERIC	CMD PROC
ALTDB	NUMERIC	SECURITY
ALTERCAP	LENGTH=50;DISPLAY;ENHANCE=NONE	obsolete
ALTERCAPD	LENGTH=80;DISPLAY	MISC
ANSWER	LENGTH=1;UPPER	WORKING VARS
ANSWERL	LENGTH=20;UPPER	WORKING VARS
BUFFER	LENGTH=65	WORKING VARS
CALLED	LENGTH=1	comments
CANCEL	NUMERIC	VALIDATION
COMCHR	LENGTH=1	CMD PROC
COMMAND	ACTION;UPPER;LENGTH=16; JUSTIFY=LEFT	CMD PROC
COMMENT	LENGTH=65;ARRAY=24	COMMENTS
COMT_ACTIVE	LENGTH=1	COMMENTS
COMT_ALTERED	LENGTH=1	COMMENTS
COMT_BREAK	LENGTH=2	COMMENTS
COMT_FILE	LENGTH=8	COMMENTS
COMT_FLD_MODNUM	LENGTH=40	COMMENTS
COMT_FLD_MODS	LENGTH=40;ARRAY=9	COMMENTS
COMT_GROUP	LENGTH=8	COMMENTS
COMT_RESET	LENGTH=160	COMMENTS
COMT_SCREEN	LENGTH=15;DISPLAY;ENHANCE=NONE	COMMENTS
COMT_START	NUMERIC	COMMENTS
COND	LENGTH=400	WORKING VARS
CURS_SWAP_LIST	LENGTH=120	MISC
DATACMDA	LENGTH=80	CMD PROC
DATACMDB	LENGTH=80	CMD PROC
DATADATE	DATE="MM/DD/CCCC";ENHANCE=I,U	MISC
DATEMODE	LENGTH=8	MISC
DATESCREEN	NUMERIC	MISC
DATEU	DATE="MM/DD/CCCC"	VALIDATION
DATE_VARS	LENGTH=80	MISC
DELET	LENGTH=1	SECURITY
DELOK	LENGTH=1;UPPER	VERIFICATION
DESC_VARS	LENGTH=80	MISC
DOADD	NUMERIC	VERIFICATION
DOMOD	NUMERIC	VERIFICATION
DONE	NUMERIC	WORKING VARS
ENDFILE	NUMERIC	WORKING VARS
ENTRY_BY	POSITION;UPPER;ENHANCE=NONE	MISC
ENTRY_DATE	DATE="MM/DD/CCCC";POSITION; ENHANCE=U	MISC
EQUALS	LENGTH=24	MISC
ERRNUM	NUMERIC	WORKING VARS
EXTRACOM	NUMERIC	WORKING VARS
FIELDNAME	LENGTH=10	WORKING VARS
FILE	LENGTH=8	WORKING VARS

TABLE 8-13. DBU GLOBAL VARIABLES

VARIABLE	DATA TYPE MODIFIERS	SUBSYSTEM
FILNAM	LENGTH=8	WORKING VARS
FLDFMT	NUMERIC	WORKING VARS
FLDLEN	NUMERIC	WORKING VARS
FLDNAM	LENGTH=15	WORKING VARS
FLDNUM	NUMERIC	WORKING VARS
FLDTYP	NUMERIC	WORKING VARS
FLDVAL	LENGTH=80	WORKING VARS
FLIP_FIELD	LENGTH=15	obsolete
FOUND	LENGTH=1	WORKING VARS
GROUP	LENGTH=8	WORKING VARS
IFOUND	NUMERIC	WORKING VARS
INMODE	ENHANCE=NONE;DISPLAY;LENGTH=6	MISC
INUM	NUMERIC	WORKING VARS
IN_ONE	NUMERIC	VALIDATION
JOINFLDS	LENGTH=90	VERIFICATION
JOINMSG	LENGTH=80	VERIFICATION
JOINTYPE	LENGTH=12	VERIFICATION
KEYFIELDS	LENGTH=70	MISC
LASTFAIL	NUMERIC	VALIDATION
LEGAL_FIELDS	LENGTH=150	LEGALS
LEGLFILE	LENGTH=8	LEGALS
LEGLFIRST	NUMERIC	LEGALS
LEGLPREF1	LENGTH=30	LEGALS
LIST	LENGTH=400	WORKING VARS
LNUM	NUMERIC	COMMENTS
LOWKEY	LENGTH=15	MISC
LOWLEN	LENGTH=2	MISC
LPUNIT	LENGTH=12	MISC
MODE	LENGTH=6	MISC
MODEDATA	LENGTH=6;UPPER	MISC
MODEDISP	LENGTH=20;DISPLAY;ENHANCE=NONE; JUSTIFY=CENTER	MISC
MODESET	LENGTH=30;UPPER	MISC
MODFY	LENGTH=1	SECURITY
MORE	LENGTH=4;DISPLAY	COMMENTS
MPECOM	LENGTH=132	WORKING VARS
NEXT_HELP	LENGTH=80	HELP
NEXT_SCREEN	LENGTH=15	SCREEN CHOICE
NLEFT	NUMERIC	COMMENTS
NOMORE	NUMERIC	WORKING VARS
NOWFILER	LENGTH=8	REPORTS
NOWGROUPR	LENGTH=8	REPORTS
NOWLEGAL	LENGTH=15	LEGALS
NOW_COMT_FILE	LENGTH=15	COMMENTS
NOW_CURSOR	LENGTH=15	FILE MGR
NOW_FILE	LENGTH=8	FILE MGR
NOW_GROUP	LENGTH=8	FILE MGR
NOW_INDEX	LENGTH=90	FILE MGR
NOW_LEG_VAR	LENGTH=15	LEGALS
NOW_SCREEN	LENGTH=15;DISPLAY;ENHANCE=NONE	SCREEN CHOICE
NOW_USER	LENGTH=8	MISC

TABLE 8-13. DBU GLOBAL VARIABLES

VARIABLE	DATA TYPE MODIFIERS	SUBSYSTEM
NUM	NUMERIC	WORKING VARS
OK	NUMERIC	WORKING VARS
PREV_SCREEN	LENGTH=160	SCREEN CHOICE
READ	LENGTH=1	SECURITY
READB	NUMERIC	SECURITY
RECURS	NUMERIC	VERIFICATION
RESET	LENGTH=160	FILE MGR
SAVE_CURSOR	LENGTH=15	COMMENTS
SAVE_RESET	LENGTH=160	COMMENTS
SAVE_SCREEN	LENGTH=15	COMMENTS
SAVMSG	LENGTH=80	VERIFICATION
SAVTYP	LENGTH=12	VERIFICATION
SCEN1	LENGTH=12	MISC
SCEN2	LENGTH=12	MISC
SCEN3	LENGTH=12	MISC
SCENARIO	LENGTH=12;DISPLAY;ENHANCE=NONE	MISC
SCENARIONM	LENGTH=12;DISPLAY;ENHANCE=NONE	MISC
SCENCOM	LENGTH=12	COMMENTS
SCENSAVE	LENGTH=12	obsolete
SCREEN	LENGTH=16	WORKING VARS
SCREEN_NAME	LENGTH=15	WORKING VARS
SCRINDX	NUMERIC	FILE MGR
SCRNUM	NUMERIC	FILE MGR
SCRPROC	NUMERIC	FILE MGR
SCRVAR	LENGTH=16	HELP
SEEIT	LENGTH=1	SECURITY
SHOWMODE	LENGTH=8;UPPER	MISC
SHOWPREF	NUMERIC	LEGALS
SRCFIELD	LENGTH=15;UPPER	VERIFICATION
SRCFILE	LENGTH=8;UPPER	VERIFICATION
SRCGROUP	LENGTH=8;UPPER	VERIFICATION
SWAPNUM	NUMERIC	FILE MGR
TGTFIELD	LENGTH=15;UPPER	VERIFICATION
TGTFILE	LENGTH=8;UPPER	VERIFICATION
TGTGROUP	LENGTH=8;UPPER	VERIFICATION
TODAY	DATE="MM/DD/CCCC"	MISC
UOK	NUMERIC	VALIDATION
UPDAT	LENGTH=1	SECURITY
USECURSOR	LENGTH=15	FILE MGR
USEFILE	LENGTH=8	FILE MGR
USEGROUP	LENGTH=8	FILE MGR
USERLEVL	NUMERIC	SECURITY
USERNAME	LENGTH=8	MISC
VERIFYFIELDS	LENGTH=90	VERIFICATION
VERTYP	LENGTH=2;UPPER	VERIFICATION
WORKFILE	LENGTH=8	WORKING VARS
WORKGROUP	LENGTH=8	WORKING VARS
Y	LENGTH=1;UPPER	WORKING VARS
ZEES	LENGTH=40	MISC

TABLE 8-14. DBU GLOBAL VARIABLES

VARIABLE	PURPOSE
	COMMAND PROCESSOR VARIABLES
AKSHUN	In addition to performing certain tasks themselves, the command processing utility screens (e.g. proc_data_cmda) can pass back a line of BUILDER code to be executed in the calling screen. This line might hold a RETURN SCREEN or SET SCREEN screen_name command, for example. AKSHUN is where this line of code is stored.
COMCHR	The primary part of any BUILDER command is a single-character code. The first act of any command processor is to strip off the first character of the COMMAND field and place it in this variable, which is then used in IF tests to determine what the command is.
COMMAND	The field which appears on EVERY screen and in which the user places his commands.
DATACMDA	List of data screen commands which are to be processed by screen PROC_DATA_CMDA.
DATACMDB	List of data screen commands which are to be processed by screen PROC_DATA_CMDB.
EXTRACOM	Boolean set to 1 if a screen has additional data add/update code in addition to that which comes standard as part of PROC_DATA_CMDB. If 1 then PROC_DATA_CMDB does not clear the layout after a delete.
	COMMENT SUBSYSTEM VARIABLES
COMMENT	An array variable in which comment text is stored (up to 24 lines). The paging algorithm selectively displays text from this array. DB reads/write data transfers made to/from this array.
COMT_ACTIVE	Set to "Y" if the comments for the current data screen tuple have been retrieved into memory already, also implying that the proper comment relation is current on the COMMENT partition.
COMT_ALTERED	Set to "Y" if the user has altered the text of the current set of comments in any way.

TABLE 8-14. DBU GLOBAL VARIABLES

VARIABLE	PURPOSE
COMT_BREAK	Set to the ASCII version of the number of fields in the current comment relation index BEFORE the LNUM field. Allows general-purpose code to do a RECORD POINT that will set up an end-of-file after the current set of comments.
COMT_FILE	File name of the comment relation for the current data screen.
COMT_FLD_MODNUM	List of numeric codes used to indicate the element of the COMT_FLD_MODS array variable to use to modify the declaration of each non-comment field on a comment screen back to what it should be in the data screen. Each code applies to the corresponding field name on the list in the KEYFIELDS variable.
COMT_FLD_MODS	An array variable which holds an assortment of BUILDER variable declaration keyword strings, representing the most commonly used type of declared variable, for use in restoring key fields shown on a comment screen from display-only status to their normal set of declarations. See the code of the INIT screen for the value of each element of this array or Section 8.4.4.1.1
COMT_GROUP	Name of the group the comment relation for the current data screen resides in.
COMT_RESET	Contents is the guts of the SELECT statement (everything after the "SELECT" keyword) which is required to set up a properly indexed and restricted path for the comment relation for the current data screen. Similar to the RESET variable.
COMT_SCREEN	Name of the comment screen associated with the current data relation.
COMT_START	Used as part of the comment text paging algorithm; gives the element in the COMMENT array from which the FIRST line of currently displayed text is drawn (note first element of array is 0, not 1).
KEYFIELDS	Holds a list of fields on the current data screen (not including entry_date) the sum of whose values must uniquely identify a tuple. All such fields should appear on the associated comment screen; this list is used by utility

TABLE 8-14. DBU GLOBAL VARIABLES

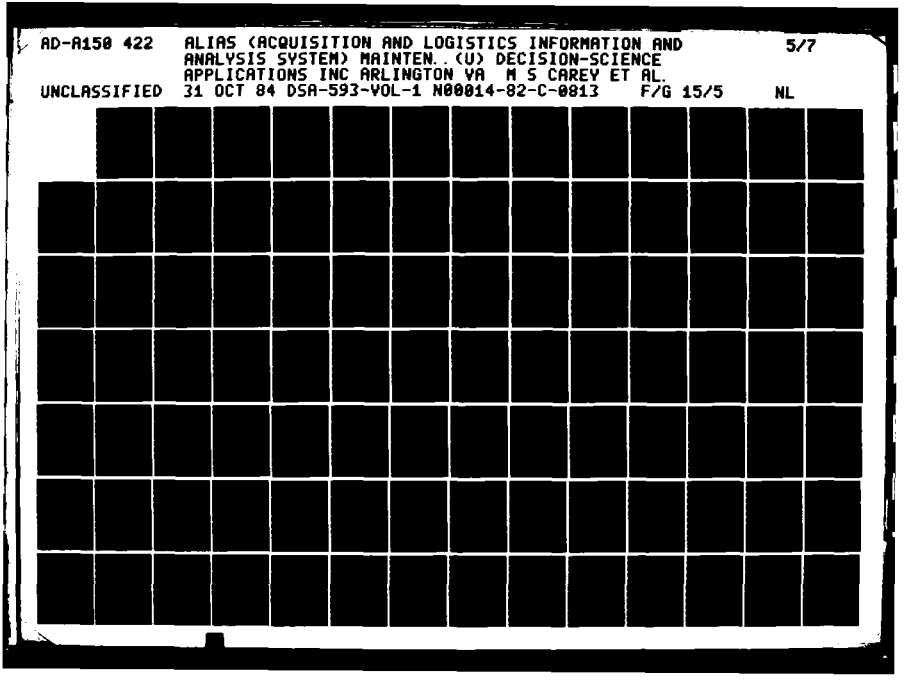
VARIABLE	PURPOSE
	code which gives MODIFY VARIABLE commands to change the variables to display-only in the comment screen and then back before exit to the data screen.
MORE	Variable appearing on the layout of every comment screen, used to indicate whether there are non-blank comment lines "below" those on the displayed page.
NLEFT	Used by the algorithm which decides whether variable MORE should be set to "more"; gives number of non-blank lines below the last one displayed.
NOW_COMT_FILE	obsolete
SAVE_CURSOR	Used to store the name of the data screen's current partition for the duration of a comment screen's operations (file utilities require "COMMENT" to be placed in the NOW_CURSOR variable.
SAVE_RESET	Used to save the current data screen's setting of the RESET variable for the duration of a comment screen's operations.
SAVE_SCREEN	Used to save the current data screen's setting of the "NOW_SCREEN" variable for the duration of a comment screen's operations.
SCENCOM	Proper scenario field key value for the current scenario for the current comment relation, as determined by a call to the GETSCENV scenario system utility.
LNUM	A field in every comment relation, used in the DBU to set the value of that field during writes to the relation.
	DISPLAY-MODE SUBSYSTEM VARIABLES
DATEMODE	Takes on the values "ACTUAL", "PLANNED", or "ALL". Used as part of the display-mode subsystem which determines which subset of all tuples to retrieve in screens which contain data which can be distinguished according to the three values may be blank if the actual/planned distinction is irrelevant in a screen.

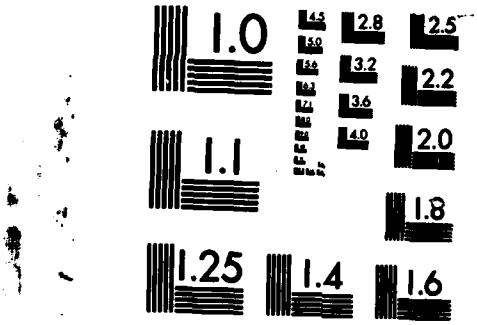
TABLE 8-14. DBU GLOBAL VARIABLES

VARIABLE	PURPOSE
DATESCREEN	Set to 1 if the current data screen is interested in the ACTUAL/PLANNED distinction.
INMODE	Used by the display-mode subsystem which determines what subset of tuples to display. Takes on values of "LATEST" or "ALL".
MODE	Obsolete.
MODEDATA	Has the value "DATA" or "DATES", depending on the nature of the current data screen. Used in constructing the MODEDISP variable's contents.
MODEDISP	Variable displayed at the top center of each data screen which gives the current display/retrieval mode subsystem setting to the user. A concatenation of the contents of the inmode, datemode, and modedata variables.
MODESET	Holds a WHERE clause to be added to the text of the SELECT statement in the RESET variable if the date retrieval mode is either "PLANNED" or "ACTUAL". The clause is of the form "AND DATETYPE=""%datemode"" ".
SHOWMODE	A "permanent" version of DATEMODE which saves the actual/planned mode setting when DATEMODE is blank.
	FILE MANAGEMENT SUBSYSTEM VARIABLES
CURS_SWAP_LIST	Contains a list of the names of the named BUILDER partitions which will be used as the current partition in GETFIL calls. ANY PARTITION WHICH IS TO WORK WITH FILES MANAGED BY THE FILE MANAGEMENT SYSTEM MUST HAVE ITS NAME ON THIS LIST.
NOW_CURSOR	Name of the "current" BUILDER partition. CAUTION: the file management subsystem assumes you have made this partition current via a USE PARTITION command before GETFIL calls.
NOW_FILE	Name of the (primary) data relation for the current data screen.
NOW_GROUP	Group the (primary) data relation is stored in.
NOW_INDEX	String version of the index which should be

TABLE 8-14. DBU GLOBAL VARIABLES

VARIABLE	PURPOSE
	used for the (primary) data relation of the current data screen. The text of this index specification will become part of the SELECT statement which the RESET variable holds.
RESET	The text of a SELECT statement (without the "SELECT" keyword) which can be used to construct a proper path for retrievals from the (primary) data relation for the current data screen. Used whenever the path must be reset after another selection has been made on it (say when the user cancels a Search condition with the B command), and to set up the path initially. The contents should set the proper index via a BY clause and include a WHERE clause which restricts records returned to those for the current scenario.
SCRINDX	Variable corresponds to the field of the same name in the filinfo.db file of the data dictionary. When a relation is opened, its record is retrieved from filinfo as part of the process, thereby returning the numeric specification for the SET INDEX command which is given after the OPEN FILE command (e.g. SET INDEX %scrindx might be interpreted as SET INDEX 1).
SCRNUM	Corresponds to the scrnum field in the filinfo.db dictionary relation. In filinfo, the field gives the location in the file management extra data segment of the partition which will be dedicated to each data relation. The location is given in terms of an indexing system, i.e. a value of 15 refers to the 15th partition in the segment; the index is converted to an address by the curswp FORTRAN routine.
SCRPROC	A field in filinfo.db like scrindx and scrnum. This field gives a numeric id for the RELATE son process which each data relation's dedicated partititon should be opened on. Values are of the form 101, 102, etc.
SWAPNUM	Indicates the number of the named BUILDER partition which is to be used in the next GETFIL call, the number being the position of the partition's name on the list in the CURS_SWAP_LIST variable. Enables the curswp routine to consult its memory of which actual


AD-A150 422 ALIAS (ACQUISITION AND LOGISTICS INFORMATION AND ANALYSIS SYSTEM) MAINTEN..(U) DECISION-SCIENCE APPLICATIONS INC ARLINGTON VA M S CAREY ET AL.
5/7
UNCLASSIFIED 31 OCT 84 DSA-593-VOL-1 N00014-82-C-0813 F/G 15/5 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

TABLE 8-14. DBU GLOBAL VARIABLES

VARIABLE	PURPOSE
	partition is currently in that named partition so its contents can be properly swapped out into the extra data segment.
USECURSOR	Name of the named BUILDER partition which is the current partition and which is to be used in the next GETFIL call.
USEFILE	Name of the relation which the next GETFIL call is to open/retrieve.
USEGROUP	Group name for the relation which the next GETFIL call is to open/retrieve.
	HELP SUBSYSTEM VARIABLES
NEXT_HELP	Name of the help screen to call if the user just gives the "?" command.
SCRVAR	Set to the name of the variable the user had the cursor in when he asked for help about that field. Also, name of a field in the scrflds.db data dictionary relation, and therefore used in some reads of that relation.
	LEGAL VALUES SUBSYSTEM VARIABLES
ALLOK	Working variable used in CHECK_LEGAL screen.
LEGAL_FIELDS	List of the legal-values type fields for the current data screen. Used during initialization (by screen legals_init) to put default values into these fields, and by screen check-legals.
LEGLFILE	Name of the relation in the .legals group which holds the legals list for the legals-type field of current interest. Name of the field in the dbflds.db relation which supplies the name.
LEGLFIRST	Working variable in the LEGALS_INIT screen which holds the element in the list in LEGLPREF1 of current interest.
LEGLPREF1	List of numeric values, one corresponding to each legals-type field named in the LEGAL_FIELDS variable, which specify the SHOWPREF value of the legal value the screen designer wishes displayed as a default on the

TABLE 8-14. DBU GLOBAL VARIABLES

VARIABLE	PURPOSE
	current data screen. The records in any legal values relation may be ordered by the SHOWPREF field's values; these values specify the order in which values will be shown when the + command is used.
NOWLEGAL	Name of the legals-type field which is "current", i.e. whose legals relation is current on the LEGALS partition.
NOW_LEG_VAR	Obsolete.
SHOWPREF	Name of the field in every legals relation whose numeric values may be used to order the legal values in order of display preference. Each value has a unique corresponding SHOWPREF value.
	MISCELLANEOUS VARIABLES
ALTERCAPD	Displays the user's data alteration capabilities at the bottom of each data and comment screen. The value of this variable is set by the SHOWPRIV screen.
DATADATE	Every data relation has a datadate field; the corresponding variable is global in the DBU for efficiency.
DATE_VARS	Part of the schedule data recalculation facility available only in the PROJ_NC_SKED screen. Holds a list of all the milestone date fields/variables on the screen, e.g. "AWARD,START,etc.".
DESC_VARS	Similar to DATE_VARS, but holds the names of fields in the schedule planning factor relation which give the standard intervals between the milestone dates.
ENTRY_BY	This field appears in all data relations, and is therefore declared globally for efficiency.
ENTRY_DATE	This field appears in all data relations, and is therefore declared globally for efficiency.
EQUALS	Holds a 24 character string with nothing but "=" signs in it. Used to help construct the ALTERCAPD variable contents for display.

TABLE 8-14. DBU GLOBAL VARIABLES

VARIABLE	PURPOSE
LOWKEY	Name of the lowest key field on the current index on the current data relation BEFORE the datadate field. Used by the "N" command processor to support the LATEST display mode option.
LOWLEN	Numeric code indicating the data type of the field named in LOWKEY. -3 indicates a date (indexed descending by assumption), -2 a number indexed descending, -1 a number indexed ascending; anything greater than zero is an alpha type field of length LOWLEN bytes.
LPUNIT	Device control. Code name of the default hard copy output device as taken from the setting in the envrn.mnurel (user environment parameters menu storage) relation.
NOW_USER	Name of the current user. Useful in retrievals from security relations and from snsusers.sysrw, and for setting the ENTRY_BY variable.
SCEN1	Proper scenario keyfield value for the current scenario for the relation currently accessible through the MAIN partition.
SCEN2	Same as SCEN1 but associated with the MAIN2 partition.
SCEN3	Same as SCEN1 but associated with the MAIN3 partition.
SCENARIO	The scenario field appears in every data relation, and is global for efficiency.
SCENARIONM	Name of the current scenario.
TODAY	Today's date, useful for setting ENTRY_DATE.
USERNAME	Field which appears in data dictionary security relations and snusers.sysrw, used in retrievals from same.
ZEES	40-byte variable filled with lower-case "z". Used in producing a string high on the ascii collating sequence as part of the Search command option processing logic.

SCREEN CHOICE SUBSYSTEM VARIABLES

TABLE 8-14. DBU GLOBAL VARIABLES

VARIABLE	PURPOSE
NEXT_SCREEN	Name of the next data or menu screen to jump to if the user gives the "=" command all by itself.
NOW_SCREEN	Name of the current data or menu screen.
PREV_SCREEN	List of the last ten data and menu screens the user has looked at. Supports the "retrace-my-steps" algorithm.
	SECURITY SUBSYSTEM VARIABLES
ADD	Set to "Y" if user may add tuples to current data relation.
ALTDB	Field in sysusr.sysro, numeric 1 if user has overall data base write priveleges.
DELET	Set to "Y" if user may delete tuples from current data relation.
MODFY	Set to "Y" if user may modify tuples in current relation (i.e. give "M" command).
READ	Set to "Y" if user may read tuples from current data relation (i.e. give "N" and "B" and "S" commands.
READB	Field in sysusr.sysro, numeric 1 if user has basic data base read priveleges.
SEEIT	Set to "Y" if the current menu or data screen can be displayed to NOW_USER.
UPDAT	Set to "Y" if the user can give the "U" command in the current data screen.
USERLEVL	A field in sysusr.sysro, gives the overall privelege level of the user. 2 for ordinary, 3 for DBA level.
	DATA VALIDATION SUBSYSTEM VARIABLES
ABORT_MODFY	Set to 1 (via assignments in data screen VARIABLE sections) if user changes value of any key field required for point needed to return to current data record just prior to issuing a RECORD UPDATE command.

TABLE 8-14. DBU GLOBAL VARIABLES

VARIABLE	PURPOSE
ADDOK	If set to "Y", validation utilties say it is OK to issue the "A" command for the data on the screen now (but not the "U" command in general).
CANCEL	Set to 1 if the user wants to cancel a deletion once it starts.
DATEU	Working variable in the VERIFYU screen.
DELOK	Set to "Y" by the VERIFYD screen if it's ok to go ahead and delete the tuple the user requested deletion of.
DOADD	Set to 1 in PROC_DATA_CMDA if all the checks required for an addition are ok.
DOMOD	Same as DOADD but for modifications
IN_ONE	Boolean set to 1 if this join check is of the "IN_ONE" variety.
JOINFLDS	Message buffer holding the names of fields on which a join must be possible; output to user if check fails. Used in VERIFYJ.
JOINMSG	A field in the filjoin.db data dictionary relation, with a message to output in the event of a join check failure.
JOINTYPE	Another field in filjoin.db, indicating the type of join VERIFYJ should test for. Valid values are "REQUIRED", "RECOMMENDED", or "IN_ONE_x".
LASTFAIL	Set to one if an "IN_ONE" (network or multiple-relations-may-satisfy) type of join check is in process AND we are at the very last try for satisfying the requirement. Used in VERIFYJ.
NOWFILER NOWGROUPR	Working storage for VERIFYD, used to transfer the name of the next relation "down the dependency tree" to process via a recursive call to VERIFYD.
RECURS	Set to the number of invocations of VERIFYD on the execution stack.
SAVMSG	Holds the value of JOINMSG to actually output,

TABLE 8-14. DBU GLOBAL VARIABLES

VARIABLE	PURPOSE
	since a read by the end of data in filjoin for the current join check will usually but the message for the NEXT check into JOINMSG itself.
SAVTYP	Similar function to SAVMSG, but for JOINTYP field values.
SRCFIELD SRCFILE SRCGROUP	Fields in filjoin.db which identify a field in a join-test source file (i.e. the relation for the current data screen which is to be part of the join specification).
TGTFIELD TGTFILE TGTGROUP	Similar to SRCFIELD, SRCFILE, and SRCGROUP except these identify the field in the join target file which must have a value matching SRCFIELD.
UOK	Set to 1 if all validation checks indicate an Update is ok.
VERIFYFIELDS	List of fields in the relation for the current data screen, the sum of whose screen variable values must be unique if an Add is to be permissible (and must NOT be unique for an Update). Used by screen VERIFYD.
VERTYP	Indicates which validation checks should be performed for the current data screen. Value may be "U", "J", or "JU", referring to Uniqueness checks, Join check, or both.
	WORKING VARIABLES USED BY SEVERAL SUBSYSTEMS
ANSWER	Text of user answer to a prompt, usually Y/N.
ANSWERL	Text of user answer to a prompt.
BUFFER	Used to move text from COMMENT array data storage into DCOMMENT comment display storage.
COND	Usually text of a WHERE clause for a SELECT.
DONE	Set to 1 if all conditions for finish of a WHILE loop have been met.
ENDFILE	Set to 1 if an end-of-file condition is encountered.
ERRNUM	Error number if an error occurs.

TABLE 8-14. DBU GLOBAL VARIABLES

VARIABLE	PURPOSE
FIELDNAME	A field in several data dictionary files, variable used in retrievals from these.
FILE	Used in dictionary file retrievals.
FILNAM	Name of a file.
FLDFMT	Holds the RELATE format code for a variable, retrieved by a $RDBINFO call. Indicates whether it's a date.
FLDLEN	Number of bytes in an alphanumeric relation field, from a $RDBINFO call.
FLDNAM	A field name.
FLDNUM	RELATE id number for a field, obtained by a call to $RDBINFO or else set as part of a loop.
FLDTYP	Data type code for a data base field, obtained by a call to $RDBINFO.
FLDVAL	Buffer to hold an ASCII version of the current value of an arbitrary field.
FOUND	Set to "Y" if a point or SELECT succeeds.
GROUP	Group name part of a file name.
IFOUND	Set to 1 if a POINT or SELECT succeeds.
INUM	Numeric variable.
LIST	Big alpha type working buffer.
MPECOM	Ostensibly text of an MPE command entered by user, also used widely as a working buffer.
NOMORE	Set to 1 on end-of-data condition.
NUM	Numeric variable.
OK	Numeric variable, usually used as a boolean.
SCREEN	Used in retrievals from dictionary relations.
SCREEN_NAME	Working storage for a screen name.
WORKFILE WORKGROUP	Temporary storage for a file name.
Y	Usually user response to a Y/N prompt.

8.4.3 DBU Screen and Subsystem Operations

This Section will describe how individual DBU screens and subsystems work. The screens are listed alphabetically and by purpose in Tables 8-15 and 8-16.

8.4.3.1 DBU Initialization

DBU initialization is done by two screens, START and INIT. Figure 8-21 presents an annotated subset of the text of these screens (comments in [ ] do not acutally appear in the code). Screen START declares all the global variables, and is thus dominated by its DECLARATIONS section. Screen INIT contains no layout and no declarations---it is a "subroutine" screen which does nothing but processing in its INITIAL section. This discussion will be concerned only with screen INIT, since global variables were discussed in Section 8.4.2.5.

INIT can be divided into seven parts for purposes of discussion. In order, they are: constant initialization, security set-up, file management subsystem initialization, data dictionary connection, report generation subsystem startup, legal values subsystem initialization, and screen choice/display mode initialization.

The first part just sets the values of global variables whose values will be constant throughout execution. Most of these are lists or arrays of reference values. Also, the data format to be used is specified and a "SYSTEM $CANCEL" is issued to enable RELATE response to control-Y.

In the second part, starting with "TODAY=$DATE" the current date and user name are retrieved and the ALIAS overall security relation SYSUSR is consulted for the user's overall priveleges. The RECORD READ on that relation returns values for the ALTDB, READB, and USERLEVL variables. Then snusers.sysrw is consulted for the name of the scenario the user is running with, and envrn.mnurel for the name of the default hard copy output device

TABLE 8-15. Alphabetical List of DBU Screens

SCREEN	TYPE/SUBSYSTEM	FILE (.screens)
BACKHELP	HELP/SYSTEM	DBUMHELP
BUG1	UTILTITY	DBUDATA
BUG2	UTILTITY	DBUMHELP
CHAINHELP	HELP/SYSTEM	DBUMHELP
CHECK_LEGAL	UTILITY/LEGALS	DBUSUBR
CLASS_CHARS	DATA	DBUDATA
CLASS_CHAR_COMT	COMMENT	DBUDATA
CLASS_CHAR_HELP	HELP/SCREEN	DBUDATA
CLASS_MENU	MENU	DBUDATA
CLASS_MENU_HELP	HELP/SCREEN	DBUDATA
COMMANDHELP	HELP/SYSTEM	DBUMHELP
COMMANDHELPL	HELP/SYSTEM	DBUMHELP
COMT_INIT	UTILITY/COMMENTS	DBUSUBR
COMT_UPDATE	UTILITY/COMMENTS	DBUSUBR
CURR_NC_SKED	DATA	DBUDATA
CURR_NC_SKED_C	COMMENT	DBUDATA
CURR_RE_SKED	DATA	DBUDATA
CURR_RE_SKED_C	COMMENT	DBUDATA
CURSORHELP	HELP/SYSTEM	DBUMHELP
CURSORHELPL	HELP/SYSTEM	DBUMHELP
C_NC_SKED_HELP	HELP/SCREEN	DBUDATA
C_RE_SKED_HELP	HELP/SCREEN	DBUDATA
DCHECKHELP	HELP/SYSTEM	DBUMHELP
DEACTIVATE	DATA	DBUDATA
DEACTIVATE_COMT	COMMENT	DBUDATA
DEACTIVATE_HELP	HELP/SCREEN	DBUDATA
DELETEHELP	HELP/SYSTEM	DBUMHELP
DEL_COMT_LINE	UTILITY/COMMENTS	DBUSUBR
FIELDHELP	HELP/SYSTEM	DBUMHELP
GETFIL	UTILITY/FILE MGR	DBUSUBR
HELPHELP	HELP/SYSTEM	DBUMHELP
HIST_NC_SKED	DATA	DBUDATA
HIST_NC_SKED_C	COMMENT	DBUDATA
HIST_RE_SKED	DATA	DBUDATA
HIST_RE_SKED_C	COMMENT	DBUDATA
H_NC_SKED_HELP	HELP/SCREEN	DBUDATA
H_RE_SKED_HELP	HELP/SCREEN	DBUDATA
IFIELDHELP	HELP/SYSTEM	DBUMHELP
INIT	UTILITY/STARTUP	DBUSUBR
INS_COMT_LINE	UTILITY/COMMENTS	DBUSUBR
JOB_TYPE_HELP	HELP/SCREEN	DBUDATA
JOB_TYPE_MENU	MENU	DBUDATA
JOINHELP	HELP/SYSTEM	DBUMHELP
JOINHELP2	HELP/SYSTEM	DBUMHELP
JUMPHELP	HELP/SYSTEM	DBUMHELP
KCLASS_CHARS	UTILITY/SCREEN CLEAR	DBUDATA
KCURR_NC_SKED	UTILITY/SCREEN CLEAR	DBUDATA
KCURR_RE_SKED	UTILITY/SCREEN CLEAR	DBUDATA
KDEACTIVATE	UTILITY/SCREEN CLEAR	DBUDATA
KHIST_NC_SKED	UTILITY/SCREEN CLEAR	DBUDATA
KHIST_RE_SKED	UTILITY/SCREEN CLEAR	DBUDATA

TABLE 8-15. Alphabetical List of DBU Screens

SCREEN	TYPE/SUBSYSTEM	FILE (.screens)
KNC_JOB_TYPES	UTILITY/SCREEN CLEAR	DBUDATA
KNC_SKED_PF	UTILITY/SCREEN CLEAR	DBUDATA
KPROJ_NC_SKED	UTILITY/SCREEN CLEAR	DBUDATA
KPROJ_RE_SKED	UTILITY/SCREEN CLEAR	DBUDATA
KRE_JOB_TYPES	UTILITY/SCREEN CLEAR	DBUDATA
KYARD_CHARS	UTILITY/SCREEN CLEAR	DBUDATA
LEGALHELP	HELP/SYSTEM	DBUMHELP
LEGALS_INIT	UTILITY/LEGALS	DBUSUBR
MAINHELP	HELP/SYSTEM	DBUMHELP
MASTER	MENU	DBUDATA
MOVEHELP	HELP/SYSTEM	DBUMHELP
MPECOMMAND	UTILITY/MISC	DBUSUBR
NC_JOB_TYPES	DATA	DBUDATA
NC_SKED_PF	DATA	DBUDATA
NC_SKED_PF_COMT	COMMENT	DBUDATA
NC_SKED_PF_HELP	HELP/SCREEN	DBUDATA
NEXTHELP	HELP/SYSTEM	DBUMHELP
NEXT_LEGAL	UTILITY/LEGALS	DBUSUBR
OPNFIL	UTILITY/FILE MGR	DBUSUBR
PREV_LEGAL	UTILITY/LEGALS	DBUSUBR
PROC_COMT_CMD	UTILITY/CMD PROCESSOR	DBUSUBR
PROC_DATA_CMDA	UTILITY/CMD PROCESSOR	DBUSUBR
PROC_DATA_CMDB	UTILITY/CMD PROCESSOR	DBUSUBR
PROC_HELP_CMD	UTILITY/CMD PROCESSOR	DBUSUBR
PROC_MENU_CMD	UTILITY/CMD PROCESSOR	DBUSUBR
PROC_REPT_CMD	UTILITY/CMD PROCESSOR	DBUSUBR
PROJ_NC_SKED	DATA	DBUDATA
PROJ_NC_SKED_C	COMMENT	DBUDATA
PROJ_RE_SKED	DATA	DBUDATA
PROJ_RE_SKED_C	COMMENT	DBUDATA
PUSHHELP	HELP/SYSTEM	DBUMHELP
P_NC_SKED_HELP	HELP/SCREEN	DBUDATA
P_RE_SKED_HELP	HELP/SCREEN	DBUDATA
READHELP	HELP/SYSTEM	DBUMHELP
REPORT	UTILITY/REPORTS	DBUSUBR
RE_JOB_TYPES	DATA	DBUDATA
RUNEDITOR	UTILITY/MISC	DBUSUBR
RUNTDP	UTILITY/MISC	DBUSUBR
SEARCH	UTILITY/MISC	DBUSUBR
SECURITY_CHECK	UTILITY/SECURITY	DBUSUBR
SET_DEVICE	UTILITY/REPORTS	DBUSUBR
SET_INSPCT_MODE	UTILITY/DISPLAY MODE	DBUSUBR
SET_MORE	UTILITY/COMMENTS	DBUSUBR
SHOWPRIV	UTILITY/SECURITY	DBUSUBR
SKED_MENU	MENU	DBUDATA
SKED_MENU_HELP	HELP/SCREEN	DBUDATA
SKED_PF_HELP	HELP/SCREEN	DBUDATA
SKED_PF_MENU	MENU	DBUDATA
SLOWHELP	HELP/SYSTEM	DBUMHELP
START	UTILITY/STARTUP	DBUSUBR
SUSPEND	UTILITY/MISC	DBUSUBR

TABLE 8-15. Alphabetical List of DBU Screens

SCREEN	TYPE/SUBSYSTEM	FILE (.screens)
VERIFYD	UTILITY/VALIDATION	DBUSUBR
VERIFYJ	UTILITY/VALIDATION	DBUSUBR
VERIFYU	UTILITY/VALIDATION	DBUSUBR
YARD_CHARS	DATA	DBUDATA
YARD_CHARS_COMT	COMMENT	DBUDATA
YARD_CHARS_HELP	HELP/SCREEN	DBUDATA
YARD_MENU	MENU	DBUDATA
YARD_MENU_HELP	HELP/SCREEN	DBUDATA

TABLE 8-16. Annotated List of DBU Screens By Subsystem

SCREEN	PURPOSE
COMMENT INSPECTION/ENTRY SCREENS	
CLASS_CHAR_COMT CURR_NC_SKED_C CURR_RE_SKED_C DEACTIVATE_COMT HIST_NC_SKED_C HIST_RE_SKED_C NC_SKED_PF_COMT PROJ_NC_SKED_C PROJ_RE_SKED_C YARD_CHARS_COMT	Each of these screens is associated with a data screen, and is intended for entry and modification of comments about the data displayed on the associated screen. Comments are always associated with a particular tuple in the associated data relation, with the implicit join made on the fields listed in KEYFIELDS but including datadate and entry_date.
DATA ENTRY/MODIFICATION SCREENS	
CLASS_CHARS CURR_NC_SKED CURR_RE_SKED DEACTIVATE HIST_NC_SKED HIST_RE_SKED NC_JOB_TYPES NC_SKED_PF PROJ_NC_SKED PROJ_RE_SKED RE_JOB_TYPES YARD_CHARS	These are the DBU's data entry screens, the reasons for its existence. Names are mnemonic, with SKED indicating schedules, HIST, CURR, and PROJ referring to historical, current, and projected epochs respectively, NC and RE to new construction type jobs (reacts and conversions too) and to repair-type jobs (includes SLEPs). CHARS stands for characteristics, PF for planning factors.
HELP SCREENS WHICH TELL ABOUT A MENU OR DATA SCREEN	
CLASS_CHAR_HELP CLASS_MENU_HELP C_NC_SKED_HELP C_RE_SKED_HELP DEACTIVATE_HELP H_NC_SKED_HELP H_RE_SKED_HELP NC_SKED_PF_HELP P_NC_SKED_HELP P_RE_SKED_HELP SKED_MENU_HELP SKED_PF_HELP YARD_CHARS_HELP YARD_MENU_HELP JOB_TYPE_HELP	These help screens have text which talks about "their" data or menu screen. They are displayed when the user enters a "?" in that screen. The names are mnemonic and similar to those of the associated data/menu screens. H, C, and P stand for HIST, CURR, and PROJ respectively.
HELP SCREENS WHICH DESCRIBE OVERALL DBU OPERATION	
BACKHELP	How to retrace your steps through the screens you have been in already.

TABLE 8-16. Annotated List of DBU Screens By Subsystem

SCREEN	PURPOSE
CHAINHELP	How to use menu screens to find your way around.
COMMANDHELP	General discussion of types of DBU commands.
COMMANDHELPL	Table of DBU command code characters.
CURSORHELP	General discussion of the screen cursor and how to move it around the screen.
CURSORHELPL	Table of control codes for cursor movement.
DCHECKHELP	General discussion of data validation which is done.
DELETEHELP	What happens when you try and delete data.
HELPHELP	How to use the help subsystem.
IFIELDHELP	Further discussion of screen enhancements.
JOINHELP JOINHELP2	Why and how join checking is done as part of the validation process.
JUMPHELP	How to have the DBU display the next screen you want to use without going through the menus.
LEGALHELP	Why and how legal value checking is done as part of the validation process.
MAINHELP	General introduction to DBU and to the system-level help screens.
MOVEHELP	General discussion of how to move from screen to screen.
NEXTHELP	How to move to the default next screen.
PUSHHELP	How help and comment screens are entered and exited.
READHELP	General discussion of how to interpret the display enhancements used for screen fields.
SLOWHELP	Why is this damn thing so slow, and inconsistently?

TABLE 8-16. Annotated List of DBU Screens By Subsystem

SCREEN	PURPOSE
	MENU SCREENS
CLASS_MENU JOB_TYPE_MENU MASTER SKED_MENU SKED_PF_MENU YARD_MENU	These screens guide the user to the proper data screen for his purposes.
	COMMAND PROCESSORS
PROC_COMT_CMD	Processes comment screen action commands.
PROC_DATA_CMDA	For data screen commands which do not manipulate data.
PROC_DATA_CMDB	Data screen actioncommands which do change the screen's contents, e.g. N, A, and D.
PROC_HELP_CMD	Action commands given in help/screen type of help screens; the system help screens are very old and often do all their own command processing.
PROC_MENU_CMD	Menu screen action commands.
PROC_REPT_CMD	Serves the REPORT hard-copy output generation utility screen; normal action commands as well as special processing needed for report execution.
	COMMENT SUBSYSTEM UTILITIES
COMT_INIT	Initializes a comment screen, including retrieval of the relation and the appropriate text from it and placement of the first page into the display buffer.
COMT_UPDATE	Updates the current comment relation with the text currently in memory. Typically called by PROC_DATA_CMDA/B when the user is finished with the current data tuple.
DEL_COMT_LINE	Removes the comment line the cursor is at on the comment screen from the buffer, and "closes up" the gap.
INS_COMT_LINE	Inserts a blank line into the comment buffer BEFORE the line the cursor is at.

TABLE 8-16. Annotated List of DBU Screens By Subsystem

SCREEN	PURPOSE
SET_MORE	Decides whether there is text in the comment storage buffer "below" the last line showing on the display page, and sets the MORE variable accordingly.
	DISPLAY MODE RESET
SET_INSPCT_MODE	This is executed when the user gives the "L" command. It prompts for the new mode settings and issues a new selection for the current path if necessary.
	FILE MANAGEMENT SUBSYSTEM
GETFIL	A call to GETFIL retrieves the relation of your choice for use on the current named BUILDER partition. The file is opened if necessary. GETFIL requires the file name to be in USEFILE and USEGROUP, and the BUILDER partition name to be in USECURSOR.
OPNFIL	Used by GETFIL and some code in the initilization screen for the actual issuance of an OPEN FILE command. Meant to detect and partially recover from system "OUT OF GLOBAL RIN" errors, which were a problem at one time.
	FIELD HELP UTILITY
FIELDHELP	This general-purpose help screen displays information about the data field the screen cursor is currently in, drawing its information from the data dictionary. This is what does the work of responding to the ESC ? command.
	LEGAL VALUES SUBSYSTEM UTILITIES
CHECK_LEGAL	Called as part of data validation (usually from PROC_DATA_CMDB) to make sure that the user has placed valid values in all legals-type fields on the screen.
LEGALS_INIT	Called during data screen initialization to place default values in the legals-type fields which appear on the screen.
NEXT_LEGAL	Shows the next legal value in the sequence for the field the cursor is currently in. This is the respondent for the ESC + command.

TABLE 8-16. Annotated List of DBU Screens By Subsystem

SCREEN	PURPOSE
PREV_LEGAL	Like NEXT_LEGAL, but shows the previous value in the SHOWPREF sequence.
	REPORT GENERATION SUBSYSTEM
REPORT	General-purpose screen which responds when the "P" command is given. Offers the user a list of reports which can be printed from the given screen and executes the report generator of the user's choice.
SET_DEVICE	Prompts and gives file equations so that printed outputs go to the hard copy device of the user's choice.
	SCREEN CLEAR UTILITIES
KCLASS_CHARS KCURR_NC_SKED KCURR_RE_SKED KDEACTIVATE KHIST_NC_SKED KHIST_RE_SKED KNC_JOB_TYPES KNC_SKED_PF KPROJ_NC_SKED KPROJ_RE_SKED KRE_JOB_TYPES KYARD_CHARS	Many DBU commands result in a screen clear when given in a data screen. The actual clearing of screen variables is done by a utility customized to the particular data screen (since each once has a different list of variables). These are the clear utilities, with names which closely resemble the data screens they serve.
	SECURITY SUBSYSTEM
SECURITY_CHECK	Queries the scrsec and filsec relations to see if *now_user* is allowed to look at *now_screen*, and what operations he can perform on *now_file.now_group*.
SHOWPRIV	Constructs the string displayed at the bottom of each data screen (in ALTERCAPD) which tells what the user's capabilities are in the given screen. Uses the output of SECURITY_CHECK.
	DBU INITIALIZATION
INIT START	Declares all the DBU global BUILDER variables. Does all the one-time DBU initialization.

TABLE 8-16. Annotated List of DBU Screens By Subsystem

SCREEN	PURPOSE
	DATA VALIDATION SUBSYSTEM
VERIFYD	Checks to see if any subsidiary will have to be deleted if the tuple requested for deletion is, and gives the DELETE commands for these subsidiary tuples after prompting the user for confirmation.
VERIFYJ	Checks to see if addition or modification of the current data tuple will violate the join conditions specified in filjoin.db, i.e. if the integrity of the data base would be compromised.
VERIFYU	Checks to see if the current data tuple would be unique in terms of its VERIFYFIELDS field values if added to the current relation.
	MISCELLANEOUS UTILITY SCREENS
MPECOMMAND	Lets the use give MPE commands (no UDC's).
RUNEDITOR	Runs the EDITOR editor.
RUNTDP	Runs the TDP editor.
SEARCH	Respondent for the "S" command available in data screens. Finds all non-blank fields on the screen, constructs a selection which will return only tuples matching those field values, and retrieves the first such matching tuple.
SUSPEND	Interface with the System Core, called whenever the user issues a "Q" command. Suspends the DBU process, and then checks what scenario is now the current one when the user re-activates the DBU.
BUG1 BUG2	These screens are present to handle a "feature" (read: "bug") of the BUILDER's compiler option which causes it to ignore the first screen found in any file it processes but the first file. These two dummies ensure that the first screens in DBUDATA and DBUMHELP are not lost after compilation.

FIGURE 8-21. DBU INITIALIZATION CODE

```
*** SCREEN START                    [declares all global variables]
*** DECLARATIONS
    SRCFILE,SRCGROUP,TGTFILE,TGTGROUP ;LENGTH=8;UPPER
    SRCFIELD,TGTFIELD ;LENGTH=15;UPPER
    Y           ;LENGTH=1;UPPER
    ANSWER      ;LENGTH=1;UPPER
    ANSWERL ;LENGTH=20;UPPER
    .
    .
    .                               [other variable declarations omitted for space]
    .
    .
    SCEN3,SCEN2,SCEN1,SCENCOM,SCENSAVE ;LENGTH=12
*** INITIAL
   CALL SCREEN INIT                 [keep the global vars on the var list]
NOTE
*** SCREEN INIT                     [start up everything]
*** INITIAL
    DISPLAY "Data maintenance module start up now in progress..."

                                    [give equivalent of FORTRAN data statements]

    ZEES := "zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz"
    DATACMDA := " ""^"","""/"","""C"","""P"","""?"","""="","""Q"","""I"","""":"",&
                ""Z"","""E"","""H"","""T"","""L"" "
    DATACMDB := " ""K"","""N"","""B"","""S"","""D"","""V"","""A"","""U"","""M"" "
    DATE_VARS := "APPROP,AWARD,START,KEEL,LAUNCH,DELIVERY,COMMISSION"
    DESC_VARS := "APPROP_AWD,AWD_ST,ST_KL,KL_LN,LN_DL,DL_COM"
    EQUALS := "=========================="
    $SUBSCRIPT := 0
    COMT_FLD_MODS := ";OPTIONAL"
    $SUBSCRIPT := 1
    COMT_FLD_MODS := ";OPTIONAL;ENHANCE=I,U"
    $SUBSCRIPT := 2
    COMT_FLD_MODS := ";OPTIONAL;DATE=""MM/DD/CCCC"" "
    $SUBSCRIPT := 3
    COMT_FLD_MODS := ";OPTIONAL;DATE=""MM/DD/CCCC"";ENHANCE=I,U"
    $SUBSCRIPT := 4
    COMT_FLD_MODS := ";OPTIONAL;UPPER"
    $SUBSCRIPT := 5
    COMT_FLD_MODS := ";OPTIONAL;UPPER;ENHANCE=I,U"
    $SUBSCRIPT := 6
    COMT_FLD_MODS := ";OPTIONAL;NUMERIC;ENHANCE=I,U"
    $SUBSCRIPT := 7
    COMT_FLD_MODS := ";OPTIONAL;NUMERIC"
    $SUBSCRIPT := 8
    COMT_FLD_MODS := ";OPTIONAL;UPPER;ENHANCE=U"
NOTE          ---get user name, date, privelege level, scenario
    SET OPTION DATE="MM/DD/CCCC"
    SYSTEM $CANCEL
    TODAY := $DATE
    SELECT NOW_USER=$USER
    RECORD READ
```

FIGURE 8-21. DBU INITIALIZATION CODE

```
    OPEN FILE SYSUSR.SYSRO;MODE=SHARED
    SET INDEX USERNAME
    RECORD POINT;KEY=NOW_USER
    IF NOT $FOUND
        DISPLAY "You are not a valid user."
        EXIT
    ELSE
        RECORD READ
    ENDIF
    CLOSE
    OPEN FILE SNUSERS.SYSRW;MODE=SHARED
    SET INDEX USERNAME
    RECORD POINT;KEY=NOW_USER
    IF NOT $FOUND
        SCROLL  "You are not in the scenario usage table.  &
                 You must be running ALIAS to use the DBU."
        EXIT
        ENDIF
    RECORD READ
NOTE          --set up the user's report output device
    OPEN FILE ENVRN.MNUREL;MODE=SHARED
    SET INDEX SCENARIO
    RECORD POINT;KEY=SCENARIONM
    RECORD READ
NOTE          --list of USE cursors for which swapping will be done
                [only these partitions can be used with screen GETFIL]
    CURS_SWAP_LIST := " ""CHECK1"",""LEGALS"",""MAIN"",""MAIN2"",""MAIN3"" ,&
        ""COMMENT"",""REPORT"",""CHECK2"",""CHECK3"",""CHECK4"" "
    CREATE PARTITION MAIN
    CREATE PARTITION MAIN2
    CREATE PARTITION MAIN3
    CREATE PARTITION CHECK1
    CREATE PARTITION CHECK2
    CREATE PARTITION CHECK3
    CREATE PARTITION CHECK4
    CREATE PARTITION LEGALS
    CREATE PARTITION COMMENT
    CREATE PARTITION REPORT
NOTE          --initialize the cursor swap system
    :SETJCW SCREENSYS,9012
    CALL PROCEDURE CURINI
NOTE          --open up files on the special-purpose cursors
    DISPLAY "Connecting to data dictionary..."
    CREATE PARTITION FILINFO
    USE PARTITION FILINFO
    OPEN FILE FILINFO.DB;MODE=SHARED
    SET INDEX GROUP,FILE
    CREATE PARTITION FILJOIN
    USE PARTITION FILJOIN
    OPEN FILE FILJOIN.DB;MODE=SHARED
    SET INDEX TGTFILE,TGTGROUP
    CREATE PARTITION SCRSEC
    USE PARTITION SCRSEC
```

FIGURE 8-21. DBU INITIALIZATION CODE

```
    OPEN FILE SCRPRIV.DB;MO=SH
    SET INDEX SCREEN,USERNAME
    CREATE PARTITION FILSEC
    USE PARTITION FILSEC
    OPEN FILE FILPRIV.DB;MO=SH
    SET INDEX FILE,GROUP,USERNAME
NOTE
    CREATE PARTITION FILSCREENS
    OPEN FILE FILSCRN.DB;MODE=SHARED
    SET INDEX FILE,GROUP
    CREATE PARTITION SCRFLDS
    USE PARTITION SCRFLDS
NOTE    OPEN FILE FLDFILE.DB;MODE=SHARED
    OPEN FILE SCRFLDS.DB;MODE=SHARED
    SET INDEX SCREEN,SCRVAR
    CREATE PARTITION DBFLDS
    USE PARTITION DBFLDS
    OPEN FILE DBFLDS.DB;MODE=SHARED
    SET INDEX FIELDNAME
    CREATE PARTITION FLDESC
    USE PARTITION FLDESC
    OPEN FILE FLDESC.DB;MODE=SHARED
    SET INDEX FIELDNAME,LNUM
    CREATE PARTITION FLDFILE
    USE PARTITION FLDFILE
    OPEN FILE FLDFILE.DB;MODE=SHARED
    SET INDEX FIELDNAME,FILE,GROUP
NOTE    --start up report partition's dedicated process
    USE PARTITION REPORT
    USEFILE := "STRT103"
    USEGROUP := "DB"
    USECURSOR := "REPORT"
    CALL SCREEN GETFIL
    CLOSE
    CREATE PARTITION REPTLIST
    USE PARTITION REPTLIST
    OPEN FILE SREPTS.DB;MODE=SHARED
    OPEN FILE REPTEX.DB;MODE=SHARED
NOTE          --start up the legals system
    DISPLAY "Connecting to the legal values reference library..."
NOTE
    USE PARTITION LEGALS
    USEFILE := "JBTPREF"
    USEGROUP := "LEGALS"
    USECURSOR := "LEGALS"
    CALL SCREEN GETFIL
    USE PARTITION FILINFO
    SELECT FILNAM=FILE,SCRINDX  WHERE SCRSET="LEGALS"
    RECORD READ
    WHILE NOT $EOF
        IF FILNAM"JBTPREF"
           USE PARTITION LEGALS
           OPEN FILE %filnam.LEGALS;MODE=SHARED
```

FIGURE 8-21. IBU INITIALIZATION CODE

```
            SET INDEX %scrindx
            ENDIF
          USE PARTITION FILINFO
          RECORD READ
          ENDWHILE
      SELECT
NOTE          --finish init and go!
      USE PARTITION MAIN
      NON_CURSOR := "MAIN"
      PREV_SCREEN := "MASTER,MASTER,MASTER,MASTER,MASTER,&
                     MASTER,MASTER,MASTER,MASTER,MASTER"
      INMODE := "LATEST"
      MODEDATA := "DATA"
      DATEMODE := "PLANNED"
      SET SCREEN MASTER
```

that is preferred. Both these relations are left open on partition DEFAULT so they can be consulted later by screen SUSPEND.

Initialization of the file management subsystem (starts with "CURS_SWAP_LIST=") requires setting up of a list of the named partitions the subsystem will be expected to work with, creation of all these partitions, and a call to CURINI with the SCREENSYS jcw set to 9012. The file management subsystem's code can serve many modules simultaneously, but it is important that each have a unique id code. SCREENSYS must be set to this code before any calls can be issued to file management FORTRAN routines.

Connection to the data dictionary, which is done through the default RELATE son process, involves a straightforward set of partition creations and OPEN FILE commands (this part starts with the 'DISPLAY "Connecting to data dictionary..."' line).

Startup of the report generator subsystem (at "USE PARTITION REPORT") marks the first use of the file management system. It is used to open a dummy relation, whose filinfo.db record causes a new RELATE son process to be created. The file is then closed, leaving a totally empty process available on the REPORT partition for use by report generators. The last two data dictionary files, REPTEX and SREPTS, are then opened.

Legal values subsystem startup (at "USE PARTITION LEGALS") employs a similar strategy to get a dedicated RELATE son process, except that the relation which is opened through the file management subsystem is "real" and left open. Once the process is started, a loop is entered which reads the names of all the relations in the legal values library from filinfo.db and which opens these on the LEGALS partition.

The final part of initialization sets the screen execution history to ten invocations of "MASTER", and the display mode to "LATEST PLANNED DATA".

The last line SETs execution to the top screen in the menu hierarchy, thereby flushing INIT, but leaving screen START with all its global variables in place in BUILDER memory.

8.4.3.2 The File Management Subsystem

The motivation for the file management subsystem was (1) the necessity of having more than one RELATE son process serve the DBU's BUILDER process, and (2) the desirability of keeping relations open once they have been opened.

The things that make the file management system possible are:

1) BUILDER uses RELATE through the HLI programmatic interface, just as ALIAS FORTRAN modules do.

2) A BUILDER partition corresponds to a RELATE HLI cursor. CREATE PARTITION xxx just causes a 50-word array to be reserved in BUILDER memory and an rdbinit call to be issued with the array as the cursor argument. Requests for RELATE operations (e.g. RECORD POINT USING xxx) on this partition cause the array to be passed as the cursor argument to HLI routines. (During BUILDER initialization a partition named DEFAULT is automatically created and an rdbinit call issued which starts up the default BUILDER RELATE son process).

3) The ONLY knowledge BUILDER has about its RELATE son process is contained in these 50-word arrays. If a second RELATE son process were somehow created, and a 50 word array which was initialized by a rdbinit call to this process were substituted into the BUILDER data area

for one of the partitions already created by BUILDER for use with the default RELATE process, BUILDER would happily use the second process in all subsequent work involving that partition.

4) The undocumented HLI routine rdbinitx allows the caller to specify WHICH RELATE PROCESS a new partition/cursor should be "hooked up" to. It is programmatically possible to create multiple RELATE son processes and to use any of them at will.

5) Any HLI routine may be called by FORTRAN subroutines called from BUILDER via CALL PROCEDURE.

6) The 50-word integer array that is the current BUILDER partition (the one used if a vanilla RECORD READ is issued, for example) is passed to any FORTRAN routine called via CALL PROCEDURE. The routine may read and write over this 50-word array.

7) Although FORTRAN routines in an SL cannot have global variables (common blocks), they can issue intrinsic calls to work with an extra data segment. Extra data segments function as indirectly addressable global storage.

We have all the elements necessary; to put them together into a system that will meet the goals (many processes and files always open) requires a concept and two rules.

The concept is to distinguish between NAMED PARTITIONS and ACTUAL PARTITIONS. A named partition is defined as the 50-word data area in BUILDER memory that is retrieved when the BUILDER wants a partition with a given name; but not the contents of this area. Think of a named partition as an empty bin. An actual partition is 50 words of data which can function as an HLI

cursor---that is, sensible results would occur if the actual partition were placed in the named partition and BUILDER operations were undertaken using the name.

The first rule is that all relations to be managed by the file system will have their own actual partition. A SHOW PATH given on that partition will always show no more than one file open (although a SELECT might be issued on top of the file).

The second rule is that when a relation is to be used in the DBU, its actual partition must be swapped into a named partition, with the named partition then being the one ostensibly worked with.

This last rule is the heart of the subsystem. The rest of this section will discuss the mechanics of conducting the swap.

The swap itself is straightforward. A call is made to the FORTRAN routine curswp, which as usual has "cursor" as one of its three formal arguments. The routine takes the current 50 words in the argument, which represent an actual partition that the DBU does not need for the moment, and stores them away for later use. The 50 words that represent the actual partition that the dbu wants now are then written into the argument and the swap is done.

Storage for all actual partitions is in an extra data segment approximately 10,000 words long (i.e. there is capacity for about 200 actual partitions or for about 200 simultaneously open relations). The trick that CURSWP has to perform is to figure out what actual partition is in the named partition passed as the formal parameter and where it should be stored in the extra data segment, and also what actual partition is desired and where it is stored.

The GETFIL utility screen, which is the interface between the DBU as a whole and the cursup routine, provides two crucial pieces of information which enable curswp to perform its trick. Two Job Control Words (JCW), numswap and cursornum, are set to the index number of the current named partition and the id number of the desired actual partition.

The index number of the current named partition refers to its place on the list of all named partitions, which is stored in the CURS_SWAP_LIST DBU global variable (see Figure 8-21 in the previous section for its initialization statement). Given the name of the named partition the DBU wants the swap done on and the list in this variable, the index number is easily found by screen GETFIL using the BUILDER's $IN function.

Subroutine curswp maintains a special array in the first few words of the extra data segment; each element of the array contains the id number of the actual partition currently in the corresponding named partition. Thus, if given the index number for the named partition passed to it, curswp can find out what actual partition is currently in it, and can then store it away.

An actual partition id number just identifies one of the 200 or so available actual partition storage spaces. Curswp retrieves an actual partition from the segment by multiplying the id number by 50 (number of words in a partition), adding a constant offset to allow for the named partition index mapping array at the start of the segment, and reading 50 words from the data segment starting at the resulting "address".

But the only information that is passed into the GETFIL screen is contained in the USECURSOR, USEFILE, and USEGROUP variables. USECURSOR contains the name of the named partition---we have seen how this is converted to an index number. USEFILE and USEGROUP name the relation that is desired when they are concatenated. But curswp requires a desired partition id number, not a desired relation name.

GETFIL converts the relation name into an id number by referencing the filinfo data dictionary relation. This has one record for each data base relation. One of the fields in these records is called SCRNUM; this field holds the id number of the actual partition which is dedicated to each relation. A RECORD POINT using the relation name on an index of GROUP,FILE followed by a RECORD READ from filinfo brings in the needed id number, which is then used in a SETJCW statement to set the cursornum JCW.

But what happens the first time a named partition is used? It will have no actual partition stored in it according to the definition, but in fact it will have a valid partition on the default RELATE son process stored in it. Where does this pseudo-actual partition go? By convention, the first 10 actual partition id numbers and their storage spaces in the data segment are reserved for storing these initial values of the named partitions. The first actual partition id number to be found in filinfo.db is 11. When curswp, given an index number, maps this to an actual partition id of 0, it knows that a named partition is being used for the first time and that the index should be used as the id number for purposes of storing the initial partition values.

What happens when an actual partition is used for the first time? It will be 50 words of zeros, unsuitable for use by the DBU. Curswp checks word 48 of every actual partition it retrieves, which is the MPE id number of the RELATE son the partition is associated with. If word 48 is zero, curswp issues an rdbinitx call to initialize the partition. Rdbinitx requires that that caller provide an id code for the process he want to use; curswp reads the id code to use from the cursorproc JCW. This JCW is in turn set by GETFIL using the value of the SCRPROC field retrieved from filinfo by the same read that retrieved

SCRNUM. Developers can "distribute" relations among DBU RELATE son processes according to the distribution of SCRPROC values in filinfo. Any relations whose filinfo records have the same values will be opened on the same process.

A list of the MPE process id numbers of active RELATE sons, and of the actual partition id of the first partition opened on each son, is maintained in words 50-150 of the extra data segment. The list is required by the routines which do transaction management for the DBU. Every time a new partition is initialized via a call to rdbinitx, curswp checks to make sure if the resulting process id is on the list, adding it if necessary.

Once the partition is initialized and swapped subroutine curswp is done. Screen GETFIL then checks to see if the relation is open by attempting to SET PATH to it. If the SET fails then screen OPNFIL is called to issue an OPEN FILE statement for the relation. OPNFIL is a late addition to the DBU caused by a chronic shortage of global RIN numbers on the PMS 392 HP 3000, a system error which causes catastrophic BUILDER aborts when an OPEN FILE is issued. OPNFIL attempts to catch these aborts and allow the user to wait and try the OPEN FILE in a few seconds, after which another user will hopefully have released a RIN.

OPNFIL also does a SET INDEX %scrindx after opening the relation. The contents of the SCRINDX variable (a numeric one) are retrieved from filinfo.db along with SCRNUM and SCRPROC, allowing developers a means of specifying what index a relation is to have available other than by issuing SELECTs.

The logic of the file management subsystem can be used to serve BUILDER applications other than the DBU. The only requirement is that each application have a separate extra data segment for storage of its actual partitions. The screensys JCW is used to specify the id number of this extra data segment; it is good practice to set this JCW to the proper value for your application before every call to CURINI and CURSWP. The data segment with id 9012 is reserved for the DBU.

Having ordered the discussion so far according to motivations and requirements, let us now describe the actual operation of the file management subsystem processors in their actual order of operations. Subroutine curini is shown in Figure 8-22, screen GETFIL in Figure 8-23, routine curswp in Figure 8-24, and screen OPNFIL in Figure 8-25.

Curini's function is to create and initialize the extra data segment used for actual partition storage. Called by screen INIT, it requires only the screensys JCW value as input. The getdseg intrinsic is called and then the 48th word of each cursor is set to zero via assignments and dmovout intrinsic calls. The named partition index conversion "array" at the start of the segment is also zerod.

GETFIL assumes its input is in the USECURSOR, USEFILE, and USEGROUP variables. It first does a point/read on filinfo.db to get the actual partition id (scrnum), son process id (scrproc), and default index (scrindx). A match on USECURSOR in CURS_SWAP_LIST is found using $IN, producing an index number identifying the named cursor. The values are stored in the JCW for transfer to curswp, and curswp is called. After it returns GETFIL checks to see if the file is open via an attempted SET PATH, calling OPNFIL if this fails.

Curswp reads the JCW GETFIL just wrote to, retrieves the extra data segment, and pulls out the list of active son processes. It then uses the named partition index to retrieve the id of the actual partition stored in it, and this id in turn to store the current state of that actual partition into the data segment. Then the requested named partition is extracted from the data segment and written into the named partition. The check is made to see if the partition has been initialized; this is done via the rdbinitx call if necessary.

FIGURE 8-22. Subroutine Curini

```
C     CURINI ****************************************
$CONTROL segment=dsa,uslinit
      SUBROUTINE curini(cursor,table,pointr)
      integer cursor(50),table(1),pointr(4)
C*                                              *** ABSTRACT ***
C$PURPOSE   Initializes a set of cursors for use by the data
C     entry system.  Works in concert with CURSWP to.  Allows use
C     of multiple RELATE processes as sons of the builder.
C     Compiled code resides in SL.PUB
C$AUDIT HISTORY
C        MSCarey          30-sep-83  AUTHOR
C$FORMAL PARAMETERS
Cin      cursor   cursor array used by screen system
Cin      table    global data storage table for screen system
Cin      pointr   pointers to argument from call
C$COMMON BLOCKS
C        none
C        data segment format is:
C           location 0-9    index of cursor now in use by system
C                           by builder USE cursor id number
C           location 50-10000 by 50's: cursor data arrays
C$CALLER CRI builder application files
C$METHOD
C     Routines resident in an SL may not have global data
C     declarations.  The screen system multiple cursor facility
C     simulates global storage for the cursors by using an
C     extra data segment.  This routine initializes that data
C     segment.
C
C     JOB CONTROL WORDS
C     are currently used to communicate the id number of the cursor
C     desired, the id number of the USE cursor to be swapped into, and
C     the id number of the son process to use.
C
C     The routine does not actually initialize any cursors; this is
C     done by CURSWP when it detects a 50-word data segment area
C     which is not yet initialized.  This routine writes codes into
C     a word of each cursor area which tell CURSWP that no
C     rcinitx call has yet been done.  A 0 is placed in word 48,
C     which RELATE uses to store son process id's in.  This word
C     will never be 0 once rcinitx has been called for a cursor.
C$LOCAL VARIABLES
C        numcur   number of cursors usable by system
C        newcur,lcurs   array of cursors to be initialized and stored
C        inuse    cursor currently in use by system
C        iarg,carg,arg  argument from call in various forms
C                 remaining arguments are for intrinsic calls
C$$
```

FIGURE 8-22. Subroutine Curini

```
C*                      *** INCLUDES and LOCAL DECLARATIONS ***
      parameter numcur = 199
      integer newcur(150),inuse(1),length,id
      integer locseg,icurs,iarg(50),end
      logical linuse,lindex,lid,lcurs(150)
      character arg*100,carg*1(100)
      character jcwnam*30
      equivalence (iarg,carg,arg)
      equivalence (lcurs,newcur)
      equivalence (lid,id),(linuse,inuse)
      system intrinsic getdseg,dmovout,findjcw
C*ENDDEC                                *** END DECLARATIONS ***
C
C
C     ---Get data segment id from job control word
      jcwnam = "SCREENSYS"
      call findjcw(jcwnam,lid,istat)
C     ---get data segment
      length = numcur*50 + 150 + 1
      call getdseg(lindex,length,lid)
C
C     ---initialize cursors and put them into data segment
C     ---do cursor already set up by screen system first
C     ---main loop
      do 300 icurs=1,numcur
         locseg = icurs*50 + 101
         newcur(48) = 0
         call dmovout(lindex,locseg,50,lcurs)
300      continue
C
C     ---set the in-use markers to 0 and finish
      do 400 i=1,150
         newcur(i) = 0
400      continue
      locseg = 1
      call dmovout(lindex,locseg,150,lcurs)
C
999   RETURN
      END
```

FIGURE 8-23. Screen GETFIL

```
*** SCREEN GETFIL
*** INITIAL
    RECORD POINT USING FILINFO;KEY=USEGROUP,USEFILE
    IF NOT FOUND
         DISPLAY "Getfil error...no %usefile.%usegroup in dictionary."
         BREAK
         ENDIF
    RECORD READ USING FILINFO
    SET OPTION QUOTES=NO
    SWAPNUM := IN(USECURSOR,%curs swap list)
    SET OPTION QUOTES=YES
    :SETJCW NUMSWAP,%swapnum
    :SETJCW CURSORNUM,%scrnum
    :SETJCW CURSORPROC,%scrproc
    SELECT
    CALL PROCEDURE CURSWP
    IGNORE ALL ERRORS
    SET PATH %usefile
    IF ERROR=713
         CALL SCREEN OPNFIL
    ELSEIF ERROR=506
         SCROLL "Unable to obtain virtual memory for DB access.
                 The DBU process will be terminated.  Please try
                 again later."
         SCROLL
         PROMPT "Hit RETURN to continue...",Y
         EXIT
    ELSEIF ERROR  0
         SCROLL "Unexpected error (number %ERROR) while fetching
                 file %usefile.%usegroup.  Report this occurrence
                 to the DBA."
         SCROLL
         PROMPT "DBU terminated.  Hit any key to continue.",Y
         EXIT
    ENDIF
    RETURN SCREEN
```

FIGURE 8-24. Subroutine Curswp

```
C     CURSWP ************************************************
$CONTROL segment=dsa
      SUBROUTINE curswp(cursor,table,pointr)
C*                        *** FORMAL PR DECLARATIONS ***
      integer cursor(50),table(1),pointr(4)
C*                                              *** ABSTRACT ***
C#PURPOSE   Swaps the cursor currently in use by the screen
C     system into cursor memory and brings in the cursor
C     requested in the argument attached to the CALL PROCEDURE call
C     to this routine (NOW READS JCW).  See CURINI.  Compiled code
C     resides in SL.PUB
C#AUDIT HISTORY
C        MSCarey          30-sep-83  AUTHOR
C#FORMAL PARAMETERS
C      Currently in use by screen system
Cin      table    screen system global memory table
Cin      pointr   pointers to argument of CALL PROCEDURE
C#COMMON BLOCKS
C        none
C#CALLER CRI builder application files
C#METHOD
C
C     Retrieve the SCREENSYS, NUMSWAP, CURSORNUM, AND CURSORPROC
C     Job Control Words, which specify the cursor memory data
C     segment id, the id of the builder USE cursor to be swapped,
C     the id number of the cursor to be swapped into 'numswap',
C     and the son process id code to be given to rdbinitx if
C     'cursornum' is not yet initialized.
C
C     Get the index of the data segment, swap out the current cursor,
C     and swap in the one desired.
C#LOCAL VARIABLES
C        !        !
C##
C# INCLUDES and LOCAL DECLARATIONS ***
      parameter numcur = 199
C     integer iarg(50),loc,end,sep
C     character arg*100,carg*1(100)
      character cproc*6,jcwnam*20,inicom*30,strng*30
      integer inuse,id,iget,length,isave,iproc,inicomi(15)
      integer istrng(15),key,loc,locp
      logical lprnt,lindex,lid,linuse,lget,lsave,lproc
      integer procs(100)
      logical lprocs(100)
      equivalence (lprocs,procs)
      equivalence (lid,id),(linuse,inuse),(lget,iget)
      equivalence (lsave,isave),(lproc,iproc)
      equivalence (inicom,inicomi),(strng,istrng)
```

FIGURE 8-24. Subroutine Curswp

```
C       equivalence (iarg,arg,carg)
C       ---FUNCTIONS
        integer dltrim,drtrim
        system intrinsic fopen,fwrite,getdseg,dmovin,dmovout,findjcw
C*ENDDEC                                *** END DECLARATIONS ***
C
        lprnt = .false.
        jcwnam = "LPRNT4"
        call findjcw(jcwnam,lprnt,istat)
        if (lprnt) ifile = fopen(,%614,%1)
C
C       ---get information from job control words
        jcwnam = "SCREENSYS"
        call findjcw(jcwnam,lid,istat)
        jcwnam = "NUMSWAP"
        call findjcw(jcwnam,lsave,istat)
        if (isave.gt.10) stop "curswp: NUMSWAP jcw value too big"
        jcwnam = "CURSORNUM"
        call findjcw(jcwnam,lget,istat)
        if (iget.gt.numcur .or.iget.le.10)
       1         stop "curswp: CURSORNUM jcw value bad"
        jcwnam = "CURSORPROC"
        call findjcw(jcwnam,lproc,istat)
C       ---get data segment and list of active son processes
        length = numcur*50 + 150 +1
        call getdseg(lindex,length,lid)
        loc = 51
        call dmovin(lindex,loc,100,lprocs)
C
C       ---do the swap, checking for init requirement
C       ---note that the first 10 locations are reserved for
C          the USE cursors as they stood on first swap call
C
C       ---find out where to swap the current cursor contents to
        call dmovin(lindex,isave,1,linuse)
        if (inuse.eq.0) inuse = isave
        loc = inuse*50 + 101
        call dmovout(lindex,loc,50,cursor)
C       ---find out where to get the desired cursor from, and note that
C          this is now the current cursor for this USE
        loc = iget*50 + 101
        call dmovin(lindex,loc,50,cursor)
        call dmovout(lindex,isave,1,lget)
C       ---see if an init call must be made for the cursor
        if (cursor(48).ne.0) go to 999
           inicom = "SEGMENT="
           cproc = str(iproc,5)
           istart = dltrim(cproc,5)
           lenp = 5 - istart + 1
           inicom[9] = cproc[istart]
           len = 9 + lenp
```

FIGURE 8-24. Subroutine Curswp

```
         inicom[len] = ";SYSTEM="
         len = len+8
         inicom[len] = cproc[istart]
         len = len+lenp
         inicom[len] =          do 400 i=1,50
            cursor(i) = 0
400         continue
         if (lprnt) call fwrite(ifile,inicomi,-len,0)
         call rdbinitx(cursor,inicomi,-1)
         call dmovout(lindex,loc,50,cursor)
         INICOM = "AFTER RDBINITX CALL"
         if (lprnt) call fwrite(ifile,inicomi,-len,0)
         if (cursor(1).ne.0) call rdberror(cursor,3,key,1)
C     ---make note if this rdbinit call started a new son process
C        words 51-150 of data segment note processes in use and
C        cursors first opened on them for use in transaction
         management
C
      nproc = 0
      do 500 i=1,50
         if (procs(i).eq.0) go to 550
         nproc = nproc+1
         if (cursor(48).eq.procs(i)) go to 999
500      continue
      stop 'curswp: proc storage overflow?'
550   procs(nproc+1) = cursor(48)
      procs(nproc+51) = loc
      loc = 51
      call dmovout(lindex,loc,100,lprocs)
      strng = 'STARTING PROC'
      strng[15] = str(cursor(48),6)
      if (lprnt) call fwrite(ifile,istrng,-30,0)
999   RETURN
      END
```

FIGURE 8-25. Screen OPNFIL

```
*** SCREEN OPNFIL
*** INITIAL
    IGNORE ERROR 900
    OPEN FILE %usefile.%usegroup;MODE=SHARED
    ERRNUM := PARTITION(2)
    IF ERRNUM=0
         SET INDEX %scrindx
         RETURN SCREEN
    ENDIF
    SCROLL "I am trying to open file %usefile.%usegroup,
            but HP is so busy it won't let me.  If
            you want me to try again (you can wait a while)
            hit the return key.  If you want to give up
            using the DBU for now, type EXIT."
    SCROLL
    WHILE ERRNUM0
         PROMPT "Hit return to try again, or type EXIT:",ANSWERL
         IF ANSWERL="EXIT"
              EXIT
         ENDIF
         IGNORE ERROR 900
         OPEN FILE %usefile.%usegroup;MODE=SHARED
         ERRNUM := PARTITION(2)
    ENDWHILE
    SET INDEX %scrindx
    SCROLL "OK, we're moving again."
    RETURN SCREEN
```

OPNFIL just issues an OPEN FILE command preceded by an IGNORE ERROR command (to prevent BUILDER aborts), and followed by error checking to identify the error that comes up when there are no more global RIN's. If that error has occurred, the user is given the choice of waiting and trying the OPEN again, or terminating the DBU process.

8.4.3.3 The Generic Data Screen: How It Works and An Example

Figure 8-26 displays the DBU data screen template, a skeletal version of a data screen that should be the starting point for creation of any new data screen. Figure 8-27 displays the code of the CURR_NC_SKED data screen, a typical example. Using these two Figures, this Section will describe how data screens work.

Data screens allow users to view, change, and add data base tuples. They ensure that data to be added or updated conforms to data base standards for consistency. They enforce security, and provide utility services to the user.

A significant restriction currently imposed by the DBU is that tuples may be displayed and changed only one at a time (exceptions being comments and certain data dictionary support screens). This can make data retrieval and inspection more difficult and time consuming under some circumstances. However, most ALIAS data records have too many fields to allow more than one to be displayed in the limited area provided by a standard screen.

By its nature, each data screen is customized to support one or a few relations. The name(s) of this relation(s) and its fields are hard-wired into the screen's data structure and logic. However, most data screens are very similar in overall form since they perform similar functions. In particular, they may all be divided into layout, declarations, initialization, and command processing sections; the discussion for each section will refer implicitly to Figure 8-27.

8.4.3.3.1 Layout

Data screen layouts are designed to provide users with a maximum of information about their context and also to keep most of the screen free for data fields.

FIGURE 8-26. Text of Data Screen Template

```
*** SCREEN <screen-name >
*** LAYOUT
SCREEN IS:[now_screen     ] [modedisp            ]   SCENARIO IS:[scenarionm  ]
 ? for help              ALIAS DATA BASE UPDATE SYSTEM              =NAME jumps
                                 <subtitle>

                          COMMAND:[command         ]

                                                        Data Source[datasource]
                                                        Data Date  [datadate  ]
                                                        Entry Date [entry_date]
                                                        Entry By   [entry_by  ]
[altercapd                                                                    ]
                                 <message>

*** DECLARATIONS
    ALTERCAPD,NOW_SCREEN,SCENARIONM,DATADATE,ENTRY_DATE,ENTRY_BY ;GLOBAL
    MODEDISP ;GLOBAL
    COMMAND ;GLOBAL;ACTION
    <other fields on screen>
*** INITIAL
    SET OPTION QUOTES=YES
    COMMAND := ""
NOTE     --SECURITY
    NOW_SCREEN := "<screen-name >"
    NOW_FILE := "<main file name>"
    NOW_GROUP := "<main file group>"
    CALL SCREEN SECURITY_CHECK
    IF SEEIT=""
        PROMPT "You do not have access to that screen.  Press any key.",Y
        NEXT_SCREEN := $ITEM(PREV_SCREEN,1,1)
        SET SCREEN %next_screen
    ENDIF
NOTE     --SET UP
    EXTRACOM := <0>  true if normal data manip logic insuff, e.g. mult files
    DATESCREEN := <0> true if this screen needs planned/actual distinction
    MODEDATA := <"DATA"  or "DATES">
    DATEMODE := <SHOWMODE or "">
    MODESET := "" or " AND DATETYPE=""%datemode"" "
    MODEDISP := $CONCAT(INMODE," ",DATEMODE," ",MODEDATA)
    NOW_CURSOR := "MAIN"
    NOW_INDEX := "<fld,fld,...for use in reset select>"
```

FIGURE 8-26. Text of Data Screen Template

```
    LEGAL_FIELDS := "<fld,fld,...lsit of legal-type fields on screen>"
    LEGLPREF1 := "<#,#,...for each legal field, default value number>"
    KEYFIELDS := "<fld,fld,...list of fields which may not be blank>"
    COMT_FLD_MODNUM := "<#1-6,#,... format spec number for comt_fld_mods>"
    LOWKEY := "<last index field before datadate>"
    LOWLEN := "<declared length of lowkey in chars>"
    VERIFYFIELDS := "<fld,fld,...>"
    VERTYP := "<J or U or JU: join or uniquness or both>"
    NEXT_SCREEN := "<default next screen>"
    NEXT_HELP := "<default help screen>"
    PREV_SCREEN := $CONCAT(NOW_SCREEN,",",$ITEMS(PREV_SCREEN,1,9))
    COMT_SCREEN := "<assoc screen name>"
    COMT_FILE := "<comt file name>"
    COMT_GROUP := "<comt group name>"
    COMT_BREAK := "<# of comment key fields thru entry_date>"
    COMT_ACTIVE := ""
    CALL PROCEDURE GETSCENV %comt_file.%comt_group SCENCOM
    COMT_RESET := "@ BY <index> WHERE SCENARIO=""%scencom"" "
    DATADATE := ""
    ENTRY_DATE := ""
    ENTRY_BY := ""
    REFRESH
    DISPLAY "Fetching data files for this screen."
NOTE     ——GET FILES
    USEFILE := NOW_FILE
    USEGROUP := NOW_GROUP
    USECURSOR := NOW_CURSOR
    USE PARTITION %now_cursor
    CALL SCREEN GETFIL
    CALL PROCEDURE GETSCENV %usefile.%usegroup SCEN1
    RESET := "@ BY %now_index WHERE SCENARIO=""%scen1"" "
    SET OPTION QUOTES=NO
    SELECT %reset %modeset
    SET OPTION QUOTES=YES
    FOUND := "N"
NOTE     ——MORE SECURITY
    SCENARIO := SCEN1
    IF SCENARIO<>SCENARIONM
        UPDAT := ""
        DELET := ""
        MODFY := ""
        ADD := ""
    ENDIF
    CALL SCREEN SHOWPRIV
    <CALL SCREEN LEGALS_INIT>
    DISPLAY
*** INITIAL EVERYTIME
    COMMAND := ""
*** VARIABLE <key1>
    ABORT_MODFY := 1
    <more variable sections>
*** FUNCTION
```

FIGURE 8-26. Text of Data Screen Template

```
    DISPLAY "That ESCape function not available here."
*** FUNCTION ?
    CALL SCREEN FIELDHELP
*** FUNCTION +
    CALL SCREEN NEXT_LEGAL
*** FUNCTION -
    CALL SCREEN PREV_LEGAL
*** ENTER
    COMCHR := $SUBSTR(COMMAND,1,1)
    SCREEN_NAME := $SUBSTR(COMMAND,2)
    SET OPTION QUOTES=NO
    IF $IN(COMCHR,%datacmda)
        CALL SCREEN PROC_DATA_CMDA
        IF AKSHUN="CMD_NOT_FND"
            DISPLAY "The %command command is not operational here."
        ELSE
            SET OPTION QUOTES=NO
            IGNORE ALL ERRORS
            %akshun
            IF $ERROR=7721 OR $ERROR=7639
                DISPLAY "%screen_name is not a valid screen name."
            ENDIF
            SET OPTION QUOTES=YES
        ENDIF
    ELSEIF $IN(COMCHR,%datacmdb)
        CALL SCREEN PROC_DATA_CMDB
    ELSE
        DISPLAY "The %command command is not operational here."
    ENDIF
NOTE    +++++++++
*** SCREEN K<screen_name >
*** INITIAL
    CLEAR VARIABLE DATADATE,DATASOURCE,ENTRY_DATE,ENTRY_BY
    <more clears>
    RETURN SCREEN
```

FIGURE 8-27. Text of CURR_NC_SKED Data Screen

```
*** SCREEN CURR_NC_SKED
*** LAYOUT
SCREEN IS:[now_screen     ] [modedisp           ]    SCENARIO IS:[scenarionm  ]
═══════════════════════════════════════════════════════════════════════════════
 ? for help            ALIAS DATA BASE UPDATE SYSTEM              =NAME jumps
═══════════════Current New Construction Schedules═══════════════════════════════

                        COMMAND:[command         ]
       ship                                          job description
    Name   [shipname                    ]            Shipyard [yard    ]
    Class  [class     ]                              Job Type [ncjobt]
    Hull   [hull]                                    Seq Type [jstyp ]
    Jobnum [comnum]                                  C Method [cmethd]
        dates                                        Customer [customer]
    Approp [approp    ]
    Award  [award     ]
    Start  [start     ]
    Keel   [keel      ]                              Actual/Proj[datetype  ]
    Launch [launch    ]                              Data Source[datasource]
    Deliv  [delivery  ]                              Data Date  [datadate  ]
    Commis [commission]                              Entry Date [entry_date]
    Days Added to Life [days]                        Entry By   [entry_by  ]
[altercapd                                                                   ]
              Place a command after COMMAND and press RETURN

*** DECLARATIONS
    ALTERCAPD,NOW_SCREEN,SCENARIONM,INMODE,DATADATE,ENTRY_DATE,ENTRY_BY ;GLOBAL
    MODEDISP ;GLOBAL
    COMMAND ;GLOBAL;ACTION
    NEXT_HELP ;LENGTH=15;UPPER
    APPROP,AWARD,START,KEEL,LAUNCH,DELIVERY,COMMISSION;DATE="MM/DD/CCCC"
    DAYS ;NAME=DAYSADDED;NUMERIC
    HULL,COMNUM ;NUMERIC;ENHANCE=I,U
    CLASS,YARD ;UPPER;ENHANCE=I,U
    DATETYPE,NCJOBT,JSTYP,CMETHD,CUSTOMER;UPPER;ENHANCE=U
*** INITIAL
    SET OPTION QUOTES=YES
    COMMAND := ""
NOTE     ---SECURITY
    NOW_SCREEN := "CURR_NC_SKED"
    NOW_FILE := "NCJODAT"
    NOW_GROUP := "CURRJ"
    CALL SCREEN SECURITY_CHECK
    IF SEEIT=""
        PROMPT "You do not have access to that screen.  Press any key.",Y
        NEXT_SCREEN := $ITEM(PREV_SCREEN,1,1)
        SET SCREEN %next_screen
    ENDIF
NOTE     ---SET UP
    EXTRACOM := 0
    DATESCREEN := 1
    MODEDATA := "DATES"
```

FIGURE 8-27. Text of CURR_NC_SKED Data Screen

```
     DATEMODE := SHOWMODE
     MODESET := "[ AND DATETYPE=""%datemode"" ]"
     MODEDISP := $CONCAT(INMODE," ",DATEMODE," ",MODEDATA)
     NOW_CURSOR := "MAIN"
     NOW_INDEX := "SCENARIO,CLASS,HULL,COMNUM,DATADATE:D,ENTRY_DATE:D"
     LEGAL_FIELDS := "DATETYPE,NCJOBT,JSTYP,CMETHD,CUSTOMER"
     LEGLPREF1 := "1,1,1,1,1"
     KEYFIELDS := "CLASS,HULL,COMNUM,DATETYPE,DATADATE"
     COMT_FLD_MODNUM := "6,7,7,9,4"
     LOWKEY := "COMNUM"
     LOWLEN := "-1"
     VERIFYFIELDS := "SCENARIO,CLASS,HULL,COMNUM"
     VERTYP := "JU"
     NEXT_SCREEN := ""
     NEXT_HELP := "C_NC_SKED_HELP"
     PREV_SCREEN := $CONCAT(NOW_SCREEN,",",$ITEMS(PREV_SCREEN,1,9))
     COMT_SCREEN := "CURR_NC_SKED_C"
     COMT_FILE := "NCJOCOM"
     COMT_GROUP := "CURRJ"
     COMT_BREAK := "7"
     COMT_ACTIVE := ""
     CALL PROCEDURE GETSCENV %comt_file.%comt_group SCENCOM
     COMT_RESET := "@ BY %now_index,DATETYPE,LNUM WHERE SCENARIO=""%scencom"" "
     DATADATE := ""
     ENTRY_DATE := ""
     ENTRY_BY := ""
     REFRESH
     DISPLAY "Fetching data files for this screen."
NOTE      --GET FILES
     USEFILE := NOW_FILE
     USEGROUP := NOW_GROUP
     USECURSOR := NOW_CURSOR
     USE PARTITION %now_cursor
     CALL SCREEN GETFIL
     CALL PROCEDURE GETSCENV %usefile.%usegroup SCEN1
     RESET := "@ BY %now_index WHERE SCENARIO=""%scen1"" "
     SET OPTION QUOTES=NO
     SELECT %reset %modeset
     SET OPTION QUOTES=YES
     FOUND := "N"
     ABORT_MODFY := 0
NOTE      --MORE SECURITY
     SCENARIO := SCEN1
     IF SCENARIO<>SCENARIONM
          UPDAT := ""
          DELET := ""
          MODFY := ""
          ADD := ""
     ENDIF
     CALL SCREEN SHOWPRIV
      CALL SCREEN LEGALS_INIT
     IF DATEMODE <> ""
```

FIGURE 8-27. Text of CURR_NC_SKED Data Screen

```
        DATETYPE := DATEMODE
    ENDIF
    DISPLAY
*** INITIAL EVERYTIME
    COMMAND := ""
*** VARIABLE CLASS
    ABORT_MODFY := 1
*** VARIABLE HULL
    ABORT_MODFY := 1
*** VARIABLE COMNUM
    ABORT_MODFY := 1
*** VARIABLE DATADATE
    ABORT_MODFY := 1
*** FUNCTION
    DISPLAY "That ESCape function not available here."
*** FUNCTION ?
    CALL SCREEN FIELDHELP
*** FUNCTION +
    CALL SCREEN NEXT_LEGAL
*** FUNCTION -
    CALL SCREEN PREV_LEGAL
*** ENTER
    COMCHR := $SUBSTR(COMMAND,1,1)
    SCREEN_NAME := $SUBSTR(COMMAND,2)
    SET OPTION QUOTES=NO
    IF $IN(COMCHR,%datacmda)
        CALL SCREEN PROC_DATA_CMDA
        IF AKSHUN="CMD_NOT_FND"
            DISPLAY "The %command command is not operational here."
        ELSE
            SET OPTION QUOTES=NO
            IGNORE ALL ERRORS
            %akshun
            IF $ERROR=7721 OR $ERROR=7639
                DISPLAY "%screen_name is not a valid screen name."
            ENDIF
            SET OPTION QUOTES=YES
            CALLED := "N"
        ENDIF
    ELSEIF $IN(COMCHR,%datacmdb)
        CALL SCREEN PROC_DATA_CMDB
    ELSE
        DISPLAY "The %command command is not operational here."
    ENDIF
NOTE    +++++++++
*** SCREEN KCURR_NC_SKED
*** INITIAL
    ABORT_MODFY := 0
    DATADATE := ""
    DATASOURCE := ""
    ENTRY_DATE := ""
    ENTRY_BY := ""
```

FIGURE 8-27. Text of CURR_NC_SKED Data Screen

```
CLASS := ""
HULL := ""
COMNUM := ""
YARD := ""
NCJOBT := ""
JSTYP := ""
CUSTOMER := ""
SHIPNAME := ""
CMETHD := ""
APPROP := ""
AWARD := ""
START := ""
KEEL := ""
LAUNCH := ""
DELIVERY := ""
COMMISSION := ""
DAYSADDED := ""
DATETYPE := ""
RETURN SCREEN
```

The top line of the layout tells the user the name of the current screen, the current display/retrieval mode (e.g. "LATEST DATES"), and the name of the current scenario. The third line reminds the user that he is using the DBU, and provides "tickler" help by giving the command which brings up the help subsystem and one of the commands to move to a different screen. The fourth line has a (hopefully) informative screen title.

Every DBU screen has a COMMAND field just below the screen title, in which the user places command characters. At the bottom of the screen the "altercapd" field (display only, appears as text to the user) tells the user his data change privileges in this screen. The bottom line has a message which further explains what to do in the screen.

The area of the layout between the COMMAND field and the bottom two lines is reserved for data fields and their labels. In the example these include "shipname", "customer", etc. Conventionally, the datasource, datadate, entry_date, and entry_by fields appear in the bottom right hand corner. The entry_date and entry_by fields are for use in audits by the DBA and are thus display-only.

Screen enhancements do not show on the paper copy of Figure 8-27, but in fact a data screen typically contains fields with a variety of enhancements. Conventions are that "regular" fields are in inverse video, legal-value fields are underlined, and key fields (those for which join/uniqueness checks will be made) are both underlined and inverse video. Fields the user cannot change are not enhanced.

8.4.3.3.2 Declarations

A data screen has access to the global DBU variables, but also requires a local data structure. Most of this structure consists of local variables which appear on the layout.

The first three lines of CURR_NC_SKED's declarations section tell BUILDER which variables appearing in the layout are global. BUILDER assumes all layout variables are local unless told otherwise; the ones shown are global in the DBU either because they actually hold global data (e.g., scenarionm) or because they are common to many screens and thus declared global to reduce overall DBU processing time.

The rest of the declarations specify the data types and enhancements of the data variables on the layout, enabling BUILDER to perform some type and range checking automatically. If a layout variable is not mentioned in the declarations section BUILDER assumes it to be an optional field (inverse video) of alphanumeric type.

8.4.3.3.3 Initialization

Before the user may perform any action in a data screen, the DBU must:

1) Do security checks to find out if the user has access to the screen itself and to the data it manages;

2) Fill in any informative variables on the layout, and set variables which tell general-purpose utilities what screen they are serving and how to serve it;

3) Retrieve the relations which hold the data the screen manages, via calls to the file management system;

4) Set up scenario security;

5) Initialize any legals-type variables on the layout.

This is all done in the initialization section. In the example in Figure 8-27, the initial security check is performed by the code between "---SECURITY" and "---SET UP". First, the name of the current screen and relation is placed in three global variables. These are read by the general-purpose SECURITY_CHECK subroutine screen, which queries the data dictionary and sets the SEEIT (can he see this screen?) flag as well as the ADD, UPDAT, MODFY, and DELET data modification privilege flags. If the user

is not allowed to use the screen, the name of the last screen used is retrieved from the "history" in the PREV_SCREEN variable and a SET back to it is performed.

Part 2 of initialization is the most complex; it involves setting a series of variables which fully describe the screen to all DBU subsystems. If these are set incorrectly, DBU errors are likely to result. This set-up is necessary since utility screens will be used to carry out most user requests, and these screens must have control information.

The variables and their meaning, in order of appearance in Figure 8-27, are as follows:

EXTRACOM A boolean which tells PROC_DATA_CMDB whether to clear the layout after a deletion has been done (extracom=0) or to leave this final step to logic in the data screen itself (extracom=1). The latter is usually necessary only when the data screen is serving multiple relations (see screen CLASS_CHARS for an example of this).

DATESCREEN A boolean which tells whether this screen contains schedule data (datescreen=1 if so). Alerts the retrieval mode subsystem to the need for a PLANNED/ACTUAL distinction.

MODEDATA A companion to datescreen which gives the type of data on the screen, either "DATES" or "DATA". Used by the retrieval mode subsystem.

DATEMODE Takes on either the value of SHOWMODE ("ACTUAL", "PLANNED", or "ALL") if this is a date screen, or is blank. SHOWMODE is global storage for this retrieval mode subsystem control value, retaining it even when a date screen is not being used. DATEMODE is "local" storage in that it must be set in each screen; it is used implicitly during path resets (re-SELECTS) by various utility screens.

MODESET Blank if DATEMODE is blank, for date screens this holds an addendum to path SELECTion WHERE clauses which ensures only that planned or actual data is retrieved if either is the current mode setting.

MODEDISP One of the variables displayed at the top of the layout, this tells the user the current settings being used by the retrieval mode subsystem.

NOW_CURSOR The name of the partition which will "contain" the primary data relation for the screen. Almost always set to "MAIN".

NOW_INDEX Field names in the index which is to be used for retrievals on the current relation. The text in this variable is substituted into the BY clauses of SELECT statements. Make sure the index specified is a permanent one or no data updates will be allowed!

LEGAL_FIELDS List of variables appearing in the layout whose values as entered by the user must conform to the standards of the legal values reference library. Every field on this list must be noted as a legals-type field in the data dictionary and must have a relation giving its legal values in the .legals group.

LEGLPREF1 Every legal value in a legals relation has an associated ordering number (a SHOWPREF number). These give the order in which the values will appear when the ESC + command option is used to "flip" through them on a data screen. LEGLPREF1 must have a list of SHOWPREF values corresponding to the LEGAL_FIELDS name list; these values specify the default value of each legal field, which will be filled into the field during the last phase of data screen initialization. For example, the NCJOBT (new construction job type) variable on the CURR_NC_SKED screen can have values of "NEWCON", "CONV", or "REACT"; these values have associated values of 1, 2, and 3 respectively. By specifying a "1" as the default value indicator in LEGLPREF1, the CURR_NC_SKED creator mandated that the value "NEWCON" would be filled in on the CURR_NC_SKED layout when the screen is initialized.

KEYFIELDS A list of the names of fields which must be filled in before the comment screen associated with the data screen can be used. The comment screen associates data records and comment records according the the values of key fields, including the ENTRY_DATE field value. KEYFIELDS must contain a list of the fields on the data screen whose values will uniquely identify any given data record, but should not include "ENTRY_DATE".

COMT_FLD_MODNUM A list of enhancement codes which correspond to the names in the KEYFIELDS list. The codes refer to elements in the COMT_FLD_MODS array (see screen INIT or Section 8.4.4.1.1). Data screen key fields always appear on the associated comment screen, but in a display-only form. The comment screen changes the variables to this form using MODIFY VARIABLE during initializion; when it returns, the comment screen must change the variables back. To do this, it must know their type/enhancement specification, which is indirectly given in the comt_fld_modnum list.

LOWKEY The name of the last field in NOW_INDEX before "DATADATE". The record retrieval utility must know this name in order to function in "LATEST" retrieval mode.

LOWLEN The retrieval utility must also know the data type of the "lowkey" field. LOWLEN should be set to a numeric code which is the length of the field in characters if it is of type alpha, -1 or -2 if of type numeric, and -3 if of type date.

VERIFYFIELDS A list of the fields which uniquely identify a given entity in the relation served by this screen, as opposed to a given data record, which generally requires that DATADATE and ENTRY_DATE be specified as well. The list is used by the uniqueness check logic to see if data for the given entity (e.g., individual ship) has been entered already. If so, ADD is not appropriate; UPDATE is.

VERTYP A one-or two-letter code specifying the kind of data validation checks to be performed before adds, modifies, or updates are posted. "J" indicates join checks, "U" uniqueness checks, and "JU" both.

NEXT_SCREEN If there is a logical next screen to work with after the user is finished in this one, place the name of that screen here. Otherwise the variable should be set to "".

NEXT_HELP The name of the help screen associated with this data screen.

PREV_SCREEN holds a list of the last ten screens used (data and menu screens only). This allows the user to retrace his steps through the DBU using the ^ command. The name of the current screen must be added to this list.

COMT_SCREEN The name of the comment screen which serves this data screen.

COMT_FILE and COMT_GROUP The name of the relation which stores comments having to do with this screen's data.

COMT_BREAK The control break level to be used in comment screen RECORD POINTS so that a simulated end-of-file will occur when all comments having to do with the current data record have been read. Set this variable to the ASCII version of the number of fields in NOW_INDEX plus any additional fields which appear in the COMT_RESET index before (below) the LNUM field.

COMT_ACTIVE A flag which the comment subsystem uses to indicate that comments for the current data record are

already in memory. This must always be .false. ("") during data screen initialization.

COMT_RESET The text of the SELECT statement which will be used to set up the retrieval path for the comment relation the associated comment screen will use.

Note also that the fortran procedure GETSCENV is called to retrieve the scenario key field value for the comment relation for the current scenario, and that this value is substituted into the COMT_RESET SELECT statement text.

Following this set-up, the three global data variables DATADATE, ENTRY_DATE, and ENTRY_BY are blanked and the layout is drawn on the screen for the user's inspection while initialization completes.

The primary data relation(s) needed by the screen are then retrieved by calls to file management system routines, which include the set-up and call to the GETFIL screen, retrieval of the proper scenario key field value via a call to GETSCENV, and set-up and execution of the selection which prepares the relation properly for retrieval requests.

Note particularly that any unusual conditions on this selection should be inserted in the RESET variable assignment line and not in the SELECT statement itself, since utility screens often find it necessary to re-issue this selection during their processing. Putting the conditions into the SELECT line rather than the RESET: = line will lead to issuance of erroneous selections.

At this point an additional security check must be made. If the scenario key field value retrieved by the GETSCENV call for the primary data relation returned, a value not equal to the name of the current scenario, the user is only allowed indirect access to the scenario's data. His data modification privilege flags must be reset. After this check his privileges can be

written into the ALTERCAPD variable on the layout; this is done by a call to the SHOWPRIV utility screen.

Legal field default values can then be filled in on the layout by a call to LEGALS_INIT. Finally, in this particular screen, the DATETYPE field on the layout is filled in to agree with the retrieval mode currently in use.

8.4.3.3.4 Command Processing

Most data screen command processing is done by the PROC_-DATA_CMDA and PROC_DATA_CMDB utility screens, which are called in the ENTER section of a data screen. The first character in the COMMAND field is stripped off and placed in the COMCHR field; the logic then looks for a match in the DATACMDA and DATACMDB command lists to decide which screen to call. PROC_DATA_CMDB handles the retrieval/update/delete commands, while PROC_DATA_CMDA handles the service commands. PROC_DATA_CMDA often returns a line of BUILDER code in the AKSHUN variable, which is then executed by the data screen (e.g. a SET SCREEN command).

BUILDER "function" commands, i.e. those prefaced by pressing the ESCAPE key, are captured in function sections of the data screen. These in turn call subroutine screens to do the actual processing. Examples in Figure 8-27 are FUNCTION ? for field help requests and FUNCTION + for legal value display requests.

Note finally that every data screen has associated with it a utility screen whose purpose is to clear all the variables on the layout, there being no BUILDER command that can do this with precision. In the example this screen is called KCURR_NC_SKED.

8.4.3.3.5 Underlying DB Update Logic and Its Limitations

Data screens also require a separate VARIABLE section for each variable named on the KEYFIELDS list (see "SET-UP" in the

previous section). These support the update logic by detecting user changes to key field values.

The user has four commands at his disposal for making changes to the contents of a main data relation, these being Add, Delete, Modify, and Update. Delete causes the DBU to find and delete the record displayed on the screen, along with any subsidiary records in other relations. Add and Update both lead to issuance of a RECORD ADD command; they differ in how uniqueness checks are applied as part of data validation. If uniqueness checks are done (as they almost always are) then Add works only if the given data item does not yet appear in the relation; that is, only if a SELECT which attempts to find a record with values for the fields listed in VERIFYFIELDS matching the values on the screen fails. Update works only if this SELECT succeeds (i.e. the item is already there) but if no record with the data date on the screen exists.

The distinction is made so users will be informed if they think they are adding data that is not already there but are mistaken; at the same time, they are able to enter new data for an item already in the data base while retaining the old data as a "track record".

The Modify command overwrites the record in the relation with the changed data from the screen. This command works only if the user has not changed any of the key fields on the screen (as indicated by a zero value of the ABORT_MODFY flag, the variable which is set in VARIABLE sections). The update logic must use the key values on the layout to find its way back to the record it read when it first displayed the data; if users were allowed to change the values and then do modifies, they could overwrite records other than the intended one.

8.4.3.4 The Generic Comment Screen: How It Works and An Example

Figures 8-28 and 8-29 display the DBU comment screen template and an example of a comment screen (CURR_NC_SKED_C, the companion to the data screen used as the example in the last section). This section will describe the purpose and operation of comment screens, using the contents of these two Figures as examples.

Comment screens give users the capability to enter textual comments pertaining to each data record they work with in a data screen. This allows information to be stored and retrieved which would not otherwise have a place in the data base. Each comment screen is implicitly attached to a particular data screen, and is accessible to the user only through the "C" command given in that data screen.

To provide this capability, a comment screen must do several things:

1) Provide a layout suitable for text entry.
2) Store and retrieve the textual data.
3) Know what textual data is associated with the particular data record which is current in the associated data screen.
4) Display enough information from the data screen to remind the user what it is he is commenting on.

8.4.3.4.1 Comment Screen Layouts

All comment screens have layouts very similar to the one appearing in Figure 8-29. The usual system status and help information is displayed at the top (e.g. current scenario name), a (hopefully) helpful message at the bottom, and a COMMAND field. A nine-line area is reserved for the comment text, with a line number labeling each line (the dnum field). Between the command field and the text area the values of the key fields from the associated data screen appear. They are display-only here (user may not change), appearing as a reminder of the identity of the data item the user is commenting on.

FIGURE 8-28. Text of Comment Screen Template

```
*** SCREEN <screen-name >
*** LAYOUT
SCREEN IS:[comt_screen    ] [modedisp            ]   SCENARIO IS:[scenarionm  ]
═══════════════════════════════════════════════════════════════════════════════
ESC L/D inserts/deletes  ALIAS DATA BASE UPDATE SYSTEM          + or - to page
═══════════════════════<subtitle>═══════════════════════════════════════════════

                        COMMAND:[command        ]
                                                  Data Date  [datadate  ]
                                                  Entry Date [entry_date]

                              Comments
  [dnum] [dcomment                                                    ]
  [dnum] [dcomment                                                    ]
  [dnum] [dcomment                                                    ]
  [dnum] [dcomment                                                    ]
  [dnum] [dcomment                                                    ]
  [dnum] [dcomment                                                    ]
  [dnum] [dcomment                                                    ]
  [dnum] [dcomment                                                    ]
  [more]                                                Entry By    [entry_by  ]
 Changes to comments will be posted on finish with this item in data screen.

*** DECLARATIONS
    ALTERCAPD,SCENARIONM,INMODE,DATADATE,ENTRY_DATE,ENTRY_BY ;GLOBAL
    COMT_SCREEN,MODEDISP ;GLOBAL
    COMMAND ;GLOBAL;ACTION
    DCOMMENT ;ALPHA
    DNUM ;NUMERIC;DISPLAY;JUSTIFY=RIGHT
    NOCHANGE ;NUMERIC
    MORE ;GLOBAL
    <other fields on screen>
*** INITIAL
    SET OPTION QUOTES=YES
    COMMAND := ""
    IF CALLED<>"Y"
         SCROLL "Comment screens may be entered only using C in data screen."
         SET SCREEN MASTER
    ENDIF
    INUM := 1
    FLDNAM := $ITEM(KEYFIELDS,INUM,1)
    WHILE FLDNAM<>""
         IF %fldnam=""
              SCROLL "You must fill in all key fields (%keyfields) before &
                      you may make any comments."
              RETURN SCREEN
         ENDIF
         INUM := INUM+1
         FLDNAM := $ITEM(KEYFIELDS,INUM,1)
    ENDWHILE
    USE PARTITION COMMENT
```

FIGURE 8-28. Text of Comment Screen Template

```
    SAVE_CURSOR := NOW_CURSOR
    SAVE_SCREEN := NOW_SCREEN
    SAVE_RESET := RESET
    SCENARIO := SCENCOM
    INUM := 1
    FLDNAM := $ITEM(KEYFIELDS,INUM,1)
    WHILE FLDNAM<>""
         MODIFY VARIABLE %fldnam;GLOBAL;DISPLAY
         INUM := INUM+1
         FLDNAM := $ITEM(KEYFIELDS,INUM,1)
    ENDWHILE
    RESET := COMT_RESET
    NOW_CURSOR := "COMMENT"
    REFRESH
    CALL SCREEN COMT_INIT
    DISPLAY
*** INITIAL EVERYTIME
    COMMAND := ""
*** VARIABLE DCOMMENT
    COMT_ALTERED := "Y"
*** FUNCTION
    DISPLAY "That ESCape function not available here."
*** FUNCTION D
    CALL SCREEN DEL_COMT_LINE
*** FUNCTION I
    CALL SCREEN INS_COMT_LINE
*** ENTER
    CALL SCREEN PROC_COMT_CMD
    IF AKSHUN="CMD_NOT_FND"
         DISPLAY "The %command command is not available here."
    ELSE
         SET OPTION QUOTES=NO
         IGNORE ALL ERRORS
         %akshun
         IF $ERROR=7721 OR $ERROR=7639
              DISPLAY "%screen_name is not a valid screen name."
         ENDIF
         SET OPTION QUOTES=YES
    ENDIF
```

FIGURE 8-29. Text of CURR_NC_SKED_C Comment Screen

```
*** SCREEN CURR_NC_SKED_C
*** LAYOUT
SCREEN IS:[comt_screen    ] [modedisp           ]    SCENARIO IS:[scenarionm  ]
═══════════════════════════════════════════════════════════════════════════════
ESC L/D inserts/deletes  ALIAS DATA BASE UPDATE SYSTEM          + or - to page
═══════════Current New Construction Schedule Comments═══════════════════════

                          COMMAND:[command        ]
  Class  [class     ]                                  Data Date  [datadate  ]
  Hull   [hull]           Datetype[datetype  ]         Entry Date [entry_date]
  Jobnum [comnum]
                                Comments
  [dnum] [dcomment                                                 ]
  [dnum] [dcomment                                                 ]
  [dnum] [dcomment                                                 ]
  [dnum] [dcomment                                                 ]
  [dnum] [dcomment                                                 ]
  [dnum] [dcomment                                                 ]
  [dnum] [dcomment                                                 ]
  [dnum] [dcomment                                                 ]
  [dnum] [dcomment                                                 ]
  [more]                                           Entry By   [entry_by  ]
[altercapd                                                                 ]
  Changes to comments will be posted on finish with this item in data screen.

*** DECLARATIONS
   ALTERCAPD,COMT_SCREEN,SCENARIONM,INMODE,DATADATE,ENTRY_DATE,ENTRY_BY ;GLOBAL
    MODEDISP ;GLOBAL
    COMMAND ;GLOBAL;ACTION
    DCOMMENT ;ALPHA
    DNUM ;NUMERIC;DISPLAY;JUSTIFY=RIGHT
    NOCHANGE ;NUMERIC
    MORE,CLASS,HULL,COMNUM,DATETYPE ;GLOBAL
*** INITIAL
    SET OPTION QUOTES=YES
    COMMAND := ""
    IF CALLED<>"Y"
         SCROLL "Comment screens may be entered only using C in data screen."
         SET SCREEN MASTER
    ENDIF
    INUM := 1
    FLDNAM := $ITEM(KEYFIELDS,INUM,1)
    WHILE FLDNAM<>""
         IF %fldnam=""
              SCROLL "You must fill in all key fields (%keyfields) before &
                      you may make any comments."
              RETURN SCREEN
         ENDIF
         INUM := INUM+1
         FLDNAM := $ITEM(KEYFIELDS,INUM,1)
    ENDWHILE
    USE PARTITION COMMENT
```

FIGURE 8-29. Text of CURR_NC_SKED_C Comment Screen

```
    SAVE_CURSOR := NOW_CURSOR
    SAVE_SCREEN := NOW_SCREEN
    SAVE_RESET := RESET
    SCENARIO := SCENCOM
    INUM := 1
    FLDNAM := $ITEM(KEYFIELDS,INUM,1)
    WHILE FLDNAM<>""
         MODIFY VARIABLE %fldnam;GLOBAL;DISPLAY
         INUM := INUM+1
         FLDNAM := $ITEM(KEYFIELDS,INUM,1)
    ENDWHILE
    RESET := COMT_RESET
    NOW_CURSOR := "COMMENT"
    REFRESH
    CALL SCREEN COMT_INIT
    DISPLAY
*** INITIAL EVERYTIME
    COMMAND := ""
*** VARIABLE DCOMMENT
    COMT_ALTERED := "Y"
*** FUNCTION
    DISPLAY "That ESCape function not available here."
*** FUNCTION D
    CALL SCREEN DEL_COMT_LINE
*** FUNCTION I
    CALL SCREEN INS_COMT_LINE
*** ENTER
    CALL SCREEN PROC_COMT_CMD
    IF AKSHUN="CMD_NOT_FND"
         DISPLAY "The %command command is not available here."
    ELSE
         SET OPTION QUOTES=NO
         IGNORE ALL ERRORS
         %akshun
         IF $ERROR=7721 OR $ERROR=7639
              DISPLAY "%screen_name is not a valid screen name."
         ENDIF
         SET OPTION QUOTES=YES
    ENDIF
```

8.4.3.4.2 Paging and Text Management: the Comment Screen Data Structure

Although the layout contains only nine lines for comment text, users may in fact enter and retrieve up to 24 lines. The layout area is managed as a window on a memory buffer which contains the actual text. The 'more' field on the layout indicates whether there is more text in the buffer 'below' that which is displayed, while the line numbers appearing on the layout indicate the relative position in the buffer of the text that is being shown. The user may 'page', i.e. move the position of the window on the buffer up and down, using the + and - commands. He may delete lines and insert blank lines using the ESC D and ESC L commands, which change both the window and the buffer.

The buffer consists of a 24-element array of 65-character alphanumeric-type elements called 'comment', which is declared as a part of the global DBU data structure. The window is the nine-element array (dcomment) declared as local on the comment screen layout. Whenever paging is requested, what actually takes place is saving of the window contents to the buffer, followed by moving of the desired lines from the buffer into the window array.

It is important to understand that comments remain in the buffer even when the user leaves the comment screen. Changes are posted to the data base only when the user finishes with the given data item in the data screen, or when the user forces posting with the COMMIT command in the comment screen. The user may therefore move between the data and comment screens when working on a particular data item with minimal processing delays.

8.4.3.4.3 Text Storage and Retrieval

Comment text related to data in a given relation is stored in a second relation which has a similar key and index structure.

When a comment screen is called up for a given data item (the current data record displayed in the associated data screen) it uses the values appearing in key fields on the data screen to do a POINT/READ type of retrieval from the comment relation. For example, if a record from the data relation can be uniquely identified by the values of the SCENARIO,YARD,DATADATE,ENTRY_DATE fields, then the associated comment relation will be indexed by SCENARIO,YARD,DATADATE,ENTRY_DATE,LNUM, where lnum serves to order the comment lines. A comment text retrieval in this case would involve issuance of a RECORD POINT;BREAK=4 command followed by RECORD READs in an array loop. Since the BREAK option of RECORD POINT is used BUILDER will stop the read loop with a simulated end-of-file as soon as one of the first four key field values changes.

Updates involve a similar strategy, involving a combination of RECORD UPDATEs, RECORD DELETEs, and RECORD ADDs in addition to the POINT/READs in order to ensure that only the updated text appear in the storage relation.

By convention, comment storage relations always have the text field named 'COMMENT', which makes transfer of text to and from the DBU text buffer transparent.

8.4.3.4.4 Comment Screen Initialization

Given these features, comment screen initialization involves:

1) Checking to make sure the user has filled in values for all the key fields.

2) The values of certain screen-dependent global variables which are consulted by the utility subroutines must be saved (so their settings for the data screen are not lost) and values appropriate for the comment screen assigned to them. The SAVE_ variables are storage for the data screen's values.

3) Changing the status of the key fields to display-only. They must be global variables if their values are to appear, i.e. the variables placed in DBU memory by the

data screen must be used, but these will not be display-only since the user must be able to change their values in the data screen. The MODIFY VARIABLE command is used to temporarily change them to display-only while the comment screen is active; it is used again in conjunction with the COMT_FLD_MODNUM variable declaration specification array to change them back to their original status just before a RETURN SCREEN to the data screen is issued (this latter processing takes place in the PROC_COMT_CMD subroutine screen).

4) Comments must be read into memory from the storage relation and/or displayed. This processing is done by the COMT_INIT utility screen.

8.4.3.4.5 Comment Screen Command Processing

Comment screen command processing is organized in a manner similar to that for data screens. Action commands (those entered in the COMMAND field) are referred to the PROC_COMT_CMD utility screen, which (sometimes) returns an executable command in the AKSHUN variable. Function commands are handled by utility screens also; the only function commands typically used in a comment screen are for comment line insertion and deletion. Finally, a VARIABLE section detects changes to the comment text and sets a flag (COMT_ALTERED) so that the changes are saved later.

It is important to note that comment screens are executed by CALL SCREEN rather than SET SCREEN statements; thus they are exitable only by RETURN SCREEN statements. The = screen selection command cannot work in this context since it depends on use of SET's to exit the current screen.

8.4.3.5 The Generic Help Screen: How It Works and An Example

Figures 8-30 and 8-31 display the DBU help screen template and an example of a help screen (C_NC_SKED_HELP, the companion to the data screen used as the example above). This section will describe the purpose and operation of help screens, using the contents of these two Figures as examples.

FIGURE 8-30. Text of Help Screen Template

```
*** SCREEN <screen-name >
*** LAYOUT
SCREEN IS: <screen-name >                              SCENARIO IS:[scenarionm ]
═══════════════════════════════════════════════════════════════════════════════
 ? for more help      ALIAS DATA BASE UPDATE SYSTEM HELP      / to leave help
═══════════════════════════<subtitle>═══════════════════════════════════════════

                          COMMAND:[command          ]









═══════════════════════════no data may be changed here═══════════════════════════
            Please place a command after COMMAND and press RETURN

*** DECLARATIONS
    SCENARIONM ;GLOBAL
    COMMAND ;GLOBAL;ACTION
    NEXT_HELP ;LENGTH=80;UPPER
*** INITIAL
    SET OPTION QUOTES=YES
    NEXT_HELP := "<default help screen>"
*** INITIAL EVERYTIME
    IF COMMAND="/"
        RETURN SCREEN
    ENDIF
    COMMAND := ""
*** FUNCTION
    DISPLAY "That ESCape function isn't available here."
*** ENTER
    CALL SCREEN PROC_HELP_CMD
    IF AKSHUN="CMD_NOT_FND"
        DISPLAY "The %command command is not operational here."
    ELSE
        SET OPTION QUOTES=NO
        IGNORE ALL ERRORS
        %akshun
        IF $ERROR=7721 OR $ERROR=7639
            DISPLAY "%screen_name is not a valid screen name."
        ENDIF
        SET OPTION QUOTES=YES
    ENDIF
```

FIGURE 8-31. Text of C_NC_SKED_HELP Help Screen

```
*** SCREEN C_NC_SKED_HELP
*** LAYOUT
SCREEN IS: C_NC_SKED_HELP                          SCENARIO IS:[scenariomm  ]
═══════════════════════════════════════════════════════════════════════════
 ? for more help      ALIAS DATA BASE UPDATE SYSTEM HELP      / to leave help
═══════════════════════what screen CURR_NC_SKED is for═══════════════════════

                        COMMAND:[command          ]

   Screen CURRent New Construction SchEDules allows inspection and updating
of the historical new construction jobs schedule file.  This file contains
schedule information for all ship construction, reactivation, and conversion
jobs which have been awarded but which have not delivered.  For a given
ship, it may contain both actual data and projections made before and
during the construction process.

   The main key fields are class, hull, and job number; these serve to
uniquely identify a ship.  Note that job number is equivalent to commissioning
number: it will always be '1' except for reactivations.  In combination with
the data date field, these keys uniquely identify a ship schedule data record.

═══════════════════════no data may be changed here═══════════════════════
          Please place a command after COMMAND and press RETURN

*** DECLARATIONS
    SCENARIOMM ;GLOBAL
    COMMAND ;GLOBAL;ACTION
    NEXT_HELP ;LENGTH=80;UPPER
*** INITIAL
    SET OPTION QUOTES=YES
    NEXT_HELP := "COMMANDHELPL"
*** INITIAL EVERYTIME
    IF COMMAND="/"
        RETURN SCREEN
    ENDIF
    COMMAND := ""
*** FUNCTION
    DISPLAY "That ESCape function isn't available here."
*** ENTER
    CALL SCREEN PROC_HELP_CMD
    IF AKSHUN="CMD_NOT_FND"
        DISPLAY "The %command command is not operational here."
    ELSE
        SET OPTION QUOTES=NO
        IGNORE ALL ERRORS
        %akshun
        IF $ERROR=7721 OR $ERROR=7639
            DISPLAY "%screen_name is not a valid screen name."
        ENDIF
        SET OPTION QUOTES=YES
    ENDIF
```

The sole purpose of help screens is display of text to tell the user what the DBU's capabilities are and what to do in a particular context. There are two varieties of help screen: those which offer guidance to use of the system as a whole, and those dedicated to explaining what a particular data or menu screen is for.

Every data and menu screen should have an associated help screen. This screen should be created using the THELP.TEMPLAT template at the same time the data/menu screen is created. The text should concentrate on description of the purpose of the associated screen; secondarily, any special commands or features which are active only in the screen being described should be covered. If there is too much text to fit on a single screen, create two or more help screens.

Help screens have essentially no data structure or initialization, and only enough command processing logic to provide the standard DBU service commands (access to editors, etc.). This command processing is as usual done by a subroutine screen.

Like comment screens, all help screens are entered using CALL SCREEN rather than SET SCREEN. However, any help screen may be displayed at any time using the '?help_screen_name' command, which employs the same concept as '=screen_name'.

Since users will typically have no idea of the name of the appropriate help screen to call up when they become confused, the help system is designed to offer a chain of help screens automatically in response to a series of '?' commands. The chain begins with the help screen that is specific to the data or menu screen last SET to, and continues on into the generic help subsystem.

This chaining is implemented by every DBU screen declaring a local variable named 'NEXT_HELP'. Whenever the user gives the '?' command all by itself a 'CALL SCREEN &next_help' command is issued. Screen names are assigned to NEXT_HELP in the help subsystem screens in such a way that a sensible progression is followed; also, some help subsystem screens offer menus of help choices.

After having chained through several help screens, the user will typically want to return directly to the menu, data, or comment screen without having to creep back up the chain one screen at a time. However, enough RETURN SCREEN commands must be issued to flush all the help screens off the BUILDER's stack. This 'pop to the top' capability is implemented by help screen checking for a '/' command after every user command is processed, issuing an immediate RETURN SCREEN if one is found. It appears to the user that he has popped directly back to the screen he started asking for help in, while at the same time RETURN SCREEN commands are properly executed.

8.4.3.6 The Generic Menu Screen: How It Works and An Example

Figures 8-32 and 8-33 display the DBU menu screen template and an example of a menu screen (SKED_MENU). This section will describe the purpose and operation of menu screens, using the contents of these two Figures as examples.

Menu screens exist to help the user find his way around the DBU. There are enough screens in the DBU that even the most experienced user is unlikely to remember all their names, and thus will not always be able to jump directly to the screen he wants to work with using the '=' command. The menu screens provide an hierarchical path to every data screen.

The logic and data structure of menu screens is fairly simple given this limited function. They need to present a layout offering several numbered choices, to accept the user's

FIGURE 8-32. Text of Menu Screen Template

```
*** SCREEN <screen-name  >
*** LAYOUT
SCREEN IS:[now_screen     ]                          SCENARIO IS:[scenarionm  ]
═══════════════════════════════════════════════════════════════════════════════
 ? for help              ALIAS DATA BASE UPDATE SYSTEM                =NAME jumps
═══════════════════choose a screen to use by number or =NAME═══════════════════

                        COMMAND:[command          ]

                        1.
                        2.
                        3.
                        4.
                        5.
                        6.
                        7.
                        8.
                        9.




═══════════════════════════No data may be changed here═══════════════════════════
     Please place a command or option number after COMAMND and press RETURN

*** DECLARATIONS
    NOW_SCREEN,SCENARIONM ;GLOBAL
    COMMAND ;GLOBAL;ACTION
    NEXT_HELP ;LENGTH=80;UPPER
*** INITIAL
    SET OPTION QUOTES=YES
    NOW_SCREEN := "<this screen's name>"
    NOW_FILE := ""
    NOW_GROUP := ""
    CALL SCREEN SECURITY_CHECK
    IF SEEIT=""
         PROMPT "You do not have access to that screen.  Press any key.",Y
         NEXT_SCREEN := $ITEM(PREV_SCREEN,1,1)
         SET SCREEN %next_screen
    ENDIF
    NEXT_SCREEN := "<default next screen>"
    NEXT_HELP := "<default help screen>"
    PREV_SCREEN := $CONCAT(NOW_SCREEN,",",$ITEMS(PREV_SCREEN,1,9))
*** INITIAL EVERYTIME
    COMMAND := ""
*** FUNCTION
    DISPLAY "That ESCape function isn't available here."
*** ENTER
    CALL SCREEN PROC_MENU_CMD
    IF AKSHUN="CMD_NOT_FND"
         DISPLAY "The %command command is not operational here."
    ELSE
```

FIGURE 8-32. Text of Menu Screen Template

```
        SET OPTION QUOTES=NO
        IGNORE ALL ERRORS
        %akshun
        IF $ERROR=7721 OR $ERROR=7639
             DISPLAY "%screen_name is not a valid screen name."
        ENDIF
        SET OPTION QUOTES=YES
    ENDIF
*** ACTION <#>
    DISPLAY "Getting screen."
    SET SCREEN <screen name>
```

FIGURE 8-33. Text of SKED_MENU Menu Screen

```
*** SCREEN SKED_MENU
*** LAYOUT
SCREEN IS:[now_screen     ]                          SCENARIO IS:[scenarionm  ]
═══════════════════════════════════════════════════════════════════════════════
 ? for help              ALIAS DATA BASE UPDATE SYSTEM              =NAME jumps
═══════════════choose a screen to use by number or >NAME═══════════════════════

                          COMMAND:[command         ]

    1.  PROJECTED  NEW CONSTRUCTION, CONVERSION AND REACTIVATION JOB SCHEDULES
    2.  PROJECTED  REPAIR & OVERHAUL, REFUELING, AND SLEP JOB SCHEDULES
    3.  CURRENT    NEW CONSTRUCTION, CONVERSION AND REACTIVATION JOB SCHEDULES
    4.  CURRENT    REPAIR & OVERHAUL, REFUELING, AND SLEP JOB SCHEDULES
    5.  HISTORICAL NEW CONSTRUCTION, CONVERSION AND REACTIVATION JOB SCHEDULES
    6.  HISTORICAL REPAIR & OVERHAUL, REFUELING, AND SLEP JOB SCHEDULES

═══════════════════════════No data may be changed here═══════════════════════════
     Please place a command or option number after COMAMND and press RETURN

*** DECLARATIONS
    NOW_SCREEN,SCENARIONM ;GLOBAL
    COMMAND ;GLOBAL;ACTION
    NEXT_HELP ;LENGTH=80;UPPER
*** INITIAL
    SET OPTION QUOTES=YES
    NOW_SCREEN := "SKED_MENU"
    NOW_FILE := ""
    NOW_GROUP := ""
    CALL SCREEN SECURITY_CHECK
    IF SEEIT=""
         PROMPT "You do not have access to that screen.  Press any key.",Y
         NEXT_SCREEN := $ITEM(PREV_SCREEN,1,1)
         SET SCREEN %next_screen
    ENDIF
    NEXT_SCREEN := ""
    NEXT_HELP := "SKED_MENU_HELP"
    PREV_SCREEN := $CONCAT(NOW_SCREEN,",",$ITEMS(PREV_SCREEN,1,9))
*** INITIAL EVERYTIME
    COMMAND := ""
*** FUNCTION
    DISPLAY "That ESCape function isn't available here."
*** ENTER
    CALL SCREEN PROC_MENU_CMD
    IF AKSHUN="CMD_NOT_FND"
         DISPLAY "The %command command is not operational here."
    ELSE
```

FIGURE 8-33. Text of SKED_MENU Menu Screen

```
        SET OPTION QUOTES=NO
        IGNORE ALL ERRORS
        %akshun
        IF $ERROR=7721 OR $ERROR=7639
             DISPLAY "%screen_name is not a valid screen name."
        ENDIF
        SET OPTION QUOTES=YES
    ENDIF
*** ACTION 1
    DISPLAY "Getting screen."
    SET SCREEN PROJ_NC_SKED
*** ACTION 2
    DISPLAY "Getting screen."
    SET SCREEN PROJ_RE_SKED
*** ACTION 3
    DISPLAY "Getting screen."
    SET SCREEN CURR_NC_SKED
*** ACTION 4
    DISPLAY "Getting screen."
    SET SCREEN CURR_RE_SKED
*** ACTION 5
    DISPLAY "Getting screen."
    SET SCREEN HIST_NC_SKED
*** ACTION 6
    DISPLAY "Getting screen."
    SET SCREEN HIST_RE_SKED
```

choice, and to perform a SET SCREEN to the appropriate menu or data screen in response.

The options are hard-wired into the layout; proper responses are implemented using the BUILDER's standard action-section capability. Note in Figure 8-33 that if the user places '1' in the COMMAND field and presses the return key, a SET to screen PROJ_NC_SKED is performed.

Miscellaneous command services are provided via the usual subroutine screen command processing logic. Security is enforced by a call to SECURITY_CHECK and a SET back to the previous screen displayed if the user does not have access to the menu.

8.4.3.7 Data Screens Which Require Further Discussion

There are a few data screens which differ from the generic model in one way or another. These are CLASS_CHARS, NC_JOB_TYPES, RE_JOB_TYPES, PROJ_NC_SKED, and YARD_CHARS.

8.4.3.7.1 CLASS_CHARS and YARD_CHARS

The CLASS_CHARS and YARD_CHARS screens are unusual in that they serve more than one data relation. CLASS_CHARS serves SHDESC.MISCJ, SHLIFE.MISCJ, and VALCLS.MNUREL. YARD_CHARS serves YARDID.YARDS and VALYDS.MNUREL.

Basic class description data and standard class life data are in separate relations against the eventuality that it will be necessary to keep life data (as well as retirement dates) on a ship-by-ship basis. As far as the user knows at this time, however, both types of data are in a single relation.

A more enduring reason for multiple file service by data screens is the necessity to service the relations holding ALIAS command system list menu candidates and statuses. Whenever information about a new yard is added to a scenario, the name of this yard must be added to any system menus which list the yards ALIAS knows about. These lists must be in a separate relation from the yard's descriptive data in order to accommodate the command system's requirement, but the two relations must be managed as one by their DBU screen.

Meeting this requirement involved relatively simple additions to the generic data screen logic. The basic strategy is to open the "ancillary" files on their own partitions, and to conduct operations on these files in tandem with those on the main file. RECORD POINTs ensure that the simulated join which is involved occurs properly.

Using CLASS_CHARS as an example (see Figure 8-34 for this screen's code), the first addition to the generic logic is

Figure 8-34. CLASS_CHARS Data Screen

```
*** SCREEN CLASS_CHARS
*** LAYOUT
SCREEN IS:[now_screen     ] [modedisp           ]   SCENARIO IS:[scenarionm  ]
? for help              ALIAS DATA BASE UPDATE SYSTEM              =NAME jumps
                        Ship Class Characteristics

                          COMMAND:[command         ]

        Class Name:    [class      ]          Owner:  [customer]

   Name for Reports:   [prntname  ]

                    SHIP SIZE                                  SHIP LIFE

          Length Beam   Height  Draft  Displacement     Standard Service Time
Waterline: [wl  ] [wb ]                                 After Delivery:[life]
Overall:   [ol  ] [ob ]                                 in Time Units: [timunt]
At Launch:              [lh ]   [ld]   [lt    ]         Data Source[datasource]
Light:                  [ih ]   [id]   [it    ]         Data Date  [datadate  ]
Full Load:              [fh ]   [dd]   [ft    ]         Entry Date [entry_date]
                                                        Entry By   [entry_by  ]
[altercapd                                                                    ]
             Place a command after COMMAND and press RETURN

*** DECLARATIONS
    ALTERCAPD,NOW_SCREEN,SCENARIONM,DATADATE,ENTRY_DATE,ENTRY_BY ;GLOBAL
    MODEDISP ;GLOBAL
    COMMAND ;GLOBAL;ACTION
    WL ;NUMERIC;NAME=LENGTH_WL
    OL ;NUMERIC;NAME=LENGTH_OA
    WB ;NUMERIC;NAME=BEAM_WL
    OB ;NUMERIC;NAME=BEAM_OA
    LH ;NUMERIC;NAME=LAUN_HITE
    IH ;NUMERIC;NAME=LITE_HITE
    FH ;NUMERIC;NAME=FULL_HITE
    LD ;NUMERIC;NAME=LAUN_DRAFT
    ID ;NUMERIC;NAME=LITE_DRAFT
    FD ;NUMERIC;NAME=FULL_DRAFT
    LT ;NUMERIC;NAME=LAUN_TONS
    IT ;NUMERIC;NAME=LITE_TONS
    FT ;NUMERIC;NAME=FULL_TONS
    PRNTNAME ;ALPHA
    CLASS ;UPPER;ENHANCE=I,U
    CANDIDATE ;LENGTH=12
    CUSTOMER ;UPPER;ENHANCE=U
    TIMUNT ;UPPER;ENHANCE=U
    LIFE ;NUMERIC
*** INITIAL
    SET OPTION QUOTES=YES
    COMMAND := ""
NOTE     --SECURITY
```

Figure 8-34. CLASS_CHARS Data Screen

```
    NOW_SCREEN := "CLASS_CHARS"
    NOW_FILE := "SHDESC"
    NOW_GROUP := "MISCJ"
    CALL SCREEN SECURITY_CHECK
    IF SEEIT=""
        PROMPT "You do not have access to that screen.  Press any key.",Y
        NEXT_SCREEN := $ITEM(PREV_SCREEN,1,1)
        SET SCREEN %next_screen
    ENDIF
NOTE    --SET UP
    ABORT_MODFY := 0
    EXTRACOM := 1
    DATESCREEN := 0
    MODEDATA := "DATA"
    DATEMODE := ""
    MODESET := ""
    MODEDISP := $CONCAT(INMODE," ",DATEMODE," ",MODEDATA)
    NOW_CURSOR := "MAIN"
    NOW_INDEX := "SCENARIO,CLASS,DATADATE:D,ENTRY_DATE:D"
    LEGAL_FIELDS := "CUSTOMER,TIMINT"
    LEGLPREF1 := "1,4"
    KEYFIELDS := "CLASS,DATADATE"
    COMT_FLD_MODNUM := "2,4"
    LOWKEY := "CLASS"
    LOWLEN := "10"
    VERIFYFIELDS := "SCENARIO,CLASS"
    VERTYP := "U"
    NEXT_SCREEN := "NC_SKED_PF"
    NEXT_HELP := "CLASS_MENU_HELP"
    PREV_SCREEN := $CONCAT(NOW_SCREEN,",",$ITEMS(PREV_SCREEN,1,9))
    COMT_SCREEN := "CLASS_CHAR_COMT"
    COMT_FILE := "SHCOMT"
    COMT_GROUP := "MISCJ"
    COMT_BREAK := "4"
    COMT_ACTIVE := ""
    CALL PROCEDURE GETSCENV %comt_file.%comt_group SCENCOM
    COMT_RESET := "@ BY SCENARIO,CLASS,DATADATE:D,ENTRY_DATE:D,LNUM &
                  WHERE SCENARIO=""%scencom"" AND HULL=0 "
    DATADATE := ""
    ENTRY_DATE := ""
    ENTRY_BY := ""
    REFRESH
    DISPLAY "Fetching data files for this screen."
NOTE    --GET FILES
    USEFILE := NOW_FILE
    USEGROUP := NOW_GROUP
    USECURSOR := NOW_CURSOR
    USE PARTITION %now_cursor
    CALL SCREEN GETFIL
    CALL PROCEDURE GETSCENV %usefile.%usegroup SCEN1
    RESET := "@ BY %now_index WHERE SCENARIO=""%scen1"" "
    SET OPTION QUOTES=NO
```

Figure 8-34. CLASS_CHARS Data Screen

```
            SET OPTION QUOTES=NO
            IGNORE ALL ERRORS
            %akshun
            IF $ERROR=7721 OR $ERROR=7639
                DISPLAY "%screen_name is not a valid screen name."
            ENDIF
            SET OPTION QUOTES=YES
        ENDIF
    ELSEIF $IN(COMCHR,%datacmdb)
        CALL SCREEN PROC_DATA_CMDB
        IF (COMCHR="N" OR COMCHR="B" OR COMCHR="S") AND FOUND="Y"
            SCENARIO := SCEN2
            RECORD POINT USING MAIN2
            RECORD READ USING MAIN2
            SCENARIO := SCEN1
            DISPLAY
        ELSEIF COMCHR="A" AND ADD="Y" AND DOADD
            RECORD ADD USING MAIN2
            CANDIDATE := CLASS
            RECORD POINT USING MAIN3
            IF NOT $FOUND
                RECORD ADD USING MAIN3
            ENDIF
            DISPLAY "New class added to scenario."
        ELSEIF COMCHR="D" AND DELOK="Y"
            SET OPTION QUOTES=OFF
            SELECT @ WHERE SCENARIO="%scenario" AND CLASS="%class"
            NOMORE := NOT $FOUND
            SELECT %reset %modeset
            SET OPTION QUOTES=ON
            RECORD POINT USING MAIN2
            RECORD DELETE USING MAIN2
            IF NOMORE
                CANDIDATE:=CLASS
                RECORD POINT USING MAIN3
                RECORD DELETE USING MAIN3
                DISPLAY "Class deleted from scenario."
            ELSE
                DISPLAY "Data deleted."
            ENDIF
            CALL SCREEN KCLASS_CHARS
        ELSEIF COMCHR="M" AND FOUND="Y" AND MODFY="Y" AND DOMOD
note
            RECORD POINT USING MAIN2
            RECORD UPDATE USING MAIN2
            DISPLAY "Data modified."
        ELSEIF COMCHR="U" AND FOUND="Y" AND UPDAT="Y" AND DOMOD
NOTE
            RECORD ADD USING MAIN2
        ENDIF
    ELSE
        DISPLAY "The %command command is not operational here."
```

Figure 8-34. CLASS_CHARS Data Screen

```
    SELECT %reset %modeset
    SET OPTION QUOTES=YES
NOTE     --ancillary files
    USEFILE := "SHLIFE"
    USECURSOR := "MAIN2"
    USE PARTITION MAIN2
    CALL SCREEN GETFIL
    SET INDEX %now_index
    CALL PROCEDURE GETSCENV %usefile.%usegroup SCEN2
    USEFILE := "VALCLS"
    USEGROUP := "MNUREL"
    USECURSOR := "MAIN3"
    USE PARTITION MAIN3
    CALL SCREEN GETFIL
    SET INDEX SCENARIO,CANDIDATE
    CALL PROCEDURE GETSCENV %usefile.%usegroup SCEN3
    USE PARTITION %now_cursor
    FOUND := "N"
NOTE     --MORE SECURITY
    SCENARIO := SCEN1
    IF SCENARIO<>SCENARIONM
        UPDAT := ""
        DELET := ""
        MODFY := ""
        ADD := ""
    ENDIF
    CALL SCREEN SHOWPRIV
    CALL SCREEN LEGALS_INIT
    DISPLAY
*** INITIAL EVERYTIME
    COMMAND := ""
*** VARIABLE CLASS
    ABORT_MODFY := 1
*** VARIABLE DATADATE
    ABORT_MODFY := 1
*** FUNCTION
    DISPLAY "That ESCape function not available here."
*** FUNCTION ?
    CALL SCREEN FIELDHELP
*** FUNCTION +
    CALL SCREEN NEXT_LEGAL
*** FUNCTION -
    CALL SCREEN PREV_LEGAL
*** ENTER
    COMCHR := $SUBSTR(COMMAND,1,1)
    SCREEN_NAME := $SUBSTR(COMMAND,2)
    SET OPTION QUOTES=NO
    IF $IN(COMCHR,%datacmda)
        CALL SCREEN PROC_DATA_CMDA
        IF AKSHUN="CMD_NOT_FND"
            DISPLAY "The %command command is not operational here."
        ELSE
```

Figure 8-34. CLASS_CHARS Data Screen

```
    ENDIF
NOTE     +++++++++
*** SCREEN KCLASS_CHARS
*** INITIAL
    CLEAR VARIABLE FOUND,ABORT_MODFY,DATADATE,DATASOURCE,ENTRY_DATE,ENTRY_BY
    CLEAR VARIABLE LENGTH_WL,LENGTH_OA,BEAM_WL,BEAM_OA,LAUN_HITE,LITE_HITE
    CLEAR VARIABLE FULL_HITE,LAUN_DRAFT,LITE_DRAFT,FULL_DRAFT,LAUN_TONS
    CLEAR VARIABLE LITE_TONS,FULL_TONS,PRNTNAME,CLASS,CANDIDATE,CUSTOMER
    CLEAR VARIABLE TIMUNT,LIFE
    RETURN SCREEN
```

opening of the additional files (right after the "---ancillary files" comment in the code), which is done using the standard file management methodology. Note, however, that no selections are issued---it is enough to set indexes on the ancillary files, since the selection on the main file will ensure that only records for the current scenario are returned.

The other change to the generic logic involves the addition of code after the call to PROC_DATA_CMDB so that the various search and update commands operate on the ancillary files. This code identifies the command issued and whether it executed successfully in PROC_DATA_CMDB, and then carries out the requisite actions.

8.4.3.7.2 NC_JOB_TYPES and RE_JOB_TYPES

These two screens (shown in Figures 8-35 and 8-36) are quite unusual. They exist to serve the VLJTYP.MNUREL and VLRJOB.MNUREL command system list menu storage relations, there being no main data relation concerned with job type codes. However, they must be bound by the contents of the NCJOBT.LEGALS and REJOBT.LEGALS legal values relations, since job type is a legals-type field on many screens. Further, the VLJTYP relation contains both new construction and repair job type codes. The screens must therefore distinguish one type of code from another as they make retrievals from the relations.

This unusual set of requirements made it impractical to use PROC_DATA_CMDB in these screens. Instead, dedicated search and update logic was implemented.

The search logic works by reading down the main relation (VLJTYP), and then looking to see if the value read from that relation is in the appropriate legals relation. This gets around the problem of repair and new construction job type codes being mixed in VLJTYP.

Figure 8-35. NC_JOB_TYPES Data Screen

```
*** SCREEN NC_JOB_TYPES
*** LAYOUT
SCREEN IS:[now_screen     ] [modedisp           ]   SCENARIO IS:[scenarionm  ]
 ? for help            ALIAS DATA BASE UPDATE SYSTEM              =NAME jumps
=Lengthen or Shorten List of New Construction Job Type Codes for this Scenario=

                        COMMAND:[command         ]


             Job Type Code:    [ncjobt]

             Code Description:[fullname                                    ]
```

```
[altercapd                                                                  ]
  Modify and Update don't work here; use ESC + or ESC - to choose type codes.

*** DECLARATIONS
    ALTERCAPD,NOW_SCREEN,SCENARIONM ;GLOBAL
    MODEDISP ;GLOBAL
    COMMAND ;GLOBAL;ACTION
    NCJOBT ;UPPER;ENHANCE=I,U;POSITION
    CANDIDATE;LENGTH=12
    FULLNAME ;DISPLAY;ENHANCE=NONE
*** INITIAL
    SET OPTION QUOTES=YES
    COMMAND := ""
NOTE     --SECURITY
    NOW_SCREEN := "NC_JOB_TYPES"
    NOW_FILE := "VLJTYP"
    NOW_GROUP := "MNUREL"
    CALL SCREEN SECURITY_CHECK
    IF SEEIT=""
        PROMPT "You do not have access to that screen.  Press any key.",Y
        NEXT_SCREEN := $ITEM(PREV_SCREEN,1,1)
        SET SCREEN &next_screen
    ENDIF
NOTE     --SET UP
    ABORT_MODFY := 0
    EXTRACOM := 1
    DATESCREEN := 0
    MODEDATA := "DATA"
    MODESET := ""
```

Figure 8-35. NC_JOB_TYPES Data Screen

```
    INMODE := "ALL"
    DATEMODE := ""
    MODEDISP := $CONCAT(INMODE," ",DATEMODE," ",MODEDATA)
    NOW_CURSOR := "MAIN"
    NOW_INDEX := "SCENARIO,NCJOBT"
    LEGAL_FIELDS := "NCJOBT"
    LEGLPREF1 := "1"
    KEYFIELDS := "NCJOBT"
    COMT_FLD_MODNUM := "2"
    LOWKEY := "NCJOBT"
    LOWLEN := "6"
    VERIFYFIELDS := ""
    VERTYP := ""
    NEXT_SCREEN := "RE_JOB_TYPES"
    NEXT_HELP := "JOB_TYPE_HELP"
    PREV_SCREEN := $CONCAT(NOW_SCREEN,",",$ITEMS(PREV_SCREEN,1,9))
NOTE     --no comment file here, relevant line removed
    COMT_ALTERED := ""
    REFRESH
    DISPLAY "Fetching data files for this screen."
NOTE     --GET FILES
    USEFILE := NOW_FILE
    USEGROUP := NOW_GROUP
    USECURSOR := NOW_CURSOR
    USE PARTITION %now_cursor
    CALL SCREEN GETFIL
    CALL PROCEDURE GETSCENV %usefile.%usegroup SCEN1
RESET := "SCENARIO,NCJOBT=CANDIDATE BY %now_index WHERE SCENARIO=""%scen1"" "
    SET OPTION QUOTES=NO
    SELECT %reset %modeset
    SET OPTION QUOTES=YES
    FOUND := "N"
NOTE     --MORE SECURITY
    SCENARIO := SCEN1
    IF SCENARIO<>SCENARIONM
        UPDAT := ""
        DELET := ""
        MODFY := ""
        ADD := ""
    ENDIF
    CALL SCREEN SHOWPRIV
    DISPLAY
*** INITIAL EVERYTIME
    COMMAND := ""
*** FUNCTION
    DISPLAY "That ESCape function not available here."
*** FUNCTION ?
    CALL SCREEN FIELDHELP
*** FUNCTION +
    CALL SCREEN NEXT_LEGAL
*** FUNCTION -
    CALL SCREEN PREV_LEGAL
```

Figure 8-35. NC_JOB_TYPES Data Screen

```
*** ENTER
    COMCHR := $SUBSTR(COMMAND,1,1)
    SCREEN_NAME := $SUBSTR(COMMAND,2)
    SET OPTION QUOTES=NO
    IF COMCHR="C"
         FINISH "No comment screen available here."
    ENDIF
    IF $IN(COMCHR,%datacmda)
         CALL SCREEN PROC_DATA_CMDA
         IF AKSHUN="CMD_NOT_FND"
              DISPLAY "The %command command is not operational here."
         ELSE
              SET OPTION QUOTES=NO
              IGNORE ALL ERRORS
              %akshun
              IF $ERROR=7721 OR $ERROR=7639
                   DISPLAY "%screen_name is not a valid screen name."
              ENDIF
              SET OPTION QUOTES=YES
         ENDIF
    ELSE
         DISPLAY "The %command command is not operational here."
    ENDIF
*** ACTION L
    FINISH "Inspection mode not relevant here, and may not be changed."
*** ACTION K
    CALL SCREEN KNC_JOB_TYPES
*** ACTION N@
    FOUND := "N"
    IF READ=""
         FINISH "You are not allowed to look at data here."
    ENDIF
    DISPLAY "Getting next type code."
    USE PARTITION LEGALS
    SELECT NCJOBT.NCJOBT,NCJOBT.FULLNAME BY NCJOBT
    NOWLEGAL := ""
    USE PARTITION MAIN
    MPECOM := NCJOBT
    IFOUND := 0
    WHILE NOT IFOUND
         RECORD READ
         IF $EOF
              NCJOBT := MPECOM
              FINISH "End of codes.  Use B to return to top of list."
         ELSE
              RECORD POINT USING LEGALS
              IFOUND := $FOUND
         ENDIF
    ENDWHILE
    RECORD READ USING LEGALS
    FOUND := "Y"
    DISPLAY
```

Figure 8-35. NC_JOB_TYPES Data Screen

```
*** ACTION B@
    FOUND := "N"
    IF READ=""
         FINISH "You are not allowed to look at data here."
    ENDIF
    DISPLAY "Returning to top of list."
    SET OPTION QUOTES=NO
    SELECT %reset
    SET OPTION QUOTES=YES
    USE PARTITION LEGALS
    SELECT NCJOBT.NCJOBT,NCJOBT.FULLNAME BY NCJOBT
    NOWLEGAL := ""
    USE PARTITION MAIN
    MPECOM := NCJOBT
    IFOUND := 0
    WHILE NOT IFOUND
         RECORD READ
         IF $EOF
              NCJOBT := MPECOM
              FINISH "End of codes.  Use B to return to top of list."
         ELSE
              RECORD POINT USING LEGALS
              IFOUND := $FOUND
         ENDIF
    ENDWHILE
    RECORD READ USING LEGALS
    FOUND := "Y"
    DISPLAY
*** ACTION S@
    FOUND := "N"
    IF READ=""
         FINISH "You are not allowed to look at data here."
    ENDIF
    DISPLAY "Looking"
    MPECOM := $CONCAT(NCJOBT," ",ZEES)
    SET OPTION QUOTES=NO
    SELECT %reset AND CANDIDATE>="%ncjobt" AND CANDIDATE<="%mpecom"
    SET OPTION QUOTES=YES
    IF NOT $FOUND
         DISPLAY "No match found."
    ENDIF
    USE PARTITION LEGALS
    SELECT NCJOBT.NCJOBT,NCJOBT.FULLNAME BY NCJOBT
    NOWLEGAL := ""
    USE PARTITION MAIN
    MPECOM := NCJOBT
    IFOUND := 0
    WHILE NOT IFOUND
         RECORD READ
         IF $EOF
              NCJOBT := MPECOM
              FINISH "End of codes.  Use B to return to top of list."
```

Figure 8-35. NC_JOB_TYPES Data Screen

```
        ELSE
            RECORD POINT USING LEGALS
            IFOUND := $FOUND
        ENDIF
    ENDWHILE
    RECORD READ USING LEGALS
    FOUND := "Y"
    DISPLAY
*** ACTION V
    CALL SCREEN CHECK_LEGAL
*** ACTION A@
    IF ADD=""
        FINISH "You are not allowed to add data here."
    ENDIF
    CANDIDATE := NCJOBT
    CALL SCREEN CHECK_LEGAL
    IF OK
        IGNORE ERROR 202
        RECORD ADD
        IF $ERROR=202
            FINISH "That code has already been added."
        ENDIF
    ELSE
        FINISH "Cannot add."
    ENDIF
*** ACTION D@
    CANDIDATE := NCJOBT
    DELOK := "Y"
    IF DELET=""
        FINISH "You are not allowed to delete data here."
    ENDIF
    IF FOUND<>"Y"
        FINISH "No current data record to delete.  Use N, B, or S first."
    ENDIF
    RECURS := 0
    CALL SCREEN VERIFYD
    USE PARTITION MAIN
    IF DELOK := "Y"
        DISPLAY "Deleting code."
        SET OPTION QUOTES=NO
        SELECT %reset
        SET OPTION QUOTES=YES
        RECORD POINT;BREAK=0
        RECORD DELETE
        CALL SCREEN K[%now_screen]
        DISPLAY "Code deleted."
    ELSE
        DISPLAY "No deletion done."
    ENDIF
*** SCREEN KNC_JOB_TYPES
*** INITIAL
    CLEAR VARIABLE NCJOBT,FULLNAME,FOUND
    RETURN SCREEN
```

Figure 8-36. RE_JOB_TYPES Data Screen

```
*** SCREEN RE_JOB_TYPES
*** LAYOUT
SCREEN IS:[now_screen     ] [modedisp           ]   SCENARIO IS:[scenarionm  ]
═══════════════════════════════════════════════════════════════════════════
 ? for help            ALIAS DATA BASE UPDATE SYSTEM              =NAME jumps
=Lengthen or Shorten List of New Construction Job Type Codes for this Scenario=

                        COMMAND:[command        ]


             Job Type Code:    [rejobt]

             Code Description:[fullname                                  ]









[altercapd                                                                  ]
  Modify and Update don't work here; use ESC + or ESC - to choose type codes.

*** DECLARATIONS
    ALTERCAPD,NOW_SCREEN,SCENARIONM ;GLOBAL
    MODEDISP ;GLOBAL
    COMMAND ;GLOBAL;ACTION
    REJOBT ;UPPER;ENHANCE=I,U;POSITION
    CANDIDATE;LENGTH=12
    FULLNAME ;DISPLAY;ENHANCE=NONE
*** INITIAL
    SET OPTION QUOTES=YES
    COMMAND := ""
NOTE     --SECURITY
    NOW_SCREEN := "RE_JOB_TYPES"
    NOW_FILE := "VLJTYP"
    NOW_GROUP := "MNUREL"
    CALL SCREEN SECURITY_CHECK
    IF SEEIT=""
         PROMPT "You do not have access to that screen.  Press any key.",Y
         NEXT_SCREEN := $ITEM(PREV_SCREEN,1,1)
         SET SCREEN %next_screen
    ENDIF
NOTE     --SET UP
    ABORT_MODFY := 0
    EXTRACOM := 1
    DATESCREEN := 0
    MODEDATA := "DATA"
    MODESET := ""
```

Figure 8-36. RE_JOB_TYPES Data Screen

```
    INMODE := "ALL"
    DATEMODE := ""
    MODEDISP := $CONCAT(INMODE," ",DATEMODE," ",MODEDATA)
    NOW_CURSOR := "MAIN"
 NOW_INDEX := "SCENARIO,REJOBT"
    LEGAL_FIELDS := "REJOBT"
    LEGLPREF1 := "1"
    KEYFIELDS := "REJOBT"
    COMT_FLD_MODNUM := "2"
    LOWKEY := "REJOBT"
    LOWLEN := "6"
    VERIFYFIELDS := ""
    VERTYP := ""
    NEXT_SCREEN := "NC_JOB_TYPES"
    NEXT_HELP := "JOB_TYPE_HELP"
    PREV_SCREEN := $CONCAT(NOW_SCREEN,",",$ITEMS(PREV_SCREEN,1,9))
NOTE     --no comment file here, relevant line removed
    COMT_ALTERED := ""
    REFRESH
    DISPLAY "Fetching data files for this screen."
NOTE     --GET FILES
    USEFILE := NOW_FILE
    USEGROUP := NOW_GROUP
    USECURSOR := NOW_CURSOR
    USE PARTITION %now_cursor
    CALL SCREEN GETFIL
    CALL PROCEDURE GETSCENV %usefile.%usegroup SCEN1
RESET := "SCENARIO,REJOBT=CANDIDATE BY %now_index WHERE SCENARIO=""%scen1"" "
    SET OPTION QUOTES=NO
    SELECT %reset %modeset
    SET OPTION QUOTES=YES
    FOUND := "N"
    USEFILE := "VLRJOB"
    USECURSOR := "MAIN2"
    USE PARTITION MAIN2
    CALL SCREEN GETFIL
    SET INDEX SCENARIO,CANDIDATE
    CALL PROCEDURE GETSCENV %usefile.%usegroup SCEN2
    USE PARTITION %now_cursor
NOTE     --MORE SECURITY
    SCENARIO := SCEN1
    IF SCENARIO<>SCENARIONM
        UPDAT := ""
        DELET := ""
        MODFY := ""
        ADD := ""
    ENDIF
    CALL SCREEN SHOWPRIV
    DISPLAY
*** INITIAL EVERYTIME
    COMMAND := ""
*** FUNCTION
```

Figure 8-36. RE_JOB_TYPES Data Screen

```
    DISPLAY "That ESCape function not available here."
*** FUNCTION ?
    CALL SCREEN FIELDHELP
*** FUNCTION +
    CALL SCREEN NEXT_LEGAL
*** FUNCTION -
    CALL SCREEN PREV_LEGAL
*** ENTER
    COMCHR := $SUBSTR(COMMAND,1,1)
    SCREEN_NAME := $SUBSTR(COMMAND,2)
    SET OPTION QUOTES=NO
    IF COMCHR="C"
         FINISH "No comment screen available here."
    ENDIF
    IF $IN(COMCHR,%datacmda)
         CALL SCREEN PROC_DATA_CMDA
         IF AKSHUN="CMD_NOT_FND"
              DISPLAY "The %command command is not operational here."
         ELSE
              SET OPTION QUOTES=NO
              IGNORE ALL ERRORS
              %akshun
              IF $ERROR=7721 OR $ERROR=7639
                   DISPLAY "%screen_name is not a valid screen name."
              ENDIF
              SET OPTION QUOTES=YES
         ENDIF
    ELSE
         DISPLAY "The %command command is not operational here."
    ENDIF
*** ACTION L
    FINISH "Inspection mode not relevant here, and may not be changed."
*** ACTION K
    CALL SCREEN KRE_JOB_TYPES
*** ACTION N@
    FOUND := "N"
    IF READ=""
         FINISH "You are not allowed to look at data here."
    ENDIF
    DISPLAY "Getting next type code."
    USE PARTITION LEGALS
    SELECT REJOBT.REJOBT,REJOBT.FULLNAML BY REJOBT
    NOWLEGAL := ""
    USE PARTITION MAIN
    IFOUND := 0
    MPECOM := REJOBT
    WHILE NOT IFOUND
         RECORD READ
         IF $EOF
              REJOBT := MPECOM
              FINISH "End of codes.  Use B to return to top of list."
         ELSE
```

Figure 8-36. RE_JOB_TYPES Data Screen

```
            RECORD POINT USING LEGALS
            IFOUND := $FOUND
        ENDIF
    ENDWHILE
    RECORD READ USING LEGALS
    FOUND := "Y"
    DISPLAY
*** ACTION B@
    FOUND := "N"
    IF READ=""
        FINISH "You are not allowed to look at data here."
    ENDIF
    DISPLAY "Returning to top of list."
    SET OPTION QUOTES=NO
    SELECT %reset
    SET OPTION QUOTES=YES
    USE PARTITION LEGALS
    SELECT REJOBT.REJOBT,REJOBT.FULLNAME BY REJOBT
    NOWLEGAL := ""
    USE PARTITION MAIN
    IFOUND := 0
    MPECOM := REJOBT
    WHILE NOT IFOUND
        RECORD READ
        IF $EOF
            REJOBT := MPECOM
            FINISH "End of codes.  Use B to return to top of list."
        ELSE
            RECORD POINT USING LEGALS
            IFOUND := $FOUND
        ENDIF
    ENDWHILE
    RECORD READ USING LEGALS
    FOUND := "Y"
    DISPLAY
*** ACTION S@
    FOUND := "N"
    IF READ=""
        FINISH "You are not allowed to look at data here."
    ENDIF
    DISPLAY "Looking"
    MPECOM := $CONCAT(REJOBT," ",ZEES)
    SET OPTION QUOTES=NO
    SELECT %reset AND CANDIDATE>="%rejobt" AND CANDIDATE<="%mpecom"
    SET OPTION QUOTES=YES
    IF NOT $FOUND
        DISPLAY "No match found."
    ENDIF
    USE PARTITION LEGALS
    SELECT REJOBT.REJOBT,REJOBT.FULLNAME BY REJOBT
    NOWLEGAL := ""
    USE PARTITION MAIN
```

Figure 8-36. RE_JOB_TYPES Data Screen

```
    IFOUND := 0
    MPECOM := REJOBT
    WHILE NOT IFOUND
         RECORD READ
         IF $EOF
              REJOBT := MPECOM
              FINISH "End of codes.  Use B to return to top of list."
         ELSE
              RECORD POINT USING LEGALS
              IFOUND := $FOUND
         ENDIF
    ENDWHILE
    RECORD READ USING LEGALS
    FOUND := "Y"
    DISPLAY
*** ACTION V
    CALL SCREEN CHECK_LEGAL
*** ACTION A@
    CANDIDATE := REJOBT
    IF ADD=""
         FINISH "You are not allowed to add data here."
    ENDIF
    CALL SCREEN CHECK_LEGAL
    IF OK
         IGNORE ERROR 202
         RECORD ADD
         IF $ERROR=202
              FINISH "That code has already been added."
         ELSE
              RECORD ADD USING MAIN2
         ENDIF
    ELSE
         FINISH "Cannot add."
    ENDIF
*** ACTION D@
    CANDIDATE := REJOBT
    DELOK := "Y"
    IF DELET=""
         FINISH "You are not allowed to delete data here."
    ENDIF
    IF FOUND<>"Y"
         FINISH "No current data record to delete.  Use N, B, or S first."
    ENDIF
    RECURS := 0
    CALL SCREEN VERIFYD
    USE PARTITION MAIN
    IF DELOK := "Y"
         DISPLAY "Deleting code."
         SET OPTION QUOTES=NO
         SELECT %reset
         SET OPTION QUOTES=YES
         RECORD POINT;BREAK=0
```

Figure 8-36. RE_JOB_TYPES Data Screen

```
        RECORD DELETE
        RECORD POINT USING MAIN2
        RECORD DELETE USING MAIN2
        CALL SCREEN K[%now_screen]
        DISPLAY "Code deleted."
    ELSE
        DISPLAY "No deletion done."
    ENDIF
*** SCREEN KRE_JOB_TYPES
*** INITIAL
    CLEAR VARIABLE REJOBT,FULLNAME,FOUND
    RETURN SCREEN
```

Only adds and deletes are supported (updates and modifies don't make sense when the only data item is a code name). These ensure that the user has privileges and cause the requisite data validation tasks to be performed, and then do the add or the delete relation(s) using standard commands.

Note that the screen layouts contain a description field (which is display-only) in addition to the code name. The contents of this field are read from the legal-values relation as part of the search logic.

Note also that the code name field is position-only. Since the user may only add names already on the legals list, the way he does this addition is by using the "+" command to flip through the list.

8.4.3.7.3 PROJ_NC_SKED

PROJ_NC_SKED, shown in Figure 8-37, is unusual only in that it is the only screen which supports the "ESC R" (Recalculate dates) command. This command makes it easy to revise a schedule to conform to a new basis date (e.g. a START date) and have it conform to schedule planning factors at the same time. The user enters the new basis date in the proper field, leaves the cursor there, and gives the ESC R command. Processing transfers to the FUNCTION R section in the screen.

To support the command, the planning factor file (NCJDAT. DESCJ) is opened on an auxiliary partition and some of its fields (e.g. AWD_ST (award-start interval)) are declared in the local variable data structure. When ESC R is given, a selection is issued which uses aggregates to ensure that the latest planning factors are returned, and a series of RECORD POINTs on this selection attempt to find planning factors appropriate for the ship on the screen layout (note that an exact match on both yard

Figure 8-37. PROJ_NC_SKED Data Screen

```
*** SCREEN PROJ_NC_SKED
*** LAYOUT
SCREEN IS:[now_screen     ] [modedisp           ]   SCENARIO IS:[scenarionm  ]
═══════════════════════════════════════════════════════════════════════════
 ? for help            ALIAS DATA BASE UPDATE SYSTEM              =NAME jumps
═══════════════Projected New Construction Type Schedules═══════════════════

                          COMMAND:[command         ]

    Name    [shipname                      ]          Shipyard [yard    ]
    Class   [class     ]                              Job Type [ncjobt]
    Hull    [hull]                                    Seq Type [jstyp ]
    Comnum  [comnum]                                  C Method [cmethd]
                                                      Customer [customer]
    Approp  [approp    ]
    Award   [award     ]                              Assigner May Modify
    Start   [start     ]                              This Schedule? [yes]
    Keel    [keel      ]
    Launch  [launch    ]                              Data Source[datasource]
    Deliv   [delivery  ]                              Data Date  [datadate  ]
    Commis  [commission]                              Entry Date [entry_date]
    Days Added to Life [days]                         Entry By   [entry_by  ]
[altercapd                                                                  ]
      Only one entry per class, hull, comnum allowed—no Update here.

*** DECLARATIONS
    ALTERCAPD,NOW_SCREEN,SCENARIONM,DATADATE,ENTRY_DATE,ENTRY_BY ;GLOBAL
    MODEDISP ;GLOBAL
    COMMAND ;GLOBAL;ACTION
    APPROP,AWARD,START,KEEL,LAUNCH,DELIVERY,COMMISSION;DATE="MM/DD/CCCC"
    DAYS ;NAME=DAYSADDED;NUMERIC
    YES ;NAME=YESNO;UPPER;ENHANCE=U
    AUTOMOD;LENGTH=4;UPPER
    CLASS,YARD;ENHANCE=I,U;UPPER
    HULL,COMNUM ;NUMERIC;ENHANCE=I,U
    NCJOBT,JSTYP,CMETHD,CUSTOMER ;UPPER;ENHANCE=U
    YRD;LENGTH=8
    STYP,TIMUNT ;LENGTH=6
    APPROP_AWD,AWD_ST,ST_KL,KL_LN,LN_DL,DL_COM ;NUMERIC
    DATEMP ;DATE="MM/DD/CCCC"
*** INITIAL
    SET OPTION QUOTES=YES
    COMMAND := ""
NOTE    ——SECURITY
    NOW_SCREEN := "PROJ_NC_SKED"
    NOW_FILE := "NCJOBAT"
    NOW_GROUP := "PROJJ"
    CALL SCREEN SECURITY_CHECK
    IF SEEIT=""
        PROMPT "You do not have access to that screen.  Press any key.",Y
        NEXT_SCREEN := $ITEM(PREV_SCREEN,1,1)
        SET SCREEN &next_screen
```

Figure 8-37. PROJ_NC_SKED Data Screen

```
    ENDIF
NOTE     ——SET UP
    ABORT_MODFY := 0
    EXTRACOM := 0
    DATESCREEN := 0
    MODEDATA := "DATES"
    DATEMODE := "PLANNED"
    INMODE := "ALL"
    MODESET := ""
    MODEDISP := $CONCAT(INMODE," ",DATEMODE," ",MODEDATA)
    NOW_CURSOR := "MAIN"
    NOW_INDEX := "SCENARIO,CLASS,HULL,COMNUM"
    LEGAL_FIELDS := "NCJOBT,JSTYP,CMETHD,CUSTOMER,YESNO"
    LEGLPREF1 := "1,1,1,1,2"
    KEYFIELDS := "CLASS,HULL,COMNUM"
    COMT_FLD_MODNUM := "2,7,7"
    LOWKEY := "COMNUM"
    LOWLEN := "-1"
    VERIFYFIELDS := "SCENARIO,CLASS,HULL,COMNUM"
    VERTYP := "JU"
    NEXT_SCREEN := ""
    NEXT_HELP := "P_NC_SKED_HELP"
    PREV_SCREEN := $CONCAT(NOW_SCREEN,",",$ITEMS(PREV_SCREEN,1,9))
    COMT_SCREEN := "PROJ_NC_SKED_C"
    COMT_FILE := "NCJCOM"
    COMT_GROUP := "PROJJ"
    COMT_BREAK := "4"
    COMT_ACTIVE := ""
    CALL PROCEDURE GETSCENV %comt_file.%comt_group SCENCOM
    COMT_RESET := "@ BY %now_index,LNUM WHERE SCENARIO=""%scencom"" "
    DATADATE := ""
    ENTRY_DATE := ""
    ENTRY_BY := ""
    REFRESH
    DISPLAY "Fetching data files for this screen."
NOTE     ——GET FILES
    USEFILE := NOW_FILE
    USEGROUP := NOW_GROUP
    USECURSOR := NOW_CURSOR
    USE PARTITION %now_cursor
    CALL SCREEN GETFIL
    CALL PROCEDURE GETSCENV %usefile.%usegroup SCEN1
    RESET := "@ BY %now_index WHERE SCENARIO=""%scen1"" "
    SET OPTION QUOTES=NO
    SELECT %reset %modeset
    SET OPTION QUOTES=YES
    FOUND := "N"
NOTE     ——GET PLANNING FACTOR FILE FOR ESC R FUNCTION SUPPORT
    USEFILE := "NCJDAT"
    USEGROUP := "DESCJ"
    USECURSOR := "MAIN2"
    USE PARTITION MAIN2
```

Figure 8-37. PROJ_NC_SKED Data Screen

```
    CALL SCREEN GETFIL
    CALL PROCEDURE GETSCENV %usefile.%usegroup SCEN2
    USE PARTITION %now_cursor
NOTE     ——MORE SECURITY
    SCENARIO := SCEN1
    IF SCENARIO<>SCENARIONM
        UPDAT := ""
        DELET := ""
        MODFY := ""
        ADD := ""
    ENDIF
    CALL SCREEN SHOWPRIV
    CALL SCREEN LEGALS_INIT
    DISPLAY
*** INITIAL EVERYTIME
    COMMAND := ""
*** VARIABLE CLASS
    ABORT_MODFY := 1
*** VARIABLE HULL
    ABORT_MODFY := 1
*** VARIABLE COMNUM
    ABORT_MODFY := 1
*** FUNCTION
    DISPLAY "That ESCape function not available here."
*** FUNCTION ?
    CALL SCREEN FIELDHELP
*** FUNCTION +
    CALL SCREEN NEXT_LEGAL
*** FUNCTION -
    CALL SCREEN PREV_LEGAL
*** FUNCTION R
    FLDNAM := $VARIABLE
    IF FLDNAM="APPROP" OR FLDNAM="AWARD" OR FLDNAM="START" OR FLDNAM="KEEL" &
    OR FLDNAM="LAUNCH" OR FLDNAM="DELIVERY" OR FLDNAM="COMMISSION"
        DISPLAY "Recalculating dates based on %fldnam."
    USE PARTITION MAIN2
    SELECT SCENARIO,CLASS,NCJOBT,YRD=YARD,STYP=JSTYP,APPROP_AWD,AWD_ST, &
        ST_KL,KL_LN,LN_DL,DL_COM,TIMUNT &
     BY SCENARIO,CLASS,NCJOBT,YRD,STYP &
     WHERE DATADATE=$MAX(DATADATE BY SCENARIO,CLASS,NCJOBT,YARD,JSTYP) AND &
        ENTRY_DATE=$MAX(ENTRY_DATE BY SCENARIO,CLASS,NCJOBT,YARD,JSTYP) &
        AND SCENARIO="%scen2"
        YRD := YARD
        STYP := JSTYP
        RECORD POINT USING MAIN2
        IF NOT $FOUND
            YRD := "ANY"
            RECORD POINT USING MAIN2
            IF NOT $FOUND
                STYP := "ORDFOL"
                YRD := YARD
                RECORD POINT USING MAIN2
```

Figure 8-37. PROJ_NC_SKED Data Screen

```
                IF NOT $FOUND
                    YRD := "ANY"
                    RECORD POINT USING MAIN2
                    IF NOT $FOUND
                        USE PARTITION MAIN
                        FAIL "No planning factors available for this &
                             class, hull, job type."
                    ENDIF
                ENDIF
            ENDIF
        ENDIF
        DISPLAY "Using planning factors for yard=%yrd, seq type=%styp"
        RECORD READ USING MAIN2
        DATEMP := %fldnam
        CALL PROCEDURE CALCDATE %fldnam %datemp
        USE PARTITION %now_cursor
        NUM := 1
        FLDNAM := $ITEM(DATE_VARS,NUM,1)
        WHILE FLDNAM <> ""
            %fldnam := %fldnam
            NUM := NUM+1
            FLDNAM := $ITEM(DATE_VARS,NUM,1)
        ENDWHILE
        DISPLAY
    ELSE
        DISPLAY "Place the cursor on the date to be used as a basis."
    ENDIF
*** ACTION L
    DISPLAY "Projected schedules inspection mode is always 'ALL PLANNED'."
*** ACTION U
    DISPLAY "The update command does not operate in this screen."
*** ENTER
    AUTOMOD := YESNO
    COMCHR := $SUBSTR(COMMAND,1,1)
    SCREEN_NAME := $SUBSTR(COMMAND,2)
    SET OPTION QUOTES=NO
    IF $IN(COMCHR,%datacmda)
        CALL SCREEN PROC_DATA_CMDA
        IF AKSHUN="CMD_NOT_FND"
            DISPLAY "The %command command is not operational here."
        ELSE
            SET OPTION QUOTES=NO
            IGNORE ALL ERRORS
            %akshun
            IF $ERROR=7721 OR $ERROR=7639
                DISPLAY "%screen_name is not a valid screen name."
            ENDIF
            SET OPTION QUOTES=YES
        ENDIF
    ELSEIF $IN(COMCHR,%datacmdb)
        CALL SCREEN PROC_DATA_CMDB
            IF COMCHR="N" OR COMCHR="B" OR COMCHR="S"
```

Figure 8-37. PROJ_NC_SKED Data Screen

```
                YESNO := AUTOMOD
            ENDIF
    ELSE
        DISPLAY "The %command command is not operational here."
    ENDIF
NOTE    +++++++++
*** SCREEN KPROJ_NC_SKED
*** INITIAL
    CLEAR VARIABLE FOUND,ABORT_MODFY,DATADATE,DATASOURCE,ENTRY_DATE,ENTRY_BY
    CLEAR VARIABLE APPROP,AWARD,START,KEEL,LAUNCH,DELIVERY,COMMISSION
    CLEAR VARIABLE YESNO,DAYSADDED,AUTOMOD,CLASS,YARD,HULL,COMNUM
    CLEAR VARIABLE SHIPNAME,NCJOBT,JSTYP,CMETHD,CUSTOMER
    RETURN SCREEN
```

and ncjobt are first tried for, with a value of "ANY" for yard and "ORDFOL" for ncjobt being tried if these fail.

The planning factor record is read to get its values into the local variables declared, and the calcdate FORTRAN procedure is called. This subroutine makes heavy use of the date utility routines in the SL to construct a complete new set of schedule dates, using the planning factor intervals to determine the interval between the user-specified basis date and each of the other schedule dates. The abstract for CALCDATE is given in Figure 8-38. Note that CALCDATE determines which BUILDER variables to operate on by reading the contents of the DATE_VARS and DESC_VARS global variables (set in the INIT screen); the contents of these variables must be modified if additional schedule dates are added to the screen.

8.4.3.8 The Command Processors

A general description of DBU command processing strategies was given in Section 8.4.1.2. One of these strategies was to pass action commands (command codes put in the COMMAND field followed by a carriage return) to subroutine screens for interpretation and processing, rather than using the BUILDER's action-section facility. This Section will describe in more detail the operation of these subroutine screens.

The motivation for this subroutinization was twofold: first, the large volume of BUILDER code required to carry out many DBU commands, if duplicated once for each screen, would result in enormous and unmanageable screen source code files. Second, even the smallest change to any of the algorithms or conventions would require a large volume of editing to implement. These problems are avoided by isolating the logic, leaving only enough code in user screens to execute the processors.

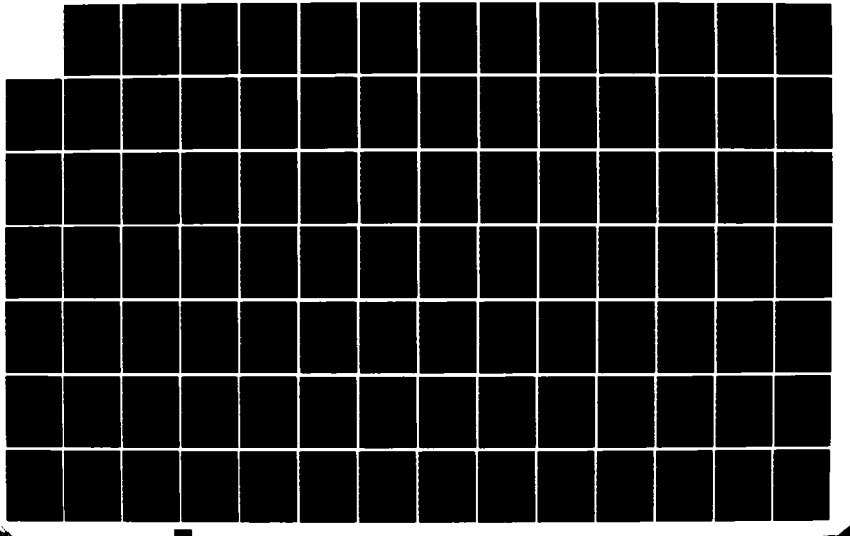
There are six action command processing screens: PROC_MENU_CMD, PROC_DATA_CMDA, PROC_DATA_CMDB, PROC_COMT_CMD, PROC_HELP_CMD, and PROC_REPT_CMD.

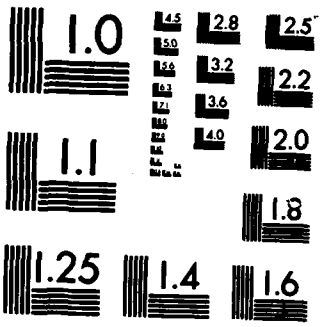
Figure 8-38. Calcdate FORTRAN Procedure Abstract

```
C     CALCDATE ***********************************************
$CONTROL segment=dsa
      SUBROUTINE  calcdate(cursor,table,pointr)
C*                      *** FORMAL PARAMETER DECLARATIONS ***
      integer cursor(50),table(41),pointr(4)
C*                                              *** ABSTRACT ***
C$PURPOSE   Implements the ESC R function for projected
C     new construction schedules DBU screen; recalculates
C     schedule dates using planning factors.
C$AUDIT HISTORY
C        MSCarey        02-apr-84  AUTHOR
C$FORMAL PARAMETERS
Cin      cursor   current DBU cursor
Cin      table    primary builder memory table
Cin      pointr   pointers to call procedure line text
C$COMMON BLOCKS
C        none
C$CALLER DBU, screen PROJ_NC_SKED
C$METHOD
C     Get the test from the call procedure line; first argument is
C     name of variable which is basis date; second argument is basis
C     date.
C     Load planning factors from builder memory.
C     Call ascdaysl for each date.
C     Place the new date in the proper variable in builder memory.
C$LOCAL VARIABLES
C        maxdat   max number of schedule dates processable
C        basind   index number of the basis date
C        basnam   name of the basis date
C        basdat   value of the basis date, ddate format
C        timunt   time units specification
C        dlist    list of dates to be processed, in ascending order
C        plist    list of planning factors, in ascending order
C        day      ddate representation of each new date
C        datnam   name of each date to process
C$$
C*                    *** INCLUDES and LOCAL DECLARATIONS ***
```

AD-A150 422 ALIAS (ACQUISITION AND LOGISTICS INFORMATION AND ANALYSIS SYSTEM) MAINTEN..(U) DECISION-SCIENCE APPLICATIONS INC ARLINGTON VA M S CAREY ET AL. 6/7

UNCLASSIFIED 31 OCT 84 DSA-593-VOL-1 N00014-82-C-0813 F/G 15/5 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

8.4.3.8.1 General Methodology

The processors share a common overall design. When the user presses the return key, the first character in the COMMAND field of the current screen is stripped off and stored in the COMCHR variable (DBU action commands are usually single-character commands). The stripping is done in the command processor except in the case of data screens, which must decide whether to call PROC_DATA_CMDA or PROC_DATA_CMDB.

The rest of the code in the processors consists of a series of IF-ELSE-ENDIF constructs which guide execution to the code for the particular command given. This code will take actions based on the current DBU global data structure and/or will construct an executable command to be passed back to the calling screen. The latter is necessary because not all actions can be completed in a subroutine screen. In particular, commands which the user gives to use a different screen must ultimately be executed by BUILDER in the screen he is currently in, not the command processor.

For example, if the user is in a help screen and wants to go back to the data screen he came into the help screen from, he is effectively asking that a RETURN SCREEN command be executed. However, a RETURN executed by PROC_HELP_CMD just returns control back to the help screen again. To carry out the user's wish, there must be a way of getting the help screen to execute its own RETURN. PROC_HELP_CMD puts the string 'RETURN SCREEN' in a global variable called 'akshun'; in the help screen, following the call to PROC_HELP_CMD, there is a line of code that reads '%akshun', which causes BUILDER to try and execute whatever text is in akshun.

If a command was given that can be completely processed within the command processor, akshun is returned blank and nothing happens when it is executed in the calling screen.

8.4.3.8.2 PROC_MENU_CMD

PROC_MENU_CMD processes all menu screen action commands EXCEPT those hard-wired into menu screen code to take care of particular menu options (e.g. if menu option 1 offers 'menu2', then the menu screen will have a section headed '*** ACTION 1' which executes the line 'SET SCREEN MENU2').

The following discussion refers to Figure 8-39.

The subroutine screen first divides the contents of the COMMAND field into its first character (into COMCHR) and the remainder (into SCREEN_NAME; almost all multiple character DBU commands are asking for a jump to a particular screen).

The first block IF statements check for exit-screen commands in all their many forms. A "^" (pop to previous screen) results in extraction of the name of the last menu or data screen used before this one from the usage history in PREV_SCREEN, popping of this name off of the PREV_SCREEN stack (so it is not duplicated when the given screen puts its name back on the stack during its initialization), and construction of the appropriate "akshun" command.

A "/" (pop to the top of this processing level) is interpreted as a call for the MASTER menu screen; the requisite "akshun" command is set up.

A "?" help request has four possible responses. If the user has appended a specific help screen name to the "?" in the COMMAND field, a call to this screen is set up providing the screen name contains the string "HELP". An error message is printed if the user has asked for a non-help screen with "?". Note that checking for an invalid screen name (i.e. one for which no screen exists) is done implicitly in the help screen itself when the "akshun" command is executed.

Figure 8-39. PROC_MENU_CMD Subroutine Screen

```
*** SCREEN PROC_MENU_CMD
*** INITIAL
    AKSHUN := ""
    COMCHR := $SUBSTR(COMMAND,1,1)
    SCREEN_NAME := $SUBSTR(COMMAND,2)
    IF COMCHR="^"
        DISPLAY "Returning to previous screen."
        NEXT_SCREEN := $ITEM(PREV_SCREEN,2,1)
        PREV_SCREEN := $CONCAT($ITEMS(PREV_SCREEN,3,10),",MASTER,MASTER")
        AKSHUN := "SET SCREEN &next_screen"
        RETURN SCREEN
    ELSEIF COMCHR="/"
        DISPLAY "Returning to master screen."
        AKSHUN := "SET SCREEN MASTER"
        RETURN SCREEN
    ENDIF
    IF COMCHR = "?"
        DISPLAY "Getting screen..."
        IF SCREEN_NAME <> ""
            IF $MATCH(SCREEN_NAME,"HELP") <> 0
                AKSHUN := "CALL SCREEN &screen_name"
            ELSE
                AKSHUN := "FAIL ""Only help screens may be pushed &
                                  into with ?NAME."" "
            ENDIF
        ELSE
            IF $MATCH(NEXT_HELP,"""")
                AKSHUN := NEXT_HELP
            ELSE
                AKSHUN := "[CALL SCREEN &next_help]"
            ENDIF
        ENDIF
        RETURN SCREEN
    ENDIF
    IF COMCHR="="
        DISPLAY "Getting screen."
        IF SCREEN_NAME <> ""
            IF $MATCH(SCREEN_NAME,"HELP")
                AKSHUN := "FAIL ""Jump into help screens by '?NAME'"" "
            ELSE
                AKSHUN := "SET SCREEN &screen_name"
            ENDIF
        ELSE
            IF NEXT_SCREEN=""
                AKSHUN := "DISPLAY ""No default screen here.  &
                           You must choose."" "
            ELSE
                AKSHUN := "SET SCREEN &next_screen"
            ENDIF
        ENDIF
        RETURN SCREEN
    ENDIF
```

Figure 8-39. PROC_MENU_CMD Subroutine Screen

```
IF COMCHR="Q"
     CALL SCREEN SUSPEND
     ARSHUN := "SET SCREEN MASTER"
ELSEIF COMCHR="E"
     PROMPT "The DBU process will be TERMINATED, not held as with Q.  &
            Do it? ",Y
     IF Y="Y"
          EXIT
     ENDIF
ELSEIF COMCHR="L"
     CALL SCREEN SET_INSPCT_MODE
ELSEIF COMCHR=":"
     CALL SCREEN MPECOMMAND
ELSEIF COMCHR="T" AND SCREEN_NAME=""
     CALL SCREEN RUNTDP
ELSEIF COMCHR="T" AND SCREEN_NAME="H"
     CALL SCREEN RUNEDITOR
ELSEIF COMCHR="Z" AND USERLEVL>=3
     IF SCREEN_NAME="ON"
          TRACE ON
     ELSEIF SCREEN_NAME="OFF"
          TRACE OFF
     ELSEIF SCREEN_NAME="VAR"
     PROMPT "Name of variable to display value for:",MPECOM
     SHOW VARIABLE %mpecom
     PAUSE 9
     ENDIF
ELSE
     ARSHUN := "CMD_NOT_FND"
ENDIF
RETURN SCREEN
```

If the user has asked for the next screen in the help chain by giving a "?" alone, then a call to the screen whose name is stored in NEXT_HELP is set up. If it happens that the user is at the end of a help chain, this will be detected by the presence of a DISPLAY message (which must always contain quotes) to that effect in the NEXT_HELP variable; the message is then sent back to the help screen for execution.

A very similar four-case method is used for executing the "=" command, the main difference being that the commands set up involve SETs rather than CALLs.

If the user asks for a return to the ALIAS command system via a "Q" command, the SUSPEND utility screen is called to suspend the DBU process and reactivate the command system father process. If the user subsequently returns to the DBU, processing will pick up at this point; a SET SCREEN MASTER is issued to implement the convention that processing always starts at screen MASTER when the DBU is entered.

Requests for various service operations ("L" to change the data retrieval mode setting; ":", "T", or "TH" to give MPE commands or use editors) result in calls to the appropriate utility screens. A "Z" command can be used to turn on the TRACE debugging support option if given by level 3 users (DBA/development personnel).

8.4.3.8.3 PROC_DATA_CMDA

This section will refer to Figure 8-40. Action command processing for data screens is divided among two command processors because of the large volume of code involved. PROC_DATA_CMDA handles "service" commands, such as screen choices and service requests. PROC_DATA_CMDB handles the data retrieval and update commands. A given data screen decides which of the

Figure 8-40. PROC_DATA_CMDA Subroutine Screen

```
*** SCREEN PROC_DATA_CMDA
*** INITIAL
    SET OPTION QUOTES=YES
    AKSHUN := ""
    IF COMCHR="^"
         IF COMT_ACTIVE◇"" AND COMT_ALTERED◇"" &
               AND FOUND="Y" AND NOT ABORT_MODFY
              CALL SCREEN COMT_UPDATE
         ENDIF
         DISPLAY "Returning to previous screen."
         NEXT_SCREEN := $ITEM(PREV_SCREEN,2,1)
         PREV_SCREEN := $CONCAT($ITEMS(PREV_SCREEN,3,10),",MASTER,MASTER")
         AKSHUN := "SET SCREEN %next_screen"
         RETURN SCREEN
    ELSEIF COMCHR="/"
         IF COMT_ACTIVE◇"" AND COMT_ALTERED◇"" &
               AND FOUND="Y" AND NOT ABORT_MODFY
              CALL SCREEN COMT_UPDATE
         ENDIF
         DISPLAY "Returning to master screen."
         AKSHUN := "SET SCREEN MASTER"
         RETURN SCREEN
    ENDIF
    IF COMCHR="C" AND READ◇""
         DISPLAY "Getting comment screen."
         CALLED := "Y"
         AKSHUN := "CALL SCREEN %comt_screen"
         RETURN SCREEN
    ENDIF
    IF COMCHR="P"
         AKSHUN := "CALL SCREEN REPORT"
         RETURN SCREEN
    ENDIF
    IF COMCHR = "?"
         DISPLAY "Getting screen..."
         IF SCREEN_NAME ◇ ""
              IF $MATCH(SCREEN_NAME,"HELP") ◇ 0
                   AKSHUN := "CALL SCREEN %screen_name"
              ELSE
                   AKSHUN := "FAIL ""Only help screens may be pushed &
                                    into with ?NAME."" "
              ENDIF
         ELSE
              IF $MATCH(NEXT_HELP,"""")
                   AKSHUN := NEXT_HELP
              ELSE
                   AKSHUN := "[CALL SCREEN %next_help]"
              ENDIF
         ENDIF
         RETURN SCREEN
    ENDIF
    IF COMCHR="="
```

Figure 8-40. PROC_DATA_OMDA Subroutine Screen

```
    IF COMT_ACTIVE<>"" AND COMT_ALTERED<>"" &
          AND FOUND="Y" AND NOT ABORT_MODFY
        CALL SCREEN COMT_UPDATE
    ENDIF
    DISPLAY "Getting screen."
    IF SCREEN_NAME <> ""
        IF $MATCH(SCREEN_NAME,"HELP")
            AKSHUN := "FAIL ""Jump into help screens by '?NAME'"" "
        ELSE
            AKSHUN := "SET SCREEN &screen_name"
        ENDIF
    ELSE
        IF NEXT_SCREEN=""
            AKSHUN := "DISPLAY ""No default screen here.  &
                       You must choose."" "
        ELSE
            AKSHUN := "SET SCREEN &next_screen"
        ENDIF
    ENDIF
    RETURN SCREEN
ENDIF
IF COMCHR="Q"
    IF COMT_ACTIVE<>"" AND COMT_ALTERED<>"" &
          AND FOUND="Y" AND NOT ABORT_MODFY
        CALL SCREEN COMT_UPDATE
    ENDIF
    CALL SCREEN SUSPEND
    AKSHUN := "SET SCREEN MASTER"
ELSEIF COMCHR="E"
    IF COMT_ACTIVE<>"" AND COMT_ALTERED<>"" &
          AND FOUND="Y" AND NOT ABORT_MODFY
        CALL SCREEN COMT_UPDATE
    ENDIF
    PROMPT "The DBU process will be TERMINATED, not held as with Q.  &
           Do it? ",Y
    IF Y="Y"
        EXIT
    ENDIF
ELSEIF COMCHR="L"
    CALL SCREEN SET_INSPCT_MODE
ELSEIF COMCHR=":"
    CALL SCREEN MPECOMMAND
ELSEIF COMCHR="T" AND SCREEN_NAME=""
    CALL SCREEN RUNTDP
ELSEIF COMCHR="T" AND SCREEN_NAME="H"
    CALL SCREEN RUNEDITOR
ELSEIF COMCHR="Z" AND USERLEVEL>=3
    IF SCREEN_NAME="ON"
        TRACE ON
    ELSEIF SCREEN_NAME="OFF"
        TRACE OFF
    ELSEIF SCREEN_NAME="VAR"
```

Figure 8-40. PROC_DATA_CMDA Subroutine Screen

```
     PROMPT "Name of variable to display value for:",MPECOM
     SHOW VARIABLE %mpecom
     PAUSE 9
     ENDIF
ELSE
     ARSHUN := "CMD_NOT_FND"
ENDIF
RETURN SCREEN
```

two to call by stripping off the first character in the COMMAND field and looking for its match in the DATACMDA and DATACMDB lists.

PROC_DATA_CMDA is an expanded version of PROC_MENU_CMD (discussed in the previous section). It handles the same commands in basically the same fashion. One difference is that commands which request exit from the data screen to another data or menu screen or to the command system trigger the saving of any comment changes for the current data item via a call to the COMT_UPDATE utility screen.

Also, PROC_DATA_CMDA handles two data screen-specific commands, "P" and "C". "C" requests the associated comment screen, for which a CALL is constructed. Note that the CALLED flag variable is set during this construction; this tells the comment screen it has been executed properly (rather than via an inadvertent SET given through an "=" command, which would violate the calling convention for comment screens).

"P" requests that the report generation utility screen be brought up, which is done by a CALL.

8.4.3.8.4 PROC_DATA_CMDB

PROC_DATA_CMDB is the largest and most complex command processor, because it must supervise data base retrievals and updates. It handles the K, N, S, D, V, M, A, and U commands, which operate only in data screens. The flow of control in this command processor is organized in the usual fashion, block IF statements testing the value of COMCHR.

Figure 8-41 gives the code of PROC_DATA_CMDB.

8.4.3.8.4.1 Screen Clearing

A screen clear request causes changes (if any) to the comments for the current data item to be saved. The comment

Figure 8-41. PROC_DATA_CMDB Subroutine Screen

```
*** SCREEN PROC_DATA_CMDB
*** INITIAL
    SET OPTION QUOTES=YES
    IF COMCHR="K"
         IF COMT_ACTIVE<>"" AND COMT_ALTERED<>""
              CALL SCREEN COMT_UPDATE
         ENDIF
         COMT_ACTIVE := ""
         FOUND := "N"
         CALL SCREEN K[%now_screen]
         RETURN SCREEN
    ELSEIF COMCHR="N"
         IF READ=""
              DISPLAY "You are not allowed to look at data here."
         ELSE
              IF COMT_ACTIVE<>"" AND COMT_ALTERED<>"" &
                AND FOUND="Y" AND NOT ABORT_MODFY
                   CALL SCREEN COMT_UPDATE
              ENDIF
              ABORT_MODFY := 0
              DISPLAY "Getting next record."
              COMT_ACTIVE := ""
              USE PARTITION %now_cursor
              IF INMODE="ALL"
                   RECORD READ
              ELSE
                   MPECOM := %lowkey
                   IF LOWLEN = -1
                        %lowkey := %lowkey +1
                   ELSEIF LOWLEN = -2
                        %lowkey := %lowkey -1
                   ELSEIF LOWLEN = -3
                        USE PARTITION DEFAULT
                        SELECT %lowkey=$NEW_DATE("%mpecom",-1)
                        RECORD READ
                        SELECT
                        USE PARTITION %now_cursor
                   ELSEIF LOWLEN>0
                        IF $LENGTH(%lowkey)=%lowlen
                        %lowkey := $CONCAT($SUBSTR(%lowkey,1,%lowlen-1),"z")
                        ELSEIF $LENGTH(%lowkey)=%lowlen-1
                             %lowkey := $CONCAT(%lowkey,ZEES)
                        ELSEIF $LENGTH(%lowkey)<>0
                             %lowkey := $CONCAT(%lowkey," ",ZEES)
                        ENDIF
                   ENDIF
                   RECORD POINT
                   RECORD READ
              ENDIF
              IF $EOF
                   IF (LOWLEN = -3 OR LOWLEN>0) AND INMODE<>"ALL"
                      %lowkey := "%mpecom"
```

Figure 8-41. PROC_DATA_CMDB Subroutine Screen

```
            ELSE
               %lowkey := %mpecom
            ENDIF
            DISPLAY "No more data.  Use B to return to top of file."
            FOUND := "N"
        ELSE
            FOUND := "Y"
            DISPLAY
        ENDIF
    ENDIF
    RETURN SCREEN
ELSEIF COMCHR="B"
    IF READ=""
        DISPLAY "You are not allowed to look at data here."
    ELSE
        IF COMT_ACTIVE<>"" AND COMT_ALTERED<>"" &
          AND FOUND="Y" AND NOT ABORT_MODFY
            CALL SCREEN COMT_UPDATE
        ENDIF
        DISPLAY "Returning to top of file."
        COMT_ACTIVE := ""
        ABORT_MODFY := 0
        USE PARTITION %now_cursor
        SET OPTION QUOTES=NO
        SELECT %reset %modeset
        SET OPTION QUOTES=YES
        RECORD READ
        IF $EOF
            DISPLAY "File empty."
            FOUND := "N"
        ELSE
            FOUND := "Y"
            DISPLAY
        ENDIF
    ENDIF
    RETURN SCREEN
ELSEIF COMCHR="S"
    IF READ=""
        DISPLAY "You are not allowed to look at data here."
    ELSE
        IF COMT_ACTIVE<>"" AND COMT_ALTERED<>"" &
          AND FOUND="Y" AND NOT ABORT_MODFY
            CALL SCREEN COMT_UPDATE
        ENDIF
        DISPLAY "Looking."
        COMT_ACTIVE := ""
        USE PARTITION %now_cursor
        CALL SCREEN SEARCH
        IF OK
            RECORD READ
            DISPLAY
            FOUND := "Y"
```

Figure 8-41. PROC_DATA_CMDB Subroutine Screen

```
            ABORT_MODFY := 0
        ELSE
            SET OPTION QUOTES=NO
            SELECT %reset %modeset
            SET OPTION QUOTES=YES
            DISPLAY "No match found."
            FOUND := "N"
        ENDIF
    ENDIF
    RETURN SCREEN
ELSEIF COMCHR="D"
    DELOK := "N"
    IF DELET=""
        DISPLAY "You are not allowed to delete data here."
        RETURN SCREEN
    ENDIF
    IF FOUND<>"Y"
        DISPLAY "No current data record to delete.  Use N, B, or &
                S to position at a record."
        RETURN SCREEN
    ENDIF
    COMT_ACTIVE := ""
    NOW_COMT_FILE := ""
    USE PARTITION %now_cursor
    RECURS := 0
    CALL SCREEN VERIFYD
    USE PARTITION %now_cursor
    IF DELOK="Y"
        DISPLAY "Deleting data."
        SET OPTION QUOTES=NO
        SELECT %reset %modeset
        SET OPTION QUOTES=YES
        RECORD POINT;BREAK=0
        RECORD DELETE
        FOUND := "N"
        ABORT_MODFY := 0
        IF NOT EXTRACOM
            CALL SCREEN K[%now_screen]
            DISPLAY "Data deleted."
        ENDIF
    ELSE
        DISPLAY "No deletion done."
    ENDIF
    RETURN SCREEN
ELSEIF COMCHR="V" OR COMCHR="A" OR COMCHR="U" OR COMCHR="M"
    DISPLAY "Verifying data."
    UOK := 1
    IF $MATCH(VERTYP,"U")
        CALL SCREEN VERIFYU
        IF NOT UOK AND COMCHR="A"
            SCROLL "The combined values of %verifyfields are not &
                    unique."
```

Figure 8-41. PROC_DATA_CMDB Subroutine Screen

```
            PROMPT "Shall I check for other errors? (Y or N):",Y
            IF Y<>"Y"
                RETURN SCREEN
            ENDIF
        ELSE
            DISPLAY "Key uniqueness test passed."
        ENDIF
    ENDIF
    ADDOK := "Y"
    IF $MATCH(VERTYP,"J")
        MODE := "change"
        CALL SCREEN VERIFYJ
        IF ADDOK="Y"
            DISPLAY "Required data found in associated files."
        ENDIF
        IF ADDOK<>"Y" AND Y<>"Y"
            SET OPTION QUOTES=NO
            SELECT %reset %modeset
            SET OPTION QUOTES=YES
            RECORD POINT;BREAK=0
            RETURN SCREEN
        ENDIF
    ENDIF
    OK := 1
    IF LEGAL_FIELDS<>""
        CALL SCREEN CHECK_LEGAL
    ENDIF
    NUM := 1
    FLDNAM := $ITEM(KEYFIELDS,NUM,1)
    WHILE FLDNAM<>""
        IF %fldnam=""
            DISPLAY "The %fldnam field may not be blank or zero."
            OK := 0
        ENDIF
        NUM := NUM+1
        FLDNAM := $ITEM(KEYFIELDS,NUM,1)
    ENDWHILE
        DOADD := 0
        DOMOD := 0
    IF UOK AND OK AND ADDOK="Y"
        DISPLAY "This data is OK for addition only."
        DOADD := 1
    ELSEIF NOT UOK AND OK AND ADDOK="Y"
        DISPLAY "This data is OK for updates and modifies only."
        DOMOD := 1
    ENDIF
    SET OPTION QUOTES=NO
    SELECT %reset %modeset
    SET OPTION QUOTES=YES
    RECORD POINT;BREAK=0
ENDIF
IF COMCHR="V"
```

Figure 8-41. PROC_DATA_CMDB Subroutine Screen

```
     PROMPT "Hit return to continue...",Y
     RETURN SCREEN
ELSEIF COMCHR="M"
     IF MODFY=""
          DISPLAY "You are not allowed to modify data here."
          RETURN SCREEN
     ENDIF
     IF FOUND<>"Y"
          DISPLAY "No current data record to modify.  Do an Add, or &
                   use N, B, or S."
          RETURN SCREEN
     ENDIF
     IF UCK
         DISPLAY "A data record with these keys is not yet in the file.  &
                  Do an Add."
          RETURN SCREEN
     ENDIF
     IF NOT DOMOD
          RETURN SCREEN
     ENDIF
     IF ABORT_MODFY
          SCROLL "You changed a key field other than the datadate &
                  field.  A modify could accidentally write over &
                  another data base record under these circumstances.  &
                  An Update may work."
          DOMOD := 0
          RETURN SCREEN
     ENDIF
     USE PARTITION %now_cursor
     RECORD POINT;BREAK=0
     IF NOT $FOUND
          SCROLL "You have changed one of the key fields for this &
          data record, making it impossible for me to find the &
          original record in the file to modify it.  Probably &
          the data date field was changed.  You must do either &
          an update or an add to get the new data into the file.  &
          Use update if you changed the datadate field, otherwise &
          do an Add."
          DOMOD := 0
          RETURN SCREEN
     ENDIF
     IF COMT_ACTIVE<>"" AND COMT_ALTERED<>""
          CALL SCREEN COMT_UPDATE
     ENDIF
     USE PARTITION %now_cursor
     RECORD POINT;BREAK=0
     RECORD UPDATE
     RECORD READ
     DISPLAY "Modified data has replaced original in data base."
     RETURN SCREEN
ELSEIF COMCHR="A"
     FOUND := "N"
```

Figure 8-41. PROC_DATA_CMDB Subroutine Screen

```
      IF ADD=""
           DISPLAY "You are not allowed to add data here."
           RETURN SCREEN
      ENDIF
      IF NOT UCK
           DISPLAY "Cannot add when key values not unique."
           RETURN SCREEN
      ENDIF
      IF NOT DOADD
           RETURN SCREEN
      ENDIF
      USE PARTITION %now_cursor
      ENTRY_BY := NOW_USER
      ENTRY_DATE := TODAY
      RECORD ADD
      FOUND := "Y"
      DISPLAY "Data added."
      IF COMT_ACTIVE<>"" AND COMT_ALTERED<>""
           CALL SCREEN COMT_UPDATE
      ENDIF
      ABORT_MODFY := 0
      DISPLAY
      RETURN SCREEN
ELSEIF COMCHR="U"
      IF UPDAT=""
           DISPLAY "You are not allowed to update data here."
           RETURN SCREEN
      ENDIF
      IF UCK
          DISPLAY "A data record with these keys is not yet in the file.  &
                Do an Add."
           RETURN SCREEN
      ENDIF
      IF NOT DOMOD
           RETURN SCREEN
      ENDIF
      USE PARTITION %now_cursor
      ENTRY_DATE := TODAY
      ENTRY_BY := NOW_USER
      IGNORE ERROR 202
      RECORD ADD
      IF $ERROR
           DOMOD := 0
           DISPLAY "Cannot update because record with the values &
                    shown on the screen for the %now_index fields &
                    already exists.  Try changing the datadate."
           RETURN SCREEN
      ENDIF
      ABORT_MODFY := 0
      FOUND := "Y"
      IF COMT_ACTIVE="Y"
           USE PARTITION COMMENT
```

Figure 8-41. PROC_DATA_CMDB Subroutine Screen

```
     ARRAY
          LNUM:= $SUBSCRIPT+1
          RECORD ADD
     ENDARRAY
ELSE
     Y := "Y"
     PROMPT "Want comments associated with old record brought &
            forward? (Y):",ANSWER
     IF ANSWER<>"" AND (ANSWER="Y" OR ANSWER="N")
          Y := ANSWER
     ENDIF
     IF Y="Y"
          USE PARTITION COMMENT
          IF NOW_COMT_FILE <> $CONCAT(COMT_FILE,".",COMT_GROUP)
               USEFILE := COMT_FILE
               USEGROUP := COMT_GROUP
               USECURSOR := "COMMENT"
               CALL SCREEN GETFIL
               NOW_COMT_FILE := $CONCAT(COMT_FILE,".",COMT_GROUP)
          ENDIF
          INUM := $MATCH(KEYFIELDS,",DATADATE")
          IF INUM
               MPECOM := $SUBSTR(KEYFIELDS,1,INUM-1)
          ELSE
               MPECOM := KEYFIELDS
          ENDIF
          IGNORE ALL ERRORS
          SELECT %mpecom BY %mpecom WHERE SCENARIO="%scencom"
          IF $ERROR
               DISPLAY "Sorry unable to bring comments forward.
               SET OPTION QUOTES=NO
               SELECT %comt_reset
               SET OPTION QUOTES=YES
               USE PARTITION %now_cursor
               RETURN SCREEN
          ENDIF
          RECORD POINT
          IF NOT $FOUND
               DISPLAY "Sorry, no comments available to bring &
                       forward."
               SET OPTION QUOTES=NO
               SELECT %comt_reset
               SET OPTION QUOTES=YES
               USE PARTITION %now_cursor
               RETURN SCREEN
          ELSE
               SAVE_DATADATE := DATADATE
               SET OPTION QUOTES=NO
               SELECT %comt_reset
               SET OPTION QUOTES=YES
               DATADATE := "12/31/2199"
               RECORD POINT
```

Figure 8-41. PROC_DATA_OMDB Subroutine Screen

```
                    RECORD READ
                    NUM := %comt_break + 1
                    RECORD POINT;BREAK=%num
                    ARRAY
                         RECORD READ
                    ENDARRAY
                    DATADATE := SAVE_DATADATE
                    ENTRY_DATE :=  TODAY
                    ARRAY
                         LNUM := $SUBSCRIPT+1
                         RECORD ADD
                    ENDARRAY
                    DISPLAY "Comments brought forward successfully."
                    USE PARTITION %now_cursor
                    ENTRY_BY = NOW_USER
               ENDIF
          ENDIF
     ENDIF
ENDIF
SCROLL "SHOULD NOT ARRIVE AT THIS POINT IN EXECUTION.  PROC_DATA_OMDB."
RETURN SCREEN
```

buffer is marked as empty. The dedicated screen clear utility for the current data screen is then called by using the naming convention.

8.4.3.8.4.2 Retrieving the Next Record

Following a security check to see if the user can read data in this screen, and following saving of any comment changes for the current data item, processing proceeds according to the data retrieval mode setting. If the mode setting is "ALL", a simple record read on the proper partition is all that is required. If the mode setting is "LATEST", however, an algorithm to skip to the latest version of the next data item is employed.

This algorithm depends on two facts. First, retrievals in the DBU are always on an index, making it possible to do RECORD POINTS. Second, if a POINT fails for given target key values, it always leaves the RELATE record pointer at the record on the index with key values just greater than the target. Thus if I want the data for the next yard after "ZZZA", but I don't know what its name is, I can be assured of getting to the proper record by pointing with a target key of "ZZZB".

In providing only the latest data, a retrieval index must be implicitly split into two parts: those fields before the DATADATE field, and those after and including DATADATE. The former identify the entity, the latter the currency of each given data record for that entity. Indexed records in ALIAS data relations appear in order of entity, with data for each entity appearing latest-to-oldest.

To return the latest data for the next entity in response to an "N" command (rather than the next latest data for the current entity) we take the least significant field in the entity portion of the index (the field just before DATADATE), increment it slightly, and perform a RECORD POINT. The POINT is almost

certain to fail, but it will leave the record pointer in the desired position.

The logic required to implement this algorithm is complicated by the fact that the manner of incrementing the field value depends on its data type. Alpha fields should have trailing z's added ('NEWPORTz'), numeric fields should be incremented or decremented by 1 depending on whether the index is ascending or descending on that field, and date fields should be decremented by one day (since dates are always indexed descending).

The name of the least significant entity field and a code giving its data type are placed in the LOWKEY and LOWLEN global variables by a data screen's creator.

Following its READ or POINT/READ, the "N" algorithm performs an end-of-file check and sets the FOUND flag. If an end-of-file was encountered and a screen field was altered by the "latest" algorithm, this field's value must be set back to its original value. This is done by storing the original value in the MPECOM buffer and using it to perform the reset.

8.4.3.8.4.3 Removing Search Restrictions/Returning to Top of File

The "B" command resets the data screen's retrieval path to normal status and top-of-file by re-issuing the selection which set up the path during the data screen's initialization. This selection is stored in the RESET global variable.

Before doing the reset any comment changes for the current data item are saved. Also, all flags are reset.

8.4.3.8.4.4 Searching For a Match On Target Field Values

The "S" command constructs a selection with current values of all non-blank data screen fields in its WHERE clause, allowing the user to move directly to the desired record if he knows

enough of its values. The bulk of the work of constructing this selection is done in the SEARCH utility screen. Before the utility is called any comments for the current item are saved (likely to be unnecessary since a "K" is usually issued before filling in the targets for the search).

The utility returns a flag indicating whether it was able to find a record meeting the search criteria. If so, the record is read and flags are set; if not, the retrieval path is reset (similar to issuing a B, but the top record is not read onto the screen).

8.4.3.8.4.5 Data Record Deletion

Data deletion is a complex process requiring that any "subsidiary" data base records in other relations be flushed along with the record showing on the layout. Most of this complexity is "pushed down" into the VERIFYD utility, leaving only security checks and deletion of the visible record to PROC_DATA_CMDB.

Deletion proceeds only if the user has deletion privileges in the current screen and if the FOUND flag indicates that what is showing on the screen represents a current record.

VERIFYD returns a flag, DELOK, which will be false if the user decided not to go through with the deletion after being told that subsidiary data was going to be deleted also. If the flag is true (=1) PROC_DATA_CMDB does a path reset (the path may be changed during VERIFYD operations), does a POINT to get to the record to be deleted, deletes it, and sets flags.

If EXTRACOM indicates that the developer of the data screen has extra data-record processing in the data screen logic then PROC_DATA_CMDB leaves the (now deleted) record's contents on the layout for reference; otherwise it calls the screen clear utility to wipe the layout clean.

8.4.3.8.4.6 Data Validation

Whenever any of the V, A, M, or U commands are given the DBU's data record, validation logic is exercised to determine the suitability of the record on the screen layout for placement in the data base. Validation is typically a four-step process: determining whether the data entity is unique (i.e. not yet in the data relation), whether any join constraints would be violated by its addition (i.e. whether any data which must be present in other relations hasn't been put there yet), whether there are any illegal values in legals-type fields (really a special case of a join failure), and whether any key fields are blank. Most of the work for the first three tests is pushed down into subroutine screens VERIFYU, VERFIYJ, and CHECK_LEGAL.

If a failure occurs during the first two stages of checking the user is prompted for a desire to continue. This prompting is done in PROC_DATA_CMDB in the event of a uniqueness failure, and in VERIFYJ for join failures. The "Y" global variable contains the user's response. If the user chooses to terminate checking the retrieval path is reset and the record pointer moved back to the record showing on the layout via a RECORD POINT.

VERIFYU returns a status flag in variable UOK, VERIFYJ in ADDOK, and CHECK_LEGAL in OK. PROC_DATA_CMDB also uses OK as a failure flag when it runs through the list of fields in the KEYFIELDS variable, checking to see if any are blank. The status flags are then combined into the two "output" status flags DOMOD (ok to do an update or modify) and DOADD (ok to do an add) (an add requires that UOK=1, while an update/modify requires UOK=0), and the path is reset to its pre-verification status.

PROC_DATA_CMDB processing terminates at this point if the V command was issued; otherwise another block IF transfers execution to the code which implements the type of data base modification which was requested.

8.4.3.8.4.7 Data Record Modification

Most of the modification logic in PROC_DATA_CMDB consists of error checking. The user must have modification privileges; the FOUND flag must indicate that the record on the layout was retrieved from the data base; the user is told to do an add, not a modify, if the record is not in the file (check on UOK); DOMOD must be true, i.e. validation checks had to be passed; the user cannot have changed any key fields in the record, including DATADATE; and a RECORD POINT must actually find the target record in the file. If all of these tests are passed, any comment changes currently in memory are posted via a call to COMT_UPDATE and the actual update is done. Note that a RECORD READ follows the RECORD UPDATE so that the record pointer is set to read the next record the next time an "N" command is given, not the current record all over again.

The motivation behind most of the tests is clear, but ABORT_MODFY deserves further comment. ABORT_MODFY is set to .true. (1) by VARIABLE sections in the data screen whenever the user changes any key field. This prevents the user from implicitly changing the data record he is operating on. Generally, an attempt to modify after changing a key field would fail anyway because the POINT which attempts to locate the record just prior to updating would fail. However, if the user happened to make a keyfield change such that the layout record corresponded to another record in the relation, the POINT would succeed. If the modify were allowed to go forward it would write over data the user had not seen, while leaving the original record, the one the user was modifying, intact! Further, the comments for the accidental target might be overwritten as well.

A user who wishes to change a record to one with different keys must do so by retrieving it, making the changes, executing an Add, and then returning to and deleting the original. The DBU's dependency on key values and POINTs to find its way to

records to be modified makes the more direct modification route impossible.

Note that the modify logic does not change the ENTRY_DATE field value. This is necessary since ENTRY_DATE values form part of the join condition which ties data and comments together. Changing of ENTRY_DATE in the data relation would result to "loss" of the associated comments through join failures.

8.4.3.8.4.8 Data Record Addition

The Add logic is also largely error-checking. The user must have add privileges, and the verification checks must have been passed. The ENTRY_DATE and ENTRY_BY fields are automatically set to the current date and user, the record is added, and any comments are added to the comment relation.

8.4.3.8.4.9 Data Record Updating

An Update can be thought of as a partial combination of an Add and a modify. The data entity to be updated must already have a record present in the relation, but the update logic will add a new version of the record, retaining the old. This new version must, however, have a unique index key (i.e. a unique DATADATE,ENTRY_DATE for the given entity).

The Update logic first ensures that the user has update privileges and that the data base integrity checks were passed. It then attempts to do the update (really a record add), which may fail if there is a record already in the data base with an exact match on all the key fields (e.g. if the user tries to do two updates with the same DATADATE on the same day). If there is success, comments in memory can be taken care of by a simple series of RECORD ADDs since we are sure that the key fields will be unique.

If comments are not in memory, a rather complicated strategy is followed to bring the latest set of comments

associated with the entity forward, if the user so desires. First the proper comment storage relation must be retrieved onto the comment partition if it is not there already. Then the keys on this relation's index that identify the data entity, i.e. those keyfields before DATADATE, are extracted and a select is given specifying these fields as the index. This allows a point to be done using the current values of the key fields but not including DATADATE. If the point succeeds then we know there are comments available for bringing forward (if DATADATE had been left in the index we would have had to approximate its value for the point and thus would not have been able to use the resulting $FOUND flag value to determine if comments were there). If there are comments the retrieval path is reset to the normal index and a point/read is issued using a very late DATADATE. This gets the first record of the comments into memory, including the DATADATE value. Then another point with a control break value equal to the number of fields in the entity key plus DATADATE can be issued, setting up issuance of a simulated end-of-file when the ARRAY loop which reads all the comments in passes the last comment line for the given entity and DATADATE. Finally DATADATE is set to its value in the newly updated (added) data record and the comments are added back into the comment storage relation.

8.4.3.8.5 PROC_COMT_CMD

The comment screen command processor, shown in Figure 8-42, processes all action commands given in comment screens. In addition to offering the usual services, this screen must also manage the comment buffer and window data structures, making sure that the two are congruent.

The basic structure of the screen is similar to that of all the others. The command character is stripped out of the comment screen's COMMAND field; this character is then used in a series of block IF tests to guide execution to the proper code.

Figure 8-42. PROC_COMT_CMD Subroutine Screen

```
*** SCREEN PROC_COMT_CMD
*** INITIAL
    AKSHUN := ""
    COMCHR := $SUBSTR(COMMAND,1,1)
    SCREEN_NAME := $SUBSTR(COMMAND,2)
    IF COMCHR="^" OR COMCHR="/"
         DISPLAY "Returning to data screen."
         NUM := 0
         WHILE NUM <= 8
              $SUBSCRIPT := NUM
              BUFFER := DCOMMENT
              $SUBSCRIPT := COMT_START+NUM
              COMMENT := BUFFER
              NUM := NUM+1
         ENDWHILE
         NOW_SCREEN := SAVE_SCREEN
         NOW_CURSOR := SAVE_CURSOR
         RESET := SAVE_RESET
         SCENARIO := SCEN1
         USE PARTITION %now_cursor
         IF NOCHANGE
              COMT_ALTERED := ""
         ENDIF
         NUM := 1
         FLDNAM := $ITEM(KEYFIELDS,NUM,1)
         SET OPTION QUOTES=NO
         WHILE FLDNAM<>""
         $SUBSCRIPT := $ITEM(COMT_FLD_MODNUM,NUM) - 1
              MODIFY VARIABLE %fldnam;GLOBAL[%comt_fld_mods]
              NUM := NUM+1
              FLDNAM := $ITEM(KEYFIELDS,NUM,1)
         ENDWHILE
         SET OPTION QUOTES=YES
         AKSHUN := "RETURN SCREEN"
         RETURN SCREEN
    ELSEIF COMCHR = "+" OR COMCHR="-"
         IF COMT_START=15 AND COMCHR="+"
              DISPLAY "Already showing last available page of comments."
              RETURN SCREEN
         ELSEIF COMT_START=0 AND COMCHR="-"
              DISPLAY "Already showing first page of comments."
              RETURN SCREEN
         ENDIF
         DISPLAY "Saving this page..."
         NUM := 0
         WHILE NUM <= 8
              $SUBSCRIPT := NUM
              BUFFER := DCOMMENT
              $SUBSCRIPT := COMT_START+NUM
              COMMENT := BUFFER
              NUM := NUM+1
         ENDWHILE
```

Figure 8-42. PROC_COMT_CMD Subroutine Screen

```
     IF COMCHR="+"
          COMT_START := COMT_START+8
          DISPLAY "Getting next page..."
          IF COMT_START>15
               COMT_START=15
          ENDIF
     ELSE
          COMT_START := COMT_START-8
          DISPLAY "Getting previous page..."
          IF COMT_START<0
               COMT_START=0
          ENDIF
     ENDIF
     NUM := 0
     WHILE NUM <= 8
          $SUBSCRIPT := COMT_START+NUM
          BUFFER := COMMENT
          $SUBSCRIPT := NUM
          DCOMMENT := BUFFER
          DNUM := COMT_START+NUM+1
          NUM := NUM+1
     ENDWHILE
     CALL SCREEN SET_MORE
     DISPLAY
     RETURN SCREEN
ENDIF
IF COMCHR = "?"
     DISPLAY "Getting screen..."
     IF SCREEN_NAME <> ""
          IF $MATCH(SCREEN_NAME,"HELP") <> 0
               AKSHUN := "CALL SCREEN %screen_name"
          ELSE
               AKSHUN := "FAIL ""Only help screens may be pushed &
                              into with ?NAME."" "
          ENDIF
     ELSE
          IF $MATCH(NEXT_HELP,"""")
               AKSHUN := NEXT_HELP
          ELSE
               AKSHUN := "[CALL SCREEN %next_help]"
          ENDIF
     ENDIF
     RETURN SCREEN
ENDIF
IF COMCHR="K"
     DISPLAY "Clearing..."
     ARRAY
          COMMENT := ""
     ENDARRAY
     ARRAY
          DCOMMENT := ""
          DNUM := $SUBSCRIPT+1
```

Figure 8-42. PROC_COMT_CMD Subroutine Screen

```
     ENDARRAY
     IF NLEFT>0
          COMT_ALTERED:="Y"
     ENDIF
     NLEFT := 0
     MORE := ""
ENDIF
IF COMCHR="D" AND DELET<>""
     LNUM := 1
     RECORD POINT;BREAK=%comt_break
     IF NOT $FOUND
          DISPLAY "No comments in data base matching the key values."
     ELSE
          DISPLAY "Deleting..."
          INUM := LNUM
          WHILE NOT $EOF
               RECORD DELETE
               RECORD READ
          ENDWHILE
          ARRAY
               DNUM := $SUBSCRIPT+1
               DCOMMENT := ""
          ENDARRAY
          ARRAY
               COMMENT := ""
          ENDARRAY
          NLEFT := 0
          COMT_ALTERED := ""
          MORE := ""
     ENDIF
     DISPLAY "Comments deleted"
     RETURN SCREEN
ENDIF
IF COMCHR="L"
     CALL SCREEN SET_INSPCT_MODE
ELSEIF COMCHR=":"
     CALL SCREEN MPECOMMAND
ELSEIF COMCHR="T" AND SCREEN_NAME=""
     CALL SCREEN RUNTDP
ELSEIF COMCHR="T" AND SCREEN_NAME="H"
     CALL SCREEN RUNEDITOR
ELSEIF COMCHR="Z" AND USERLEVL>=3
     IF SCREEN_NAME="ON"
          TRACE ON
     ELSEIF SCREEN_NAME="OFF"
          TRACE OFF
     ELSEIF SCREEN_NAME="VAR"
     PROMPT "Name of variable to display value for:",MPECOM
     SHOW VARIABLE %mpecom
     PAUSE 9
     ENDIF
ELSEIF COMCHR="="
```

Figure 8-42. PROC_COMT_CMD Subroutine Screen

```
     DISPLAY "Direct screen jumps may not be made from comment &
              screens.  Use / first."
ELSEIF COMCHR="Q" OR COMCHR="E"
     DISPLAY "Sorry, you cannot Quit from a comment screen.  Use / first."
ELSE
     AKSHUN := "CMD_NOT_FND"
ENDIF
RETURN SCREEN
```

Since comment screens are CALLed rather than SET into (unlike menu and data screens), user issuance of an exit command (here either ^ or /) implies that a RETURN SCREEN command should be sent back to the comment screen in the AKSHUN variable. Three additional tasks must be completed before the comment screen can be exited, however. First, the contents of the comment window array (dcomment) must be saved into the proper location of the global comment buffer, so that any user changes are not lost when the local dcomment data structure disappears. Second, the key field values appearing on the layout, which were changed to display-only fields during comment initialization, must be changed back again so that the user can work with them in the data screen. Third, the values of global variables, such as RESET, which were set by the comment screen, must be changed back to the data screen's values so that its processing can proceed smoothly.

A unique responsibility of PROC_COMT_CMD is implementation of the "+" and "-" window paging commands, which are implemented only in comment screens. The paging commands require a series of loops over the window/buffer, first to move the current window contents into the buffer, then to move the desired section of the buffer into the window. The algorithm sets the page size at 8 lines even though the window has nine, thus ensuring that one line from the previous "page" the user saw always appears on the new "page". Once the new window is set up, utility screen SET_MORE is called to decide if there is more text "below" the window in the buffer; if so, the layout variable more is set to "more".

Response to a help ("?") request is implemented in the standard fashion (see Section 8.4.3.8.1).

Screen clearing in a comment screen flushes both the window and the buffer, but leaves the key fields intact.

Deletion ("D") is allowed in a comment screen if the user has privileges. This both flushes the window and buffer and deletes any comments in the underlying relation which match the key field values shown. Note that there is an implicit assumption that there is no data subsidiary to the comments, since the VERIFYD utility is not called.

PROC_COMT_CMD also provides the usual user-convenience functions, such as editor access; see Section 8.4.3.8.1 for a discussion.

Note that comment screens also support the "L" and "D" function commands, for text line insertion and deletion. These are implemented by calls to the INS_COMT_LINE and DEL_COMT_LINE utility screens in FUNCTION L and FUNCTION D sections of the comment screen itself.

8.4.3.8.6 PROC_HELP_CMD

The help screen command processor, shown in Figure 8-43, works very much like PROC_MENU_CMD, except that a more limited range of user actions is supported. The user may not Quit back to the command system from a help screen, and may not use the "=" command to jump to a data or menu screen. These actions would disrupt flow of control between screens, since help screens are always CALLed into. The actions are therefore limited to data and menu screens.

Another difference is that the "/" and "^" commands merely cause a RETURN SCREEN command to be issued in the help screen, rather than construction of a SET SCREEN command.

8.4.3.8.7 PROC_REPT_CMD

The report generation utility command processor, shown in Figure 8-44, is a very special-purpose command processor which supports only the REPORT general-purpose utility screen. REPORT is an unusual utility screen in that it presents the user with a

Figure 8-43. PROC_HELP_CMD Subroutine Screen

```
*** SCREEN PROC_HELP_CMD
*** INITIAL
    AKSHUN := ""
    COMCHR := $SUBSTR(COMMAND,1,1)
    SCREEN_NAME := $SUBSTR(COMMAND,2)
    IF $IN(COMCHR,"^","/")
        AKSHUN := "RETURN SCREEN"
        RETURN SCREEN
    ENDIF
    IF COMCHR = "?"
        DISPLAY "Getting screen..."
        IF SCREEN_NAME <> ""
            IF $MATCH(SCREEN_NAME,"HELP") <> 0
                AKSHUN := "CALL SCREEN %screen_name"
            ELSE
                AKSHUN := "FAIL ""Only help screens may be pushed &
                                  into with ?NAME."" "
            ENDIF
        ELSE
            IF $MATCH(NEXT_HELP,"""")
                AKSHUN := NEXT_HELP
            ELSE
                AKSHUN := "[CALL SCREEN %next_help]"
            ENDIF
        ENDIF
        RETURN SCREEN
    ENDIF
    IF COMCHR="L"
        CALL SCREEN SET_INSPCT_MODE
    ELSEIF COMCHR=":"
        CALL SCREEN MPECOMMAND
    ELSEIF COMCHR="T" AND SCREEN_NAME=""
        CALL SCREEN RUNTDP
    ELSEIF COMCHR="T" AND SCREEN_NAME="H"
        CALL SCREEN RUNEDITOR
    ELSEIF COMCHR="Z" AND USERLEVL>=3
        IF SCREEN_NAME="ON"
            TRACE ON
        ELSEIF SCREEN_NAME="OFF"
            TRACE OFF
        ELSEIF SCREEN_NAME="VAR"
        PROMPT "Name of variable to display value for:",MPECOM
        SHOW VARIABLE %mpecom
        PAUSE 9
        ENDIF
    ELSEIF COMCHR="Q" OR COMCHR="E"
        DISPLAY "Sorry, you cannot quit from a help screen. Use / first."
    ELSEIF COMCHR="="
        DISPLAY "Sorry, direct screen jumps may not be made from help &
                 screens.  Use / first."
    ELSE
        AKSHUN := "CMD_NOT_FND"

    ENDIF
    RETURN SCREEN
```

Figure 8-44. PROC_REPT_CMD Subroutine Screen

```
*** SCREEN PROC_REPT_CMD
*** INITIAL
NOTE      --CATCH THE REQUEST FOR A SIMPLE PRINT
    IF NUM = 1
        DISPLAY "Preparing printout"
        USE PARTITION &now_cursor
        SET OPTION QUOTES=NO
        IF INMODE="LATEST"
            INUM  := $MATCH(RESET,"BY")+3
          MPECOM := $SUBSTR(RESET,INUM ,$MATCH(RESET,"DATADATE")-INUM -1)
            SELECT &reset &modeset AND &
                          AND ENTRY_DATE=$MAX(ENTRY_DATE [BY &mpecom])
        ELSE
            SELECT &reset &modeset
        ENDIF
        SET OPTION QUOTES=YES
        CALL SCREEN SET_DEVICE
        SYSTEM $CANCEL
        PRINT:P:F:D
        SET OPTION QUOTES=NO
        SELECT &reset &modeset
        SET OPTION QUOTES=YES
    ELSEIF NUM>1
NOTE      --DO THE REQUESTED PRINT
        DISPLAY "Preparing printout"
        $SUBSCRIPT := NUM-2
        USE PARTITION REPTLIST
        SELECT SREPTS.@,REPTEX.@ BY CALLER,REPTFILE WHERE &
          SREPTS.REPTFILE=REPTEX.REPTFILE
        RECORD POINT
        IF $FOUND
            RECORD READ USING REPTLIST
            SELECT
            USE PARTITION &now_cursor
            IF REPTYPE="SCREEN"
NOTE            --BUILDER-TYPE REPORT SPECIFICATION
                CALL SCREEN SET_DEVICE
                [USE PARTITION &reptpartn]
               IF $MATCH(REPTFILE,".") <>0
                SCREEN_NAME:= $SUBSTR(REPTFILE,1,$MATCH(REPTFILE,".")-1)
               ENDIF
                IGNORE ERROR 7721
                CALL SCREEN &screen_name
                IF $ERROR
                    PROMPT "Can't find report file in group .screens.  &
                            Press any key to go on.",Y
                    USE PARTITION &now_cursor
                    RETURN SCREEN
                ENDIF
            ELSE
NOTE            --STRAIGHT RELATE EXECUTE FILE, SUBST IN SCENARIO NAMES
                CALL PROCEDURE PREPREPT &reptfile
```

Figure 8-44. PROC_REPT_CMD Subroutine Screen

```
                CALL SCREEN SET_DEVICE
                [USE PARTITION &reptpartn]
                SYSTEM $CANCEL
                EXECUTE DBURTEMP
            ENDIF
NOTE
            USE PARTITION &now_cursor
            RETURN SCREEN
        ELSE
            DISPLAY "No report defined for that number."
            SELECT
        ENDIF
    ENDIF
    USE PARTITION &now_cursor
    RETURN SCREEN
```

layout and accepts commands there. It is designed, however, to be called from any data screen, using the data screen's data structure values and information from the data dictionary to decide what actions to take.

REPORT uses the PROC_HELP_CMD utility to process all "standard" command options. PROC_REPT_CMD is called only if the user places the number of one of the menu choices which REPORT presents to him in the COMMAND field. PROC_REPT_CMD's task is to figure out what report the user is asking for and to give it to him.

There are three basic types of report which the DBU can produce. The most general is always option 1 on the REPORT layout menu: this is a simple RELATE PRINT of a selection on the current data relation. If the data retrieval mode is "ALL" the selection is the normal retrieval path; if "LATEST", the selection must include an aggregate to force return of data for only the latest DATADATE and ENTRY_DATE values for each entity. The SET_DEVICE subroutine screen is called to ensure that output goes to the hard copy device of the user's choice.

The other two types of report are a RELATE EXECUTE file, presumably containing CREATE commands, or a BUILDER subroutine screen (which may also contain CREATE commands). For menu options numbered greater than 1 the name of the executable file will be in the appropriate element of the REPTFILE array of the REPORT screen, which is of course available to PROC_REPT_CMD. The $SUBSCRIPT variable is set to this element, and a RECORD POINT is issued on a selection of data dictionary files which contains a list of the report files executable from the given data screen (a similar selection was used to retrieve the menu list the user viewed). The characteristics of the report file, most notably its name and type, are then retrieved by a RECORD READ.

If the report is a BUILDER screen, a CALL to it is constructed. It is presumed to reside in the .screens group. SET_DEVICE is first called to set the output device, and the default BUILDER partition is changed to the one specified in the data dictionary for this report. Note that in general reports should use the REPORT partition, which is open on a dedicated RELATE process, in order to avoid memory-induced aborts resulting from the crowding of most DBU RELATE sons.

If the report is a RELATE EXECUTE file it is presumed to reside in the .rprocs group. One problem with EXECUTE files is that they will typically need to know the scenario the user is running under in order to conform to scenario security, but they do not have access to BUILDER variables containing that information. The preprept FORTRAN procedure takes care of this problem by pre-processing EXECUTE files before they are sent to RELATE for execution. Preprept looks for instances of '[file.group]' in the execute file lines, substituting the proper scenario key field value for the given relation (file.group) under the current scenario into the execute file where the bracketed expression appears. Report developers can thus write general purpose SELECTions (e.g. SELECT SHPDATA.@ WHERE SCENARIO="[SHPDATA.DATABASE]") and be confident that scenario security will be preserved. Preprept writes the processed version into a temporary file named DBURTEMP, which is then executed by PROC_REPT_CMD. See Section 10.3 for more information about preprept.

8.4.3.9 Comment Screen Support Utilities

DBU comment screens are supported by six utility screens: PROC_COMT_CMD (see Section 8.4.3.8.4), COMT_INIT, COMT_UPDATE, SET_MORE, INS_COMT_LINE, and DEL_COMT_LINE.

It is assumed that the reader is already familiar with the information in Sections 8.4.3.4 and 8.4.3.8.4.

8.4.3.9.1 COMT_INIT

This screen is responsible for the part of comment screen initialization which retrieves a comment from the data base and displays it. The code for the screen is given in Figure 8-45.

There are two major cases: either the comment for the current data screen record is not already in memory or it is. If it is, the only requirement is to transfer its first nine lines from the COMMENT buffer variable into the DCOMMENT layout "window" variable.

If it is not in memory, then:

1) Access to the appropriate relation must be swapped into the COMMENT partition if it hasn't already been, via standard file management system methods.

2) A look to see if there is in fact an existing comment in the relation must be taken via a RECORD POINT.

3) If a comment is found then it must be read in and displayed on the layout. During the read, any lines of a previous comment must be flushed from the buffer if the new comment is less than 24 lines long. Note that the "LNUM25" logic is a relic of days when the BREAK option of RECORD POINT did not work; this logic is now effectively non-functional.

4) If there is no existing comment, both the buffer and the window must be cleared.

Figure 8-45. COMT_INIT Subroutine Screen

```
*** SCREEN COMT_INIT
*** INITIAL
    IF SCENARIO<>SCENARIONM
        DISPLAY "Warning: scenario security will forbid any changes here."
    ELSE
        DISPLAY "Working."
    ENDIF
    IF COMT_ACTIVE <> "Y"
NOTE    --GET THE FILE IF NECESSARY
        IF NOW_COMT_FILE <> $CONCAT(COMT_FILE,".",COMT_GROUP)
            DISPLAY "Fetching data files for this screen."
            USEFILE := COMT_FILE
            USEGROUP := COMT_GROUP
            USECURSOR := "%now_cursor"
            CALL SCREEN GETFIL
            SET OPTION QUOTES=OFF
            SELECT %reset %modeset
            SET OPTION QUOTES=ON
            NOW_COMT_FILE := $CONCAT(COMT_FILE,".",COMT_GROUP)
        ENDIF
NOTE    --LOOK FOR AN EXISTING VERSION OF THE COMMENT
        COMT_ACTIVE := "Y"
        COMT_ALTERED := ""
        LNUM := 1
        RECORD POINT;BREAK=%comt_break
        IF $FOUND
NOTE        --FOUND EXISTING VERSION, READ IN AND DISPLAY IT
            LNUM := 0
            INUM := 0
            WHILE NOT $EOF AND LNUM <> 25
                $SUBSCRIPT := INUM
                RECORD READ
                INUM := INUM+1
            ENDWHILE
            WHILE INUM<=23
                $SUBSCRIPT := INUM
                COMMENT := ""
                INUM := INUM+1
            ENDWHILE
            COMT_START := 0
            CALL SCREEN SET_MORE
            NUM := 0
            WHILE NUM <= 8
                $SUBSCRIPT := NUM
                DCOMMENT := COMMENT
                DNUM := NUM+1
                NUM := NUM+1
            ENDWHILE
        ELSE
NOTE        --NO EXISTING COMMENT, CLEAR MEMORY
            ARRAY
                COMMENT := ""
```

Figure 8-45. COMT_INIT Subroutine Screen

```
                ENDARRAY
                ARRAY
                     DCOMMENT := ""
                     DNUM := $SUBSCRIPT+1
                ENDARRAY
                MORE := ""
           ENDIF
      ELSE
NOTE       ——WE HAVE THE COMMENT IN MEMORY, DISPLAY IT
           NUM := 0
           WHILE NUM <= 8
                $SUBSCRIPT := COMT_START+NUM
                BUFFER := COMMENT
                $SUBSCRIPT := NUM
                DCOMMENT := BUFFER
                DNUM := COMT_START+NUM+1
                NUM := NUM+1
           ENDWHILE
      ENDIF
      RETURN SCREEN
```

8.4.3.9.2 COMT_UPDATE

This screen, displayed in Figure 8-46, is responsible for storing the comment in memory in the comment storage relation.

It assumes that the proper path is already set up on the COMMENT partition, and does the usual RECORD POINT. If it finds no records already in the comment relation, a simple loop of RECORD ADDs can perform the task.

If, however, there are existing lines in the relation, the task is complicated by the necessity to ensure that the new version fully overwrites them. This is done by a mixture of RECORD UPDATEs, RECORD ADDs, and RECORD DELETEs. The logic first counts the number of non-blank comment lines in the memory buffer. It then UPDATEs new lines over lines in the relation either until there are no more new lines or until there are no more in the relation. In the former case the remainder in the relation are deleted; in the latter case the remainder of the new lines are added.

8.4.3.9.3 SET_MORE

Shown in Figure 8-47, SET_MORE's task is simply to fill in the "more" variable on the comment screen layout. This variable should say "more" if there are additional lines in the comment buffer below those showing in the window; otherwise it should be blank.

The screen makes its determination by finding the last non-blank line in the buffer and combining this with the buffer location of the first line showing in the window.

8.4.3.9.4 INS_COMT_LINE and DEL_COMT_LINE

These screens (shown in Figures 8-48 and 8-49) implement the ESC L and ESC D commands, the ones which allow the user to insert a blank comment line into the middle of existing comment

Figure 8-46. COMT_UPDATE Subroutine Screen

```
*** SCREEN COMT_UPDATE
*** DECLARATIONS
    ENTRY_BY ;ALPHA
*** INITIAL
    DISPLAY "Saving your changes to associated comments."
    USE PARTITION COMMENT
    LNUM := 1
    RECORD POINT;BREAK=&comt_break
    IF NOT $FOUND AND ADD="Y"
NOTE          --ADD
         ENTRY_BY := NOW_USER
         ARRAY
              LNUM := $SUBSCRIPT+1
              RECORD ADD
         ENDARRAY
    ELSEIF MODFY="Y" AND $FOUND
NOTE          --DO MODIFY
         NUM := 23
         WHILE BUFFER="" AND NUM>=0
              $SUBSCRIPT := NUM
              BUFFER := COMMENT
              NUM := NUM-1
         ENDWHILE
          NUM := NUM+2
          IF NUM=1 AND COMMENT=""
               LNUM := NUM+1
          ELSE
               LNUM := 1
          ENDIF
          ENTRY_BY := NOW_USER
          WHILE LNUM<=NUM AND $FOUND
               $SUBSCRIPT := LNUM-1
               RECORD UPDATE
               LNUM := LNUM+1
               RECORD POINT;BREAK=&comt_break
          ENDWHILE
          IF $FOUND
NOTE
               WHILE NOT $EOF AND LNUM <> 25
                    RECORD DELETE
                    RECORD READ
               ENDWHILE
               IF NUM=1 AND BUFFER=""
                    RECORD DELETE
               ENDIF
          ELSE
               WHILE LNUM<=NUM
                    $SUBSCRIPT := LNUM-1
                    RECORD ADD
                    LNUM := LNUM+1
               ENDWHILE
          ENDIF
```

Figure 8-46. COMT_UPDATE Subroutine Screen

```
ELSE
    DISPLAY "Comments could not be saved for security reasons."
ENDIF
USE PARTITION &now_cursor
RETURN SCREEN
```

Figure 8-47. SET_MORE Subroutine Screen

```
*** SCREEN SET_MORE
*** INITIAL
    NUM := 23
    BUFFER := ""
    WHILE (BUFFER="") AND (NUM > 8)
         $SUBSCRIPT := NUM
         BUFFER := COMMENT
         NUM := NUM-1
    ENDWHILE
    NLEFT := NUM-(COMT_START+9) + 1
    IF NLEFT > 0
         MORE := "more"
    ELSE
         MORE := ""
    ENDIF
    RETURN SCREEN
```

Figure 8-48. INS_COMT_LINE Subroutine Screen

```
*** SCREEN INS_COMT_LINE
*** INITIAL
    IF $VARIABLE <> "DCOMMENT"
        DISPLAY "You must place the cursor on the line to follow the insert."
        RETURN SCREEN
    ENDIF
    INUM := $SUBSCRIPT
    DISPLAY "Inserting line..."
NOTE    --SAVE THE CURRENT PAGE
    NUM := 0
    WHILE NUM <= 8
        $SUBSCRIPT := NUM
        BUFFER := DCOMMENT
        $SUBSCRIPT := COMT_START+NUM
        COMMENT := BUFFER
        NUM := NUM+1
    ENDWHILE
NOTE    --SEE IF THERE IS ANY MORE SPACE
    $SUBSCRIPT := 23
    IF COMMENT <> ""
        DISPLAY "No more space.  A maximum of 24 lines of comments allowed."
        RETURN SCREEN
    ENDIF
NOTE    --REWRITE THE BUFFER DOWNWARDS
    NUM := 22
    WHILE NUM >= INUM
        $SUBSCRIPT := NUM
        BUFFER := COMMENT
        $SUBSCRIPT := NUM+1
        COMMENT := BUFFER
        NUM := NUM-1
    ENDWHILE
    $SUBSCRIPT := INUM
     COMMENT := ""
NOTE    --RETRIEVE THE CURRENT PAGE AND SET MORE
    NUM := 0
    WHILE NUM <= 8
        $SUBSCRIPT := COMT_START+NUM
        BUFFER := COMMENT
        $SUBSCRIPT := NUM
        DCOMMENT := BUFFER
        INUM := COMT_START+NUM+1
        NUM := NUM+1
    ENDWHILE
    NLEFT := NLEFT+1
    IF NLEFT > 0
        MORE := "more"
    ELSE
        MORE := ""
    ENDIF
    DISPLAY
    RETURN SCREEN
```

Figure 8-49. DEL_COMT_LINE Subroutine Screen

```
*** SCREEN DEL_COMT_LINE
*** INITIAL
    IF $VARIABLE <> "DCOMMENT"
        DISPLAY "You must place the cursor on the line to be deleted."
        RETURN SCREEN
    ENDIF
    DISPLAY "Removing line..."
    INUM := $SUBSCRIPT
NOTE    --SAVE THE CURRENT PAGE
    NUM := 0
    WHILE NUM <= 8
        $SUBSCRIPT := NUM
        BUFFER := DCOMMENT
        $SUBSCRIPT := COMT_START+NUM
        COMMENT := BUFFER
        NUM := NUM+1
    ENDWHILE
NOTE    --REWRITE THE BUFFER UPWARDS
    NUM := INUM
    IF NUM=23
        $SUBSCRIPT := NUM
        COMMENT := ""
    ELSE
        WHILE NUM < 23
            $SUBSCRIPT := NUM+1
            BUFFER := COMMENT
            $SUBSCRIPT := NUM
            COMMENT := BUFFER
            NUM := NUM+1
        ENDWHILE
        $SUBSCRIPT := 23
        COMMENT := ""
    ENDIF
NOTE    --RETRIEVE THE CURRENT PAGE AND SET MORE
    NUM := 0
    WHILE NUM <= 8
        $SUBSCRIPT := COMT_START+NUM
        BUFFER := COMMENT
        $SUBSCRIPT := NUM
        DCOMMENT := BUFFER
        INUM := COMT_START+NUM+1
        NUM := NUM+1
    ENDWHILE
NOTE    --RESET 'MORE'; WORK BACKWARDS, SEE IF LAST NON-BLANK ON SCREEN
    NLEFT := NLEFT-1
    IF NLEFT > 0
        MORE := "more"
    ELSE
        MORE := ""
    ENDIF
    COMT_ALTERED := "Y"
    DISPLAY

    RETURN SCREEN
```

text or to delete a comment line. They represent a minimal, primitive word processing capability for comment management.

Line insertion and deletion requires only artful management of the comment buffer and window arrays. To insert a line, for example, the contents of the window are first saved to the buffer so no information is lost. The contents of the buffer from the last non-blank line to the line to be blanked are then "pushed down" one line by an assignment loop which moves each line from COMMENT into a temporary buffer, and then from this buffer back into the next-lower line of COMMENT. The requisite line can then be blanked, and the appropriate buffer page moved into the window.

8.4.3.10 Legal Values Subsystem Utilities

The legal values subsystem ensures that users have placed only valid values in legals fields, and makes it easier for them to do so. It consists of four utility screens: LEGALS_INIT, NEXT_LEGAL, PREV_LEGAL, and CHECK_LEGALS.

8.4.3.10.1 LEGALS_INIT

Figure 8-50 shows the code for LEGALS_INIT. This screen is responsible for displaying default values in each legal field on a data screen as part of the data screen's initialization. It does this by running a processing loop over each variable mentioned in the LEGAL_FIELDS list. This loop queries the dbflds.db data dictionary relation to get the name of the .legals relation the field's legals list is stored in. A selection is issued on the LEGALS partition to make this relation current with the SHOWPREF (order-of-appearance) field as its index field. The legal value desired as the default for this screen, as specified by a SHOWPREF number extracted from the LEGLFIRST list, is then retrieved by the usual POINT/READ strategy and the loop continues on to the next field.

Figure 8-50. LEGALS_INIT Subroutine Screen

```
*** SCREEN LEGALS_INIT
*** INITIAL
    USE PARTITION LEGALS
    NUM := 1
    SCREEN := NOW_SCREEN
    SCRVAR := $ITEM(LEGAL_FIELDS,NUM,1)
    WHILE SCRVAR <> ""
        NOWLEGAL := SCRVAR
NOTE        USE PARTITION SCRFLDS
NOTE        RECORD POINT
NOTE        RECORD READ USING SCRFLDS
        RECORD POINT USING DBFLDS;KEY=SCRVAR
        RECORD READ USING DBFLDS
NOTE        USE PARTITION LEGALS
        SELECT %leglfile.%scrvar,%leglfile.SHOWPREF BY SHOWPREF
        LEGLFIRST := $ITEM(LEGLPREF1,NUM,1)
        RECORD POINT;KEY=LEGLFIRST
        RECORD READ
        NUM := NUM+1
        SCRVAR := $ITEM(LEGAL_FIELDS,NUM,1)
        ENDWHILE
    USE PARTITION %now_cursor
    RETURN SCREEN
```

Note that all the legals files are presumed to be opened simultaneously on the single LEGALS partition, which is somewhat unusual in DBU file management. This greatly simplifies the legals subsystem's code, however, and the routines of this subsystem are in general the only ones which use the LEGALS partition or legals relations.

8.4.3.10.2 NEXT_LEGAL and PREV_LEGAL

These two screens, shown in Figures 8-51 and 8-52, implement the ESC + and ESC - data screen commands.

They identify the legal field the user is working with by detecting cursor position on the layout with the $VARIABLE facility. If this field was the last one served by the legals subsystem (it is NOWLEGAL), then processing time can be saved because its path is already set up, with the record pointer in the appropriate place, on the LEGALS partition. In this case a simple RECORD READ gets the next legal value for the field (note that the field must always have the same name on the screen as it does in the legals file, making transmission of the value to the layout transparent). In the event of ESC - the ordering variable (SHOWPREF) must be decremented and a POINT/READ issued.

If the proper legals file is not current on the partition, the data dictionary must be consulted to find out the name of the file, with the field name being used as the key for the query. Then a selection must be given with the field name as the indexing variable, so that a point can use the variable's value on the layout to get to the record in the relation corresponding to the value. A READ is done to get the associated SHOWPREF value into the SHOWPREF variable, and then the select ordering the relation by SHOWPREF can be issued. In the NEXT_LEGAL case a POINT on this path will again get to the same record as is implicitly showing on the layout, and two reads will move along to the next record. For PREV_LEGAL SHOWPREF can just be decremented and the usual POINT/READ done.

Figure 8-51. NEXT_LEGAL Subroutine Screen

```
*** SCREEN NEXT_LEGAL
*** INITIAL
    SCRVAR := $VARIABLE
    IF SCRVAR = NOWLEGAL
         RECORD READ USING LEGALS
         IF $EOF
              RECORD REWIND USING LEGALS
              RECORD READ USING LEGALS
         ENDIF
         RETURN SCREEN
    ELSE
         USE PARTITION LEGALS
         DISPLAY "One moment while I retrieve the legals list for this field."
         SCREEN := NOW_SCREEN
note          RECORD POINT USING SCRFLDS
note          RECORD READ USING SCRFLDS
         RECORD POINT USING DBFLDS;KEY=SCRVAR
         RECORD READ USING DBFLDS
         IF LEGLFILE=""
         USE PARTITION %now_cursor
         DISPLAY "The field the cursor is in has no legal values list."
         RETURN SCREEN
         ENDIF
         SELECT %leglfile.%scrvar,%leglfile.SHOWPREF BY %scrvar
         RECORD POINT;KEY=%scrvar
         RECORD READ
SELECT %leglfile.%scrvar,%leglfile.SHOWPREF,%leglfile.FULLNAME  BY SHOWPREF
         RECORD POINT;BREAK=0
         RECORD READ
         RECORD READ
         IF $EOF
              RECORD REWIND
              RECORD READ
         ENDIF
         NOWLEGAL := SCRVAR
         USE PARTITION %now_cursor
    ENDIF
    DISPLAY
    RETURN SCREEN
```

Figure 8-52. PREV_LEGAL Subroutine Screen

```
*** SCREEN PREV_LEGAL
*** INITIAL
    SCRVAR := $VARIABLE
    IF SCRVAR = NOWLEGAL
         SHOWPREF := SHOWPREF-1
         RECORD POINT USING LEGALS
         RECORD READ USING LEGALS
         RETURN SCREEN
    ELSE
         DISPLAY "One moment while I retrieve the legals list for this field."
         SCREEN := NOW_SCREEN
         USE PARTITION LEGALS
note         RECORD POINT USING SCRFLDS
note         RECORD READ USING SCRFLDS
         RECORD POINT USING DBFLDS;KEY=SCRVAR
         RECORD READ USING DBFLDS
         IF LEGLFILE=""
         USE PARTITION %now_cursor
         DISPLAY "The field the cursor is in has no legal values list."
         RETURN SCREEN
         ENDIF
         SELECT %leglfile.%scrvar,%leglfile.SHOWPREF BY %scrvar
         RECORD POINT;KEY=%scrvar
         RECORD READ
SELECT %leglfile.%scrvar,%leglfile.SHOWPREF,%leglfile.FULLNAME BY SHOWPREF
         SHOWPREF := SHOWPREF-1
         RECORD POINT
         RECORD READ
         NOWLEGAL := SCRVAR
         USE PARTITION %now_cursor
    ENDIF
    DISPLAY
    RETURN SCREEN
```

8.4.3.10.3 CHECK_LEGALS

Shown in Figure 8-53, CHECK_LEGALS validates values in the legal fields on the data screen as part of the overall validation checks overseen by PROC_DATA_CMDB. This screen is similar to LEGALS_INIT in that it loops over the fields named in the LEGAL_FIELDS list. For each field it must (unless the field is NOW_LEGAL) query the data dictionary to get the legals relation name, make that relation the current path on the LEGALS partition via a select, and do a POINT to see if the screen value can be found in the relation. The OK flag is the output of the screen.

The last field processed is set up as NOW_LEGAL at the end of the screen's processing by giving the appropriate selection and POINT/READ commands.

8.4.3.11 Data Validation Utilities

The DBU performs four kinds of data validation, in addition to the data type checking that BUILDER automatically enforces (e.g. forcing dates to be entered in date fields). These are join checking, key uniqueness checking, ensuring that complete deletions occur, and legals-field value verification. A utility screen is responsible for the bulk of the work in each case: VERIFYJ, VERIFU, VERIFYD, and CHECK_LEGALS. CHECK_LEGALS is discussed in Section 8.4.3.10.3.

These screens are the DBU's primary line of defense against data base integrity degradation.

8.4.3.11.1 VERIFYJ

The code of VERIFYJ is shown in Figure 8-54. This complex screen is responsible for ensuring that "companion" data is present in other relations whenever a given record is to be added to a given relation. In other words, it checks that joins can be made; it allows network and hierarchy relationships to be enforced in a fundamentally relational data base.

Figure 8-53. CHECK_LEGAL Subroutine Screen

```
*** SCREEN CHECK_LEGAL
*** INITIAL
        DISPLAY "Checking legal-type (underlined) field values."
    SCREEN := NOW_SCREEN
    ALLOK := 1
    USE PARTITION LEGALS
    NUM := 1
    SCRVAR := $ITEM(LEGAL_FIELDS,NUM,1)
    WHILE SCRVAR <> ""
        IF SCRVAR = NOWLEGAL
            SELECT %leglfile.%scrvar,%leglfile.SHOWPREF BY %scrvar
            RECORD POINT USING LEGALS
            OK := $FOUND
        ELSE
            RECORD POINT USING DBFLDS;KEY=SCRVAR
            RECORD READ USING DBFLDS
            SET PATH %leglfile
            SELECT %leglfile.%scrvar,%leglfile.SHOWPREF BY %scrvar
            RECORD POINT;KEY=%scrvar
            OK := $FOUND
            NOWLEGAL := ""
        ENDIF
        IF NOT OK
            SCROLL  "The value of the %scrvar field is invalid.  &
                     Tab to that field and use ESC +."
        ALLOK := 0
        ENDIF
        NUM := NUM+1
        MPECOM := SCRVAR
        SCRVAR := $ITEM(LEGAL_FIELDS,NUM,1)
    ENDWHILE
    OK := ALLOK
    IF OK
        RECORD READ
    ENDIF
SELECT %leglfile.%mpecom,%leglfile.SHOWPREF,%leglfile.FULLNAME  BY SHOWPREF
    NOWLEGAL := FLDNAM
    IF OK
        DISPLAY "Legal field values ok."
        RECORD POINT
        RECORD READ
    ENDIF
    USE PARTITION %now_cursor
    RETURN SCREEN
```

Figure 8-54. VERIFYJ Subroutine Screen

```
*** SCREEN VERIFYJ
*** INITIAL
NOTE          ---SUBROUTINE SCREEN TO CHECK THAT JOIN CONDITIONS MET
NOTE          ---REQUIRES %mode,%now_file,%now_group, RETURNS %y,%addok
    USE PARTITION FILJOIN
    SELECT @ BY TGTFILE,TGTGROUP WHERE SRCFILE="%now_file" AND &
                                       SRCGROUP="%now_group"
NOTE      ---IF NO CONDITIONS THEN CAN QUIT
    IF NOT $FOUND
        USE PARTITION %now_cursor
        RETURN SCREEN
    ENDIF
    DISPLAY "Verifying that key values exist elsewhere in data base."
    COND := ""
    ADDOK := "Y"
    Y := "Y"
    USE PARTITION CHECK1
    JOINFLDS := ""
NOTE      ---SET UP FOR FIRST CONDITION
    RECORD READ USING FILJOIN
    MPECOM := JOINMSG
    WORKFILE := TGTFILE
    WORKGROUP := TGTGROUP
    DISPLAY "Checking for join in file %workfile.%workgroup."
    SAVMSG := JOINMSG
    SAVTYP := JOINTYPE
    IN_ONE := SAVTYP="IN_ONE:A"
    USEFILE := WORKFILE
    USEGROUP := WORKGROUP
    USECURSOR := "CHECK1"
    CALL SCREEN GETFIL
    NOMORE := 0
NOTE      ---CONSTRUCT CONDITION BY GETTING FIELD NAMES
    WHILE NOMORE = 0
        FLDNUM := $RDBINFO(TGTFIELD)
        LIST := $RDBINFO(201,FLDNUM)
        FLDTYP := $ITEM(LIST,3)
        FLDFMT := $ITEM(LIST,11)
        FLDVAL := %srcfield
        JOINFLDS := $CONCAT(JOINFLDS,",",TGTFIELD)
NOTE      ---ALPHA/DATE  OR  NUMERIC   FIELD TYPE
        IF FLDTYP=1 OR (FLDTYP=4 AND $BITS(FLDFMT,13,3)=1)
            COND := $CONCAT(COND," %tgtfield=","  ""%fldval"" ","  AND ")
        ELSE
            COND := $CONCAT(COND," %tgtfield=","%fldval","  AND ")
        ENDIF
        RECORD READ USING FILJOIN
        ENDFILE := $EOF
        IF ENDFILE OR TGTFILE<>WORKFILE OR TGTGROUP<>WORKGROUP
            LASTFAIL := IN_ONE AND (ENDFILE OR &
            ($SUBSTR(JOINTYPE,1,6)<>"IN_ONE" OR $SUBSTR(JOINTYPE,8,1)="A"))
            COND := $SUBSTR(COND,1,$LENGTH(COND)-4)
```

Figure 8-54. VERIFYJ Subroutine Screen

```
                SET OPTION QUOTES=OFF
                SELECT LINE=$LINE WHERE %cond
                SET OPTION QUOTES=ON
                IFOUND := $FOUND
NOTE    BREAK
                IF NOT IFOUND AND (NOT IN_ONE  OR LASTFAIL)
                     USE PARTITION FILSCREENS
                     SELECT MPECOM=SCREEN WHERE FILE="%workfile" AND &
                                                GROUP="%workgroup"
                     RECORD READ
                     LIST := ""
                     WHILE NOT $EOF
                          LIST := $CONCAT(LIST,MPECOM," or ")
                          RECORD READ
                     ENDWHILE
                     IF LASTFAIL
                          WORKGROUP := "*"
                          SAVTYP :="REQUIRED"
                     ENDIF
                     LIST := $SUBSTR(LIST,1,$LENGTH(LIST)-3)
                     JOINFLDS := $SUBSTR(JOINFLDS,2)
                     SCROLL "Matching key values not found in &
                          %workfile.%workgroup for fields %joinflds.  &
                %savmsg  Use screen %list to add %savtyp values."
                     IF  SAVTYP="REQUIRED"
                          IF ADDOK="Y"
                               PROMPT "Cannot %mode data base.  Check for &
                                       other problems?(%y):",ANSWER
                               IF ANSWER<>"" AND (ANSWER="Y" OR ANSWER="N")
                                    Y := ANSWER
                               ENDIF
                               ADDOK := " "
                          ENDIF
                          IF Y="N"
                               ADDOK := ""
                               SELECT
                               USE PARTITION %now_cursor
                               RETURN SCREEN
                          ENDIF
                          DISPLAY "Checking..."
                     ENDIF
                     USE PARTITION CHECK1
                ENDIF
                SELECT
                IF ENDFILE
                     NOMORE := 1
                ENDIF
                IF NOT NOMORE
NOTE                 --SET UP FOR NEXT JOIN TARGET FILE
                     IF IFOUND AND IN_ONE AND NOT LASTFAIL
                          TGTGROUP := $CONCAT(TGTGROUP,ZEES)
                           RECORD POINT USING FILJOIN
```

Figure 8-54. VERIFYJ Subroutine Screen

```
                RECORD READ USING FILJOIN
                NOMORE := ENDFILE
                WHILE NOT NOMORE AND $SUBSTR(JOINTYPE,1,6)="IN_ONE"&
                     AND JOINTYPE<>"IN_ONE:A"
                    TGTGROUP := $CONCAT(TGTGROUP,ZEES)
                    IF COND=TGTGROUP
                        TGTFILE := $CONCAT(TGTFILE,ZEES)
                    ENDIF
                    COND := TGTGROUP
                    RECORD POINT USING FILJOIN
                    RECORD READ USING FILJOIN
                    NOMORE := $EOF
                ENDWHILE
            ENDIF
        ENDIF
        IF NOT NOMORE
            IN_ONE := NOT ENDFILE AND $SUBSTR(JOINTYPE,1,6) = "IN_ONE"
            COND := ""
            WORKFILE := TGTFILE
            WORKGROUP := TGTGROUP
        DISPLAY "Checking for join in file %workfile.%workgroup."
        USEFILE := WORKFILE
        USEGROUP := WORKGROUP
        USECURSOR := "CHECK1"
        CALL SCREEN GETFIL
            JOINFLDS := ""
        ENDIF
    ENDIF
    SAVMSG := JOINMSG
    SAVTYP := JOINTYPE
    LASTFAIL := 0
ENDWHILE
DISPLAY
USE PARTITION %now_cursor
RETURN SCREEN
```

VERIFYJ's job is to ensure that the conditions specified in the filjoin.db data dictionary relation are satisfied. This relation's records contain the fields SRCFILE, SRCGROUP, SRCFIELD, TGTFILE, TGTGROUP, TGTFIELD, JOINTYPE, FLEV, DATANAME, and JOINMSG. Several records with the same SRCFILE,SRCGROUP and TGTFILE,TGTGROUP values comprise a condition. Each record specifies a pair of fields which are equated in a SELECT WHERE clause joining the two files; a set of such pairs forms a condition; the condition is that the resulting join must return at least one record.

There are (at present) three types of join for these purposes, with the type specified by the JOINTYPE field value for any given condition/set. REQUIRED joins must be satisfied or the data base modification cannot be allowed. RECOMMENDED joins cause a warning to be issued to the user. IN_ONE_x join requirements themselves come in groups: at least one of the conditions in the group must be satisfied. An example of an IN_ONE join is the requirement that a ship deactivation record to be added to deact.miscj must have a corresponding ship construction/conversion/reactivation record in one of ncjodat.projj, ncjodat.currj, or ncjodat.histj. Thus required joins implement hierarchies, while in_one joins implement networks.

The FLEV field gives the order of significance of the fields (records) in a given set. This ordering is done implicitly in that FLEV appears in all FILJOIN indexed even though it is not specified in the VERIFYJ select BY clauses. The DATANAME and JOINMSG fields provide text for messages to the user in the event of a failure.

In order to perform its task, then, VERIFYJ must loop over all sets of filjoin records (each set comprising a single join condition) for the file the calling data screen is serving. The

initial set-up for this is done by issuing a selection on the FILJOIN partition which orders the relation by TGTFILE,TGTGROUP and restricts it to SRCFILE="%now_file" AND SRCGROUP="%now_group". The "now_" variables are set during data screen initialization.

The major steps taken during each iteration of the loop are: retrieve the target relation onto the CHECK1 utility partition via the file management subsystem; construct a WHERE clause from the filjoin records; issue a SELECT on CHECK1 using this clause; go on to the next condition if a record is $FOUND on the selection; issue error messages and prompt for continuation of checking if the join condition is violated by a $FOUND failure.

In detail, the processing is:

1) The first 25 lines conduct initialization already discussed and retrieve the first target file. Note also that certain field values from filjoin must be saved at the start of working with a given set (these values will be the same in all records in the set) because the way the screen detects the end of a set is by reading past it into a new one. If the values of JOINMSG, JOINTYPE, TGTFILE, and TGTGROUP were not saved the logic would end up using the values for the next set, the one read into, by mistake.

2) A gigantic WHILE loop forms the rest of the screen. Its continuation condition flag, NOMORE, is true only when an end-of-file is returned on the FILJOIN partition, signifying that all conditions for the SRC_ relation have been processed. The loop executes ONCE PER RECORD READ FROM FILJOIN.

3) The TGTFIELD name retrieved by each filjoin read is used in $RDBINFO query on the SRC relation's partition to determine the data type of the field, and this information is then used to concatenate an additional 'AND %tgtfield=appropriate layout value' clause onto the WHERE condition (COND) being constructed. Note the trick of moving the value of the "source" variable into a variable of known name (FLDVAL) by a simple assignment statement.

4) The next filjoin record is read. If an end-of-file is returned or we have read into the first record of the next condition/set (identifiable by a change in the values of the TGTFILE and TGTGROUP fields), then processing goes to step 5. Otherwise, we drop through to the ENDWHILE statement and the loop continues.

5) LASTFAIL is set to true (1) if we have read past the end of the set of conditions forming an IN_ONE, network type join requirement. LASTFAIL is irrelevant otherwise.

6) The selection is issued and $FOUND consulted to see if the join requirement is met. If so, or if LASTFAIL is false (i.e. there are still network paths to try), then we can move on to the next condition (set of filjoin records). We drop to the IF ENDFILE statement. If there are more conditions to process, then the next target file is retrieved onto CHECK1 by the file management subsystem.

   If the successful join was of the network variety, it is necessary to read past any additional IN_ONE sets for this join in filjoin.db to get to the next join set. Otherwise, spurious join failures would be produced. This is done in the code immediately after 'SET UP FOR NEXT JOIN TARGET FILE' by incrementing the values of TGTFILE and TGTGROUP and POINTing, a strategy similar to the one used in PROC_DATA_CMDB's Next methodology.

   COND is then cleared, the TGT variables saved in their WORK_ counterparts, and IN_ONE set (if we have a new network-type join test coming up). Finally, GETFIL can be called.

7) If, on the other hand, there is a join failure when the selection is issued, the user must be informed. First the name of the screen serving the target relation is retrieved from the filscrns.db data dictionary relation for inclusion in messages (note that there may be more than one such screen---thus the concatenation into the LIST variable). The message is issued, including the name of the screen to go use, the names of the fields the join failed on (in JOINFLDS, which was built up during the loop operation), and SAVMSG (JOINMSG) from filjoin.

8) If it was a required join and it's the first failure, the user is prompted to see if he wants checking to continue. If not, the screen returns.

8.4.3.11.2 VERIFYU

Screen VERIFU is shown in Figure 8-55. Relatively simple in comparison with VERIFYJ, this screen finds out whether a record exists in the current main relation with the same key values as are showing on the layout. It finds out which fields to include as key fields by looking at the list in VERIFYFIELDS. Looping over the elements of this list, it constructs a select WHERE clause in COND by consulting RELATE's information about the current relation's fields via the $RDBINFO command (which provides data type information). The select is then issued; if a record is found then uniqueness is violated. The main data retrieval path is then reset and the record pointer returned to the current record (if possible).

8.4.3.11.3 VERIFYD

This most complex of DBU subroutine screens is shown in Figure 8-56. VERIFYD's job is to delete all records "below" the record currently on the layout (the one the user has just asked to have deleted) in any implicit data base hierarchies and networks. Where VERIFYJ ensures that no data can be added in the middle of a hierarchy or network without "father" or higher-up data already being in place, i.e. that no data gets into the data base without proper companion data, VERIFYD ensures that hierarchy and network branches are trimmed out completely, i.e. that no data is left in the data base without proper companion data.

The screen uses the contents of the filscrn.db relation in a manner similar to VERIFYJ (readers unfamiliar with VERIFYJ should read Section 8.4.3.11.1 before continuing with this Section), but in a reverse fashion. Instead of using the current main data relation as the source and looking for join targets which must be satisfied, the current relation is treated as the target for the join requirements of other relations (sources).

Figure 8-55. VERIFYU Subroutine Screen

```
*** SCREEN VERIFYU
*** INITIAL
    DISPLAY "Verifying that the values for %verifyfields are unique."
    COND := ""
    FLDNAM := $ITEM(VERIFYFIELDS,1,1)
    IF FLDNAM = ""
         RETURN SCREEN
         ENDIF
    NUM := 1
    WHILE FLDNAM <> ""
         FLDNUM := $RDBINFO(FLDNAM)
         LIST := $RDBINFO(201,FLDNUM)
         FLDTYP := $ITEM(LIST,3)
         FLDFMT := $ITEM(LIST,11)
         FLDVAL := %fldnam
         IF FLDTYP=1 OR (FLDTYP=4 AND $BITS(FLDFMT,13,3)=7)
            COND := $CONCAT(COND," %fldnam=","""%fldval""","," AND ")
         ELSE
            COND := $CONCAT(COND," %fldnam=","%fldval"," AND ")
         ENDIF
         NUM := NUM+1
         FLDNAM := $ITEM(VERIFYFIELDS,NUM,1)
         ENDWHILE
    COND := $SUBSTR(COND,1,$LENGTH(COND)-4)
    SET OPTION QUOTES=NO
    SELECT DATEU=ENTRY_DATE WHERE %cond
    SET OPTION QUOTES=YES
    IF $FOUND
         UOK := 0
         RECORD READ
    ELSE
         UOK := 1
    ENDIF
    SET OPTION QUOTES=NO
    SELECT %reset %modeset
    SET OPTION QUOTES=YES
    RECORD POINT;BREAK=0
    DISPLAY
    RETURN SCREEN
```

Figure 8-56. VERIFYD Subroutine Screen

```
*** SCREEN VERIFYD
*** DECLARATIONS
    TGTF,TGTG,WORKFILE,SRCFILE,WORKGROUP,SRCGROUP;LENGTH=8;UPPER
    NOWFILER,NOWGROUPR ;GLOBAL
    DONE,RECURS,CANCEL ;GLOBAL
    ASKME ;GLOBAL
    DCOND,DCOND2 ;LENGTH=300
    Y ;LENGTH=1;UPPER
    NOMORE ;NUMERIC
    SAVTYP   ;LENGTH=12
    SAVMSG   ;LENGTH=20
*** INITIAL
NOTE     ---SUBROUTINE SCREEN FORCING DELETION OF CHILDREN IF PARENT DELETED
NOTE     ---REQUIRES %recurs,%now_file,%now_group, RETURNS %delok
NOTE     ***ROUTINE IS RECURSIVE; USE CARE IN CHANGING ITS DATA STRUCTURE
    DELOK := "Y"
    IF RECURS
NOTE     ---CALLED BY ITSELF
         TGTF := NOWFILER
         TGTG := NOWGROUPR
         RECURS := RECURS+1
    ELSE
NOTE     ---FIRST CALL
         TGTF := NOW_FILE
         TGTG := NOW_GROUP
         RECURS := 1
         DONE := 0
         CANCEL := 0
         ASKME := ""
         Y := ""
         DISPLAY "Making sure no other data in the data base depends &
                  on this data."
    ENDIF
    USE PARTITION FILJOIN
    SELECT @ BY SRCFILE,SRCGROUP WHERE TGTFILE="%tgtf" AND &
                                       TGTGROUP="%tgtg"
    IF NOT $FOUND
         DISPLAY
         RECURS := RECURS-1
         RETURN SCREEN
    ENDIF
    RECORD READ USING FILJOIN
NOTE     ---SET UP CHECK OF FIRST CHILD FILE
    SAVTYP := JOINTYPE
    SAVMSG := DATANAME
    WORKFILE := SRCFILE
    WORKGROUP := SRCGROUP
    NOMORE := 0
NOTE     ---CONSTRUCT FIELD LIST FOR THIS CHILD
    WHILE NOMORE=0
         USE PARTITION %now_cursor
         FLDNUM := $RDBINFO(SRCFIELD)
```

Figure 8-56. VERIFYD Subroutine Screen

```
        LIST := $RDBINFO(201,FLDNUM)
        FLDTYP := $ITEM(LIST,3)
        FLDFMT := $ITEM(LIST,11)
        IGNORE ERROR 7313
        FLDVAL := %tgtfield
        NUM := $ERROR
       IF NOT NUM
        IF (FLDTYP=1 OR (FLDTYP=4 AND $BITS(FLDFMT,13,3)=1))
             DCOND := $CONCAT(DCOND," %srcfield=","""%fldval""","," AND ")
             DCOND2 := $CONCAT(DCOND2," %tgtfield=","""%fldval""","," AND ")
        ELSE
             DCOND := $CONCAT(DCOND," %srcfield=","%fldval"," AND ")
             DCOND2 := $CONCAT(DCOND2," %tgtfield=","%fldval"," AND ")
        ENDIF
        RECORD READ USING FILJOIN
        NOMORE := $EOF
       ELSE
        SRCFILE := $CONCAT(SRCFILE,ZEES)
        SRCGROUP := $CONCAT(SRCGROUP,ZEES)
       ENDIF
        IF NOMORE OR SRCFILE<>WORKFILE OR SRCGROUP<>WORKGROUP OR NUM
             DCOND := $SUBSTR(DCOND,1,$LENGTH(DCOND) - 4)
             USE PARTITION [CHECK[%recurs]]
             SELECT
             USEFILE := WORKFILE
             USEGROUP := WORKGROUP
             USECURSOR := "[CHECK[%recurs]]"
             CALL SCREEN GETFIL
             SET OPTION QUOTES = OFF
             SELECT LINE=$LINE WHERE %dcond
             SET OPTION QUOTES=ON
NOTE         --IF A MATCH IS FOUND, ADDITIONAL DATA MAY HAVE TO BE DELETED, BUT
NOTE         --ONLY IF THERE IS NO DUPLICATE ON THIS KEY IN CURRENT FILE
             IF $FOUND
NOTE              --IS THERE A DUPLICATE IN THIS FILE?
NOTE                 TEST APPLIES ONLY WHEN RECURS = 1
                  IF RECURS = 1
                       USE PARTITION %now_cursor
                  ELSE
                       OK := 1
                  ENDIF
                  IF RECURS = 1
                  DCOND2 := $SUBSTR(DCOND2,1,$LENGTH(DCOND2)-4)
                  SET OPTION QUOTES=OFF
                  SELECT LINE=$LINE WHERE %dcond2
                  SET OPTION QUOTES=ON
                  RECORD READ
                  RECORD READ
                  OK := $EOF
                  SELECT
                  ENDIF
                  IF OK
```

Figure 8-56. VERIFYD Subroutine Screen

```
NOTE                  --if no duplicate, do deletion or give warning
                      IF SAVTYP="REQUIRED" OR &
                         $SUBSTR(SAVTYP,1,6)="IN_ONE"
NOTE                       --prompt for prompting and first-time stuff
                           IF DONE=0
                            SCROLL "Data in other files which depends on &
                                this will have to be deleted before this &
                                data can be.  The DBU will do this &
                                automatically, but will prompt for &
                                approval before taking action if you wish."
                            WHILE ASKME<>"Y" AND ASKME<>"N"
                                 ASKME := "Y"
                             PROMPT "Do you want to approve &
                                    step-by-step?(%askme):",ANSWER
                             IF ANSWER<>"" AND (ANSWER="Y" OR ANSWER="N")
                                  ASKME := ANSWER
                             ENDIF
                            ENDWHILE
                            DONE := 1
                            IF ASKME="Y"
                                CALL PROCEDURE BGTRNS
                            ELSE
                                Y := "Y"
                            ENDIF
                           ENDIF
                           IF ASKME="Y"
                               WHILE Y<>"Y" AND Y<>"N"
                                Y := "Y"
                                PROMPT "Delete %savmsg associated with &
                                       this data?(%y): ",ANSWER
                                IF ANSWER<>"" AND (ANSWER="Y" OR ANSWER="N")
                                     Y := ANSWER
                                ENDIF
                               ENDWHILE
                           ENDIF
                           WHILE Y="N"
                               Y :="Y"
                               PROMPT "Cancel this deletion and &
                               restore any deleted associated data? &
                               (%y): ",ANSWER
                               Y := ANSWER
                               IF Y="Y"
                                    CALL PROCEDURE ABTRNS
                                    CANCEL := 1
                                    DELOK := "N"
                                    USE PARTITION %now_cursor
                                    RETURN SCREEN
                               ELSE
                                    Y := "Y"
                                    PROMPT "Only choices are cancel &
                                    or go through with whole deletion.  &
                                    Do deletion?(%y): ",ANSWER
```

Figure 8-56. VERIFYD Subroutine Screen

```
                                    IF ANSWER<>"" AND (ANSWER="Y" OR &
                                                       ANSWER="N")
                                        Y := ANSWER
                                    ENDIF
                                    PAUSE 4
                                ENDIF
                            ENDWHILE
                            USE PARTITION [CHECK[%recurs]]
                            SET OPTION QUOTES=NO
                            SELECT
                            DELETE FOR %dcond
                            SET OPTION QUOTES=YES
NOTE                        --now recurs to catch sons of sons
                            NOWFILER := WORKFILE
                            NOWGROUPR := WORKGROUP
                            CALL SCREEN VERIFYD
                            IF CANCEL
                                RETURN SCREEN
                            ENDIF
                            USE PARTITION FILJOIN
                            SELECT @ BY SRCFILE,SRCGROUP &
                             WHERE TGTFILE="%tgtf" AND TGTGROUP="%tgtg"
                            RECORD POINT
                            RECORD READ
                        ELSE
                            SCROLL "Remember to delete %savmsg &
                            if appropriate in this case."
                        ENDIF
                    ENDIF
                ENDIF
                DCOND := ""
                DCOND2 := ""
                WORKFILE := SRCFILE
                SAVMSG := DATANAME
                SAVTYP := JOINTYPE
                WORKGROUP := SRCGROUP
            ENDIF
        ENDWHILE
        DISPLAY
        IF ASKME="Y" AND RECURS=1
            CALL PROCEDURE DOTRNS
        ENDIF
        RECURS := RECURS-1
        RETURN SCREEN
```

As an example, we would expect a record in ncjodat.projj for a DDG-51 class ship to have companion data in the (target) relation shdesc.miscj describing the DDG-51 class as a whole. In turn, we would expect comments for the ship in ncjocom.projj to be joinable to the main ship schedule record in ncjodat.projj. If the DDG-51 record in shdesc is deleted, then both the ncjodat and ncjocom records must also be deleted.

This requirement that VERIFYD feel its way down chains of data dependency made it necessary to give the screen a recursive structure. When VERIFYD tries to delete the record in ncjodat.projj in the above example (a relation which can be thought of as one step removed from shdesc.miscj), it must call itself to ensure that any records dependent on the ncjodat.projj record are also deleted (e.g., the comments).

This is why VERIFYD, alone among utility screens, has a significant DECLARATIONS section. Since there can be multiple "copies" of VERIFYD on the BUILDER execution stack at any one time, each one must have some local variables so that the copies do not interfere with one another.

Table 8-17 describes the purpose of each of VERIFYD's declared local variables.

The general approach used in VERIFYD is as follows:

1) Construct a select WHERE clause using the information in filjoin.db and on the layout. The selection is to be given on the subsidiary file that is suspected of having data which will need deletion (the SRC file as specified in filjoin.db).

2) Give the selection and see if any records are returned. If any are, they will have to be deleted unless there is another record in the TGT file (the "father" in the hierarchy or network) with the same key values as the one to be deleted. For example, if we are attempting to delete a record for the ship DDG-67, but there is another record in the file for DDG-67 (perhaps with a later DATADATE value), then we neither need nor want to delete subsidiary data.

TABLE 8-17. VERIFYD SPECIAL VARIABLES

VARIABLE	PURPOSE
	GLOBAL VARIABLES WITH SPECIAL USES
ASKME	Set to "Y" if the user wants to be prompted before each deletion.
CANCEL	Set to 1 if user wants to abort the deletion. Tells recursive invocations of VERIFYD to quit what they're doing and RETURN.
DONE	Set to 1 if the user has been asked whether he wants to be prompted before each deletion.
NOWFILER NOWGROUPR	When VERIFYD calls itself, it must tell its son invocation what data base relation to treat as the root of its particular tree (i.e. what relation will be the "TGT" relation for purposes of filjoin.db queries. The name of this relation is put into these two variables.
RECURS	Indicates the number of times VERIFYD has been called. If greater than 1, recursive processing is taking place.
	VARIABLES LOCAL TO EACH INVOCATION OF THE SCREEN
TGTF TGTG	Name of the "TGT" relation for filjoin.db queries during this invocation.
SRCFILE SRCGROUP	Name of the "SRC" relation which for which the current join check is being conducted. In the context of the example in the text, shdesc.miscj would be a TGT and ncjodat.projj a SRC. The SRC relations are those in which data will be deleted. Note that as the recursion chain is moved down relations not on one end or the other will be processed both as sources and as targets.
WORKFILE WORKGROUP	Working storage for SRCFILE, SRCGROUP. The latter can take on the name of the file for which the next check will be made, so the WORK versions are the ones generally used.
DCOND DCOND2	Buffer variables in which the SELECT WHERE /DELETE FOR clauses are conducted. DCOND is for the query for deletable data on the SRC relation; DCOND2 is used only when RECURS=1, to find out if there

TABLE 8-17. VERIFYD SPECIAL VARIABLES

VARIABLE	PURPOSE
	is a duplicate record in the TGT relation which will permit deletion to be avoided.
Y	User prompt responses.
NOMORE	Set to 1 if there are no more join conditions to test, i.e. we have an end-of-file from filjoin.db.
SAVTYP SAVMSG	Buffers in which messages destined for the user, and read from filjoin.db, are stored.

This reason for not deleting subsidiary data can apply only during the first invocation of VERIFYD (i.e. the one where TGT is the relation the calling data screen is concerned with), not during any recursive invocations. This is because we are only deleting one record in the "top" relation, but are doing wholesale deletions in the subsidiaries. For example, say we are deleting a DDG-51 class description record in shdesc.miscj. We have to check and see if there is a second DDG-51 description in shdesc.miscj before we delete all the individual schedules for ships of this class in ncjodat.projj. Say it's the only record and the schedules therefore have to go. Now when VERIFYD calls itself to see if there is any data subsidiary to the schedules, and finds comments, we know immediately the comments will have to go, since we're going to delete ALL the DDG-51 schedules (VERIFYD will be issuing a DELETE FOR command).

3) Having decided that subsidiary data will have to be deleted, the user must be prompted to see if he wants to abort the whole deletion process (maybe he didn't realize that all that vital data would be flushed). He can have an abort option before every deletion decision, or can give a blanket approval.

4) If the user wants to approve at each step, RELATE's transaction management facility must be invoked so that data can be un-deleted if the user decides late in the game (i.e. after several approvals) that he wants to back out. In principle this is simple: just give a BEGIN TRANSACTION command, which tells RELATE not to post any DB changes until a COMMIT TRANSACTION command is given. However, the DBU is working with multiple RELATE son processes, and a BEGIN TRANSACTION command must be issued on each process (not on each partition, just once per process). But there is no guarantee that the named BUILDER partitions of the DBU contain a set of actual partitions such that each process is represented. In order to ensure that BEGINs are issued everywhere, a FORTRAN procedure (bgtrns) is called which uses the contents of the file management system extra data segment to do the job. Similar routines (dotrns and abtrns) are called to issue DBU-global COMMIT TRANSACTION and ABORT TRANSACTION commands when these are necessary.

5) If the user gives the go-ahead for deletion of this particular data, a DELETE FOR command (whose FOR clause is the same as was used in the select WHERE clause which detected the deletable data) is issued.

6) VERIFYD calls itself to check the next level down the tree (if any). Eventually this call will return, and processing proceeds to the next subsidiary listed in filjoin.db.

Thinking of the network of relationships between relations in the form of the standard inverted tree, with the relation the calling data screen serves at the top, VERIFYD works its way all the way down each branch before going on to the next branch.

Note that the entire process is implictly based on the field values showing on the layout of the current data screen. These values are used to make up the condition clauses of every SELECT and DELETE statement that is issued.

Now considering the processing in detail in the order in which it appears in the code, VERIFYD does the following:

1) First the screen must initialize itself. If this is the original call (i.e. one made by PROC_DATA_CMDB and not by itself) then the global flags must be set up: RECURS is set to 1, indicating that the first call is in progress, DONE and CANCEL are set to 0 to indicate that there is more to check and that the user has not backed out, and ASKME and Y to blanks to indicate that the user has not yet been prompted. These global variables are the means of communication between multiple copies of VERIFYD on the execution stack.

   The name of the target file to work with, i.e. the name of the file for which a deletion is proposed, is put in TGTF and TGTG, which must be local since each invocation of the screen works with a different target. On the first call, the target name is NOW_FILE from the data screen; on recursive calls the target name is communicated by the calling invocation through the NOWFILER and NOWGROUPR global variables.

   Each time a recursive call is made the RECURS variable is incremented; each RETURN decrements. Thus VERIFYD knows that it's all done when the value is reduced to 0.

2) A selection is issued on filjoin.db to set up retrieval of the sets of records which specify join conditions for the target relation. If there are no conditions an immediate RETURN can be issued.

3) Processing starts in on a loop similar in outline to the one in VERIFYJ; the loop increments once per filjoin record read. The name of the source relation and relevant messages are put in working variables before the loop starts so they are not inadvertantly erased by a filjoin read past the end of a join condition set.

4) The loop can be divided into two parts: the small amount of code at the top which constructs the WHERE clause used to test for deletable data, and the large amount of code which issues the testing select, prompts the user, etc. Construction of the clause is basically the same as in VERIFYJ, with two extensions.

   First, it may happen during a recursive call that a field named in a filjoin-specified condition will not appear on the current data screen's layout, i.e. it will not be in the data screen's main relation. For example, there is no HULL field in the shdesc.miscj relation, but this is a part of the join between ncjodat.projj and ncjocom.projj. VERIFYD will attempt to discover the data type of HULL by a $RDBINFO call on the partition serving shdesc. This will fail, as is appropriate since there is no HULL value to be used in conditions on the current layout anyway. Condition construction for the given join will be terminated on such a failure, and the test will be made only for a join on the fields already processed.

   Since fields are read from filjoin in order of significance (this is what the FLEV field in filjoin specifies), failure of a field to be significant (relevant) to the situation means any subsequent fields will also be irrelevant. In the running example used so far, we only need to see if there are comments for any DDG-51 class ship in shcomt, since we're going to delete then for ALL HULLS in this case.

   Second, two condition clauses must be constructed (these are placed in DCOND and DCOND2). One uses the field names of the "source" file; this condition is the one which tests whether there is data requiring deletion in the subsidiary. DCOND2 uses the field names in the "target" (father) relation; this condition is used to test for duplicate records there.

5) Condition construction terminates on a $RDBINFO query failure (as just explained), when a read on the filjoin selection returns an end-of-file, or when a check of the record returned from filjoin indicates it is for the next join set.

6) The trailing AND is stripped from the condition, and the "source" (son) relation is retrieved onto the utility partition corresponding to the current recursion level (e.g. CHECK3 if this is the third call to VERIFYD). Each invocation of VERIFYD must use a different CHECK# partition, so that the invocations do not interfere with one another. Note that this currently LIMITS THE RECURSION LEVEL TO 4, SINCE THAT IS HOW MANY CHECK

PARTITIONS ARE DEFINED. The selection testing for data requiring deletion is issued, and $FOUND evaluated.

7) If there is a deletion requirement, and this is the first call (RECURS=1), then the check for a duplicate record in the father is made.

8) If $RECURS_1$ or there is no duplicate (OK=1), and the join is of the REQUIRED type or the IN_ONE type, then the subsidiary data has to go, but first the user must be prompted. A one-time prompt asks if he wants to be prompted before each deletion (DONE ensures that it's a one-time prompt); his answer is stored in ASKME. The *bgtrns* FORTRAN routine issues a global BEGIN TRANSACTION if prompting is on.

9) If prompting is on, then he is asked if it's ok to delete the data. His answer is placed in the Y variable.

10) If he says no to the deletion, then he is forced to choose between canceling the entire deletion or changing his mind and goind ahead. If he decides to cancel, the FORTRAN routine abtrns aborts all deletions up to this point, CANCEL is set to 1 (so that if this was a recursive call the "higher" invocations of VERIFYD will know to cease processing, and the screen returns.

11) If deletion is approved, either explicitly or by no-prompting being chosen, then the DELETE FOR %dcond command is issued on the CHECK# partition to flush the subsidiary data.

12) The source file (subsidiary) in the current screen is then set to be the target (primary, NOWFILER) relation for the next recursive invocation of VERIFYD and the recursive CALL SCREEN VERIFYD is issued.

13) When this recursive call returns the first thing done is to reset the retrieval path on filjoin, since it is sure to have been disturbed. This involves reselection and a RECORD POINT/READ. Note that SRCFILE and SRCGROUP, the index variables on filjoin, are local in VERIFYD, so their values will not have been disturbed by the recursive call. Note also that SRCFILE and SRCGROUP will have the values for the <u>next</u> set of filjoin join text specification records if construction of the current test's condition was terminated by a read into the record's for this next test; if termination was for end-of-file then the POINT is irrelevant because the WHILE loop is about to terminate (NOMORE=1); if termination was from a $RDBINFO failure then SRCFILE and SRCGROUP were incremented just before the "IF NOMORE..." statement, ensuring that the POINT would jump to the next condition set.

14) Processing for a single condition is complete at this point in the code; the condition buffers are cleared and the working variables (WORKFILE, SAVMSG, SAVTYP, and WORKGROUP) are set to the values for the new set just read from the set's first filjoin record. The WHILE loop begins incrementing over this new set.

15) If deletions have been done, the user has asked to be prompted, and the primary (RECURS=1) invocation of VERIFYD is about to return, then the FORTRAN routine dotrns must be called to commit all the deletions that were done.

Extreme care must be used in modifying this screen because of its tremendously complex data structure. It is easy to get seemingly random errors caused by the changing of a global DBU variable by a recursively called screen, when the calling invocation is expecting this variable to retain the value it put there. Similar problems occur with partitions, since they are global resources.

8.4.3.12 Security Utilities

The DBU enforces three kinds of security: screen security, which keeps a user from even seeing screens he is not allowed to use; file security, which can be used to limit the functions he can perform in a given data screen; and scenario security, which ensures that he sees and changes only data belonging to the scenario he is working with (and that he makes no changes to data the scenario uses indirectly).

Screen and file security are both centered around the SECURITY_CHECK utility screen, which is called during data and menu screen initialization. SECURITY_CHECK is shown in Figure 8-57.

The screen just queries the two data base security privelege relations scrpriv.db and filpriv.db. First appropriate values for the current screen are moved into variables which have the names used in these relations (e.g. FILE and GROUP). Then a POINT is made into scrpriv.db to see if the user can access the

Figure 8-57. SECURITY_CHECK Subroutine Screen

```
*** SCREEN SECURITY_CHECK
*** INITIAL
    FILE := NOW_FILE
    GROUP := NOW_GROUP
    SCREEN := NOW_SCREEN
    USERNAME := NOW_USER-      RECORD POINT USING SCRSEC
    IF NOT $FOUND
         USERNAME := "ANY"
         RECORD POINT USING SCRSEC
         IF NOT $FOUND
              SEEIT := ""
         ELSE
              RECORD READ USING SCRSEC
         ENDIF
    ELSE
         RECORD READ USING SCRSEC
    ENDIF
    USERNAME := NOW_USER
    RECORD POINT USING FILSEC
    IF $FOUND
         RECORD READ USING FILSEC
    ELSE
         USERNAME := "ANY"
         RECORD POINT USING FILSEC
         IF NOT $FOUND
              READ := ""
              ADD := ""
              DELET := ""
              MODFY := ""
              UPDAT := ""
         ELSE
              RECORD READ USING FILSEC
         ENDIF
    ENDIF
    IF NOT READB
         READ := ""
    ENDIF
    IF NOT ALTDB
         ADD := ""
         DELET := ""
         MODFY := ""
         UPDAT := ""
    ENDIF
    RETURN SCREEN
```

given screen (a successful point either on the user's own name or on a user name of "ANY" will do). If no record can be found, then the user is automatically denied priveleges; it one is found then the subsequent RECORD READ sets the value of the SEEIT flag to that in the scrpriv record (i.e. a user can be denied (SEEIT="N") as well as permitted by an entry in the relation---and a specific denial for a user overrides an ANY entry).

A similar strategy is followed in querying the filpriv.db file usage privelege relation. Note that the priveleges specified in the relation apply only in the DBU. RELATE security applies everywhere else.

The filpriv RECORD READ returns values for all five of the file privelege flags. The ADD, DELET, MODFY, and UPDAT flags are used to determine the user's ability to use the corresponding commands in the given screen; the READ flag indicates whether he can see data at all.

Note that if the user does not have ALTDB=1 (alter data base) priveleges in his sysusr.sysro record (the master ALIAS security relation, read in INIT and SUSPEND screens), then all the data modification priveleges are automatically denied.

The SHOWPRIV subroutine screen (code in Figure 8-58) is called at the end of data screen initialization to fill in the ALTERCAPD variable at the bottom of the data screen layout with a message telling the user what his priveleges are. SHOWPRIV makes up a standardized message using the flag values set in SECURITY_CHECK as a guide, and then centers this message in a line of "====".

Scenario security is enforces as part of file management operations by including a WHERE clause specifying the appropriate scenario field key value in the selections issued to construct

Figure 8-58. SHOWPRIV Subroutine Screen

```
*** SCREEN SHOWPRIV
*** INITIAL
    ALTERCAPD := "Your priveleges in this screen are: "
    IF READ<>""
         ALTERCAPD := $CONCAT(ALTERCAPD," inspect,")
    ENDIF
    IF ADD<>""
         ALTERCAPD := $CONCAT(ALTERCAPD," add,")
    ENDIF
    IF MODFY<>""
         ALTERCAPD := $CONCAT(ALTERCAPD," modify,")
    ENDIF
    IF UPDAT<>""
         ALTERCAPD := $CONCAT(ALTERCAPD," update,")
    ENDIF
    IF DELET<>""
         ALTERCAPD := $CONCAT(ALTERCAPD," delete,")
    ENDIF
    INUM := $LENGTH(ALTERCAPD)-1
    ALTERCAPD := $SUBSTR(ALTERCAPD,1,INUM)
    IF INUM=34
         ALTERCAPD := "You may not look at or change data in this screen."
         INUM := 50
    ENDIF
    MPECOM := $SUBSTR(EQUALS,1,(78-INUM+1)/2 -1)
    ALTERCAPD := $CONCAT(MPECOM,ALTERCAPD,MPECOM)
    RETURN SCREEN
```

data screen retrieval paths. The getscenv FORTRAN procedure is called to extract the appropriate key value from the scenario system's extra data segment. See Section 8.4.3.3.3.

8.4.3.13 The Report Generation Subsystem

The report generation subsystem has already been discussed to some extent in Section 8.4.3.8.7 (PROC_REPT_CMD). This subsystem ensures that users always have a way to obtain a hard copy of the data they can access in a given data screen. It also provides a platform for addition of specific, special purpose report generators to support particular data screens.

The subsystem consists of the REPORT, PROC_REPT_CMD, and SET_DEVICE screens and the preprept FORTRAN subroutine.

The subsystem is invoked by the user giving the "P" command in a data screen. It can be thought of as comprising an additional screen type (i.e. in addition to menu, data, comment, and help screen types), since to the user an apparently different, customized screen appears when P is given in different data screens. In fact it is the REPORT screen which is always called. This screen's layout consists of a menu of numbered options, the contents of which is filled in based on the name of the data screen REPORT was called from.

There are four basic means of producing formatted output of data base data: the RELATE PRINT command, CREATE report generation commands given in an EXECUTE file or a BUILDER procedure, BUILDER output via SCROLL or DISPLAY, and FORTRAN subroutine output. REPORT provides for creation of reports using each method by allowing a report to take the form of a RELATE EXECUTE file or a BUILDER screen file (PRINT can be given in an EXECUTE file, and FORTRAN routines can be called from a BUILDER screen). In addition, REPORT always offers the user the option of a simple PRINT executed on the current main retrieval partition of the calling data screen.

The subsystem must perform three basic functions: present a menu of reports available to the user and accept the user's choice, find out which output device the report is to go to, and cause the report to be produced. These tasks are handled by the REPORT, SET_DEVICE, and PROC_REPT_CMD screens, respectively. See Section 8.4.3.8.7 for a discussion of PROC_REPT_CMD.

Screen REPORT (shown in Figure 8-59) both presents the menu and acts as the executive for the subsystem. In presenting the menu, it performs a selection on the REPTLIST partition which joins the srepts.db and reptex.db data dictionary relations. Srepts contains a list of the reports executable via REPORT from each data screen (screen name in field CALLER, and file name in REPTFILE). Reptex contains descriptive information about each REPTFILE (note that a given report file could be executed by more than one screen---thus the breakdown of the data into the two relations). The selection returns a list of reports executable from screen CALLER (:="%now_screen"), along with a description of each one. The description goes onto the REPORT screen layout, while the names of the report files go into an array variable local to REPORT.

The user may choose an option by number from the resulting menu, or may give any of the standard help screen commands. If a numeric character is the first one in COMMAND, then PROC_REPT_CMD is called; otherwise PROC_HELP_CMD is called.

Note that the sophisticated user may specify his menu option number as part of his "P" command when calling for REPORT from the data screen (e.g. "P5"). Display of the REPORT screen layout is bypassed and control is passed more or less directly to PROC_REPT_CMD.

Screen SET_DEVICE, whose function is to issue an appropriate 'FILE RDBLIST=' MPE command so that output is

Figure 8-59. REPORT Subroutine Screen

```
*** SCREEN REPORT
*** LAYOUT
SCREEN IS:  REPORT                                  SCENARIO IS: [scenariornm  ]
═══════════════════════════════════════════════════════════════════════════════
 ? for help              ALIAS DATA BASE UPDATE SYSTEM          ^ for data screen
═══════════════choose a report to print by number═══════════════════════════════

                          COMMAND: [command         ]

          1    Regular print, according to Inspection mode.
        [dnum]  [reptdesc                                                    ]
        [dnum]  [reptdesc                                                    ]
        [dnum]  [reptdesc                                                    ]
        [dnum]  [reptdesc                                                    ]
        [dnum]  [reptdesc                                                    ]
        [dnum]  [reptdesc                                                    ]
        [dnum]  [reptdesc                                                    ]
        [dnum]  [reptdesc                                                    ]
        [dnum]  [reptdesc                                                    ]
        [dnum]  [reptdesc                                                    ]
        [dnum]  [reptdesc                                                    ]
        [dnum]  [reptdesc                                                    ]

═══════════════no data may be changed here═══════════════════════════════════
*** DECLARATIONS
    DNUM; NUMERIC;ENHANCE=NONE;JUSTIFY=RIGHT;DISPLAY
    REPTDESC;ENHANCE=NONE;DISPLAY
    REPTFILE;LENGTH=8;ARRAY=12
    REPTYPE;LENGTH=8
    CALLER,REPTPARTN;LENGTH=15
    COMMAND;GLOBAL;ACTION
    SCENARIONM;GLOBAL
*** INITIAL
    DISPLAY "Getting report options screen"
    CALLER := NOW_SCREEN
         USE PARTITION REPTLIST
         SELECT SREPTS.@,REPTEX.@ BY CALLER,REPTFILE WHERE &
           SREPTS.REPTFILE=REPTEX.REPTFILE AND CALLER="%caller"
    ARRAY
         RECORD READ
         DNUM := $SUBSCRIPT+2
    ENDARRAY
    SELECT
    USE PARTITION %now_cursor
NOTE     ---DID THEY PASS IN A REPORT NUMBER?
    NUM := 0
    IF $SUBSTR(COMMAND,2,1) <> "" AND $NUMERIC($SUBSTR(COMMAND,2,1))
         MPECOM := $SUBSTR(COMMAND,2)
         NUM := %mpecom
    ENDIF
    IF NUM<>0
         CALL SCREEN PROC_REPT_CMD
```

Figure 8-59. REPORT Subroutine Screen

```
        RETURN SCREEN
    ENDIF
*** INITIAL EVERYTIME
    COMMAND := ""
*** FUNCTION
    DISPLAY "No ESCape functions available in this screen."
*** ENTER
    IF $NUMERIC(COMMAND)
NOTE     ---GAVE A NUMERIC OPTION
         NUM := COMMAND
         CALL SCREEN PROC_REPT_CMD
    ELSE
         CALL SCREEN PROC_HELP_CMD
         IF AKSHUN="CMD_NOT_FND"
              DISPLAY "The %command command is not operational here."
         ELSE
              SET OPTION QUOTES=NO
              %akshun
              SET OPTION QUOTES=YES
         ENDIF
    ENDIF
```

directed to the proper device, is discussed in Section 10.4. The FORTRAN subroutine preprept, called by PROC_REPT_CMD, is discussed in Section 10.3. This routine searches an editor-type file for instances of '[filename.groupname]', where the name is expected to be that of a scenario-dependent data relation, and substitutes the proper scenario key field value for that relation for the current ALIAS scenario. This allows developers to write scenario-independent RELATE EXECUTE files while still maintaining ALIAS scenario security.

Addition of new reports for a given data screen is quite easy: create the report file (EXECUTE file in the .rprocs group, screens in the .screens group), and make an addition to each of the srepts.db and reptex.db data dictionary relations.

8.4.3.14 The Field Help Subsystem

The FIELDHELP screen, shown in Figure 8-60, is the respondent to a user ESC ? command. The screen determines which field the user has the cursor in, and then extracts and presents information about this field from the data dictionary.

The name of the variable is placed in SCRVAR, and a point/read from the scrflds.db relation is done. This returns FIELDNAME (usually the same as SCRVAR) and FLDTYPE, and also the FILE.GROUP the field resides in. Then the DATATYPE, MAXVALUE, UNITYPE, and FORMATSPEC are extracted from dbflds.db (these are unique to the field), and the CODERESP and PRIMSOURCE and SECSOURCE are retrieved from fldfile.db (these values can vary from file to file for the same field). If additional reads on scrflds for the given SCREEN and SCRVAR can be made (note the BREAK condition on the original RECORD POINT) this indicates that the field is in multiple files underlying the screen from which FIELDHELP was called. These file names are indicated in the ALSO_IN variable on the layout. Finally, the text description for the variable is read from fldesc.db into the DESCRIP array.

Figure 8-60. FIELDHELP Subroutine Screen

```
*** SCREEN FIELDHELP
*** LAYOUT
SCREEN IS: FIELDHELP                                SCENARIO IS: [scenarionm  ]
═══════════════════════════════════════════════════════════════════════════════
 ? for help          ALIAS DATA BASE UPDATE SYSTEM HELP          ^ to return
═══════════════description of the field the cursor was in═══════════════════════

                          COMMAND: [command         ]

 Fieldname [fieldname ] in file [filgrp           ] is a [fldtype   ] field
                [also_in                                             ]
 Data type is [datatype     ] with max value [maxvalue  ] in [unitype ] units
                              [formatspec               ]
 Code [coderesp] is responsible.  Primary source is [primsource       ]
                                  Secondary source  [secsource        ]
      [descrip                                                       ]
      [descrip                                                       ]
      [descrip                                                       ]
      [descrip                                                       ]
      [descrip                                                       ]
      [descrip                                                       ]
      [descrip                                                       ]
      [descrip                                                       ]
═══════ESC functions: F-other files with this field; J-join files/fields═══════

*** DECLARATIONS
    COMMAND ;ACTION
    SCENARIONM;GLOBAL
    FIELDNAME,FILGRP,FLDTYPE,DATATYPE;DISPLAY;ENHANCE=NONE
    MAXVALUE,UNITYPE,FORMATSPEC,CODERESP;DISPLAY;ENHANCE=NONE
    PRIMSOURCE,SECSOURCE,DESCRIP,ALSO_IN;DISPLAY;ENHANCE=NONE
    NEXT_HELP ;LENGTH=80
*** INITIAL
    NEXT_HELP := "COMMANDHELPL"
    SCREEN := NOW_SCREEN
    SCRVAR := $VARIABLE
    IF SCRVAR="COMMAND"
         SCROLL "The COMMAND field is present in every DBU screen, &
                 but not in any data base file.  You should place &
                 the proper DBU command code in this field each time &
                 you wish to perform an action, and then press the &
                 RETURN key.  For a list of available DBU command codes, &
                 place a ? in the command field and press RETURN."
         RETURN SCREEN
    ENDIF
    RECORD POINT USING SCRFLDS;KEY=SCREEN,SCRVAR;BREAK=2
    IF NOT $FOUND
         SCROLL "No help available for that field."
         RETURN SCREEN
    ENDIF
    RECORD READ USING SCRFLDS
    RECORD POINT USING DBFLDS
```

Figure 8-60. FIELDHELP Subroutine Screen

```
    IF NOT $FOUND
         SCROLL "No help available for that field."
         RETURN SCREEN
    ENDIF
    RECORD READ USING DBFLDS
    FILGRP := $CONCAT(FILE,".",GROUP)
    RECORD POINT USING FLDFILE
    RECORD READ USING FLDFILE
NOTE      --SEE IF THERE ARE MULTIPLE FILES UNDERLYING HERE
NOTE     RECORD POINT USING SCRFLDS;BREAK=2
NOTE     RECORD READ USING SCRFLDS
    RECORD READ USING SCRFLDS
    WHILE NOT $EOF
         ALSO_IN := $CONCAT(ALSO_IN,FILE,".",GROUP,",")
         RECORD READ USING SCRFLDS
    ENDWHILE
    ALSO_IN := "[also in file  %also_in]"
    RECORD POINT USING SCRFLDS;KEY=SCREEN,SCRVAR;BREAK=2
    RECORD READ
NOTE
    RECORD POINT USING FLDESC;BREAK=1
    IF $FOUND
         ARRAY
              RECORD READ USING FLDESC
         ENDARRAY
    ENDIF
*** INITIAL EVERYTIME
    COMMAND := ""
*** ENTER
    CALL SCREEN PROC_HELP_CMD
    IF AKSHUN="CMD_NOT_FND"
         DISPLAY "The %command command is not operational here."
    ELSE
         SET OPTION QUOTES=NO
         IGNORE ALL ERRORS
         %akshun
         IF $ERROR=7721 OR $ERROR=7639
              DISPLAY "%screen_name is not a valid screen name."
         ENDIF
         SET OPTION QUOTES=YES
    ENDIF
*** FUNCTION
    DISPLAY "That ESCape function isn't available here."
*** FUNCTION F
    USE PARTITION SCRFLDS
    SET PATH FLDFILE
    SELECT FILE,GROUP,SCRFLDS.SCREEN BY FILE,GROUP &
           WHERE FIELDNAME="%fieldname" AND FILE=SCRFLDS.FILE AND &
                 FIELDNAME=SCRFLDS.FIELDNAME AND GROUP=SCRFLDS.GROUP
    $EOF := 0
    SCROLL "DATA BASE FILES HOLDING THE %fieldname FIELD"
    SCROLL
```

Figure 8-60. FIELDHELP Subroutine Screen

```
    SCROLL "  FILE    GROUP      DBU SCREEN"
    SCROLL "-------- -------- ---------------"
    SYSTEM $CANCEL
    PRINT:S:N
    SELECT
    SET PATH SCRFLDS
    USE PARTITION %now_cursor
*** FUNCTION J
    USE PARTITION FILJOIN
    SCROLL "FILES/FIELDS WHICH MUST ALREADY HAVE A VALUE MATCHING THIS FIELD'S"
    SCROLL
    SELECT TGTFILE,TGTGROUP,TGTFIELD,SRCFIELD BY TGTFILE,TGTGROUP &
          WHERE SRCFILE="%file" AND SRCGROUP="%group"
    RECORD READ
   WHILE NOT $EOF
    MPECOM := ""
    COND := ""
    WORKFILE := TGTFILE
    WORKGROUP := TGTGROUP
    WHILE NOT $EOF AND WORKFILE=TGTFILE AND WORKGROUP=TGTGROUP
         MPECOM := $CONCAT(MPECOM,TGTFIELD,",")
         COND := $CONCAT(COND,SRCFIELD,",")
         RECORD READ
    ENDWHILE
    MPECOM := $SUBSTR(MPECOM,1,$LENGTH(MPECOM)-1)
    COND := $SUBSTR(COND,1,$LENGTH(COND)-1)
    SCROLL
    SCROLL "Values for fields %cond must have matches in fields %mpecom &
           in file %workfile.%workgroup."
   ENDWHILE
NOTE
    SCROLL
    SCROLL
    SCROLL "FILES/FIELDS WHICH WILL EXPECT TO FIND A MATCHING VALUE HERE"
    SCROLL
    SELECT SRCFILE,SRCGROUP,SRCFIELD,TGTFIELD BY SRCFILE,SRCGROUP &
          WHERE TGTFILE="%file" AND TGTGROUP="%group"
    RECORD READ
   WHILE NOT $EOF
    MPECOM := ""
    COND := ""
    WORKFILE := SRCFILE
    WORKGROUP := SRCGROUP
    WHILE NOT $EOF AND WORKFILE=SRCFILE AND WORKGROUP=SRCGROUP
         MPECOM := $CONCAT(MPECOM,SRCFIELD,",")
         COND := $CONCAT(COND,TGTFIELD,",")
         RECORD READ
    ENDWHILE
    MPECOM := $SUBSTR(MPECOM,1,$LENGTH(MPECOM)-1)
    COND := $SUBSTR(COND,1,$LENGTH(COND)-1)
    SCROLL
    SCROLL "Values for fields %mpecom in file %workfile.%workgroup &
```

Figure 8-60. FIELDHELP Subroutine Screen

```
        will expect matches in fields %cond in the current file."
ENDWHILE
 SELECT
 USE PARTITION %now_cursor
```

Command processing in the screen is limited to the services offered by PROC_HELP_CMD, except that two special capabilities are provided. The user can request a dump to the screen of all the data base relations the given screen appears in (ESC F), produced by a selection on fldfile.db followed by a formatted print. He can also request a list of all the join conditions on the file the field is in by giving the ESC J command. This causes selections on the filjoin.db relation to be made, and lists of fields in each join condition to be constructed and output.

8.4.3.15 Miscellaneous Utilities

See Section 10.4 for a discussion of the MPECOMMAND, RUNTDP, RUNEDITOR, and SEARCH utility screens. These are sufficiently general in purpose, and independent of the DBU data structure, that they might prove useful in other screen applications.

The SUSPEND utility is shown in Figure 8-61. This is the screen which transfers control back to the ALIAS command system when the user issues a DBU "Q" command. It does so by calling the spsusp FORTRAN procedure, which merely issues an ACTIVATE system intrinsic call which suspends the DBU's BUILDER process and activates the (father) command system process. If and when the user wishes to re-enter the DBU, processing will pick up at this point. The DBU must then check to see what scenario the user is using (via a query of the snsusers.sysrw relation), and also what default hard copy output device is specified on the user environment parameter menu. The SCREENSYS Job Control Word must be reset to 9012, the DBU's value, so that the file management subsystem is sure to be able to find its extra data segment. Finally, a SYSTEM $CANCEL is issued to ensure that the CONTROL-Y keyboard option will be effective.

The SET_INSPCT_MODE utility is shown in Figure 8-62. This screen is the one called when the user issues an "L" command. It

Figure 8-61. SUSPEND Subroutine Screen

```
*** SCREEN SUSPEND
*** INITIAL
    DISPLAY "Returning to main menu system..."
    CALL PROCEDURE SPSUSP
    USE PARTITION DEFAULT
    SET PATH SNUSERS
    RECORD POINT;KEY=NOW_USER
    RECORD READ
    SET PATH ENVRN
    RECORD POINT;KEY=SCENARIO
    RECORD READ
    USE PARTITION %now_cursor
    :SETJCW SCREENSYS,9012
    SYSTEM $CANCEL
    SCROLL "Back in the DBU now."
    SCROLL "    "
    RETURN SCREEN
```

Figure 8-62. SET_INSPCT_MODE Subroutine Screen

```
*** SCREEN SET_INSPCT_MODE
*** INITIAL
    IF INMODE="ALL"
         MPECOM := "LATEST"
    ELSE
         MPECOM := "ALL"
    ENDIF
    PROMPT "Change data inspection mode from %inmode to %mpecom?(Y):",Y
    IF Y="Y" OR Y=""
         INMODE := MPECOM
    ENDIF
    NUM := 0
    WHILE NUM<1 OR NUM > 3
         PROMPT "Date type inspection mode: 1) Actual; 2) Planned; &
                 3) All :",NUM
    ENDWHILE
    IF NUM=1
         SHOWMODE := "ACTUAL"
    ELSEIF NUM=2
         SHOWMODE := "PLANNED"
    ELSE
         SHOWMODE := ""
    ENDIF
    IF DATESCREEN
         DATEMODE := SHOWMODE
         MODESET := "[ AND DATETYPE=""%datemode"" ]"
         USE PARTITION %now_cursor
         SET OPTION QUOTES=NO
         SELECT %reset %modeset
         SET OPTION QUOTES=YES
    ENDIF
    MODEDISP := $CONCAT(INMODE," ",DATEMODE," ",MODEDATA)
    RETURN SCREEN
```

is the means by which the retrieval mode setting may be changed from ALL to LATEST or vice versa, and from ACTUAL to PLANNED or ALL and vice versa. The settings are obtained from the user via prompts, and then are put into effect by revising the retrieval mode data structure and by issuing a select to reset the retrieval path on the current main data relation. Note that the ACTUAL/PLANNED/ALL distinction is implemented by clauses in the retrieval path select which depend on the existence of a DATETYPE field in the current data relation.

8.4.4 Expansion of the DBU

8.4.4.1 Step-By-Step Guide to Screen Creation

The DBU was designed explicitly to make creation of new screens to support an expanding ALIAS data base as easy and fast as possible. This design goal underlies much of the DBU's complex utility-oriented processing structure: since most functions are carried out by subroutine-like processors, the amount of code required to make up a new screen is very modest.

Giving the DBU the capability to support a new data relation typically requires creation of three or four screens, in addition to creation of the relation itself and the work of entering the definition of the relation in the data dictionary. Data, comment, and help screens must be made up, and either a new menu screen or addition of an option to an existing menu must be made.

The FIRST step in the process should always be entry of the new relation's description in the data dictionary, followed by creation of the relation using the MAKFIL.DBA utility screen. See the Data Base Reference Guide for instructions.

Screen creation is made easier by the existence of four screen templates, one for each type, in the files TDATA.TEMPLAT, TCOMT.TEMPLAT, THELP.TEMPLAT, and TMENU.TEMPLAT. Each template contains a complete skeleton of its screen type which conforms to DBU conventions; the developer need only fill in the layout section and any bracketed (" ") areas in the code where, e.g., names must be inserted.

Once the relation has been created, perform the following steps:

1) Text the data screen template into one of the editors, fill it in, and save it in a file in your working group. Do the same for the comment and help templates and, if necessary, the menu template, saving each in a separate

file. Follow the step-by-step instructions in the next few sections in filling in these templates. Be especially careful not use a name for any of your screens that has already been used for an existing screen.

2) Text in the file DBUDATA.SCREENS, which contains the code for menu, data, and associated help and comment screens (dbusubr.screens holds the utility screens, while dbumhelp holds the help screens which form the overall help subsystem). You must put an entry on at least one existing menu screen to provide users with a menu-path to your new data screen and/or menu screen. Make sure to both put the entry on the menu layout and to add an action section which calls your screen when the user chooses that entry number.

3) Use the editor's Join command to pull the code for your new screens into the dbudata text file, and then keep (save) it.

4) The version of the DBU code which ALIAS executes is a preprocessed or "compiled" version stored in the file DBU.SCREENS. This version must be re-created so that it is in agreement with the ASCII code files. Use the "CS" (compile-screen) UDC, while runs the BUILDER in compilation mode. Respond with "DBU.SCREENS" to its prompt for an output file, and then in response to its requests for input files type 'dbusubr.screens', 'dbudata.screens', 'dbumhelp.screens', and 'datdic.screens'.

5) Text your new screen thoroughly, paying particular attention to data validation. The most common errors are caused by erroneous or missing data dictionary entries, which often show up only after a period of time in the form of data relations whose contents do not conform to integrity standards.

Follow the instructions in the next sections when filling in the templates.

8.4.4.1.1 Guide to Filling in the Data Screen Template

The template shown in Figure 8-26 (stored in file TDATA.TEMPLAT) was created to make the task of data screen creation less burdensome and risky, and to promote standardization of screen formats and design. It contains a complete skeleton for a "standard" data screen, with places where

the developer must fill things in marked out by " ". Always use this template when creating a new data screen!

This section will give step-by-step instructions for filling in the template, taking the bracketed lines on the template in the order of their appearance. Be aware that following these instructions without reading Section 8.4.3 and understanding what is going on may well lead to trouble; this section is meant as a quick reference guide.

1) Fill in the screen name on the first line and in the "NOW_SCREEN:=" line. The name must be no more than 14 characters (not 15), less if possible, must not contain the string "HELP" anywhere, and should be mnemonic.

2) Fill in an informative subtitle at the top of the layout, and any message you feel would help the user at the bottom.

3) Place the data fields and associated labels into the blank area of the layout. Note that the layout is already arranged to give a blank area of maximum size. If there are more fields than will fit in the area, consider making some smaller by giving them alternative names (using the "NAME=" renaming capability in the DECLARATIONS section). As a last resort you can delete the message line at the bottom and add one more line in the middle of the layout.

4) Fill in the names and data types of all the variables you put on the layout in the declarations section.

5) Fill in the name of the data relation the screen will serve in the NOW_FILE:= and NOW_GROUP:= assignments lines. If there is to be more than one data relation, use the name of the relation the primary retrieval selection is to be issued on (see the CLASS_CHARS screen for an example).

6) Set EXTRACOM equal to 0 unless you will be working with multiple relations or must hard-wire some data base update logic into this screen. Understand the PROC_DATA_CMDB screen thoroughly before trying to write a screen where EXTRACOM must be 1.

7) Set DATESCREEN to 0 unless the data is divided into actual and planned; then set it to 1.

8) Set MODEDATA according to the type of data on the screen (choices shown on template).

9) Set DATEMODE:="" unless DATESCREEN was set to 1; then set DATEMODE:=SHOWMODE.

10) Fill in the fields in the retrieval index for the primary relation on the NOW_INDEX:= assignment line.

11) Fill in the names of the legals-type fields on the screen on the LEGAL_FIELDS:= line, and the associated default-value specification codes for each legal field on the LEGLPREF1 line.

12) Fill in the list of fields required to uniquely identify a given data record on the KEYFIELDS:= line, including DATADATE but not ENTRY_DATE, in the order they appear in now_index. A data record ID consists of an entity ID plus the datadate, e.g. YARD,DATADATE uniquely identifies the data for a given yard as of a given date.

13) For each field named in the KEYFIELDS list in the last step, determine its data type and enhancement specification code from the following list:

CODE	SPECIFICATION
1	;OPTIONAL
2	;OPTIONAL;ENHANCE=I,U
3	;OPTIONAL;DATE="MM/DD/CCCC"
4	;OPTIONAL;DATE="MM/DD/CCCC";ENHANCE=I,U
5	;OPTIONAL;UPPER
6	;OPTIONAL;UPPER;ENHANCE=I,U
7	;OPTIONAL;NUMERIC;ENHANCE=I,U
8	;OPTIONAL;NUMERIC
9	;OPTIONAL;UPPER;ENHANCE=U

and place the list of codes, in the same order as the variable names, on the COMT_FLD_MODNUM variable line.

14) Set LOWKEY to the name of the last NOW_INDEX field before DATADATE.

15) Set LOWLEN to the length of "lowkey" if "lowkey" is an alpha field; to -1 if it is a numeric field indexed increasing; to -2 if it is a numeric field indexed decreasing; and to -3 if it is a date field.

16) If uniqueness checking is to be done as part of modify/update/add/delete data validation, assign to the VERIFYFIELDS variable a list of the names of the fields before DATADATE which are required to uniquely identify a record, e.g. SCENARIO,CLASS,HULL,COMNUM are required to uniquely identify a given construction-type job.

17) Set VERTYP equal to "J", "U", or "JU" to enable join or uniqueness validation checks or both on data records to be added or updated.

18) Set NEXT_SCREEN and NEXT_HELP to the names of a logical next screen to work with ("" if there isn't one) and to the name of the help screen dedicated to this data screen; put the name of the dedicated comment screen in the COMT_SCREEN variable.

19) Put the name of the relation which holds comments related to the data screen in the COMT_FILE and COMT_GROUP variables. Set COMT_BREAK to the number of fields in the index on this comment relation BEFORE the LNUM field.

20) Fill in the field list describing this index in the bracketed portion of the COMT_RESET:= line (usually you can just put '%now_index' here).

21) If there are legals-type fields on the data screen, remove the brackets from the "CALL SCREEN LEGALS_INIT" line; otherwise delete the line.

22) Put in VARIABLE sections to detect changes by the user to keyfield values.

23) Give a name to the screen variable clearing utility screen at the bottom of the template. This name must be Kdata_screen_name.

24) Fill in all the local variable names on CLEAR VARIABLE lines in the clear utility.

8.4.4.1.2 Guide to Filling in the Comment Screen Template

The comment screen template file (stored in tcomt.templat) was shown in Figure 8-28. This template requires much less work to produce a finished screen than the data screen template does. Only the screen name and the key field data structure needs to be filled in.

The comment screen name filled in on the '*** SCREEN' line must be the same as that specified in the 'COMT_SCREEN:=' line filled in on the data screen template, and must be no more than 15 characters.

Key fields in addition to DATADATE and ENTRY_DATE should be filled in on the layout section above the comment text area. Be sure to add lines to the DECLARATIONS section declaring these variables as global as well, or the comment screen will not have access to the values shown on the data screen.

The remainder of the comment screen's logic is generic; it will work as long as all variables were set appropriately when the data screen was created.

8.4.4.1.3 Guide to Filling in the Help Screen Template

The contents of the comment screen template file were shown in Figure 8-30. This template is the simplest of all to fill in: place the screen's name on the '*** SCREEN' line and the top line of the layout, fill in the help text in the blank area of the layout, and set the name of the next help screen in the chain on the 'NEXT_HELP:=' line.

There are two rules. The first, which is very important, is that the help screen's name MUST be the same as the name assigned to NEXT_HELP in the data or menu screen which the help screen serves, and the name MUST have the string 'HELP' in it. The processing logic which implements the '?help_screen_name' and '=screen_name' commands must have a way of catching user errors (e.g. requesting a help screen with the = command). This is crucial because ? uses a CALL oriented logic while = uses SET. Very messy errors will result if a screen which expects to be CALLed is SET to, or vice versa. The error checking depends on all help screens having the string 'HELP' somewhere in their names, and no other screens having 'HELP' in their names.

The second rule is that NEXT_HELP should in general be set to 'COMMANDHELPL'. After an explanation of his context, a user is most likely to need a summary of available commands. COMMANDHELPL also is the start of the general help subsystem screen chain.

If the help being created requires multiple screens, this convention should be modified so that the screens form a chain of their own via appropriate NEXT_HELP settings, the last one then chaining into COMMANDHELPL.

8.4.4.1.4 Guide to Filling in the Menu Screen Template

The contents of the menu screen template file (tmenu.templat) were shown in Figure 8-32. To create a new menu from the template, first fill in the menu screen's name on the '*** SCREEN' line and the 'NOW_SCREEN:=' line. Place text after the option numbers to inform the user what each one represents, and remove any numbers which will be unused. For each option to be available, place an ACTION # section (e.g. '*** ACTION 5') at the end of the file. The section should perform a 'SET SCREEN' to the menu or data screen the option points to.

If there is a sensible default next screen to chain to after the new menu, put its name on the 'NEXT_SCREEN:=' line; otherwise set NEXT_SCREEN to "".

Finally, set NEXT_HELP to the name of the help screen for this menu. The remainder of the menu processing logic is generic.

8.4.4.2 Adding Subsystems to the DBU

Adding or changing DBU subsystems is a task much like that of modifying any large program. It is likely to be time consuming, and a thorough understanding of the existing logic and data structure is usually necessary.

If the subsystem takes the form of processors to implement a new command, it may only be necessary to make modifications to the command processing subroutine screens (see Section 8.4.3.8), and perhaps to create a new subroutine screen or two. Likewise, additional data validation logic is likely to require changes to

PROC_DATA_CMDB and perhaps to the existing validation utilities (see Section 8.4.3.11).

The code for DBU initialization screens and utility screens is maintained in file dbusubr.screens. If a new subsystem requires initialization activity at the start of DBU processing additions should be made to the INIT screen; if initialization each time the user returns to the DBU from the ALIAS command system is needed then additions should be made to the SUSPEND screen as well. New global variables should be declared in the START screen.

Particular care must be taken that new utilities do not accidentally modify existing global data structures; otherwise the operations of other parts of the DBU may be unintentionally disturbed.

8.4.4.3 Adding and Changing DBU FORTRAN Procedures

There are three reasons to use a FORTRAN subroutine to implement a DBU capability (instead of BUILDER code). First, certain operations can be performed faster by (compiled) FORTRAN than by (interpreted) BUILDER, e.g. computations in a loop. Second, it is much easier to implement some algorithms in FORTRAN, particularly those requiring complex logic or manipulation of arrays. Third, some things simply cannot be done from BUILDER, such as calling of System Intrinsics to manage extra data segments or for process handling.

The FORTRAN procedures (subroutines conforming to the BUILDER calling standard) which are part of some DBU utility subsystems have source code in slproc.src and sldate.src, and object code in sl.pub (the account Segmented Library).

These routines must reside in the SL (BUILDER dynamically links them into its executable image when a CALL PROCEDURE line is interpreted; thus they must reside in a run-time linkable

segment), and must have three formal parameters: a 50-word integer array which BUILDER will make the current partition available in, a 41-word integer array which offers pointers into the BUILDER's memory map, and a 10-word integer array which contains pointers to and descriptors of the text on the CALL PROCEDURE line itself (see the documentation in subroutine abstracts in the code in slproc for detail on the contents of these BUILDER-supplied arrays).

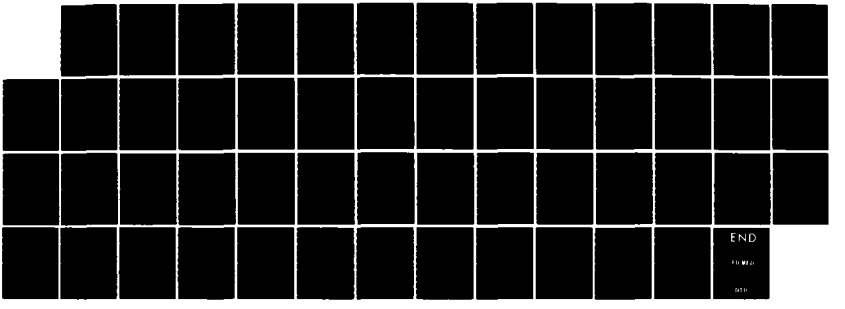
The information provided in these parameters allows a FORTRAN routine to perform operations on the current path using the 'raw' RELATE rdbxxxx interface routines, to retrieve information from the screen being interpreted via variables substituted onto the CALL PROCEDURE line after the name of the routine (e.g. 'CALL PROCEDURE GETSCENV %scen' makes the contents of the SCEN variable available to the routine), and allows data to be read from and written to BUILDER variables.

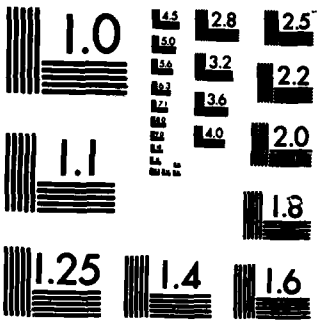
Making use of the information in the second and third formal parameters can be difficult because BUILDER, written in SPL, provides the locations of data in the form of word and byte addresses. In the absence of an SPL compiler, these addresses cannot be read directly. However, a feature of HP FORTRAN noted in the Appendices of the FORTRAN Reference Manual allows a programmer to instruct the compiler to compile a routine with call-by-value rather than call-by-reference addressing on some parameters. The programmer can then place the byte or word address BUILDER provides in the actual parameter corresponding to the one expecting by-value calling, and the formal variable or array will be mapped by FORTRAN onto the given address. In this way data can be read from or written to the BUILDER-specified locations.

Several utilities resident in SLPROC and SLDATE make this process fairly transparent to the ALIAS developer. In particular, getvar and putvar will retrieve and overwrite the

AD-A150 422 ALIAS (ACQUISITION AND LOGISTICS INFORMATION AND ANALYSIS SYSTEM) MAINTEN..(U) DECISION-SCIENCE APPLICATIONS INC ARLINGTON VA M S CAREY ET AL. 7/7

UNCLASSIFIED 31 OCT 84 DSA-593-VOL-1 N00014-82-C-0813 F/G 15/5 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

contents of a BUILDER variable whose name is provided as an argument. Dsafetch is a lower-level utility which can be used to retrieve text from the CALL PROCEDURE line.

The interested programmer should see Section 10.3 for a full listing and discussion of these utilities.

When changing or adding FORTRAN routines to the slproc and sldate libraries, the following steps should be followed EXACTLY:

1) Make the changes or additions to the source code. Make sure any new routines are assigned to segment 'dsa'. Note that routines which are to reside in an SL may not have COMMON, DATA, READ, WRITE, or FORMAT statements in their source code.

2) Compile both slproc and sldate, one after the other, using the syntax 'FORTRAN slxxxx.src,,$null' (*lp or any other file may be substituted for $null if the listing is desired).

3) Give the command 'GLUE addsl'.

Since only complete segments may be added to a Segmented Library, these steps are necessary to fully regenerate the object code for the 'dsa' segment (which contains all the DBU support routines), to delete the old version of the segment from the SL, and to add the new one (the GLUE udc, using the addsl.merge segmenter command file, takes care of the latter two steps).

The BUILDER features used to implement the DBU FORTRAN routines are largely undocumented and unsupported by CRI (the RELATE vendor), and so should be used sparingly and with care. Programmers will note that existing routines use a wide variety of methods to accomplish their purposes, where good programming practice would indicate standardization. The variety is the result of bugs in the CALL PROCEDURE interface (all now fixed) which had to be worked around initially.

8.4.5 Interfaces With Other System Units

The DBU interfaces with four ALIAS system units in addition to the data base and data dictionary. These are the command system, scenario system, security system, and output devices.

The command system interface is process-oriented rather than data-oriented. The first request for the DBU by the user leads to creation of a BUILDER process which begins interpreting the code in the dbu.screens file. Process creation occurs in the mrunp FORTRAN routine stored in the mrunit.src source file. When the user asks to exit the DBU (using the 'Q' command) the DBU's BUILDER process is suspended rather than terminated, via a call to the spsusp FORTRAN procedure (in slproc.src) from the SUSPEND utility screen. Subsequent requests for the DBU cause the process to be reactivated (rather than recreated), at considerble savings in initialization processing. Should the DBU process be terminated either by an abort or by exit from the DBU via the 'E' command, the mrunp routine will subsequently intercept the activation error and just create the DBU process again.

The DBU must interface with the scenario system in order to retrieve the name of the scenario the user is currently working with, and to retrieve scenario field key values for each relation the user works with in the DBU. The interface takes place in two ways: first, the DBU opens the snusers.sysrw relation, and retrieves from it the name of the user's current scenario at DBU initialization time and each subsequent time the user returns to the DBU from the command system; second, the DBU uses the getscenv FORTRAN routine (in slproc.src) to query the scenario system's extra data segment for scenario field key values. Getscenv is provided with a relation name and a target DBU variable name as arguments on its CALL PROCEDURE line; it searches the relation name list in the extra data segment for a match, and uses the match location to retrieve the key field value for the relation for the current scenario from the segment in the same manner as the scenario system's snrlsn FORTRAN utility function. The value is then placed into the BUILDER variable for use by DBU code in making queries and selections. The scenario system interface is thus a read-only one from the DBU's point of view, but it is crucial to DBU operations. If an attempt to run the DBU directly using the SCREEN command (rather

than running it from ALIAS) is made it is likely to fail because the required extra data segment and entry in snusers.sysrw are not present (a nice security enhancing feature).

The DBU interface to the overall ALIAS security system takes the form of queries to the sysusr.sysro, scrsec.db, and filsec.db user privelege recording relations. The user's priveleges are read from sysusr.sysro during initialization; the other two relations are consulted by the SECURITY_CHECK subroutine screen each time a new data or menu screen is requested to see if the user has access to it and to determine his data-modification priveleges.

Hard copy output device 'interfacing', i.e. assignment of hard copy to the proper device, is handled by the SET_DEVICE subroutine screen. Currently, all DBU hard copy output is generated by the REPORT utility screen, so this is the primary user of SET_DEVICE. Output is all via RELATE, so it is directed by issuing an MPE file equation equating the RDBLIST formal file name used by RELATE to the device number or name desired by the user.

8.4.6 Security Considerations

One of the primary purposes of the DBU is to promote ALIAS data base security and integrity by providing a controlled means of performing data base updates. The RELATE security scheme which forms a part of ALIAS security forbids ordinary users from making changes to data base relations using interactive RELATE (which they can run only when logged on under their 'R' user names), but does allow changes to be made programmatically (i.e. through the RELATE Host Language Interface) (when user is logged on under his 'A' name).

It is thus very important to add screens to the DBU to support new relations as they are created, since it will

otherwise be impossible to put security into effect for the relations.

Users can be kept from seeing or using any screen in the DBU by entries in the scrsec.db screen usage privelege recording relation, and can also be restricted to no or only a subset of Add, Modify, Update, and Delete capabilities by entries in the filsec.db relation. These entries are enforced only by the DBU, making it possible to give users broad fundamental capabilities using RELATE security (allowing them free use of ALIAS modules) while still restricting their activities in the DBU.

8.4.7 DBU Files and Screen Storage Conventions

The BUILDER screen source code that forms the heart of the DBU is currently stored in four files in the .screens group: dbusubr.screens (system initialization and general-purpose utilities), dbudata.screens (all menu, data, and comment screens and dedicated help screens), dbumhelp.screens (overall DBU help screens), and datdic.screens (data dictionary service screens). The executable, or compiled, version of this code is combined in the dbu.screens file.

The screen source code appears in no particular order in these files, except that it is crucial that screen START appear at the top of the dbusubr file and that this file name be the first one fed to the BUILDER when it is run in compile mode.

FORTRAN routines which are executed by the DBU using the BUILDER's CALL PROCEDURE capability have source code in slproc.src and sldate.src, and executable in sl.pub. No intermediate object code is maintained, since the routines are not processed by a segmenter merge/PREP step as are normal FORTRAN program routines.

The DBU also naturally makes use of nearly all data dictionary relations and all ALIAS data base relations.

9.0 ADDING NEW MODULES TO ALIAS

This section discusses expansion of ALIAS. It is meant as a summary, and as such concentrates more on procedures than on rationales. It presupposes a familiarity with ALIAS design goals and standards and with the architecture of the system as described in the rest of this manual.

Section 9.1 summarizes standards and procedures which should be followed in developing a new module; Section 9.2 presents step-by-step procedures for connecting new modules to the system.

9.1 DEVELOPING THE MODULE

In some respects ALIAS modules should be thought of more as stand-alone entities than as modifications to an existing system. Although the overall ALIAS architecture and standards place some restrictions on modules' form and processing (use of the integrated data base, provision of a consistent user interface, security, etc.), the modularity standard requires them to be essentially independent units. The flexibility of the System Core allows these units to be implemented in a variety of ways, thus allowing authors to use the most efficient and appropriate means available.

Module development should therefore go through a cycle similar to the standard system development cycle. A requirements study and functional description should be prepared, and a feasibility assessment made. A system design should be prepared. In most cases these phases need not be particularly formal or time consuming, since the next step will often involve fast implementation of a prototype to test the design concepts and to allow users to refine their statements of requirements. The prototype can then be modified or rewritten to conform to new specifications (and to operate more efficiently, if necessary) while the operational prototype takes care of immediate user needs.

The most important considerations with regard to integration of a new module into the ALIAS system are choice of language, process location, interfacing, input/output conventions, security, and documentation. These should all be taken into account in both the initial and final system design phases (note that a prototype and a final design may be very different in terms of language choice, process location, etc.). It is expected that the implementers of new modules will be aware of and comply with all ALIAS development standards and conventions.

9.1.1 Choice of Language

An ALIAS module may take six different forms, ranging from simplest to implement to most complex:

1) packaged program which is completely independent of ALIAS, such as the HP editor; it is a module only in that it is directly accessible from ALIAS

2) DBMS procedure file producing reports and/or graphics

3) BUILDER procedure file which causes BUILDER to present and manage data entry screens, or to perform other functions such as report generation/graphics production

4) simple higher-level language program consisting of a few subroutines not requiring a large data area

5) batch job which executes a stand-alone program or DBMS procedure file

6) large, separate high-level language program peforming special functions or analyses.

A module may use the RELATE DBMS query/report generation/graphics language, the superset of this language that is provided by the BUILDER, or a traditional high-level programming language. For simple to moderately complex functions, especially those that are query-oriented, direct use of the DBMS facilities via procedure files is recommended. These are relatively easy to design and debug, and are usually more efficient than other alternatives. The BUILDER language should be used for functions requiring extensive data entry by the user or other screen-

oriented processing. Because of its powerful dynamic substitution capability, BUILDER is often also the best choice for complex queries and for report generation.

Very complex functions, or those that are cpu-intensive, must be implemented using a traditional high-level language. Such programs will normally need to access the data base, and should do so using the record-level functions provided by the data base interface library routines.

FORTRAN should be the default high-level language of choice. The reasons for choosing FORTRAN are fully discussed in section 2.3.7. An alternative language should be chosen only if implementation of the given module is not feasible in FORTRAN.

9.1.2 Process Location

In most cases, choice of language implies the 'location' of the process that will execute the module. DBMS procedure files will be executed by the RELATE son process serving the ALIAS Core; FORTRAN programs may be linked into the Core program if their data stack requirements are not large (the Core already uses much of the 64K byte capacity); BUILDER procedures must be executed by a BUILDER son process; large FORTRAN programs must execute as son processes and start up their own RELATE sons, and may execute in the background (i.e. they will not tie up the screen) if desired; and batch jobs will execute as main processes in separate sessions.

The major choices are whether to run foreground or batch/background, whether to link programs into the core or not, and whether to run BUILDER procedures in the BUILDER service process or to start up a separate BUILDER process.

In general, modules which require NO user input and which do not write to the screen should run in the background, so that the user's terminal is not tied up. More interactive modules

must run in the foreground of the user's ALIAS session. Note that very severe aborts by any son process will cause termination of the ALIAS session (most aborts merely terminate the son process).

Background sons are preferred to batch jobs because the sons have easier access to the control and security data provided by the System Core; the batch jobs must start their own ALIAS session to access this data. As of this writing, the scenario system will not reliably allow a batch job to access a scenario concurrently with the user starting it up, and utilities for dynamically constructing appropriate batch job control files have not been developed. Modules which do not need access to current ALIAS security and control data may be implemented as batch jobs without difficulty at this time.

Linking FORTRAN modules into the System Core promotes efficiency (since no son process startup overhead is incurred), but the amount of data space now available in the core program is severely restricted. Also, errors in modules linked into the core, especially array-bounds violations, could destroy tables containing security and control information. In general, modules should be implemented as son processes. Note that son processes need not terminate when they are finished, but may simply deactivate themselves and wait for a reactivation signal from the Core. Great care to properly reinitialize module variables on reactivation must be taken if this capability is used.

A utility/template routine (mrunp in mrunit.src) is provided for startup of son processes: it contains one entry point for each module running as a son process. The creator of a new module which is to run as a son should add a new entry point for his module, using the instructions in the mrunp documentation prologue as a guide.

A second, similar utility/template routine controls the operations of the BUILDER service son process. This process is started up by the Core during initialization, and may be used for execution of small BUILDER procedures which execute in the foreground. Its function is intended to be similar to that of the RELATE service process: it allows execution of procedures without the overhead of starting up a son process each time one is to be used. Procedures using the service process must reside in a separate file, must have the same name inside the file as the file itself, must NOT use the cursor swapping system (see Section 8.4.3.2), should remove all partitions (cursors) and variables from the BUILDER stack when finished, should close all DB files when finished, and must terminate by a RETURN SCREEN command.

To use this facility to implement a module, place a new entry point into routine mrunb (also in mrunit.src); the code associated with the entry point mainly defines the name of the procedure file that is to be run. When executed, the routine places this name in a communication area and activates the BUILDER son, which retrieves the name and performs a CALL SCREEN using it.

Note that modules running their own BUILDER son process and using the cursor swapping system must use GREAT CARE in initializing this son process and the cursor swapping system. The BUILDER uses the rdbinitx facility to start its RELATE service process; the existence of multiple active BUILDER sons opens the possibility of collisions between the processes over use of the process id and the communications data segment. A fix of this problem is being worked on by CRI. The cursor swapping system was designed to be used by multiple BUILDER sons, each managing its own swaps separately. Each such son must set the SCREENSYS job control word to a unique value. This JCW is used by the swap system to initialize and retrieve a data segment in which the cursors are stored. A list of values already in use is maintained in the scrnref.doc documentation relation; this relation should be updated as more values are used.

9.1.3 Interfacing

ALIAS development standards allow modules to interface with the System Core and with the data base, but never with one another. This discussion is concerned with data transfer interfaces; process interfacing (e.g. RUN/TERMINATE) was discussed in the previous section.

9.1.3.1 Interfacing With the System Core

The Core contains three types of data of potential interest to a module: security information such as the current scenario name, user name, privelege level, etc.; basic system information of interest to FORTRAN programs such as I/O unit assignments; and parameter menu variable values. The information may be obtained in three ways, depending on the nature and process location of the module.

FORTRAN modules linked into the Core may directly access all security information and basic system information by including the proper common blocks in their subroutines: see Section 8.2.1 for a summary of Core common blocks and their contents. They may directly access current parameter menu values by including the parameter data structure files into their source code. Note that the code in these data structure files is written by the menu generator and thus changes each time the menu generator is run, necessitating recompilation of any routines using the data structure. By convention, source code for all such routines is to be maintained in file recomp.src, which is automatically recompiled after every menu generator run. It is suggested that a module designer have an initialization routine which transfers the desired variable values into variables local to his module; this convention makes it necessary to maintain only the code for that routine in recomp.src.

Modules which run as son processes started by the mrunp utility routine may load security and control information and the parameter data structure by calling the iniprc (in utlo.src)

utility routine as part of their initialization. Mrunp optionally writes this information into a communications segement, which iniprc then reads. Note that in order to use the information the module must include the same common blocks as would one linked into the Core.

Authors of modules written in a language other than FORTRAN may take advantage of these facilities by writing interface routines which have data structures with storage formats matching the FORTRAN common blocks in question. The existence of these 'duplicate' structures MUST be clearly documented in the FORTRAN include files so that maintenance personnel making changes to the common blocks will be aware of the need to change the 'duplicate' structures as well.

Modules implemented using DBMS procedures or the BUILDER may access parameter menu data values and some security information (FORTRAN system information is of no interest to such modules). The scenario in use is recorded in the snusers.sysrw relation, keyed by user name; the security priveleges of the current user are recorded in the sysusr.sysro relation, again keyed by user name; and parameter menu data values are stored in a series of relations in the .mnurel group, one relation per parameter menu, keyed by scenario name. Modules must open the files of interest and do pointed record retrievals or SELECTs in order to obtain the information.

The preprept FORTRAN utility, callable from BUILDER, may be used to substitute proper scenario key field values into EXECUTE file selects. See Section 10.3.

9.1.3.2 Interfacing With the Data Base

All calls to RELATE DBMS services by FORTRAN (or other high-level language) programs should be made using the high-level routines in the data base interface library (DBIF, in dbif.src). Generally, the names of these routines begin with 'r'; routines

in the library beginning with another letter, particularly 'd', are low-level routines meant to serve the needs of the high-level library routines. The interface insulates modules from the impact of converting to use of a DBMS other than RELATE, and also performs a number of housekeeping functions such as word-alignment of command strings. See section 10.2 for a full discussion of the DBIF.

Note that the interface does not itself enforce scenario security. Scenario field values are not necessarily the same as a scenario name throughout the data base, and the user may not have write priveleges for every data base file. When a file is opened using the interface, the proper scenario field key value and the write privelege setting for that file will be placed in the cursen and wrtprv arrays of the /scenar/ common block at the index location of the file's cursor number. Module authors are responsible for using this information when extracting and updating data base records to ensure that only legal updates occur, and that only data for the user's current scenario is retrieved. Particular attention should be paid to this function during module debugging.

9.1.4 Module Input and Output Conventions

Modules should draw all their input data from the data base, and should not make any changes to the data base. Exceptions to this principle are the concern of the rest of this action.

9.1.4.1 Sources of Module Input

ALIAS modules may draw inputs from four sources: from data provided by the System Core, from the data base, from other files, and directly from the user.

Data provided by the Core (see Section 9.1.3.1) may be used freely; it is more efficiently accessible to FORTRAN programs as provided by the Core than it is through direct retrieval from the

data base. Likewise, data in the data base may be drawn on freely, though with careful attention to the appropriateness of the DB fields used.

Use of non-data base files for user-modifiable module inputs is to be avoided, since the DBU typically cannot supervise modifications to such files, and it is cumbersome to make them scenario-dependent. The exception to this case is order-dependent textual input: RELATE does not provide good facilities for handling textual data, so it is necessary to have the user place such data in a standard HP editable file using EDITOR or TDP. As an example, the current system uses editable report format control files as a primary input to the Force Level Report Generator and the Battle Group Report Generator; these files contain report generator control keywords, the order of appearance of which controls the order of appearance of lines on the reports.

If editable files must be used, a separate HP file group should be created for them to facilitate their management.

Non-data base files not intended to be seen by the user, such as files holding module initialization data, may be used more freely. They must be thoroughly documented and should be placed in the .sysro group.

Obtaining inputs from the user by query should be avoided except for heavily interactive modules such as the assigner and the DBU. Use of queries to initialize report-generation-oriented modules should in particular be avoided; initialization settings should be obtained from parameter and list menus supervised by the Core menu system. Determination of the number, nature, and contents of a module's parameter and list menus should be made early in its design.

9.1.4.2 Module Output

Module outputs should go to the user's screen, to a printer or plotter, to an editable file for customization by the user, or into the data base.

The decision to place outputs in an editable file should be controlled by a parameter menu setting (as for the Force Impact Module), or if that is the usual output target, the user should always be informed of the name of the file the output has been placed in. Choice of screen or printer should be made according to the setting of the lpunit (Device to Print To) parameter of the envrn (User Environment Parameters) parameter menu.

Special care must be taken when designing and implementing a module which makes modifications to the data base. The data base is the central resource of the system, and must be reliable in structure and contents. The DBU ensures that users do not accidentally destroy data base integrity; module designers are responsible for ensuring that their modules do not damage the data base. No partial updates or deletions should be performed. Particular attention should be paid to the needs and expectations of existing modules. See the Data Base Reference Guide for more information on this subject.

9.1.4.3 Fitting a Module to the Data Base

In the same vein, module designers must be careful not to damage the data base by making inappropriate modifications to the data base structure. Many new modules will require some extension of the scope of the data base: new files for new subjects, and new fields for data elements not previously tracked.

A module designer should take the following steps as part of his design process:

1) Identify in detail the data elements which the module will require. This includes the data type and size of each element as well as the general nature of its contents. If the module is expected to change or grow

over time, make an attempt to determine its long-term data element requirements.

2) Identify the keys which each element depends on. These are the fields whose values would have to be given in a retrieval request to uniquely identify the elements. Some elements may have more than one possible set of keys, and some may function as keys themselves.

3) Search the data dictionary thoroughly for elements which match those required, and which reside in relations with proper key structures. Where close but not exact matches are found, attempt to use the existing element. Where this is not feasible and the meaning of the existing element is the same as that of the one desired but its format is unusable (e.g. not enough characters available in an alphabetic field), modify the existing field. Take care to modify the DBU and any modules using the modified element to reflect the changes.

   Where an element exists but not in a relation with appropriate keys, assess whether the keys in the exsisting relation could be derived from the keys desired using a cross-reference relation or rules. Assess the performance penalty which would be suffered. Assess whether the key structure of the existing relation could be modified without undue trouble or impact.

   Modifications to the existing key structure of the data base must be made with great care, since other modules may depend on that structure, and since the DBU is set up to maintain integrity according to the existing structure.

4) Group the elements which remain into relations according to the rules of normalization, with attention to performance issues. Take the key structures of the resulting relations and compare them to the structures of existing relations recorded in the data dictionary. Where matches are found, add the elements to the appropriate existing relations. Where there is similarity in relation purpose but not a perfect match on key structure, an assessment of the appropriateness and viability of modifying the structure of the existing relations must be made.

5) Implement as new relations any remaining elements.

When presented with a possible requirement to modify the data base structure, a good rule of thumb is that the modification should be made (regardless of cost/trouble) if it makes

the data base have a more precise and useful correspondence to reality. Otherwise the modification should not be made.

Note that all changes to the data base structure must be completely documented in the data dictionary, that appropriate alterations must be made to the DBU, and that data base security measures must be updated.

9.1.5 Security

The principal security issue requiring attention from module designers is that of scenario security, as discussed in Section 9.1.3.2 and Sections 7 and 8.3. In addition, authors should be aware that the DBMS will not allow creation of indexes on a data base relation by anyone other than the user creating the relation (i.e. the Data Base Administrator); thus, no module should expect to be able to create permanent indexes dynamically (temporary index creation via SELECT BY clauses is allowed).

Authors implementing modules not to be freely available to every ALIAS user are supported by a feature of the System Core which permits the System Supervisor to grant module run priveleges on a user-by-user basis. The feature is the existence of the modprv logical array in the /uzrprv/ common block; this array is initialized from the sysusr.sysro system privelege relation. A location in the array may be reserved for granting of run priveleges for a particular module. If the proper syntax is placed in the menu definition file, the Core will check this array before running a module.

9.1.6 Documentation

Maintenance of system documentation is not something to be ignored during module design; it is a vital activity which must go on from the beginning of design and implementation. All programs and procedures must be documented internally according to the standards in Section 2; all should be listed in the apropriate relations in the .doc group (see Section 6.3.5).

Particular care must be taken to update the relations which record usage of the global lprnts and uzrprv arrays, use of screensys job control word values, and use of fortran I/O unit numbers. Such updates should be made at the BEGINNING of implementation, not at the end, in order to 'reserve' values for the module to be developed. That way separate teams will not unwittingly use the same values.

9.2 HOOKING ONTO THE SYSTEM CORE

After a new module has been developed, it must be 'hooked on' to the System Core and tested. The process of hooking on primarily involves the generation of a new version of the system menus so that appropriate menus and menu options for the new module are displayed (it is assumed that required changes to the data base, the DBU, and system documentation, as discussed in the previous section, are made during the development process).

Generation of new menus mainly involves changing the contents of the files in the .mnufil and .mnurel groups. Relations which store parameter and list menu settings are stored in .mnurel, while other system files (such as those containing the text of the menus) are in .mnufil. Almost all of the necessary changes are made automatically by the menu generation program (mnug.prog). Exceptions are the update required to the system map in sysmap.sysro if any completely new menus are added, and an update to overall system help in syshlp.sysro should the changes be so major as to require revisions to that on-line help. Also, the contents of the sysusr.sysro relation should be modified to give users access priveleges to the new module if it is using the module security feature described in Sections 7 and 8.2.1.4.3.

A small amount of FORTRAN code modification is also usually required, along with some recompilation and linking.

The first step in regeneration of the system involves editing of the file which defines the menus the system displays. It is usually best to make a copy of this file and work with the copy until the menus it results in are satisfactory (take care that no other development team is doing the same thing at the same time!). The menu generation program mnug.prog then reads this file, and converts its information into the .mnufil and .mnurel files, as well as four files of FORTRAN source code. Mnug may only be run by a user logged onto the .makmenu group; the program will place all of its files in this group, so the

group must be empty of files before it is run. After the run, several procedures may be used to copy current parameter menu and list menu values from the existing .mnurel relations into those just generated.

At this point, a system creator has the choice of constructing a new development/debug version of ALIAS, which he can test and use without affecting normal ALIAS users, or of replacing the existing production system with the one just generated.

In either case, the next step is to recompile key source code files and link a new ALIAS core system. The recompilation is necessary for two reasons. First, mnug writes out the source code of a fortran subroutine named rnproc which is responsible for the first step in running modules. To run a module, the routine calls a user-written routine (often an entry point in a mrunit.src routine) named in the mnug input file. This user-written routine does the actual work of module startup. In order to run a new module, the new source code for rnproc must be compiled and linked with the rest of the core.

The second reason for recompilation is to implement the new parameter data structure produced by mnug (see section 3.1.10 for a full discussion of the parameter data structure). Three include files of source code (pvalue, pvdecl, and pveqiv) form this data structure. Subroutines which include this data structure must be recompiled and linked so that the values of any new parameters are properly displayed and stored. Such subroutines are by convention stored in recomp.src and mrunit.src.

The difference between generation of a new development or a new production system lies in the link procedure used, and in whether or not the new files/relations are moved from .makmenu into .mnufil and .mnurel.

9.2.1 Step-By-Step System Regeneration Procedure

1) Develop the new module.

Write the necessary programs or procedures, making sure to conform to the development standards listed in Section 2. Make any necessary changes to the data base and the DBU.

In writing programs/procedures, be aware of the final process location of the module and of the subroutines which will have to be provided in order to execute the module. At least one subroutine or entry point must be created for each module; the menu system will call this routine (which must be named in the menu generator input file) when the user requests execution of that module. In many cases utility routines are already written for this purpose, and require only slight modification to service additional modules. The following paragraphs describe the usual method used to prepare each type of module for running by ALIAS.

> DBMS PROCEDURE: Such procedures should be implemented as RELATE EXECUTE files in the .rprocs group. Utility routine mrunrp (in mrunit.src) is designed to function as a platform for the execution of such procedures. Procedure authors should add an entry point to this routine named after the module name as specified in the mnug input file. The routine ensures that output goes to the proper device and causes the Core RELATE service process to execute the procedure file.
>
> PACKAGED PROGRAM: Packaged programs require no data from the System Core, and thus need only a process creation and activation call. Utility routine mrunp (in mrunit.src) functions as a platform for these modules. Add an entry point named <module name> to this routine; make sure to specify the entry point type code properly so that unnecessary data swapping is avoided (see documentation in mrunp itself).
>
> BUILDER PROCEDURE: Small BUILDER procedure files may be executed by the BUILDER service son process (discussed in section 9.1.2); large ones should have their own process started up. In the latter case, an entry point should again be added to utility mrunp; take care that the entry point gives the proper file equations so that the desired procedure file is the one used.

Small BUILDER procedures may be handed to the service process by adding an entry to utility mrunb (again in mrunit.src). See the documentation in that routine for a discussion of the code to be supplied at the entry point.

PROGRAM LINKED INTO CORE: For modules which will be linked into the core, the name of the top or executive routine of the module should be the module name given in the mnug input file. No other steps are necessary.

BATCH JOB: ALIAS does not have any utilities at this time to support batch modules. Creators of such modules must write custom startup routines.

SEPARATE PROGRAM: Modules implemented as stand-alone programs may be serviced by an entry point in the mrunp utility. In contrast to packaged programs, these modules should have an entry point type code which causes mrunp to swap System Core data into the communications segment, and the first step the module program should take is a call to the iniprc utility.

Source and procedure code files should be placed in appropriate groups after development is complete (.src for source code, .screens for BUILDER procedures, .rprocs for RELATE EXECUTE files). Program files should be kept in the .prog group.

The dual nature of the ALIAS system allows two usable versions of each program or procedure file to be executable at any time. The main version, used by the production system, should be stored in the proper group under its regular name (e.g. ASGN.PROG for the assigner). A test version, executable by the development system, should have a name which tacks the character 'T' onto the beginning of the ordinary name (e.g. TASGN.PROG). When a user is running in DEVELOP mode ALIAS will automatically attempt to run the 'T' versions of modules; when it fails to find such a version, the ordinary one will be run.

New and modified modules should always be implemented first as 'T' versions so that testing may take place without disturbing other users.

2) **Modify MALIAS.DBA**

In order to generate a new set of displayable menus, with options for the new module included, the file describing the new menus to the menu generation program (mnug.prog) program must be modified. The version of the file which produced the current production system is by convention stored as MALIAS.DBA. The exact format of the menu additions, including number and format of new options to be added to existing menus, placement and contents of entirely new menus, and text of the on-line help for each new menu and menu option must be decided before work can proceed. Ideally, this should have been decided early in the module design, especially for modules requiring new menus and new parameters.

Warning: New options for existing menus should ALWAYS be added to the END of such menus. Otherwise, the numbers by which existing menu options are referenced will change, possibly changing the actions taken by existing command files. Likewise, particular care should be taken to check for affected command files if existing menu options are deleted.

2.1) **COPY MALIAS,workcopy**

Do not work directly with MALIAS itself, since accidental editing errors could ruin the specification for the existing system. Make all changes to the copy, give the copy to mnug as input, and then replace MALIAS with the copy once it is thoroughly tested.

2.2) **EDIT workcopy**

Appropriate changes should now be made to the working copy. To make such changes, it is necessary to understand what program mnug does and the syntax that it expects in the input file. Mnug produces a data structure composed of data relations and regular

files, which is read by the run-time system (program mnur) whenever ALIAS is run. The menus that mnur displays, and the actions taken in response to user commands, depend on the contents of the relation/file data structure.

The relations in the data structure are used to hold user-modifiable, scenario-dependent data. The regular files are used only by mnur for internal purposes, and should never be produced or modified with any program other than mnug.

In order to produce the data structure, mnug needs to know four things: first, the number of menus of each type (choice, parameter, list) which are desired and the exact text of the options to be displayed on each; second, a description of the data structures of interest to users and programmers which are to be produced, including the data type of each parameter and the maximum length in characters allowed for each list menu candidate name; third, the number of types of list menu which will be displayed and the name of each relation which is to be created to store lists for that type; fourth, the text of all on-line help to be made available by the menu system (except the system map and overall system help).

Mnug will produce a separate relation to store the values for each parameter menu, and will generate the necessary fortran source code for the parameter data structure into which the values will be loaded at run-time. However, it need not produce a separate relation for each list menu. If two list menus are to display the same list of candidates (such as a list of all shipyards known to the system), they may both use the same relation for data storage (each menu will have separate storage for on-off statuses). This is an advantage because ALIAS need only keep track of the one list of candidates and make sure it is always complete, rather than two. Thus, before adding new list menus to mnug, a determination of the appropriateness of existing list type relations as storage locations for the new list should be made.

Warning: if new list type relations are created, the DBU must be expanded and changed to ensure that the list the relation contains is always up to date.

Mnug will also write a FORTRAN subroutine. When a module is to be run, this routine makes a call to a routine with the module's name as specified in the input file.

Mnug expects its input file to be divided into three sections. The first section names the types of list menu which will be created later on (optionally up to nine menus per type). The type names will be used to name of the storage relations which will be created. The second section defines zero or more parameter menus; the third section one or more choice menus. The first menu in the choice menu section will be treated as the system master menu (top menu in the hierarchy). The order of appearance of menus of each type in the input file is otherwise unimportant. Options defined for each menu will be numbered in the run-time system according to their order of appearance in the input file.

The file syntax follows the following rules:

---Any line beginning with '%' will be ignored by mnugen. This allows documentation to be placed in-line in the file.

---The file is in three sections: list menu type relation definition, parameter menu definition, and choice menu definition. The sections must appear in that order, each terminated by an 'END' line. There may be only one of each section. Text beyond the 'END' on 'END' lines is ignored; it is suggested that comments be placed in this space indicating what is being ended. This convention aids future users and reduces the likelihood of nesting errors.

---The list menu type relation section must name all relations to be created to hold list menu candidate lists and statuses. One relation must be created for each type of list. If there are to be two list menus with lists of all ship classes known by the system, only one relation need be created; but one menu with a list of yards and one with a list of classes require two relations. The name of each relation must be followed with the maximum length a candidate name may have. One 'name,length' per line.

---The format of the parameter section is given in Table 9-1. Words shown in capitals in the table indicate the POSITIONING of key words which program mnug is expecting; words in lower case are explanatory. With the exception of the 'END' key word, the key words shown are not those which are actually required; rather, the words indicate the nature of those which should be supplied. For example, NAME must be a unique name for each menu; PARAMETERNAME must be a unique name for each parameter; LINE OF HELP may be any text. Note that the name given for each parameter will be used to specify its variable name in the parameter data structure. Thus, names should be chosen early in development so programmers will know the names of the variables they will work with.

The section may be broken down into repeating, non-repeating, and optional subsections. Table 9-2 shows the structure from this point of view. Required subsections are marked with a '|', repeating ones with a '>', conditional ones with a '?', and optional ones with a '['. Nesting is shown by indentation: a major section may be optional, but subsections within it may be required. Basically, there must be one full section for each parameter menu, with the name and title of the menu and menu-level on-line help appearing at the top of each such section. Subsections describing parameters follow within the section, one subsection per parameter, giving parameter name, data type, displayed text, and on-line help text. Optionally, if a parameter is of Type 6 (list menu gate), a list-menu definition subsection must be nested inside the parameter subsection. The end of definitions for a parameter menu must be marked with an end card, and the end of definitions for all parameter menus must be likewise marked.

TABLE 9-1

FORMAT OF PARAMETER MENU DEFINITION SECTION

```
% the following block must be repeated in broad outline once for
% each parameter menu; within the block for each menu, the option
% definition sub-blocks must be repeated once for each option
%
NAME of the menu, also used as name of data relation.  6 chars.
TITLE of the menu for display.  Max 72 chars.
LINE OF HELP
LINE OF HELP  as many lines of help describing the overall
LINE OF HELP  purpose of the menu as are desired, ended with an
END OF MENU LEVEL HELP
CODE,PARAMETERNAME,[LEN]  type code and name of a parameter
LEFT MENU TEXT (RIGHT MENU TEXT--VALUE OPTIONS) displayed info
LINE OF HELP                    help for this parameter
LINE OF HELP
END OF PARAMETER HELP
%     the code, parametername/text/help section must be repeated
%     once for each parameter to be displayed on the menu
%     the CODE is a numeric code indicating the data type of the
%     parameter.  The legal values are 1=real,2=integer,
%     3=character,4=True/False,5=date,6=list menu.
%     Note that if type 3 is specified, mnugen will expect the
%     length of the character variable to be given after the
%     variable name.  Note that no variable name may exceed 6
%     characters, and all must be unique.
%
%     If type 6 is specified, mnugen will expect a list menu
%     definition section to follow the END of parameter help
%     for its 'gate' parameter.  That section should look like:
LIST_TYPE_NAME,LIST_MENU_NAME
TITLE of list menu, for display
LINE OF HELP
LINE OF HELP      menu-level help for this list menu
LINE OF HELP
END OF LIST MENU HELP
%     where the list_type_name must match one of the names given
%     in the list type relation section of the file, and the
%     menu name must be a unique name.  Both should be no longer
%     than six characters.
%
%     Finally, the parameter menu should be ended with
END OF PARAMETERS FOR THIS MENU
%     and the section ended with
END OF PARAMETER MENU SPECIFICATIONS
```

TABLE 9-2

STRUCTURE OF PARAMETER MENU DEFINITION SECTION

```
|PARAMETER MENU DEFINITION SECTION (required)
|   [>    SECTION DEFINING A PARAMETER MENU (optional, repeatable)
|   [>    |  name of the parameter menu
|   [>    |  displayed title of the parameter menu
|   [>    [> menu-level help text
|   [>    |  END card for menu-level help text
|   [>    [> INDIVIDUAL PARAMETER DEFINITION SECTION (optional,
|   [>    [>                                          repeatable)
|   [>    [> |  data type code, parameter name, [max number chars]
|   [>    [> |  text to display left of = (text right of =)
|   [>    [> [> parameter-level help text
|   [>    [> |END card for parameter level help text
|   [>    [> ?  LIST MENU DEFINITION SECTION (reqd if data type=6)
|   [>    [> ?  |name of list type relation, name of menu
|   [>    [> ?  |title to be displayed on menu
|   [>    [> ?  [> list menu help text
|   [>    [> ?  |END card for list menu help text
|   [>    |  END card for the parameter menu section
|  END card terminating parameter menu definitions
```

---The format of the choice menu section is given in Table 9-3. This table is similar to that for the parameter menu section: words in capital letters indicate the kind of information expected at each position in the file: NAME is the name of the menu, TITLE its displayed title, etc. As many choice menu definition sections as are desired may be inserted, each containing specifications for up to fifteen menu options. Each option is defined by a line specifying the option type (CODE) and the menu text for the option, followed by lines of help text, followed by the name of the unit that the option refers to and its security index.

Choice options of type 1 cause another choice menu to be displayed when the user chooses the option; the name of this choice menu must be given as the option unit. Type 2 options cause a parameter menu named 'action_unit_name' to be displayed. Type 3 options cause a subroutine named 'action_unit_name' to be called; this subroutine (or entry point) should supervise startup of a module.

Figure 9-1 shows the contents of input file MALIAS.DBA as of 1 February 1984 as an example. Note the use of in-file comment lines (%) and END line annotations for documentation and clarity.

When the file has been updated, the next step is to run mnug.

3) HELLO DBA.SEA90,MAKMENU

Program mnug should never be run by any other user or from any other group. It creates system relations which DBA must supervise; creation of these relations by a user other than DBA restricts DBA's ability to create indexes and to perform REORGANIZations.

4) PPPKIL ...Clear the .makmenu group

The .makmenu group must be empty of files before mnug can be run. Since the group holds the menu system data structure for the current development system, a mnug run will destroy that system. If multiple teams are doing ALIAS development work

TABLE 9-3

FORMAT OF CHOICE MENU DEFINITION SECTION

```
NAME of choice menu
TITLE for menu to be displayed by mnurun
LINE OF HELP
LINE OF HELP        description of the overall purpose of the menu
LINE OF HELP
END of menu level help
CODE,CHOICE TEXT
LINE OF HELP
LINE OF HELP        description of what this option does
LINE OF HELP
END of option help
ACTION_UNIT_NAME, SECURITY_INDEX
%
%         where the CODE indicates the type of action which
%         action_name represents: code 1 is to display another
%         choice menu named action_name, 2 is to display a
%         parameter menu named action_name, and 3 is to call
%         a subroutine called action_name
%
%         If the code is 2, action_name must be the name of
%         one of the parameter menus defined in the preceding
%         section.  action_name should never exceed six chars.
%
%         The security index following action-name is optional.
%         If omitted, any user will be able to access the menu or
%         run the module.  If given, it should be an integer
%         number between 1 and 50 corresponding to a column in
%         sysus.sysro (e.g., 17 for M17).  The true/false status
%         in this column for a given user will be used to
%         determine if he has access to the menu or module.
%
%         The specifications for an individual menu should be
%         finished up with
END of options for this menu
%
%         and, when all choice menus have been defined, the file
%         should be terminated by
END of input file
```

FIGURE 9-1. MALIAS.DBA

```
VALYDS,12
VALCLS,12
VLRJOB,12
VLJTYP,12
END of list type relation specifications
%
%              *** BEGIN PARAMETER MENU SPECIFICATIONS
%
ENVRN
USER ENVIRONMENT PARAMETERS
      The environment parameter menu lets you "customize" your
ALIAS working environment.  For example, you may manually tell
ALIAS what type of terminal you are using, if it has made an
incorrect guess of terminal type.
END of menu level help for envirn
3,TTYTYP,8
TERMINAL TYPE (HP2623,SBRAIN,HZ15,HZ14,HP2647,PC)
      The setting of this parameter tells ALIAS what type of
terminal you are using.  It guesses your terminal type when you
first call it, but can make mistakes under certain circumstances.
If this parameter is not properly set, screen-control functions
may not work properly.
END of item help for ttytyp
3,LPUNIT,12
DEVICE TO PRINT TO (TERMINAL,DAISY,LP,PRINTER)
Many ALIAS modules produce printed output.  This parameter lets
you choose which printer this output will be sent to.  The usual
choice is SEA 90 DAISY, the local printer.  However, the PMS 392
LP line printer is much faster.  Note that printed output is
formatted for a 132 column device; if you have it sent to your
TERMINAL, it may not look very neat.
END of item help for LPUNIT
4,PSMNUH
PRINT COMMANDS ON MENUS? (YES,NO)
Allows you to turn off the tickler print at the top of each menu
of the most used ALIAS menu system commands.
END of item help for PSMNUH
END of parameters for envirn
% ************************************************************
ASNPRM
MANUAL ASSIGNER MODULE INITIALIZATION PARAMETERS
     This menu displays parameters which may be used to adjust
the ships and yards displayed during the manual ship assignment
process.
END of menu help for asnprm
3,PDURAT,6
TIME UNIT (FiscYr,CalYr,Qtr,Month,Week,Day)
     Each period will be of the duration set by this parameter.
Fiscal-year and Calendar-year are of course both twelve months
```

FIGURE 9-1. MALIAS.DBA

```
long; future plans call for their use to be differentiated but
they are identical at this time.  Qtr means quarters, the first
of which encompasses the months January through March.  In
conjunction with the Starting Date and Ending Date, this
parameter determines the number of periods into which the
assigner time window will be divided.
END of parameter menu item help for pdurat
5,DPFRST
STARTING DATE (MM/DD/YYYY)
     This is the first date of the first period to be displayed
by the assigner.  In conjunction with the Time Unit and Ending
Date parameters, this determines the number of periods and
defines the time window to be reviewed and updated by the
assigner.
END item help for dpfrst
5,DPLAST
ENDING DATE (MM/DD/YYYY)
     This is the last date of the last period to be displayed by
the assigner.  In conjunction with the Time Unit and Ending Date
parameters, this determines the number of periods and defines the
time window to be reviewed and updated by the assigner.
END item help for dplast
6,ASNYDS
CANDIDATE SHIP YARDS (ALL/LIST)
     This item allows you to specify those yards which it is
permissible to display and update during assigner processing.
END item help for ASNYDS list
VALYDS,CHSYDS
CHOOSE THE SET OF VALID YARDS TO WHICH SHIPS MAY BE ASSIGNED
     This list menu presents a list of all the shipyards known
by the system.  To assign ships to any yard, that yard must be
included (starred).  Yards not starred will not be displayed;
they cannot be added nor can their assignments be modified.
END of list menu help for chsyds
6,ASNCLS
CANDIDATE SHIP CLASSES (ALL/LIST)
     This item allows you to specify those ship classes which
may be reviewed and updated during assigner processing.
END item help for ASNCLS list
VALCLS,CHSCLS
CHOOSE THE SET OF VALID SHIP CLASSES WHICH MAY BE ASSIGNED
     This list menu presents a list of all the ship classes
known by the system.  To assign ship classes to any yard, that
class must be included (starred).  Ship classes which are not
starred are not eligible to be added since they are not
displayed.
END of list menu help for chscls
6,ASNJTP
```

FIGURE 9-1. MALIAS.DBA

```
CANDIDATE JOB TYPES (ALL/LIST)
      This item allows you to specify those job types which may
be reviewed and updated during assigner processing.
END item help for ASNJTP list
VLJTYP,CHJTYP
CHOOSE THE SET OF VALID JOBS WHICH MAY BE ASSIGNED
      This list menu presents a list of all the job types known
by the system.  To assign jobs of a given type to any yard, that
type must be included (starrred).  Jobs not starred will not be
displayed even if they were added in a previous session, nor can
they be added in a current session.
END of list menu help for chjtyp
3,DISBAS,8
DISPLAY BASIS (Approp,Awd,Start,Keel,Lnch,Deliv)
     When the manual assigner is initialized, it includes all
starred ships assigned to all starred yards already in the
database.  The date of assignment is ambiguous in that it could
refer to the ship award date, start date, or whatever.  This
parameter resolves that ambiguity so that the assignment date can
be translated into a specific period on the assigner display.
This parameter also resolves the same ambiguity when the date
conversion is performed in reverse; that is, when the assigner
periods are converted back into dates of assignment.
END of item help for disbas
3,ADJBAS,8
ADJUST BASIS (Approp,Awd,Start,Keel,Lnch,Deliv)
When multiple ships of a given class are assigned to a yard in a
given period, the actual assignment dates must be spread over the
entire period in some reasonable way; otherwise a yard would be
effectively forced by the data to begin constructing all these
ships simultaneously.  This parameter determines which of the
several possible milestone dates is used in spacing the
assignment dates over the period.
END of item help for adjbas
3,ADJMOD,12
ADJUST MODE (None,Program,Complx-Group)
When multiple ships of a given class are assigned to a yard in a
given period, the actual assignment dates must be spread over the
entire period in some reasonable way; otherwise a yard would be
effectively forced by the data to begin constructing all these
ships simultaneously.  This parameter determines how many ship
classes to consider as one group when they start in the same
period.  None means perform no spacing of ships over the period.
Program means each class should be spread over the periods to
which it is assigned separately from other classes.  Complexity-
group means that each general category of ship (for example, all
destroyers, or all carriers) are to be considered of the same
class for spacing over period purposes.  The Complexity-group
```

FIGURE 9-1. MALIAS.DBA

```
option is NOT IMPLEMENTED in the sense that it performs a
function identical to that implemented for the program option.
END of item help for adjmod
3,JEPOCH,10
JOBS EPOCH OPTION (ALL,CURR/PROJ,PROJ)
     The ship-jobs data base is divided into historical,
current, and projected jobs.  Those in the historical and current
sections may be displayed with the assigner module, but
modifications to those assignments have no effect on the data
base (since this obviously would imply an ability to "change
history").  With this parameter set to ALL, data from all three
sections will be displayed.  When set to CURR/PROJ, no data from
the historical section will be provided.  Finally, setting this
parameter to PROJ inhibits the display of all but the projected
section of the data base, H =I Z> #!e part for which modifi-
cations are permitted.  Limiting the number of sections displayed
can improve the efficiency of the assigner module both during
initial display generation (since parts of the database need not
be considered) and while modifying the database (since fewer
assignment modifications are discarded).
END of item help for USCURR
3,SRTCLS,11
SHIPCLASS SORT ORDER (Alphabetic,Input Order)
     This item determines how the shipclasses for each yard will
be ordered when they are shown on the display screen.  By input
order it is meant simply that the user is free to specify the
order in which they appear (via the Insert command).
END of item help for srtcls
3,SRTYRD,11
SHIPYARD SORT ORDER (Alphabetic,Input Order)
      This item determines how the yards will be ordered when
they are shown on the display screen.  The input order option
allows you to specify the position of new yards via the Insert
command.  Yards input in a previous session will always be
displayed alphabetically, however.
END of help for srtyrd
3,REFRSH,3
AUTO REFRESH (ON,OFF)
     This item determines whether the display screen in
refreshed at the conclusion of most Manual Assigner commands.
Usually it is wise to leave this parameter "ON", but if the
Assigner is being run from a terminal whose transfer (BAUD) rate
is slow, the experienced user may find himself waiting for output
to the screen to complete unless this parameter is set to "OFF".
END of item help for refrsh
END of parameters for this menu
```

FIGURE 9-1. MALIAS.DBA

```
% ************************************************************
FLREPT
FORCE LEVEL AND BATTLEGROUP REPORT GENERATOR PARAMETERS
      This menu displays parameters which can be set to control
the operation of the force level or the battlegroup report
generators.  You will specify exercise beginning and end dates,
whether you would like to keep an on-line copy of the report
generated, the exercise time period length, and other necessary
information.  A list menu holding repair jobs can be accessed to
set those jobs which will remove a ship from the force.
END of menu help for flrept
4,RPKEEP
KEEP REPORT ON-LINE (YES,NO)
      The setting of this parameter tells ALIAS whether you
would like to keep a copy of the generated force level or
battlegroup report, in addition to the copy sent to the printer,
on the computer. The file will be named FLREPT in your group and
you may edit it to make any changes, such as format changes, that
you might desire.
END of item help for flkeepf
5,RPBXER
REPORT START DATE (MM/DD/YYYY)
      The setting of this parameter tells the report generator
which date you would like this exercise to begin on.  The date
should be entered with the form MONTH/DAY/YEAR as in 12/29/1983.
Be sure that the exercise beginning date is prior to the end
date, or you will get an error message and your report will not
be generated.
END of item help for flbxer
5,RPFXER
REPORT END DATE (MM/DD/YYYY)
      The setting of this parameter tells the report generator
which date you would like this exercise to end with.  The date
should be entered with the form MONTH/DAY/YEAR as in 12/29/1983.
Be sure that the exercise beginning date is prior to the end
date, or you will get an error message and your report will not
be generated.
END of item help for flfxer
3,RPMRET,8
RETIRE SHIPS BY (LIFE,DATE)
      The setting of this parameter tells the report generator
whether you would like to have any ship which will retire during
this exercise, after today's date, retire on the date specified
as the ship's actual retirement date in the relate data base, or
whether you would rather use the ship's standard life as given in
the data base to compute the retirement date to be used.
END of item help for flmret
3,RPPLEN,8
```

FIGURE 9-1. MALIAS.DBA

```
TIME PERIOD LENGTH (DAY,WEEK,MONTH,QTR,CALYR)
      Set this parameter to specify which length of time period
you would like your report to be generated for. DAY will use the
exercise beginning as day 1. MONTH will go from the month of the
exercise beginning to that of the end. QTR will go from the
first of the quarter in which the beginning date falls to the end
of the quarter in which the end falls. YEAR will go from the
year of the exercise beginning to the year of its end. **NOTE**
Be sure that you do not specify dates and a period length that
will give more than 20 periods as that will cause an error. If
you do, you will be given the option of leaving the report
generator or just doing the first 20 periods.
END of item help for flplen
3,RPINFR,8
IN FORCE DAY (BEGIN,END)
      This parameter specifies which day of the period should be
examined to compute the force level for the period. For example,
suppose a ship's life is to begin on 6/20/1984, your exercise is
to go from 1/1/1984 to 1/1/1985 with period length=mnth. If in
force date=begin, the ship will not be counted as in the force
for the month of JUNE 84 because its life hadn't begun yet, but
if in force date=end, its life will have begun by the end of the
month of June, and thus would be in the force JUNE 84.
END of item help for flinfrc
3,RPPMLS,8
PROGRAM MILESTONE (APPROP,AWD,DELIV)
      This parameter tells which of the ship's milestones should
be used to compute which program a ship should be classified in.
For example, suppose this report will have two programs,
inventory from 1/1/1900 to 6/30/1983 and pom/epa from 7/1/1983 to
the end of the exercise. Suppose that a ship's milestones are
appropriation date:1/1/1978, award date:1/1/1979, and delivery
date:4/20/1985. Selecting a program milestone of either approp
or award will cause the ship to be put in the inventory program,
a program milestone of deliv will put it in the pom/epa program,
unless the exercise ends before 4/20/2985 in which case the ship
will not be in either program.
END of item help for flpmlst
6,RPRJOB
OUT OF FORCE REPAIR JOBS (All/List)
      Set this parameter to list to enter the list menu in which
you choose which repair jobs are major enough to remove a ship
from the force level for the duration of the job.
END of parameter menu item help for flrjob
```

FIGURE 9-1. MALIAS.DBA

```
VLRJOB,FLREPT
REPAIR JOBS THAT REMOVE A SHIP FROM FORCE DURING EXECUTION
      This list menu presents a list of repair jobs which may be
performed on a ship.  Star only those repair jobs which are major
enough to remove any ship from the force level for the duration
of the repair job.
END of list menu help for flrjob
END of parameters for this menu
% ************************************************************
END of parameter menu specifications
%
%            *** BEGIN CHOICE MENU SPECIFICATIONS
%
TOP
TOP LEVEL ALIAS COMMAND MENU
      Give the command S in the help subsystem to find out about
ALIAS and the basic organization of its command system.  You are
now positioned in the top, or highest level, ALIAS command menu.
Other menus are spread down a hierarchical tree which you may
move through.  All menus/functions may be reached by a series of
commands which you give starting from this menu.  Current
functions include user environment set-up, an ability to call
non-ALIAS processors from within ALIAS, and a data base
maintenance subsystem.
END of top level menu help
1,0,CUSTOMIZE USER ENVIRONMENT
Option 1 will put you in a choice menu from which you may modify
your working environment.  You may set terminal type by going
down farther to the environment parameter menu.
END of option help
ENVIRC
1,0,CALL NON-ALIAS PROCESSORS
Option 2 will put you in a choice menu from which other programs
may be called.  Editors may be accessed from within ALIAS in this
fashion.
END of option help
HPROGS
3,1,DATA BASE UPDATING SYSTEM
Option 3 will move you into the data base maintenance system.
END of option help
DMAINT
1,2,MANUAL ASSIGNMENT EDITOR
Option 4 places you in a choice menu from which you may set
parameters for and execute the manual assignment editor module,
commonly referred to as the manual assigner or simply as the
assigner.  This module allows you to modify the portion of the
database concerned with the actual award dates of ships to the
various shipyards.
END of option help
```

FIGURE 9-1. MALIAS.DBA

```
ASSIGN
1,3,FORCE LEVEL REPORT GENERATOR
Option 5 places you in a choice menu from which you may set
parameters for and execute the force level report generator
module.
END of option help
FLRPTG
1,0,SCENARIO CHOICE/MAKEUP SYSTEM
      This option will bring up a menu from which you will be
able to choose a different scenario to work with, create new
scenarios, delete scenarios or change their structure, and obtain
listings of scenario availability and composition to your
terminal and/or the printer of your choice.
END of option help
SCEN
END of menu specs
% ***************************************************************
HPROGS
HP PROGRAMS
      Non-ALIAS procedures may be called from this menu as an
alternative to leaving ALIAS in order to call them from the
operating system.  Once called from this menu, these procedures
will function exactly as they would if called from MPE, but you
will be returned to ALIAS when you are finished using them.
END of menu help for hpprogs
3,0,EDITOR
The EDITOR option calls the HP standard editor.
END of option help
RUNED
3,0,TDP
The TDP option calls the Text/Document Processor editor.
END of option help
RUNTDP
3,0,GRAPH
The GRAPH option puts you in Decision Support Graphics.
END of option help
RUNGPH
3,4,RELATE
The RELATE option will run the RELATE data base management system
for your use in interactive mode.  This option will function only
for the data base manager/system administrator when security is
in effect, though.
END of option help

RUNREL
3,5,MENU
The MENU option will run the user-friendly, screen-oriented
version of the RELATE DBMS.
END of option help
```

FIGURE 9-1. MALIAS.DBA

```
RUNSCF
3,0,SPOOK
     This option will run HP SPOOK, which is an operator-
oriented service program.  It can purge spooled files and kill
undesirable son processes.
END of option help
RSPOOK
END of specifications for this choice menu
% ****************************************************************
ENVIRC
ENVIRONMENT CONTROL
     This is the command menu for the user environment control
and customization area.  You may set debugging switches here or
move on to the environment parameter menu, where terminal type
may be set.
END of menu level help
2,0,ENVIRONMENT PARAMETERS
Option one move you to the environment parameter menu, where
terminal type may be set manually.
END of option help
ENVRN
3,6,SET LPRNTS (DEBUG SWITCHES)
Mentioning a number to the subroutine called by this flips the
status of the corresponding lprnt.
END of option help
SETLPR
END of menu specifications
% ****************************************************************
ASSIGN
MANUAL ASSIGNER SPECIFICATIONS
     This is the command menu for the manual assignment editor
module.  You may set various initialization parameters, and then
execute the assigner itself from this menu, which allows review
and alteration of the assignment dates for ships to shipyards.
END of menu level help
2,7,ASSIGNER INITIALIZATION PARAMETERS
Option 1 moves you to a menu from which you may alter initial-
ization parameters for the assigner, such as the time unit
associated with each period, milestone dates, etc.
END of option help
ASNPRM

3,8,EXECUTE THE ASSIGNER
Initializes the manual assigner using the parameters set by the
assigner initialization parameters menu.  A display will be shown
which indicates ship assignments by yard and period.  You may
then utilize various commands to review other parts of the
display, or add, delete, or modify these assignments.  At the
prompt, "(?=help) >", you may as it says type a question mark
(follow it with a carriage return as always) to enter a menu from
which you may request assistance on the capabilities/description
```

FIGURE 9-1. MALIAS.DBA

```
of the assigner.
END of option help
ASSIGN
END of menu specifications
% ************************************************************
FLRPTG
FORCE LEVEL GENERATOR
      This is the command menu for the force level report
generator module.  You may set various control parameters, and
then execute the force level report generator itself, from this
menu which along with a file which specifies what your report
will be made up of, will produce your report at the printer.
END of menu level help
2,9,FORCE LEVEL REPORT INITIALIZATION PARAMETERS
Option 1 moves you to a menu from which you may alter initial-
ization parameters for the force level report generator, as in
the exercise beginning and ending dates, period length, etc.
END of option help
FLREPT
3,10,EXECUTE FORCE REPORT GENERATOR
Runs the force level report generator.  Along with the parameters
set from the menu at option one, a file is needed to describe
things like your report titles, how many programs you wish to
examine, their start dates, the labels to be associated with
each, which ships will make up a ship type, which ships will be
repaired by a specific job, etc.  After the report has been
generated a copy will be sent to the printer you specifed in the
ALIAS menu system.
END of option help
FLREPT
3,11,PREPARE BATTLE GROUP REPORT
     Runs a report generator which determines the number of
battle groups deployable in each period, according to 'recipes'
and targets which you provide in a report control file.
END of option help
BTLGRP
END of menu specifications
% ************************************************************
SCEN
SCENARIO SYSTEM MENU
      Whenever you are in ALIAS, you are operating with a
'current    scenario', one which you chose when you started the
system.  The scenario's name is a key value which is used by the
various ALIAS modules when retrieving data from the data base for
their work.  Thus, which scenario you are working with controls
the sorts of reports you will achieve.  The existence of alter-
native scenarios lets you work on several studies simultaneously
over a period of days or weeks.
```

FIGURE 9-1. MALIAS.DBA

You may choose to work with a scenario other than the one you picked on entering ALIAS at any time by moving to this menu and giving the PICK A NEW SCENARIO TO WORK WITH option as a command. You may also create new scenarios, delete old ones (only those you have created), and examine the makeup of scenarios.
END of menu level help
3,0,CHOOSE A DIFFERENT SCENARIO TO WORK WITH
Typing '1' in response to the command prompt will place you in the scenario choice subsystem, where you will be able to change from using your current scenario to another scenario. You will also be able to list available scenarios and their makeup, and to create a new scenario if none of the ones available suit you.
END of option help
SNPICK
3,12,CREATE A NEW SCENARIO
Typing '2' in response to the command prompt will place you in the scenario creation subsystem, where you will be able to construct a new scenario. After it has been created, it will become your current scenario.

The ALIAS data base is divided into sections/families of related files/relations (shipyard descriptions and projected schedules constitute separate families). The scenario creation processor's main job is to find out what source you wish to use for the data for each family, and to implement your wishes.

New scenarios can be made up in several ways. A complete copy of an existing scenario can be made, so that the original can be maintained unchanged. A partial copy can be made, with the new scenario using the uncopied parts of the original in read-only fashion. Finally, the new scenario can be a blank slate in whole or in part. The partial copy is usually the best, since it lets you avoid the tedious data entry required to make a blank slate usable, but is not as time- and disk- consuming to construct as a complete copy. If it should turn out that you need to change an uncopied (read-only) portion of the scenario later, you can always make a changeable copy of that portion using the CHANGE SCENARIO MAKEUP option.
END of option help
SNMAKE
3,13,DELETE CURRENTLY EXISTING SCENARIOS
Typing '3' in response to the command prompt will place you in the scenario deletion subsystem, where you will be able to delete scenarios which you created from the data base. It is a good idea to delete any scenarios which you are no longer using, since this will free up disk storage resources. You will not be

FIGURE 9-1. MALIAS.DBA

```
able to delete any scenarios created by someone else, though,
even if you are able to look at and change the data of those
scenarios.
END of option help
SNDEL
3,0,LIST CURRENTLY EXISTING SCENARIOS
      Typing '4' in response to the command prompt will cause a
list of currently existing scenarios to appear on the terminal,
one page at a time.  Similar lists can be obtained inside the
scenario choice, creation, deletion, and modification subsystems.
END of option help
SNSHOW
3,0,SEND LIST OF EXISTING SCENARIOS TO LINE PRINTER
      Typing '5' in response to the command prompt will cause a
list of currently existing scenarios, in either summary or
detailed form, to be sent to the device you have specified in the
USER ENVIRONMENT PARAMETER MENU as the ALIAS system list device.
ENy { {Ation help
SNPRNT
3,14,MODIFY THE MAKEUP OF AN EXISTING SCENARIO
      Typing '6' in response to the command prompt will place you
in the scenario modification subsystem, where you will be able to
change certain aspects of any scenarios which you created.

      The modification facility lets you change data base
families in a scenario from 'read-only' to 'read/write' status,
lets you 'grab' data from another scenario and use it to replace
that in one or more of the DB families for the scenario you are
modifying, and lets you change the source data on a file-by-file
basis.  It is strongly recommended that you use extreme care in
using the file-by-file change faciltity, as the normal checks
made by ALIAS to ensure the integrity of your scenario's data are
bypassed there.

      Modifications to a scenario's structure are best made in
cooperation with the data base administrator.
END of option help
SNMDFY
END of menu options
% ****************************************
```

simultaneously, make sure that development system purging and regeneration is coordinated so that no one's progress is disturbed.

The .makmenu group may be cleared by udc PPPKIL. Note that the udc is available only to DBA.

5) **RRUN MNUG.PROG**

Run the menu generator; it will prompt for the name of the desired input file. Give it the name of the updated working version. Expect processing to take a few minutes. Make sure that account disk space is adequate for the creation of necessary files.

6) **RELATE**

Run the DBMS interactive command interface in preparation for step 7.

7) **EXECUTE CPRELS.DBA**

This procedure file will copy parameter and list menu values and candidates/statuses into the relations just created by mnug. When created, the relations contain only nominal data; before they can be used, even for testing, they must be loaded with current system data. **FAILURE TO PERFORM THIS STEP BEFORE SUBSEQUENT STEPS CAN LEAD TO LOSS OF CRUCIAL SYSTEM DATA.**

8) **Patch in new relation data**

If any new relations have been created, their contents must be 'bootstrapped' to conform to current on-line scenarios. Since no relation exists from which data can be copied, these relations still contain only nominal data. The bootstrap is best performed manually.

For parameter relations, open the relation (OPEN FILE), do a DELETE of all data now in it, and then open another parameter relation which already existed and which thus now holds data. Use a COPY statement to copy the names of all existing scenarios from the 'old' relation into the new one. Only the value of the scenario field need or should be copied, but it is crucial that parameter relations contain exactly one record for each current scenario.

For example, assume that a parameter menu named SREPT is being created for the first time on this run, and that the ENVRN parameter menu is one which has been in existence for some time. Assume further that SREPT contains one Type 6 (list menu) parameter named SLIST. The following procedure would be followed to 'bootstrap' the SREPT relation:

```
:RELATE
1) OPEN FILE SREPT.MAKMENU
2) CREATE INDEX BY SCENARIO;UNARY
3) OPEN FILE ENVRN.MAKMENU
4) COPY TO TEMP;FIELDS=,(TEMPORARY,A,24;F=U)
5) CLOSE FILE ENVRN
6) OPEN FILE TEMP
7) CREATE INDEX BY SCENARIO;UNARY
8) SELECT TEMP.,SOURCE=SREPT.SLIST
9) LET TEMPORARY=SOURCE
10) SET PATH TEMP
11) ERASE FILE SREPT
12) COPY SREPT.SCENARIO=SCENARIO,SREPT.SLIST=TEMPORARY
TO #
       SREPT.MAKMENU
13) PURGE FILE TEMP
14) SET PATH SREPT
15) PURGE INDEX SCENARIO
16) EXIT
```

See step 14 for a discussion of the necessity of creating a temporary file and transferring the value of the SLIST field into it.

For new list type relations, add at least one candidate name record for each scenario to each.

9) Exit RELATE and Recompile

The newly-written version of rnproc (placed in .makmenu by mnug) and any routines using the parameter data structure must now be recompiled. This is done with udc FTNM,; this udc uses a special include pre-processor which looks first in .makmenu for include files, then in .incl. Files which always must be recompiled with FTNM are rnproc.makmenu, recomp.src, and mrunit.src. A standard command sequence would thus be:

```
:FTNM RNPROC,MAKMENU
:FTNM RECOMP,SRC,,4026
:FTNM MRUNIT,SRC
```

10) Merge object code segments.

The object code version of the run-time program (mnur.obj) must now be rebuilt using the newly compiled routines. The command 'GLUE mnur' will run the segmenter with file mnur.merge as its driver; this will perform the task. In the event that a new module is being added whose routines are to be linked into mnur, the segmenter should be used manually to add the necessary segments to the new object code file. If, for example, object code for the new module is in file newmod.obj, and the object code is in segments 'newmod1' and 'newmod2', the following command series should be performed:

```
SEGMENTER
-USL MNUR.OBJ
-AUXUSL NEWMOD.OBJ
-COPY SEGMENT,NEWMOD1
```

```
-COPY SEGMENT,NEWMOD2
-EXIT
```

11) Link

The new version of ALIAS is now ready for creation. At this point, a new development (i.e. test) system should be created. The production system should be replaced with the test system only after changes are thoroughly tested.

To link, give the command 'LINK tmnur', which will cause a job performing a PREP mnur.obj,tmnur.prog to stream. Developers of new fortran modules should make up stream files to link their programs and place them in the .link group. Both test and production versions of the stream files should be developed.

12) Test

Test the new system. To enter test mode, give the command 'DEVELOP'. This sets a mode switch which will cause tmnur.prog to run instead of mnur.prog.

Make sure to test the <u>whole</u> system, not just additions. Unintentional errors while editing the mnug input file can cause errors to occur in previously working parts of the system; these must be checked for before a new production system is put in place.

CREATION OF A NEW PRODUCTION SYSTEM

13) Edit mnur.merge

If modules were added which consist of programs to be linked into the System Core, the segmenter input file which is used to build the object code version of the Core must be expanded. Edit this file and insert any necessary AUXUSL and COPY SEGMENT statements. Verify that the new version works by using the GLUE udc with it.

14) **Edit CPRELS, MVRELS, and PPPUDC.DBA**

The RELATE procedure files used to make parameter and list menu relation data/file transfers must be updated if any changes were made to parameter menus.

CPRELS.DBA copies data from the .mnurel relations into newly generated versions in .makmenu. It OPENs each pair of relations, ERASEs the .makmenu version, and then does a COPY of the .mnurel data into the .makmenu relations. For each new relation, a new section performing these action must be added to the file.

CPRELS also creates indexes on the new relations. Make sure that the proper indexes are created. Parameter relations must have an index on scenario, and list type relations on scenario,candidate.

The fields in parameter storage relations which correspond to list menu options are a special problem. They contain mnug-generated pointer information in addition to the ALL/LIST value; this pointer information must NOT be overwritten by old pointers from the relations in .mnurel. Relations which contain these fields require extra CPRELS procedure code to copy the .mnurel data into a temporary relation, then copy the new pointers over the old pointers in the temporary (the temporary is necessary to protect the production system). It is then safe to erase the .makmenu file and copy the 'old' data into it. See the text of CPRELS, or step 8 above, for examples of this.

MVRELS.DBA is a much simpler RELATE procedure file which purges the .mnurel files and copies (renames) the new .makmenu version into .mnurel. This procedure is used to convert a fully tested development system into a new production system. When new menus cause the creation of new relations, lines to accomplish the purging/renaming of these relations must be added to MVRELS.

PPPSAV moves all non-relational files from .makmenu into .mnufil, and PPPKIL will purge all files (relations and others) produced by mnug in the event that bugs are discovered and another mnug run must be made. Lines should be added to each of these udc's for any new relations.

15) Edit SYSMAP.SYSRO and SYSHLP.SYSRO

Review these two help files to make sure that they properly describe the new system. Make changes as appropriate.

16) Execute MVRELS

Enter RELATE, EXECUTE MVRELS to move the new system relations into .mnurel, and EXIT relate. Make sure no one is using the production system when this step is performed.

17) Move other files

Use udc PPPSAV to move all non-relation menu system files from .makmenu into .mnufil. These files will overwrite the existing versions. After execution of PPPSAV and MVRELS, .makmenu should once again be empty. Any files left in .makmenu indicate that step 14 was not performed correctly.

18) Replace System Core program

Purge the production version of the mnur program (stored in mnur.prog), and rename the development version (tmnur.prog) into its place.

19) Replace MALIAS.DBA

Purge the old version of the mnug input file and rename the working version used to regenerate the system into its place.

System regeneration is complete at this point. If all steps have been faithfully executed, future regenerations will automatically incorporate the updates done in this one. A final test should be done to make sure that the transfer of development to production systems has been successful.

9.2.2 Testing

In various places in the text above, the fact that ALIAS as a whole encompasses both a regular, or production system, and an isolated 'development' system was alluded to. This section will discuss in more detail how the two systems are separated and how to use one in preference to the other.

The dual architecture supports continuous expansion and development work. Without it, day-to-day users would continuously have their work disrupted by errors caused by software still in the debugging stage. Likewise, developers would be unable to perform certain key features such as system linking as long as any ordinary users were running the system; this would seriously hamper their productivity.

Additional features support testing of new or modified modules by allowing them to execute stand-alone. ALIAS takes a long time to start up; if the start-up step must be done prior to every test of a new module, debugging can take much longer than if the module can be executed stand-alone.

9.2.2.1 Dual-Version Architecture

9.2.2.1.1 Choosing Which Version To Use

The only overt action required to use the test version of ALIAS instead of the production version, or vice versa, is setting of a single mode switch. When set to 1, the job control word USEVERSION causes the running of the development system; when it is set to 0, the production system is run. Giving the (udc) command DEVELOP sets the value to 1, while the PRODUCTION command resets it to zero. To inspect the current value of the jcw, the mpe command SHOWJCW may be used. If USEVERSION does not appear on the current list of jcw's, ALIAS assumes its value to be zero and runs the production version.

9.2.2.1.2 How Dual Versions Are Implemented

Development personnel should have an understanding of how ALIAS is divided into development and production systems, so that care is taken to maintain the division for new modules (maintenance of the division is fairly automatic as long as the mrunit.src utilities are used for module start-up).

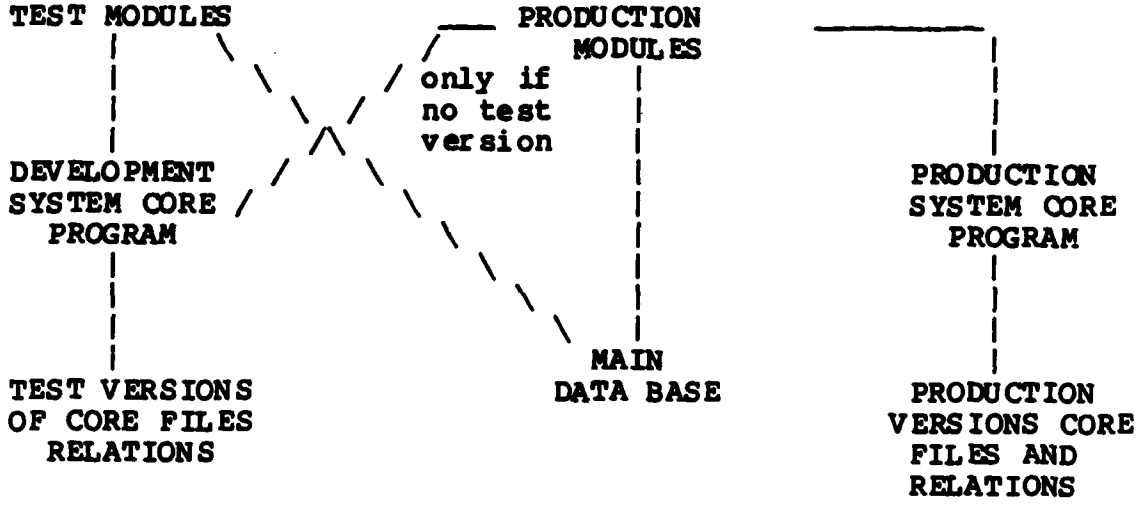
Figure 9-2 displays the architecture. Duality does not extend to the data base: both systems draw on the same relations. Separate data bases could in theory be implemented without great trouble (by modifying the DBIF utility system to open test-group relations instead of production group relations when running in develop mode), but there is not now sufficient storage on the HP to allow this. Currently, developers must be very careful to ensure that new/modified modules do not destroy user data.

It is suggested that developers make up a test scenario for use with new/modified modules, in order to maximize the isolation of their testing activity. Multiple teams doing concurrent work should each have their own test scenarios.

Dual versions of the System Core and its data base can be maintained, however. Menu system relations and files reside in .mnurel and .mnufil for the production system, and in .makmenu for the development version. Mnur routine inimnu checks the USEVERSION mode setting on start-up, and then opens files in the appropriate group(s). Inimnu also sets the rmmode variable in the /uzrprv/ common block so that other system routines will not need to query the jcw. Thus, a user cannot change from production to development mode, or vice versa, in the middle of an ALIAS session.

Dual versions of modules are also maintainable. Conventionally, test versions of modules have names which concatenate a 'T' with the regular module name, and test and development versions reside in the same group (for example, the two versions

Figure 9-2. Dual-Version ALIAS Architecture



note: 'modules' include DBU here

of the force level report generator would be FLRP.PROG and TFLRP.PROG). The module startup utility routines in mrunit.src automatically consult the setting of the rnmode variable and concatenate a 'T' onto the module file name supplied by the author of an entry point as appropriate. Thus, implementers who use these utilities need give no special attention to the dual architecture of the system.

If no 'T' version of a module can be found when ALIAS is running in DEVELOP mode, the production version will automatically be used. Thus, test versions need only be made up for the modules to be tested, and the rest of the services of the system remain available to the tester.

9.2.2.2 Stand-Alone Testing of Modules

As mentioned above, testing new modules separately from ALIAS can save considerable time in many cases. ALIAS provides support for this activity, the nature of the support depending on the nature of the module.

DBMS procedure files may be tested from the interactive, command-oriented interface to RELATE (run with the RELATE udc) by simply giving the EXECUTE <modulename> command. Note that output from report-generating procedures destined for the printer (i.e. those using the :P option on the REPORT command) will send output to the device specified as RDBLIST in the RELATE udc (SEA 90 daisy wheel at present). When run from ALIAS, the RDBLIST device setting will automatically be set to match the 'DEVICE TO PRINT TO' parameter on the user environment parameters menu.

BUILDER procedure files may be tested by supplying the procedure file name as the argument to the SCREEN udc. Note that a temporary patch must be placed in such procedures to supply the current user name, scenario name, and date if these are required and if the procedure is expecting these values to be supplied automatically by the BUILDER son service son process.

Modules which consist of subroutines to be linked into the System Core may be tested separately only if the developer writes a test driver to supply data that would otherwise be supplied by the Core common blocks.

Modules which are implemented as separate programs to be run as sons are offered limited stand-alone testing support. First, the iniprc routine will prompt the user for changes to the settings of the array in the /lprnts/ common block if the LPRNTON jcw is set to 1. The DEBUG and NORMAL udcs set this jcw to 1 and 0, respectively. This allows the switches which control the printing of debugging information to be set at module startup, without any explicit coding by the developer being necessary. This feature can be duplicated by setting lprnts in the User Environment menu of ALIAS, since /lprnts/ is part of the data transferred to iniprc, but this of course involves running ALIAS to do the sets.

Should module authors wish to insert code to prompt for lprnt settings at additional points in their module's execution, the (fortran) statement:

```
IF (debug) CALL setlpr
```

will have the desired effect. The debug logical function queries the LPRNTON job control word, while setlpr prompts for lprnt settings. Both reside in UTLR.

END

FILMED

4-85

DTIC