

AD-A042 126

TRW DEFENSE AND SPACE SYSTEMS GROUP REDONDO BEACH CALIF
THE SOFTWARE DESIGN AND VERIFICATION SYSTEM (SDVS). (U)
MAY 77 M E HOLLOWICH, M G MCCLIMENS

F/G 9/2

UNCLASSIFIED

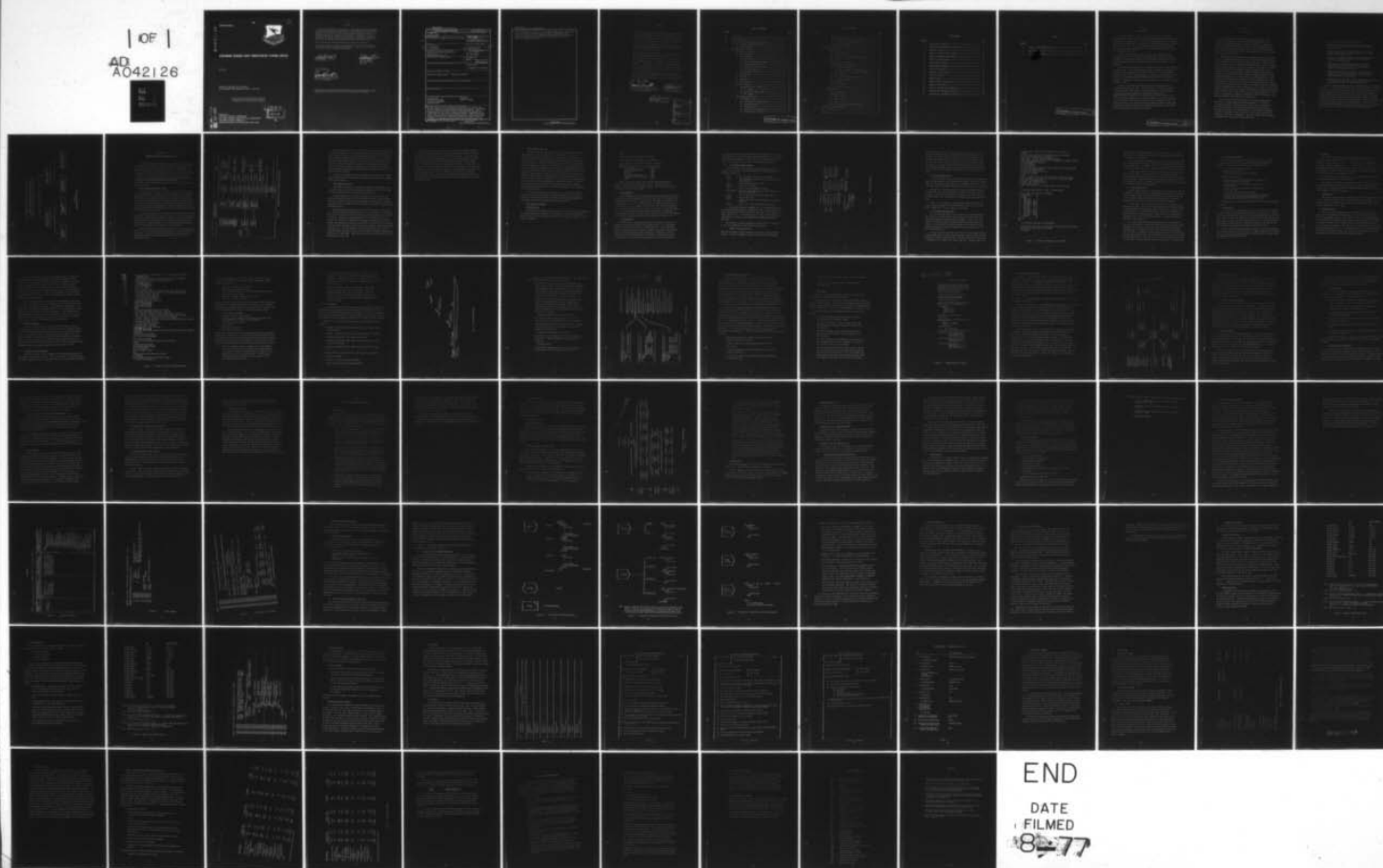
AFAL-TR-76-200

F33615-74-C-1159

NL

| OF |

AD
A042126



END

DATE
FILMED
8-77

AD A 042126

AFAL-TR-76-200



SOFTWARE DESIGN AND VERIFICATION SYSTEM (SDVS)

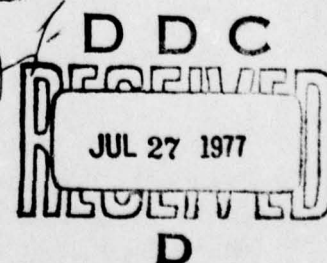
MAY 1977

TECHNICAL REPORT AFAL-TR-76-200
FINAL REPORT FOR PERIOD JUNE 1974 - JUNE 1976

Approved for public release; distribution unlimited

Prepared For
AIR FORCE AVIONICS LABORATORY
AIR FORCE WRIGHT AERONAUTICAL LABORATORIES
AIR FORCE SYSTEMS COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433

DDC FILE COPY

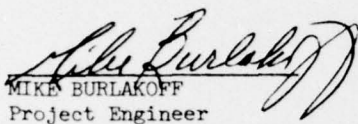


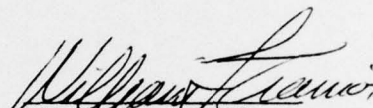
NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely related Government procurement operation, the United States Government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

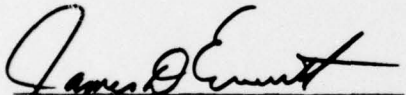
This technical report has been reviewed and is approved for publication.

This report has been reviewed by the Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.


MIKE BURLAKOFF
Project Engineer


WILLIAM L. TRAINOR
Technical Manager
DAIS Software Group
DAIS ADPO

FOR THE COMMANDER


JAMES D. EVERETT, Colonel, USAF
Chief, System Avionics Division
AF Avionics Laboratory

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

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DD FORM 1473
1 JAN 73

EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

409 637

Ince

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

various SDVS functions available are presented in addition to a description of the development methodology used for the design, development, and validation of the SDVS. The methodology included application of top-down design, development, and test techniques, in addition to the use of structured programming concepts.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

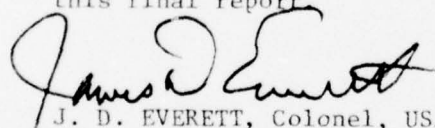
PREFACE

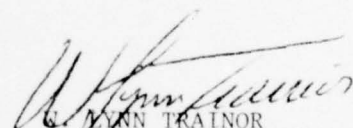
This report was prepared by TRW Defense and Space Systems Group, Redondo Beach, California, 90278 under contract F33615-74-C-1159 with the Air Force Avionics Laboratory, Digital Avionics Information System (DAIS) Software Group.

This final report summarizes the Software Design and Verification System (SDVS) which was developed between June 1974 and June 1976 by TRW.

Acknowledgement and appreciation is extended to the TRW Defense and Space Systems Group in Dayton, Ohio who participated in the SDVS development. In particular, M. E. Hollowich and M. G. McClimens were the principal authors of this report. H. M. Hart, J. Boekhout and R. M. Hart also contributed to the document. Gay Moffitt and Karen Lacy assisted in the production of the report.

This Technical report replaces, in full, AFAL-TR-75-31, same title. TR-75-31 was the interim technical report and does not fully reflect the SDVS system as documented in this final report.


J. D. EVERETT, Colonel, USAF
Chief, System Avionics Division
AF Avionics Laboratory


G. M. TRAINOR
DAIS Software Group
Technical Manager


NIKE BURLAKOFF
SDVS Project Engineer

ACCESSION for		
NTIS	Write Section	☒
DDC	Buff Section	☐
UNANNOUNCED		☐
JUSTIFICATION		
BY		
DISTRIBUTION/AVAILABILITY CODES		
Dist.	AVAIL. and	SPECIAL
A		

TABLE OF CONTENTS

<u>SECTION</u>	<u>PAGE</u>
I. Introduction -----	1
II. DAIS Objectives -----	2
III. Overview of the DAIS Integrated Test Bed -----	5
1. DAIS Information Management System -----	5
2. DAIS Support Facility -----	7
IV. Description of the SDVS -----	9
1. SDVS Modes of Operation -----	9
a. Mode Selection -----	9
b. File Generation -----	10
(1) File Structure -----	10
(2) File Conversational Commands -----	11
(3) Mission Software Files -----	13
(4) Simulation Test Case Files -----	13
(5) Post Run Edit Files -----	15
c. Set Up and Run Simulation -----	16
d. Post Run Edit -----	16
e. Rollback -----	17
f. Delete Mode -----	17
g. Supervisor Mode -----	17
2. SDVS User Languages -----	18
a. Simulation Control Language (SCL) -----	18
(1) SCL Example -----	21
b. Data Processing Language (DPL) -----	25
(1) DPL Example -----	26
3. SDVS Simulation Facilities -----	28
a. SDVS Support Facility -----	30
(1) Code Executor Linker/Loaders -----	30
(2) Snapshot/Rollback -----	30
(3) Data Recording -----	31
(4) Subsystems Data Formatting -----	31
(5) Simulated Real-Time Clock -----	32

PRECEDING PAGE BLANK NOT FILMED

TABLE OF CONTENTS

<u>SECTION</u>	<u>PAGE</u>
b. SDVS Simulators -----	32
(1) Interpretive Computer Simulator (ICS) -----	33
(2) Statement Level Simulator (SCS) -----	33
(3) Data Bus Simulation -----	34
V. Design and Implementation Techniques -----	35
1. Top-down Approach -----	35
a. Hierarchial Design -----	37
(1) Program Hierarchy -----	37
(2) Evaluation of Using Hierarchial Design -----	40
b. Rehostibility -----	41
c. Control and Data Interface -----	44
d. Software Development Standards -----	49
(1) Structured Programming -----	49
(a) SDVS Structured Programming Constructs -----	49
(b) Evaluation of Structured Programming -----	50
(2) Program Modularity -----	51
(3) Use of a High Level Language -----	56
(4) Programming Standards -----	58
e. Software Testing -----	62
(1) Test Planning -----	62
(2) Configuration Control -----	69
2. Program Plan -----	70
a. Overall Philosophy -----	70
b. Development Concept -----	70
(1) Phase I SDVS -----	70
(2) Phase II SDVS -----	72
(3) Phase III SDVS -----	73
3. Productivity Using SDVS Development Techniques -----	74
VI. Conclusions and Recommendations -----	77
VII. List of Acronyms -----	80
References -----	81

ILLUSTRATIONS

<u>FIGURE</u>		<u>PAGE</u>
1	DAIS Software Organization -----	4
2	Simplified Block Diagram of ITB Facility -----	6
3	SDVS File Structure -----	12
4	Example of Creating a File in SDVS -----	14
5	Example of Supervisor Mode Operation -----	19
6	Sample SDVS Flight Profile -----	22
7	Sample SDVS SCL Program -----	24
8	Sample SDVS DPL Program -----	27
9	The SDVS View of DAIS -----	29
10	Hierarchy of SDVS Software -----	38
11	Access CP Interface -----	46
12	SCPEES Compool -----	47
13	SDVS/J73 Constrol Structures -----	51
14	Example of SDVS Module -----	61
15	Phase III SDVS Testing Configurations -----	64
16	SDVS Phase III Configuration Testing -----	65
17	Configuration 4 Standalone SLS and SCP -----	68

TABLES

<u>NUMBER</u>		<u>PAGE</u>
1	Example of SDVS DEFINE Strings -----	59
2	SDVS Capabilities by Phase -----	71
3	Size of SDVS Programs -----	75

PRECEDING PAGE BLANK-NOT FILMED

SECTION I

INTRODUCTION

The purpose of this report is to give an overview of the Software Design and Verification System (SDVS) and present the design and implementation techniques used to develop SDVS. The SDVS has been developed as an integrated collection of software tools in support of the AFAL Digital Avionics Information System (DAIS) program. DAIS is an information management system approach to avionics processing applications. The overall objectives of DAIS and SDVS are presented in section two.

Section three describes the DAIS facility being developed at AFAL. Included is a discussion of the information management system core elements and the overall system architecture. The DAIS support facility which will be used to evaluate the real-time operation of the DAIS core elements is also described.

The DAIS mission software will be developed using the SDVS tools discussed in section four. Each of the seven SDVS modes of operation associated with the configuration management, simulation and testing of mission software is presented. The SDVS high level languages for defining simulated DAIS mission scenarios and processing simulation data are described with illustrative examples. Section four concludes with a description of the simulation functions available in SDVS.

Section five presents the techniques and tools employed by TRW in designing and building the SDVS. The basic design philosophy was based on a top-down design, development, and testing approach. Topics concerning structured programming techniques, programming standards, test planning and configuration control, rehostability of SDVS, definition of control and data interfaces, etc. are also presented. Section five continues with a discussion of the overall program plan, the development of SDVS in a three phase effort, and concludes with some quantitative data concerning the productivity achieved using the various development techniques. Conclusions and recommendations are outlined in section 6.

PRECEDING PAGE BLANK NOT FILMED

SECTION II

DAIS OBJECTIVES

The purpose of the Digital Avionics Information System (DAIS) project is to demonstrate a coherent solution to the problem of proliferation and nonstandardization of aircraft avionics, to develop and test in a "hot bench" configuration (known as the Integrated Test Bed - ITB) the DAIS concept, and to permit the Air Force to assume the initiative in the specification of avionics configurations for future Air Force weapon systems acquisitions.

Historically mission information requirements have been established along semi-autonomous subsystem areas such as navigation, weapon delivery, stores management, flight controls, communications, etc. Within each of these functional areas the trend has been toward digital systems each with its own unique processing, transfer and display of information. There has been an integration of requirements between functional areas only as necessary for interaction purposes. The DAIS concept proposes that the processing, multiplex and display functions be common and service all the previously described areas or subfunctions on an integrated basis. When coupled with other existing programs and facilities, the DAIS "hot bench" will contain the flexibility to evaluate a spectrum of multiplex, processing and display approaches such that decisions regarding interface standards, processing architectures, display concepts, etc., can be made. As technology becomes available and the "hot bench" is programmed to solve desired aircraft avionic problems, the built-in flexibility will accept adaptation. In this manner, an evolutionary growth will continually update the "hot bench" configuration whenever the capability or need exists.

The DAIS design approach reflects a total system concept which is functionally oriented rather than hardware oriented. For example, a "navigation subsystem" in DAIS does not refer to a set of black boxes which are identifiable functions which are performed various places throughout the system. Not that the system is not dedicated exclusively to doing the navigation function alone; it is also used to perform the functions of many other "subsystems". DAIS will certify ideas and classes of equipment that can satisfy weapon system needs.

Specific objectives of the DAIS program include:

- a. Develop an AFSC in-house capability to define, demonstrate, test, and evaluate evolutionary changes and requirements in digital avionics.
- b. Define and design a "hot bench" configuration for a limited hardware demonstration with growth potential to accommodate a large class of weapon systems.
- c. Identify and recommend standards, criteria, and specifications which must be instituted to reduce the proliferation and complexity of avionics systems.
- d. Provide means for quantitatively evaluating cost (both acquisition and life cycle) aspects and for exploiting potential increases in reliability, maintainability, and versatility of future weapon systems.
- e. Influence the design and development of sensors via input-output format specifications which will allow the new sensors to be compatible with the DAIS concept and ensure optimal information transfer and management.
- f. Identification of many diverse programs, offices, etc., involved in digital avionics with the resulting integration of their requirements and actions into one coherent program.

The DAIS software hierarchy necessary to support these goals is depicted in Figure 1. The mission software represents the Operational Flight Program (OFP) for a particular DAIS simulated mission. The support software includes all the necessary non real-time software tools to aid in the development, testing, verification, and maintenance of the mission software. These tools include configuration management aids, functional and bit level simulations of the DAIS "hot bench", and user interface languages. They are known collectively as the Software Design and Verification System (SDVS). In addition to the SDVS, the DAIS support software includes the Integrated Test Bed (ITB) Support Software for real-time control and monitoring of the DAIS "hot bench".

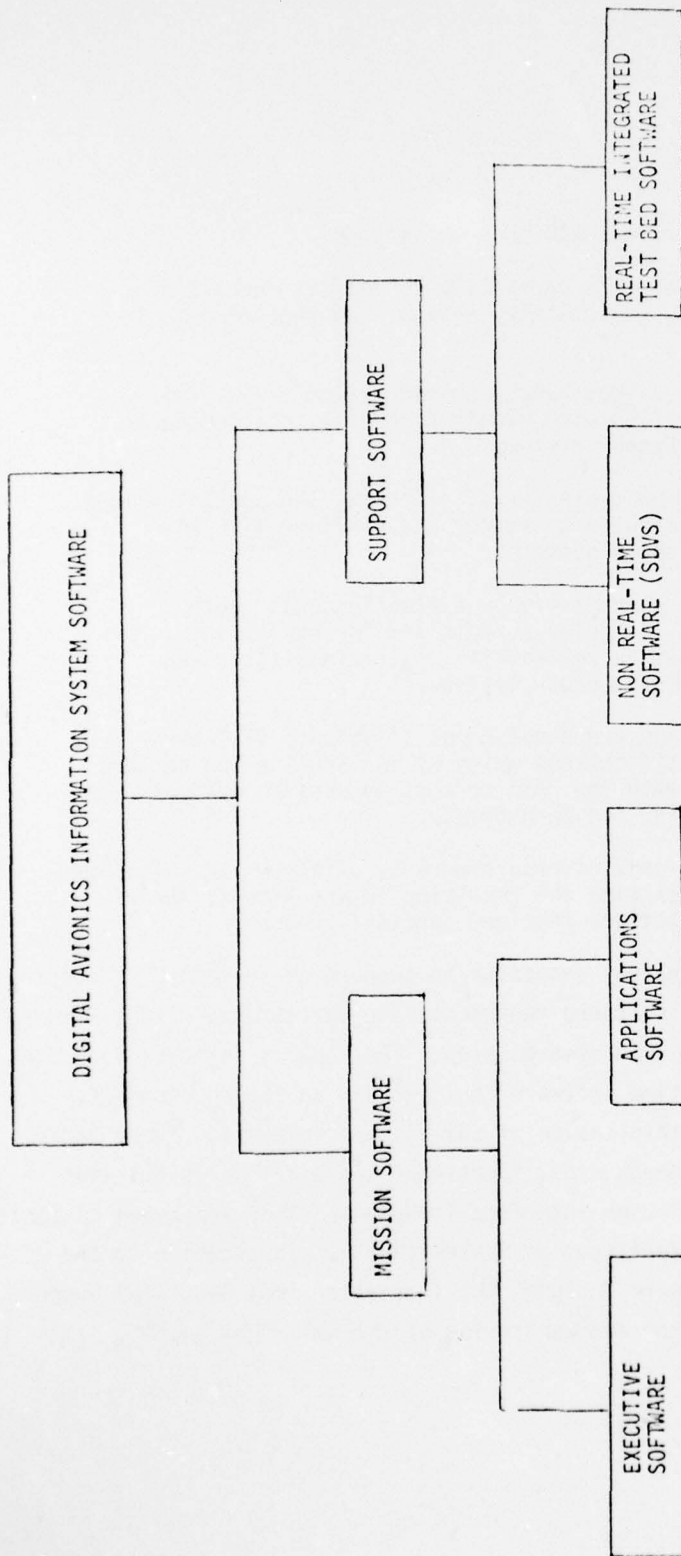


Figure 1
DAIS Software Organization

SECTION III

OVERVIEW OF THE DAIS INTEGRATED TEST BED

The DAIS "hot bench", or Integrated Test Bed (ITB) is an information management system consisting of a set of federated processors interfaced to each other, to avionic sensors, and to control and displays by a MIL-STD-1553A multiplex data bus; and a support facility to perform the information management system monitor and control functions. Figure 3-1 is a simplified block diagram of the DAIS Integrated Test Bed. Control of information management system functions is performed by the DAIS mission software which is partitioned among the DAIS processors. The following paragraphs highlight the functions of the various Integrated Test Bed components.

1. DAIS Information Management System

The DAIS core elements shown in Figure 2 are based on a federated processor architecture. Each DAIS processor is connected to a Bus Control Interface Unit (BCIU) which initiates data transmission over a redundant multiplex bus system between the processors and remote terminals (RT). The latter being the interface between the data bus and the simulated avionic equipment. Each BCIU is actually an intelligent I/O channel which executes I/O commands stored in the DAIS processor's memory. Centralized single point data bus protocol is performed by a processor resident software executive and a selected master BCIU.

The remote terminals provide an interface between the bus and aircraft equipments. Conceptionally, it functions similar to a BCIU by transferring data to or from the equipment to which it interfaces. The RT contains interface modules which can be interchanged to provide the correct electrical interface for different equipment. It can also be programmed to define the mapping of data between the bus and the aircraft equipments.

The mission software is distributed among the set of processors in the system. It consists of application software, which performs the processing required for a specific aircraft/mission application, and the executive software, which performs system control and provides services to the application software.

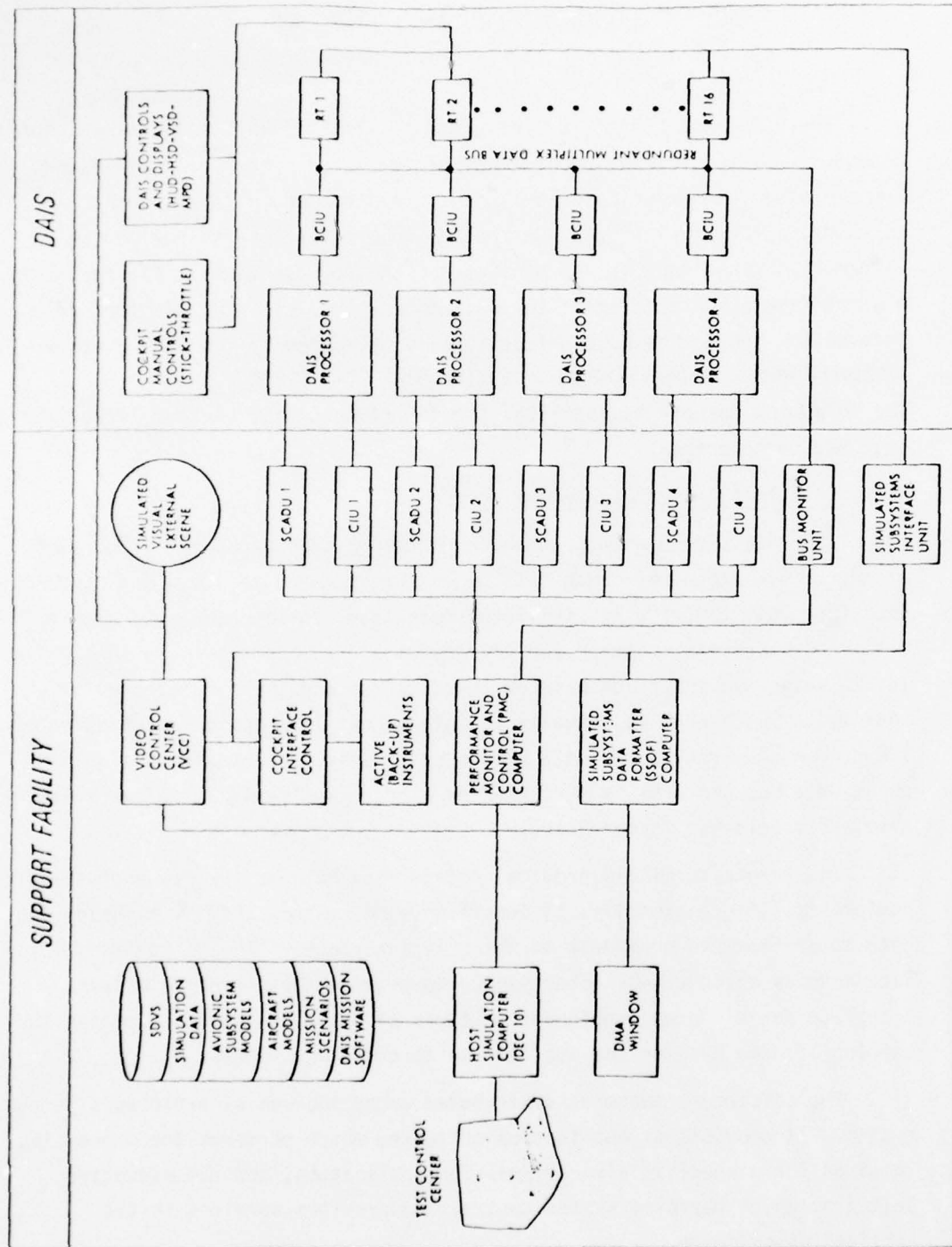


Figure 2. Simplified Block Diagram of ITB Facility

The executive software is further divided into the master executive, and the local executive. The master executive, which is responsible for system control, resides in one processor designated the master processor. The monitor executive resides in the monitor processor, and it provides a backup to the master executive. In the event of master processor/master BCIU failure, the monitor executive will assume system control. A copy of the local executive is located in each processor and provides real-time services, including data read and write, task control, etc., to the application software.

The mission software will be implemented in the JOVIAL J73/I higher order language utilizing structured programming techniques, and a modular architecture approach.

2 DAIS Support Facility

The Support Facility will provide the necessary interfaces to set up, provide real-time control and monitoring functions, and collect data for post run analysis for all DAIS testing activities.

A DEC10 computer will be used to execute real-time aircraft and environment models, compile (in J73) the DAIS executive and applications software, generate simulated mission scenarios, perform post run analysis, and maintain all the above files and simulation data under a configuration management system.

The Performance Monitoring and Control (PMC) computer in Figure 3-1 is a PDP 11/40 interfaced with the DEC10 via a DMA window. This machine will be used to load the mission software from DEC10 storage onto the DAIS processors. Operation of each processor will be monitored by a Super Control and Display Unit (SCADU) and a Console Interface Unit (CIU) which monitor the processor's memory bus and perform such functions as monitoring specified memory addresses, tracing branch instructions, breakpointing based on events, etc. The PMC computer will interact with the user to set up SCADU monitoring parameters and can also use canned scenarios stored on the DEC10 to set up the SCADU. Real time display of system performance will be available on a local CRT.

The Simulated Subsystems Data Formatting (SSDF) computer controls the transfer of data between the simulation models executing on the DEC10 and the DAIS mission software via the multiplex system. It obtains data from the simulation models needed to drive the backup cockpit instruments and supplies it to the cockpit interface and the Video Control Center (VCC). The SSDF also provides a mass memory simulation. During real-time testing, the VCC receives the simulated aircraft's position, altitude, and attitude data from the simulation models. These parameters are used, along with the necessary display map information, to provide the simulated external scene display. Other VCC functions are the digital recording of the backup cockpit instrument data and the video recording of DAIS controls and displays data.

4. Description of the SDVS

Section 3 described the DAIS concept and presented the hardware configuration for the Integrated Test Bed Facility. This section describes the SDVS which consists of an integrated collection of software tools to aid in the development, coding and testing of the actual flight software for the DAIS processors. Through the SDVS, an applications programmer will have such capabilities as the automated control of software versions and changes, an all-digital simulation capability for executing and testing the DAIS Mission Software without use of the actual hardware, the automatic control of simulation runs and the editing and analysis of the test-data generated, etc. SDVS will be used by both Air Force and other contract's personnel to aid in the design, implementation and testing of all Mission Software for the DAIS hot-bench. The SDVS tools essentially automate much of the manual setup and housekeeping activities previously required for software development in an integrated user oriented facility.

The following paragraphs summarize the basic functions the SDVS provides users. Each of the basic operation modes will be discussed in addition to the SDVS programming languages for defining simulated DAIS scenarios and for editing and displaying simulation data.

1. SDVS Modes of Operation

a. Mode Selection

Upon entering the SDVS, (by performing an "R SDVS") the user is prompted for the desired mode of operation. The following scenario illustrates a user entering SDVS and requesting (via the "HELP" command) a display of the modes of operation.

R SDVS

WELCOME TO SDVS VER.3B(061176), YOUR NAME

(LOGS NAMES L14370 ASSIGNED TO THIS SDVS RUN)

SDVS IS READY. WHICH MODE OF OPERATION IS DESIRED?

+++HELP

PLEASE ENTER NAME OR INITIALS OF ONE OF FOLLOWING:

FILE GENERATION	(FG)
SET UP & RUN SIMULATION	(SURS)
POST RUN EDIT	(PRE)
ROLLBACK	(RB)
DELETE MODE (MANAGER ONLY)	(DM)
SUPERVISOR MODE (MANAGER ONLY)	(SM)
LOGOFF	(LOG)

Based on the user's input, SDVS will enter the selected mode of operation and perform the desired user actions. Whenever the user is finished in SDVS, he enters the LOGOFF command. The following paragraphs describe the seven basic SDVS modes.

b. File Generation

The File Generation mode provides the necessary tools and configuration management aids for maintenance of all files associated with the development, test, and verification of the DAIS software. An extensive cataloging system is maintained for a number of different types of software controlled by SDVS including; DAIS mission software, SDVS test case files (defining simulation scenarios and data collection requirements), environment and aircraft models, and post simulation data reduction and analysis programs. Manipulation of files cataloged in SDVS is provided for by a number of conversational commands (e.g., editing, compiling, printing, etc.) described in section 4.1.2.2.

(1) File Structure

Each file type is cataloged by SDVS on a version/revision basis. For example, when the user creates a mission software file, it is cataloged as version 1, revision 0 and stored in a "baseline file". As the user edits the file in later sessions, he creates a number of revisions. Each revision results in the edited changes being cataloged as a unique record in the "difference file" for the particular file version. At any point in time, he can combine all the revisions associated with a particular version and

make a new version with the conversational NEXT-VERSION command. Under SDVS the user can access any version and revision number for a file since each EDIT session generates a unique entry in the difference file for a particular file version. Figure 4-1 illustrates this capability.

(2) File Conversational Commands

SDVS provides the user a number of commands to perform various actions on SDVS files. The following commands illustrate some of the functions provided for:

HELP	- List the format of all SDVS user commands
ACCESS	- Make available a specific file, version, revision for processing
EDIT	- Perform text editing on a file
COMPILE	- Compile a J73 program
COPY	- Copy a file onto another file name
NEXT-VERSION	- Generate a new version from all the revisions of the version specified
PRINT	- Print a file on the system hard copy device
CREATE	- Create a new file
ENTER	- Enter a file from the host computer into the SDVS catalogs
OUTPUT	- Output a file from SDVS to the host computer

In interpreting the above commands, SDVS will interrogate the Configuration Management catalogs to determine if the user has authority to access the desired file (file security is specified in the Supervisor Mode, section 4.1.7). If he does not, he is output an error message and asked what else he wants to do; otherwise the requested operation is performed.

For example, if the user desires to compile version 5, revision 2 of the file, NAVIGATION, he would enter the following:

COMPILE NAVIGATION/5/2/MSW

where MSW indicates a mission software file type requiring the JOVIAL compiler. SDVS will assemble a single text file from the difference

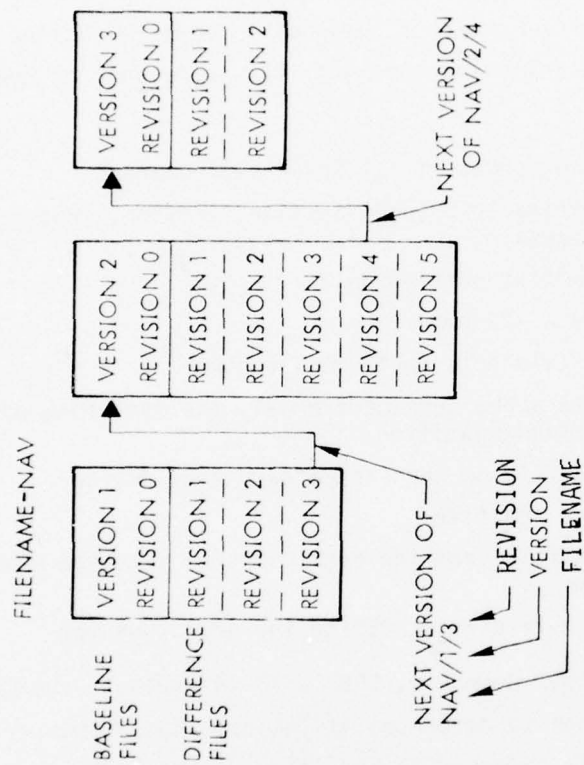


Figure 3 SDVS File Structure

file and baseline file stored in the catalogs (see Figure 3) and call the system JOVIAL compiler. Upon completion of the compilation, the compiler will return control to SDVS which will now automatically link the translated code to version 5 revision 2 of file NAVIGATION. Note, that the user does not have to keep track of source file names, object file names, etc.; these functions are handled automatically by the SDVS. The following paragraphs discuss the three basic file types maintained by the SDVS; mission software, simulation test case, and post run edit files.

(3) Mission Software Files

The SDVS catalogs provide configuration control for the development, test, and maintenance of the DAIS mission software. The user will have the capability to create and edit JOVIAL code and COMPOOL data files. SDVS will automatically link COMPOOL data files to program files in the catalogs.

The user will be able to produce listings, save newly created and updated files, and invoke the JOVIAL J73 compiler or the DAIS processor assembler. The SDVS will automatically catalog all revisions made to a mission software file, and catalog object modules from successful compilations.

Figure 4 is an example of an interactive SDVS session used to create a mission software file (MSW-PQT-DRIVER-1) that is to be linked with a COMPOOL file. All user inputs are outlined.

(4) Simulation Test Case Files

The File Generation mode of SDVS operation also provides the user the capability to create, modify, and translate source test case files containing Simulation Control Language statements. These source test case files provide the directives which define the initialization and control of a simulation run including sequences of operations, failure conditions, outputs to the rough output tape for post processing in the Post Run Edit Mode, etc.

The user inputs to build test case files are of two types, Conversational Language and Simulation Control Language. The user will enter Conversational Language commands to enter the appropriate file handling mode of operation, i.e., create, edit, print, copy, etc. The test case files,

```

SDVS IS READY.  WHICH MODE OF OPERATION IS DESIRED?
+++FG
YOU ARE NOW ENTERING FILE GENERATION MODE OF OPERATION
WHAT ACTION DO YOU WANT TO PERFORM?
ENTER "HELP" TO LIST THE AVAILABLE COMMANDS
ENTER "HELP COMMAND-NAME" TO GET THE SYNTAX OF A GIVEN COMMAND.
+++CREATE MSW-PQT-DRIVER-1
FILE TYPE:MSW
SECURITY LEVEL:2
LIST COMPOOL FILES, ONE AT A TIME, USING THE FOLLOWING FORMAT
FILENAME/VERSION/REVISION
TERMINATE THE LIST WITH A "#".
+++MSW-CMP-01/4/1
+++#
(Note: COMPOOL files must already exist and be compiled)
LIST COPY FILES, ONE AT A TIME, USING THE FOLLOWING FORMAT
FILENAME/VERSION/REVISION
TERMINATE THE LIST WITH A "#".
+++#(no COPY files)
(Note: COPY files must already exist, to be specified)

```

*** ENTERING TECO CREATE SESSION -- START WITH 'I':

```

*I !COMPOOL 'MSW-CMP-01' (CMPSRT);
PROC DRIVER;
BEGIN
  NUM = 4;
  TIMES [1] = 4;
  TIMES [2] = 49;
  TIMES [3] = 19;
  TIMES [4] = 14;
  SORTPG;
  TIMES [1] = 49;
  TIMES [2] = 17;
  TIMES [3] = 11;
  TIMES [4] = 19;
  SORTPG;
END;
[ $$ ]

```

*** RETURNING FROM EDIT SESSION OR J73

```

VERSION 1 OF THIS FILE HAS BEEN CREATED AND LOCKED TO THIS USER.
WHAT ACTION DO YOU WANT TO PERFORM?
+++LOGOFF

```

Figure 4 Example of Creating a File in SDVS

themselves, will contain statements in the Simulation Control Language which are, at user request, translated to an internal form for later use in directing a Simulation run.

The primary output of building test case files are the internal test case files which will be used to control the initialization and execution of simulation runs. In addition, the user will obtain interactive outputs during file manipulation, such as successful completion and error messages.

The test case directives file will reference mission software files that are to be used in the simulation. It will provide directives to generate the rough output tape which will be analyzed after the simulation run is complete. The test case directives source and internal files are maintained in the SDVS file catalogs. Section 4 No. (1) presents an example of a Simulation Control Language program.

(5) Post Run Edit Files

The File Generation mode of SDVS operation also provides the user the capability to create, modify and translate source Post Run Edit directives files to internal form. These PRE directives files contain the Data Processing Language commands which define the processing to be done on a rough output tape in the SDVS Post Run Editor mode of operation.

The user inputs are of two types, Conversational Language and Data Processing Language. The user will enter Conversational Language Commands to select the desired file handling mode of operation, i.e., create, edit, print, copy, etc. The PRE directives files, will contain statements in the Data Processing Language, which at user request, are translated to an internal form for input to the Post Run Editor. A sample PRE program is presented in section 4 No. (1).

The primary output of this mode of operation are the internal PRE directives files which are used in the Post Run Editor SDVS mode to perform data editing functions on a particular rough output tape generated by a simulation run. In addition, the user will obtain interactive outputs, such as successful completion and error messages, during the file manipulation and translation process. The PRE directives files, source and internal are cataloged by SDVS. A PRE directives file may be specified by the user in a test case file to be automatically run at the end of a simulation run.

c. Set Up and Run Simulation

This mode of operation is used to submit a simulation run based on a test case directives file that has previously been created and translated. The user inputs for this mode of operation include:

- o Specification of the test case file
- o Maximum simulation time
- o Time of day for executing the simulation

SDVS will automatically:

- o Retrieve the test case file and load the specified mission software for simulation
- o Interact with the machine operator to mount the necessary tapes
- o Perform initialization commands specified in the user's test case
- o Execute the desired simulation scenario
- o If specified in the test case, automatically transfer control to the Post Run Edit mode and perform post run data analysis and editing on the simulation data.

The software tools used for simulation of a DAIS mission are described in section 3.

d. Post Run Edit

The Post Run Edit mode provides the user with the ability to analyze the data recorded on a Rough Output Tape during a simulation run. The Post Run Editor will access the user-specified translated Post Run Edit directives file. The directives will specify what data is to be selected from the ROT, what analysis is to be run on that data, what format is to be used to display the analysis results, what user routines are to be used, and what devices are to receive the output files created by the Post Run Editor.

The Post Run Editor provides tabular printouts, interactive displays and data plots based on user directives. An important feature is the ability for a user to write an analysis routine in JOVIAL and have it execute within the framework of the Post Run Editor.

e. Rollback

The rollback function will provide the user the capability to restart and rerun an SDVS simulation from a point during a previous simulation run as stored on a snapshot tape. The user may change the test case to obtain additional output or alter existing conditions, following the point saved on the snapshot tape. The user will do this by generating a Rollback test case file which will be merged with the one used for the earlier simulation by SDVS.

The user will input a specification of the Rollback test case file to be used and the original Test Case file.

The outputs of this mode of operation will be a simulation run which (if no changes were made in the test case file) will exactly match the previous one. Changes may be made to provide further analysis of a simulation run which is of special interest.

f. Delete Mode

This mode of operation is only available to the SDVS manager and provides him the capability to delete files from the various SDVS catalogs. This function was made a manager level function to allow manager level control over the disposition of all files.

g. Supervisor Mode

This mode, like the Delete mode, is available only to the SDVS manager for configuration control purposes. One of the features of the SDVS file management scheme is that before a user may generate any file in File Generation mode, specifications must have been created for that file, including the provision of a list of users who are authorized to write (e.g., create new versions) on the file and, if appropriate, a list of users who are free only to read it. The creation of file specifications must be performed from Supervisor mode. This mode can be entered only by a SDVS user logged in on the special Manager programmer number.

In Supervisor mode, statements in the Conversational Language will be entered. One of these statements will allow the manager to create specifications for a particular file, and enter the initial list of users who have authority to read and/or write that file. Another statement allows the manager to change the authority of a user from read only to read/write or vice versa, or add new users to the list of authorized users, or remove users from the list. Both of these commands may be completely specified by the user or he may elect to be partially or entirely prompted for the necessary information.

There is only one type of output to the authorized user from Supervisor mode. This output consists of information relayed to the user about the result of processing his request. This might be a description of any syntax error which has been detected by SDVS or an indication that an error occurred while the request was being processed, or a message indicating that the request was satisfied. The specification files and the lists of authorized users are inaccessible to any SDVS user whether in Supervisor mode or not. Figure 4-3 illustrates the use of this mode.

2. SDVS User Languages

In the File Generation SDVS mode, the user can create simulation test case and post run edit files. These files are user programs written in the SDVS Simulation Control Language (SCL) and Data Processing Language (DPL). The Simulation Control Language provides the mechanism for the user to control a simulation by specifying initial conditions, scheduling events to occur based on time or conditions, and specifying output to be generated. The Data Processing Language is used to specify the output data and format desired (hardcopy printout, terminal display or Plot).

a. Simulation Control Language

The SDVS Simulation Control Language is a programming language used to specify simulated DAIS missions. The language syntax is patterned after the JOVIAL language and can be categorized into (1) non-executable statements,

BEST AVAILABLE COPY

SDVS IS READY. WHICH MODE OF OPERATION IS DESIRED?
++[SUPERVISOR]
YOU ARE NOW ENTERING SUPERVISOR MODE OF OPERATION
WHAT ACTION DO YOU WANT TO PERFORM?
++[CREATE-SPECS]
FILENAME[PROC-33A]
FILE TYPE:[MSW]
SKIP TECO (CR=YES)?
ENTER LINES OF THE FOLLOWING FORMAT, ONE AT A TIME:
ADD PROJECT-NUMBER, PROGRAMMER-NUMBER, ACCESS-CODE
WHERE ACCESS-CODE MAY BE:
R (READ-ONLY ACCESS) OR
W (READ/WRITE ACCESS)
TERMINATE THE LIST WITH A #.
* [ADD 3202,1224,W]
* [ADD 3202,653,R]
* [ADD 3202,1111,R]
* [#]
WHAT ACTION DO YOU WANT TO PERFORM?
++[CHANGE-AUTHORITY FOR PROC-33A/MSW]
ENTER LINES OF THE FOLLOWING FORMAT, ONE AT A TIME:
ADD PROJECT-NUMBER, PROGRAMMER-NUMBER, ACCESS-CODE
REMOVE PROJECT-NUMBER, PROGRAMMER-NUMBER
CHANGE PROJECT-NUMBER, PROGRAMMER-NUMBER, NEW-ACCESS-CODE
WHERE ACCESS-CODE MAY BE:
R (READ-ONLY ACCESS) OR
W (READ/WRITE ACCESS)
TERMINATE THE LIST WITH A #.
* [REMOVE 3202,1111]
3202,1111 HAS BEEN REMOVED FROM THE LIST OF AUTHORIZED USERS.
* [CHANGE]
PROJECT NUMBER? [3202]
PROGRAMMER NUMBER? [653]
NEW ACCESS CODE? [W]
3202, 653 HAS HAD HIS AUTHORITY CHANGED TO W.
* [ADD]
PROJECT NUMBER? [3202]
PROGRAMMER NUMBER? [457]
ACCESS CODE? [R]
ACCESS RECORD STORED.
* [#]
WHAT ACTION DO YOU WANT TO PERFORM?
++[LOGOFF]
DO YOU WANT YOUR LOG FILE PRINTED? [NO]
*** SDVS TERMINATION ***

Figure 5 Example of Supervisor mode operation

(2) sequential statements, and (3) asynchronous statements. The non-executable statements are used to convey control information to the simulation system such as

- o the mission software to be simulated
- o the flight profile tape to be used
- o the Post Run Edit program to be executed after the simulation
- o the variables to be traced
- o the type of computer simulation (ICS or SLS)
- o the type of rollback (and time)

Certain sequentially executed statements provide many of the capabilities of conventional programming languages such as FORTRAN, PL-1, and JOVIAL. Other sequential statements direct the Simulation Control Program to perform a simulation-related function. These statements are used to:

- o assign values to variables
- o transfer control to other statements
- o evaluate a logical expression and execute one of two statements depending on the value of the expression
- o activate simulated computers
- o collect data
- o turn traces on and off
- o terminate the simulation

Asynchronous statements are not executed sequentially; they are executed asynchronously as the result of a user-specified condition becoming true. In a sense, the true state of the condition behaves as a software interrupt which triggers the execution of the statement. There are three types of conditional statements providing different types of asynchronous control:

- o The WHEN statement contains a condition and a sub-statement. The first time the condition becomes true, the sub-statement is executed. An English language analogy is "When Harry gets home ask him to go to the grocery store". The first time Harry gets home he is asked to go to the store; he is not asked every time he gets home.

- o The WHENEVER statement also contains a condition and a sub-statement. Everytime the condition becomes true, the sub-statement is executed. An analogy is "Whenever the car runs low on gas, fill it up". The car gets filled up everytime it runs low on gas.
- o The WHILE statement contains a condition, a repetition frequency and the name of an SCL procedure. Until the condition becomes false the SCL procedure is performed repeatedly at that specified rate. The old saying "Keep doing it until you get it right" might be implemented using a WHILE statement.

(1) SCL Example

To illustrate use of the SCL in testing mission software, consider the development of a new navigation algorithm, NAV, that has been created and compiled in the SDVS mission software file catalogs. The user is interested in generating a flight profile such that the SDVS avionic and sensor models will generate realistic navigation sensor data for input to the NAV routine.

Figure 6 is a pictorial representation of the following flight scenario:

- o Takeoff is at latitude 35° , longitude 117° with a thrust command of 12000 pounds.
- o When the X velocity > 170 fps, pitch the aircraft at 2° /second.
- o When the pitch angle $> 20^{\circ}$, maintain that pitch.
- o When altitude exceeds 10000 feet, level the aircraft by setting a negative pitch rate.
- o When the pitch angle is less than zero, terminate the simulation.

Using the SDVS SCL, the user builds a test case file to specify:

- o The flight profile.
- o Data to be recorded for post processing.
- o Sensor data to be used by the NAV routine.

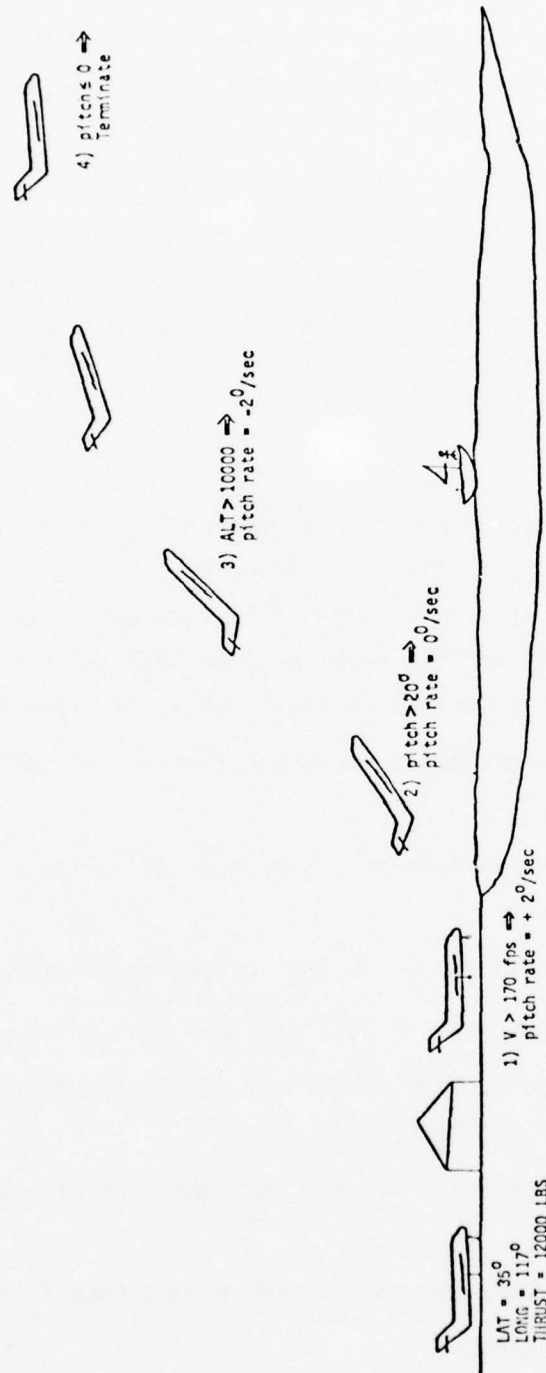


Figure 6 SAMPLE SDVS FLIGHT PROFILE

Figure 7 is an SCL program for this example. The reader should note the following points in this sample program:

- o The CONFIGURE statement specifies the STANDALONE mode which allows a user to interface directly with environment model data via an EFS (Executive Functional Simulation) block instead of using the real DAIS executive software. It also specifies the SDVS simulator (the SLS) and the files to be loaded (version 1 revision 0 of NAV).
- o The EFS Control Block (EFS-NAV-INPUT) defines the assignment of environment mode sensor data (denoted by the prefix E:) to variables in the program, NAV. These assignment statements will be executed periodically at 32 times per second prior to executing the NAV routine as defined by the statement,

PERFORM EFS-NAV-INPUT EVERY .03125 UNTIL 1000;.

- o The INCLUDE statement allows the user to copy in other test case files to be included as part of the test case.
- o The ROT-SIM-DATA block defines model and mission software position data that is to be recorded once a second as defined by the statement,

PERFORM ROT-SIM-DATA EVERY 1 UNTIL 1000;.

- o The CON-INIT block defines all the initial conditions at ground zero. These assignments are executed by the statement,

PERFORM CON-INIT;.

- o The statements defining the mission profile correspond to the flight profile illustrated in Figure 4-4.

BEST AVAILABLE COPY

EXAMPLE OF SDVS SIMULATION CONTROL LANGUAGE

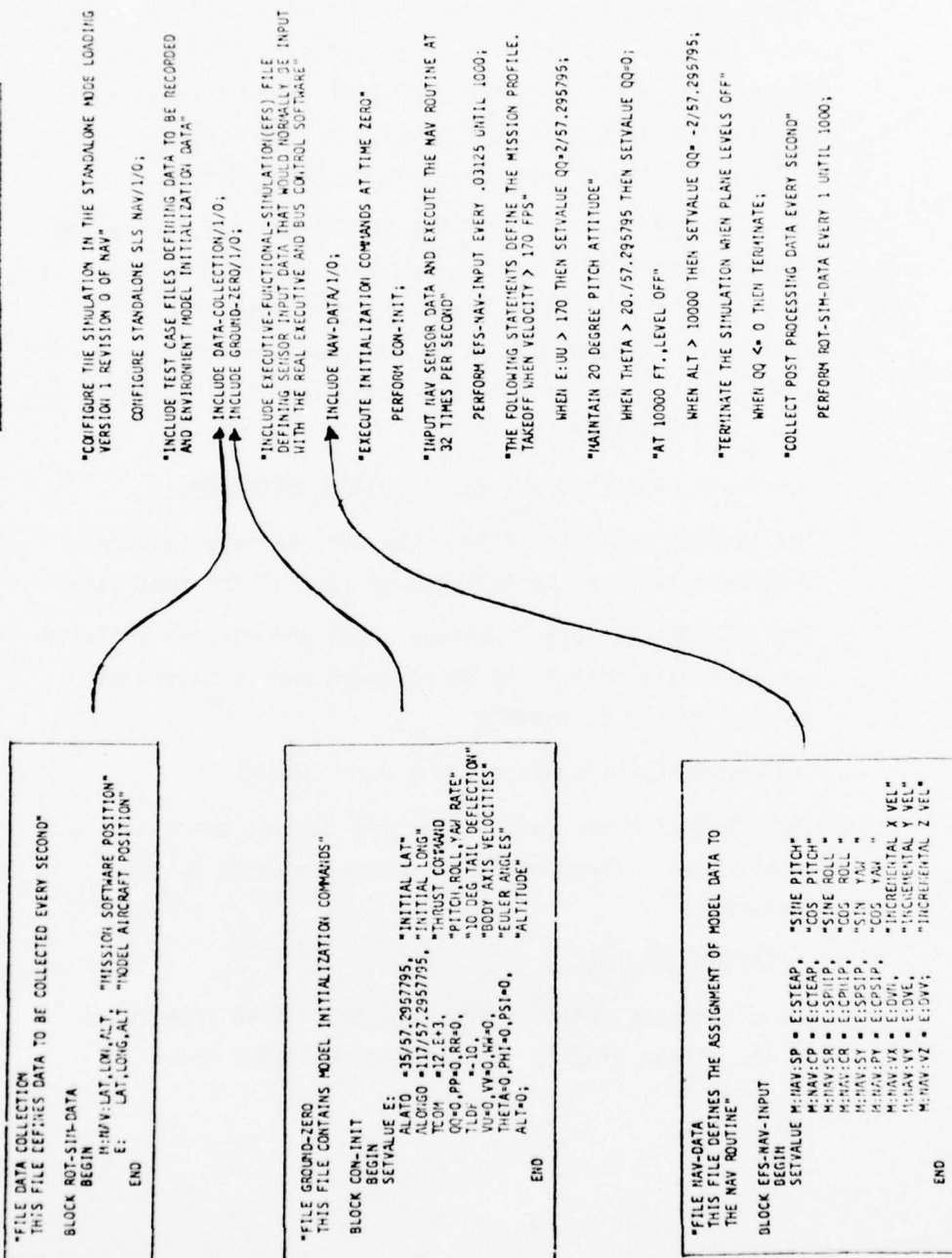


Figure 7 Sample SDVS SCL Program

f. Data Processing Language (DPL)

A SDVS simulation generates a volume of data collected at numerous points in a simulation. With the SCL, the user can specify both conditional and unconditional events that result in the output data to a Rough Output Tape. This tape contains all the simulator trace outputs, a load map of the mission software for each DAIS processor, run time error and warning messages from the various simulators, data from the environment models and mission software defined by the user, etc., as they occur in simulated time. Figure 7 illustrates the use of a Rough Output Tape (ROT) block defining the variables to be recorded every second during a simulation. From the vast volume of data, the user must be able to sort out and display the information in a meaningful format.

The SDVS Data Processing Language has been designed to provide the SDVS user an easy to use flexible tool to select for analysis, printout or plotting the specific parameter data he desires. In selecting data to be output, the user does not have to worry about conversion of mission software or environment model data from binary to the correct output format, this is all handled automatically by SDVS. To determine the correct formats, SDVS reads the symbol tables generated for mission software and environment model programs by the JOVIAL and FORTRAN compilers, and extracts the necessary information. This SDVS tool removes much of the drudgery sometimes associated with data analysis. The DPL provides the following user oriented functions:

- o Generation and editing of data files containing user defined variables from the ROT.
- o A PRINT capability to output generated data files to the printer.
- o A DISPLAY capability to output information to the user's interactive terminal.
- o A statistical package to compute statistical information of simulation data.

- o Automatic generation of plots based on collected simulation data.
- o Execution of user supplied analysis routines using a simulation ROT.

(1) DPL Example

Figure 8 is a DPL program that can be used to process data collected in the SCL example shown in Figure 7.

This DPL program is used to print out the environmental model and mission software nav data. This data will be printed on the line printer, and will be analyzed by a user routine, error analysis, to determine the mission software error. This error is then plotted as a function of time.

The reader should note the following points from this sample program:

- o The CONFIGURE statement specifies the user routine to be executed, and its language (JOVIAL).
- o The GENERATE statement is used to create a data file, NAV-DATA, which includes all the data on the ROT block, ROT-SIM-DATA.
- o The FORMAT statement is used to define the format of the data file (NAV-ACCURACY) which is computed by the user's routine.
- o The PRINT statement will automatically print out all the data contained in the data file, NAV-DATA. The output is in a tabular format and is time tagged.
- o The PLOT commands shown allow the user to specify the plot title, the axis titles, and any desired data conversions. The user could also specify the X and Y axis lengths, the minimum and maximum X and Y values allowed, and any biases to be added or subtracted from variables to be plotted.

BEST AVAILABLE COPY

EXAMPLE OF SDVS POST RUN PROCESSING

```
"THIS POST-RUN-EDIT PROGRAM IS USED TO PRINT OUT THE"  
"ENVIRONMENTAL MODEL AND MISSION SOFTWARE NAV DATA FROM"  
"A SIMULATION RUN. THIS DATA WILL BE PRINTED ON THE LINE"  
"PRINTER, AND WILL BE ANALYZED BY A USER ROUTINE, ERROR"  
"ANALYSIS, TO DETERMINE THE MISSION SOFTWARE ERROR. THIS"  
"ERROR IS THEN PLOTTED AS A FUNCTION OF TIME"  
  
"SPECIFY THE USER ERROR-ANALYSIS ROUTINE"  
  
    CONFIGURE USER-ROUTINE JCVIAL ERROR-ANALYSIS/1/0;  
  
"GENERATE THE DATA FILE, NAV-DATA, CONTAINING"  
"THE PARAMETERS IN ROT BLOCK, ROT-SIM-DATA."  
  
    GENERATE NAV-DATA ROT-SIM-DATA;  
  
"DEFINE THE DATA FILE CONTAINING THE OUTPUT OF THE USER"  
"ROUTINE. THIS OUTPUT IS THE NAVIGATION ERROR FOR LATITUDE,"  
"LONGITUDE, ALTITUDE, AND THE SIMULATION TIME, TIME"  
  
    FORMAT NAV-ACCURACY  
    BEGIN  
        FLOATING: LAT-ERR,  
        FLOATING: LON-ERR,  
        FLOATING: ALT-ERR,  
        FLOATING: TIME  
    END;  
  
"PRINT OUT ALL THE MSW AND EES NAV DATA"  
  
    PRINT NAV-DATA;  
  
"EXECUTE THE USER ROUTINE, ERROR-ANALYSIS, WHICH COMPUTES"  
"THE NAVIGATION ERROR"  
  
    "INPUT FILE: NAV-DATA"  
    "OUTPUT FILE: NAV-ACCURACY"  
  
    EXECUTE ERROR-ANALYSIS  
  
        NAV-DATA:NAV-ACCURACY;  
  
"PLOT THE COMPUTED NAVIGATION ERRORS AS A FUNCTION OF TIME"  
  
    PLOT NAV-ACCURACY SIM-TIME, LAT-ERR(RAD-DEG)  
        TITLE='LATITUDE ERROR VS. TIME'  
        XLABEL='TIME(SEC)'  
        YLABEL='LATITUDE ERROR(DEG)';  
  
    PLOT NAV-ACCURACY SIM-TIME, LON-ERR(RAD-DEG)  
        TITLE='LONGITUDE ERROR VS. TIME'  
        XLABEL='TIME(SEC)'  
        YLABEL='LONGITUDE ERROR (DEG)';  
  
    PLOT NAV-ACCURACY SIM-TIME, ALT-ERR  
        TITLE='ALTITUDE ERROR VS. TIME'  
        XLABEL='TIME(SEC)'  
        YLABEL='ALTITUDE ERROR(FEET)';
```

Figure 8 Sample SDVS DPL Program

3. SDVS Simulation Facilities

Section 4, c. described the SDVS Set Up and Run Simulation Mode in which a user specifies a test case file defining the desired simulation scenario. Section 4, a. presented an introduction into the test case language used to describe simulation scenarios. The purpose of this section is to present the SDVS simulation tools that simulate the DAIS hardware (processors, data bus, remote terminals) shown in the right half of Figure 2 and provide analogous "support" capabilities shown in the left half of the same figure.

Figure 9 presents the DAIS Integrated Test Bed Facility from a SDVS simulation point of view.

The left half of Figure 9 illustrates the SDVS support facilities for mission software testing and validation functions. The heart of this support facility is the Simulation Monitor and Control function. It can be viewed by the user as a "virtual machine" performing functions of the Performance Monitor and Control and Simulated Subsystems Data Formatter computers of the DAIS support facility. The virtual SDVS machine is actually much more powerful than the actual computers since there are no hardware limitations, and all monitoring and control functions can occur in zero simulation time.

The right half of Figure 9 shows the SDVS simulators of the DAIS components. The only simulator not included in the current version of SDVS are for the DAIS Controls and Displays which may be added at a future date. The SDVS code executors consist of both a Statement Level Simulator (SLS) and an Interpretive Computer Simulator (ICS). The SLS executes mission software generated by the J73 compiler for execution on the host DEC10, while the ICS executes the actual DAIS processor code with bit level accuracy. Since the SLS actually executes on the DEC10, the throughput is very high compared to the interpretive execution of the ICS. These two tools provide the user a choice of fidelity in the simulation of mission software code.

BEST AVAILABLE COPY

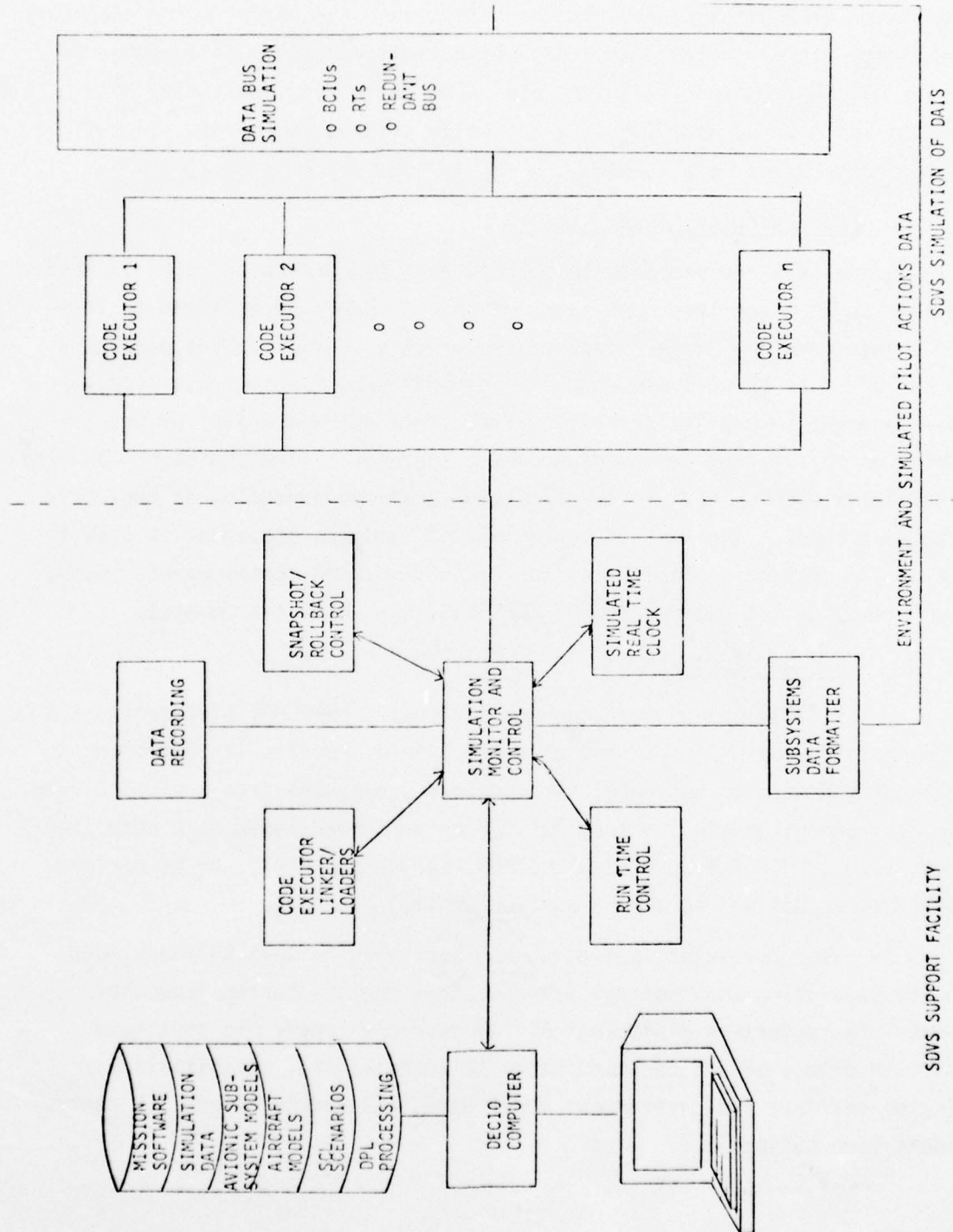


Figure 9 The SDVS View of DAIS

a. SDVS Support Facility

The SDVS Support Facility consists of a number of tools that SDVS uses to implement the simulation scenario defined in the user's test case file. The Simulation Monitor and Control function interprets the user's translated Simulation Control Language test case and performs the necessary functions. The translated test case essentially consists of a number of tables and instructions defining data to be collected, monitoring of conditional events, user run time control of simulation events, execution of the environment models, etc.

(1) Code Executor Linker/Loaders

Since SDVS can simulate both DEC10 and DAIS processor code, it must be able to link and load both types of object code. In addition to loading object code the linker/loaders also process special tables generated by the J73 mission software compiler for SDVS use. These tables include the statement boundaries for each JOVIAL statement and a list of the variables that can be set by each JOVIAL statement. The statement boundary information enables SDVS to trace mission software execution at the JOVIAL statement level. The list of variables set for each statement is used to set up the monitoring facilities for evaluating conditional events (e.g., Simulation Control Language WHEN, WHENEVER, and WHILE statements).

(2) Snapshot/Rollback

The Simulation Control Language includes a SNAPSHOT statement which, when executed during the course of a simulation, results in saving the state of the mission software, the code executors, the data bus model, and the environment models. A snapshot can be performed based on a conditional event (e.g., WHEN A > 50 OR B < 30 THEN PERFORM SNAPSHOT), or at periodic intervals as defined in the Simulation Control Language.

The user can later initiate a rollback via the SDVS Rollback mode to any simulation snapshot point and restart the simulation from that point. In restarting a simulation, the user can modify his test case files to delete or add new conditions to be evaluated, re-initialize mission software and environment model data, and add or delete SCL control blocks (subroutines).

The use of this tool can save many hours of DEC10 computer time by not having to start simulation runs from time zero to get more debugging information, or to analyze performance by changing critical variables in the mission software or environment models.

(3) Data Recording

SDVS can record a wealth of information to assist the user in the debugging or validation of mission software. The data that can be recorded during a simulation includes the following:

- (a) Code executor trace information (statement trace, transfer trace, ICS register trace, ICS instruction trace, etc.).
- (b) Data bus simulation trace information (I/O interrupts generated, bus commands, bus traffic, etc.).
- (c) Environment model trace information defining the time each model was executed.
- (d) Values of variables selected to be traced by the user each time they are updated.
- (e) Simulation run time warning or error messages detected by SDVS.
- (f) User defined mission software or environment model data as requested by the user.

As described in section 4.2.2.1, the user can analyze data collected by a simulation by constructing a program in the SDVS Data Processing Language.

(4) Subsystems Data Formatting

Since SDVS uses the same environment models that will be used on the actual facility, the Subsystems Data Formatting function is very similar for both SDVS simulations and DAIS facility. The only difference is that for SDVS, this function interfaces with the simulation of the remote terminals included as part of the data bus simulation. As the

mission software executive initiates I/O commands which address variables computed by the environment models, this function will signal the Simulation Monitor and Control function to pass data to, or receive data from a model, and if necessary, execute the appropriate model at that instant of simulation time. This technique of executing environment models on demand insures accurate sensor data for the mission software.

(5) Simulated Real-Time Clock and Run Time Control

These two functions provide the sequencing of simulation events which drive the Simulation Monitor and Control function. The simulated clock along with several event queues provide the ability to simulate in SDVS all the parallel operations that occur simultaneously on the real DAIS facility. The parallel operations include the operation of multiple processors and multiple BCUs.

Run time control includes the queuing of events for data recording, execution of environment models, simulation of transmission delays over the data bus, evaluation of conditional events, and performance of Simulation Control Language "subroutines". This function provides control of a Master Event Table which dispatches the events described above for execution.

f. SDVS Simulators

The SDVS simulators of the actual DAIS hardware provide user tools to debug and validate DAIS mission software in conjunction with, or to the exclusion of, the actual DAIS hardware. The code executors and data bus simulation each contain a small interface for communication with the Simulation Control and Monitor function. This interface provides data to the simulators to indicate simulation of error conditions (bad parity on the bus, invalid bus protocol, etc.) and receives data from the simulators to be logged by the Data Recording function. This interface is well defined and documented such that it would be possible to substitute a different ICS for a new DAIS processor, or a different Data Bus simulation for a different multiplex protocol. This ability to substitute SDVS simulators is also possible because the SDVS Support Facility shown in

Figure 6, and the Simulation Control and Data Processing languages have been designed to be as independent as possible from the simulators. This capability to substitute simulators opens up a whole new world of potential SDVS applications. As much as DAIS is to be used to evaluate and study new avionic hardware and software concepts, SDVS could be used as an engineering facility to evaluate the DAIS concept with different processor capabilities, data bus protocol, etc. Since the Simulation Control Language test case programs and Data Processing Language post processing programs are independent of the simulators, a number of standardized programs in these languages can be developed to evaluate different mixes of the DAIS simulators and compare their relative performance.

(1) Interpretive Computer Simulator (ICS)

The ICS provides a bit level simulation of the DAIS processor to support the testing and validation of mission software. The ICS will simulate the operation of the DAIS processor at the instruction level, such that the resulting contents of registers and memory after execution is the same as the results obtained on the actual processor. The ICS also simulates the input/output operations of the processor, the interrupt system, all addressable registers in the CPU, and all the processor memory. SDVS can simulate multiple ICSs communicating over the data bus.

(2) Statement Level Simulator (SLS)

The SLS allows the mission software designer to check out his equations and program design without being concerned with the details of implementation on the DAIS processor. The SLS executes DEC10 code and will run many times faster than the ICS, thus allowing faster testing of the software design.

The SLS runs on the DEC10 computer and executes JOVIAL source statements in DEC10 code. The JOVIAL compiler provides the code to be executed by the SLS, along with traps to indicate JOVIAL statement boundaries. The SLS may be thought of as a code processor similar to the ICS, but at a

higher level. The ICS executes one instruction at a time, the SLS executes one JOVIAL source statement at a time. Multiple SLSs may run concurrently in the SDVS.

(3) Data Bus Simulation

The SDVS Data Bus Simulation is a tool that functionally simulates the DAIS multiplexed data bus architecture: It will simulate the command/response characteristics of the data bus with respect to I/O requests by the SDVS code executors (ICS and SLS), and the transmission of data to/from the various modeled sensors. The bus simulation will model the various components of the data bus such that the interface presented to the ICS or SLS is the same that would be presented to the actual DAIS processors.

The simulation was designed to be independent of the DAIS executive and I/O control software. This is done by interpretively simulating the BCIU registers; the response of the simulation to requests from the code executors is based on the state of the simulated registers. The bus simulation also provides the Simulation Monitor and Control function "event profiles" which define the time sequenced events commensurate with a bus operation (e.g., master BCIU commanding a remote BCIU to receive data). "Event profiles" are also constructed to simulate the events associated with error conditions that can occur in the real world. The user can specify the occurrence of simulated errors in SCL test case program.

SECTION V

DESIGN AND IMPLEMENTATION TECHNIQUES

1. Top-Down Approach

The purpose of this section is to describe TRW's approach to the development of the SDVS software and to comment on some of the techniques employed. The basic design philosophy was based on a top-down approach to software design, development, and test. Top-down design, as applied to the SDVS software development includes the following techniques which are discussed in detail in the following sections:

- o Overall design of the program hierarchy and successive refinements of this design from software requirements to specifications. A basic design criteria was to develop the SDVS to be re-hosted on another large scale system with minimal modifications.
- o Definition of all control and data interfaces according to the hierarchy. Standard "Interface Diagrams" are used to convey this information to all project members.
- o Establishment of programming standards defining the tools and rules to be used for software development. This includes use of structured programming techniques and a high level language which supports the structured programming conventions.
- o Integration and testing according to a three phase delivery schedule in which each phase provided enhanced SDVS capabilities. Formal testing procedures were based on testing the functional requirements reflected in the program hierarchy. Testing included developing tests that followed the successive refinement process involved in the basic design in addition to standalone testing of various functions.
- o Configuration control procedures for both the development and testing of the software. During development, procedures were used for controlling data shared among programs and utility functions. During testing, strict configuration control of all files was enforced and a formal problem reporting system was employed.

As can be seen from the overview of the above techniques, the TRW approach to top-down design incorporated a number of new techniques in software technology being advocated in the literature today. TRW's approach for the SDVS software development was to integrate these new techniques into an overall systems approach to develop a top-down strategy for the design development, and testing of the SDVS.

The following paragraphs describe in more detail the top-down techniques used in the SDVS development. Since many of these techniques are relatively new and little data is available on the advantages and disadvantages of their application, an evaluation of their use for SDVS will be presented.

a. Hierarchial Design

The design of the SDVS was based on the program hierarchy shown in Figure 5-1. Each box shown, except for the grouping of DEC10 services, represents a deliverable SDVS program. The following paragraphs briefly describe the hierarchial organization shown. This section concludes with some conclusions concerning experience gained during implementation of the program hierarchy.

(1) Program Hierarchy

SDVS Control Program

This program interacts with the user in determining the desired mode of operation and then transfers control to the appropriate second level routine. All host processor functions are performed by this program (I/O handling, calling the system compiler, etc.) based on conversational commands input by the user from the File Generation mode and from second-level program requests. It will also submit simulation runs into the batch system.

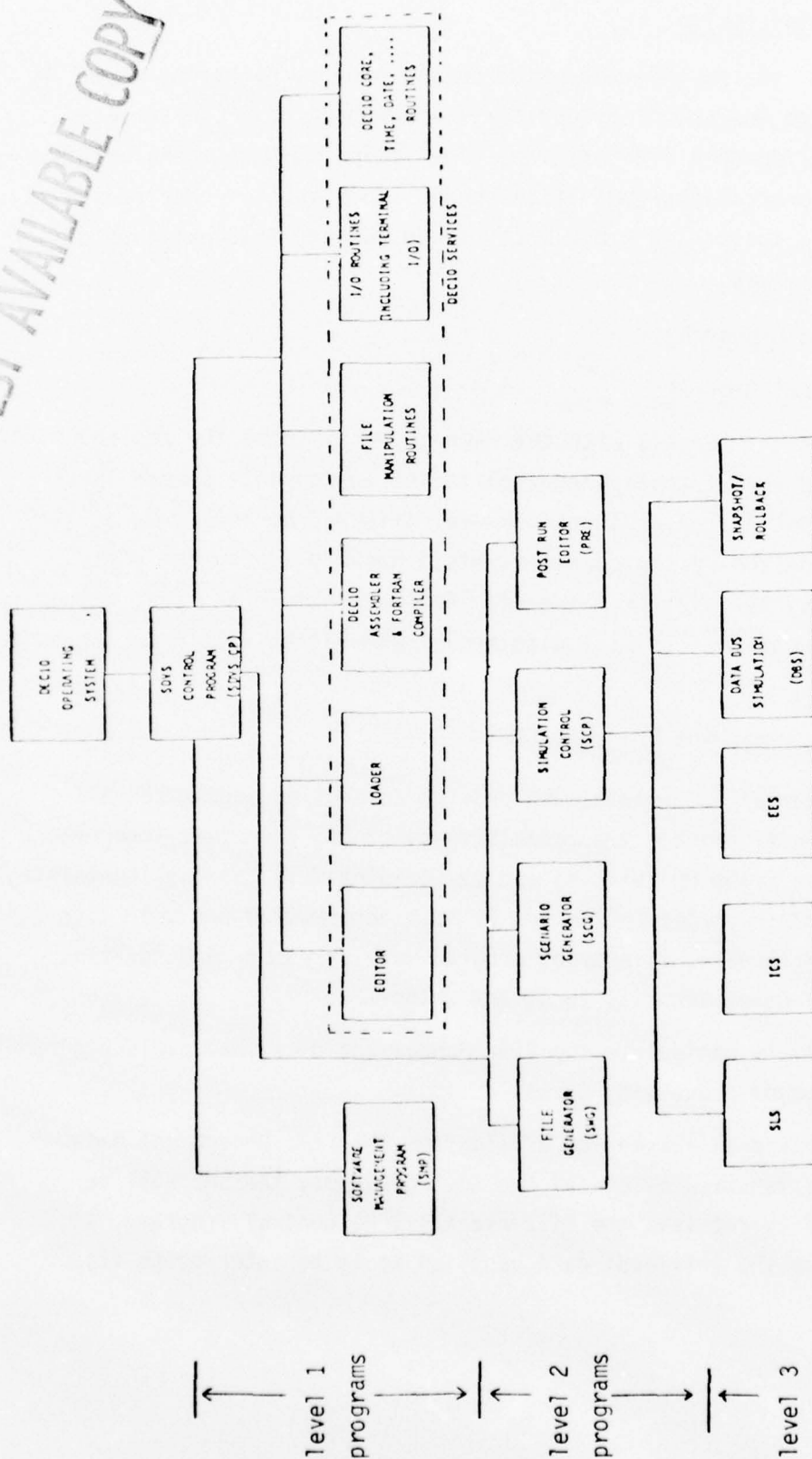
Software Management Program (SMP)

This program will maintain, and provide controlled access to, all SDVS files. It will provide the capability to store, retrieve, interrogate and protect these files by building and maintaining file catalogs containing information about the files. Examples of such information are user file name, internal file name, file type, program version number and revision number, creation date, security lock, and author.

To effectively manipulate the File Management data base, this program performs three major processing tasks, as follows:

- o File Retrieval - Whenever a file from the File Management data base is required by one of the SDVS programs, the SMP will be invoked to retrieve the file via the SDVS Control Program. It will use the retrieval data supplied to it to interrogate its

BEST AVAILABLE COPY



Hierarchy of SDVS Software
Figure 10

file catalogs and locate the internal name of the requested file. This internal name will be passed to the SDVS CP, which will use it to actually locate and retrieve the file from secondary storage.

- o File Disposition - When a new file is created by an SDVS program or a user, it must be placed in the File Management data base under control of the SMP. The program which generates the file (e.g., Scenario Generator, Simulation Control, etc) will issue a file creation request. This request will contain the qualified name under which the file is to be controlled, the file type, and the name of the person responsible for the file. The request will be passed to the SMP where the data contained in the request will be used to create a new catalog entry for the file. The cataloged file will then be written to secondary storage by the SDVS CP.
- o File Protection - The third major function of the SMP will be to provide security locks on the files such that a given file can only be accessed by someone possessing the matching security key. Each programmer will have security protection over his files until his software has been completely tested and is ready to become part of the official system. However, the project manager will always have access to all files in the data base.

File Generator

This program processes file manipulation commands input by the user. Section 4, (2) discusses these conversational commands.

The File Generator will not actually perform these functions itself (e.g., COMPILE, EDIT, ACCESS, etc.), but rather will pass the user's requests to the SDVS Control Program which will in turn pass them to the DEC10 monitor and/or the Software Management Program.

Scenario Generator (SCG)

This program is divided into two program translators, a Simulation Control Language (SCL) translator and a Data Processing Language (DPL) translator. The SCL defines the user's simulation scenario to be executed; the DPL defines the data processing to be performed on simulation data. The SCG is executed by a conversational command to translate either a SCL or DPL file. The Scenario Generator will retrieve the desired file from the SMP catalogs, translate the source code, and catalog the translated test case in the SMP catalogs.

Simulation Control - Snapshot/Rollback

These programs are used to sequence a simulation scenario defined by a translated SCL program. They will initialize the necessary simulators (ICS, SLS, data bus, environment) for execution, load the users mission software to be tested, perform rollbacks, and execute a simulation by invoking the various simulators.

Simulators (ICS, SLS, Data Bus, EES)

The programs simulate the DAIS hardware. Each of these programs simulate appropriate events (e.g., execution of an instruction, a bus transmission) upon direction of the Simulation Control Program.

(2) Evaluation of Using Hierarchical Design

The top-down design process involved designing the programs at the highest level in the hierarchy first. Based on this design, the interfaces to the next lower level of programs would be defined. These interfaces would be used to drive the design of the next level of programs in the hierarchy. This procedure would proceed down to the lowest level. Data and control paths between program elements on the same level must pass through a higher program element which is common to both. No lateral communications are allowed. One important aspect of top-down design is that interfaces are based upon the requirements of the higher level program and that the lower level programs are designed to fit these interfaces.

The SDVS Control and Software Management Programs, though shown at the top of the hierarchy, are actually a collection of utility routines that are used by all the second level programs. The exception is the SDVS Control Program function that interacts with the user and selects the mode of operation. The hierarchy diagram, as drawn, represents a functional organization and not the exact control interfaces between the various programs. To have implemented this structure in a strictly top-down manner where control and data interfaces reflect the hierarchy diagram would be impractical.

Based on this hierarchy diagram, the control and data interfaces between the SDVS programs first developed during the design of the SDVS CP provided a good first cut. Once the design of the second level programs began these interfaces had to be modified to accommodate newly defined functions. The interfaces had to again be modified when the third level programs were being designed in detail. So, we found that design of the program hierarchy is an interactive process going in a top-down, bottom-up cycle. Avoidance of this cycle is practically impossible. By spending the time to make the original interfaces more complete and more general, however, the necessity for changes is reduced and much less modification of code must be performed. Only after several top-down, bottom-up cycles can a final "top-down" hierarchy diagram be drawn.

b. Rehostibility

One of the basic SDVS design goals was that SDVS should be structured to facilitate rehosting on another system. This was done by isolating all host processor dependent software in the SDVS Control Program (e.g., I/O handling, core control, text editor and compiler interfaces, etc.). The Control Program would therefore be the only program requiring assembly language code; all other programs would be coded entirely in JOVIAL to facilitate rehosting.

This approach worked out extremely well. The only additional program requiring assembly language code was in the SLS which involved interfacing with the JOVIAL object code. The SDVS Control Program, originally estimated to be coded 90% in assembly language, required less than 10% of the coding to be in assembly language. The following paragraphs address some of the major tasks of rehosting the SDVS.

Data Structures

Specified Tables were used throughout all SDVS programs. In fact these tables represent a machine dependency since actual bit locations *within a word are referenced*. In rehosting SDVS to a new computer many of these tables would have to be changed. This would involve extensive, but straightforward modification of data declarations only. The actual code which processes these tables would remain unchanged.

SDVS Control Program

As previously mentioned, all machine dependent code was incorporated into the design of this program. A major concern in rehosting will be the investigation of available techniques for interacting with a new host monitor/operating system. There are major Control Program techniques for interfacing with the DEC10 monitor which are probably realized differently on other processors. These include:

- o Use of pseudo teletypes
- o Paging swapping considerations
- o File directory operations
- o Account-code schemes
- o Tape Mounting Utilities
- o Interfacing with system processors (editors, compilers)
- o I/O schemes
- o Special compiler and text editor interfaces

Software Management Program (SMP)

The SDVS SMP is implemented with the DEC Data Base Management System (DBMS) software developed in COBOL. Rehosting this program requires an Information Management System package that would provide for the same

interfaces as DEC's DBMS. Questions that must be answered include:

- o Is it a "standalone" system or callable from other programs (e.g., SDVS)?
- o How similar are its cataloging facilities to those of DBMS?
- o How similar are its operators and calling sequences to those of DBMS?
- o Is it COBOL hosted?

c. Control and Data Interfaces

As mentioned above, the control interfaces in SDVS were completely defined by the hierarchy structure. A program could be called only by a program of a higher level and in turn it can only call programs of the next lower level. Programs of the same level have no direct interface. A controlling program of the next higher level must pass any necessary information exchange between two programs of the same level. After following this approach completely at the start of SDVS, the need for a set of common utility routines (I/O, parser, conversion, etc.) available to all programs soon became obvious. However this was the only major diversion from the hierarchy defined control structure.

All DEC10 services (Action Processors) in SDVS are requested via Control Points. A Control Point is a data area with enough fields to describe any available service. For example, one field contains a code for the type of service, another contains the address of a buffer area (used on I/O requests only), etc. When a program requests a particular service, it fills only those fields in the Control Point pertaining to that service. The use of Control Points provides one data area for the passing of information. In SDVS one Control Point was used for passing information to the action processor and another for returning results to the calling program. Thus all data input to the Action Processor remained intact even after the action had been serviced. This has proved very beneficial in debugging the logic of programs requesting DEC10 services and in decreasing the complexity of data manipulation in the Control Program itself. Insofar as meaningful names are assigned to the various Control Point fields, they are very self-documenting and make it very easy for a programmer to incorporate calls on new Action Processors. Unfortunately, in SDVS, as the number of Action Processors expanded, several Control Point fields took on duplicate unrelated meanings in calls on different Action Processors. This ambiguity made the Control Points harder to use and introduced greater possibility of errors.

Each SDVS Action Processor was described on an Interface Diagram which listed the Control Point fields needed by the Action Processor on input and described the meanings of each possible completion code or output generated

from the service request. These diagrams provided a clear description of the interfaces required for a given service request and a central definition point to facilitate updates and modifications of these interfaces (see Figure 5-2 for an example).

In addition to Control Point interfaces most SDVS programs, especially in the simulation system, need to share several data items with their controlling program. These data interfaces were controlled by requiring all data common to two programs to reside in a separate compool. One copy of this compool was kept so that changes to the interface were immediately reflected in both programs. Figure 12 shows the compool interface between the SCP and EES programs.

INTERFACES

FROM _____ to SDVS CP CONTROL POINT ACTION PROCESSORS

CONTROL POINT ACTION	INFORMATION SUPPLIED TO SDVS CP IN CONTROL POINT "A"	REASON FOR ISSUING CONTROL POINT "A" REQUEST	INFORMATION TO BE RETURNED IN CONTROL POINT "B"
1 ACCESS FOR EDIT-DISPLAY	Field 3: File name 4: Extension (sub-type) 5: Type 6: Version number 7: Revision number 8: (-1 means "latest") 9: (-1 means "latest")	Make a cataloged file "local" in the environment of the current SDVS job in this SDVS mode of operation, available for future editing or displaying (or copying or compiling) in this session.	Field 13: Completion Code Range: 0: File accessed OK; see field 12 for level of write authority. 2: File already local. 3: Unknown revision number 4: Unknown version. 5: Unknown file name. 6: Unknown Type Extension. 7: Insufficient Control-Point A information. 8: File previously deleted. 9: Version previously deleted. 14: File already local for sequential I/O. 19: Insufficient authority to read this file. 22: Can't access this subtype for edit-display. 26: DECIO file error 32: Too many local files. 98: SMP system error. 2: File size. 6: Last version number of this name. 7: Last revision of requested version. 12: Write-Authority Code: 1: NO writing on file name, read only. 2: NO writing on version. 3: Full write authority.

```

JOVIAL V.011276 3/23/76 11:58 MODULE:SCPEE3.COM PAGE 1
SCPEE3,SCPEE3=SCPEE3.COM,SCPEE3/A/M/NOI

0. 03300 " TITLE: SCP EES COMMON AREA "
1. 03400 " " PHASE III "
1. 03500 " " JAN 10, 76 "
1. 03600 " " THIS COMPOOL CONTAINS THE VARIABLES WHICH
1. 03700 " " ARE REFERENCED BY SCP AND EES. "
1. 03800
1. 03900 COMPOOL SCPEES;
2. 04000 BEGIN "SCPEES"
2. 04100
3. 07900 !CUPY 'DEFINS.COP(3202,6621)';
38. 04300 !EJECT ;

```

Figure 12

SCPEES Compool

3/23/76 11:58 MODULE:SCPEES3.COM *****

JOVIAL V.011276

```

38. 04400
39. 04500
40. 04600
41. 04700
42. 04800
43. 04900
44. 05000
45. 05100
46. 05200
47. 05300
48. 05400
49. 05500
50. 05600
51. 05700
52. 05800
53. 05900
54. 06000
55. 06100
56. 06200
57. 06300
58. 06400
59. 06500
60. 06600
61. 06700
62. 06800
63. 06900
64. 07000
65. 07100
66. 07200
67. 07300
68. 07400
69. 07500
70. 07600
71. 07700
72. 07800
73. 07900
74. 08000
75. 08100

```

" *****
 S C P E E S C O M M U N *****

 ITEM RESEVT INTEGER ; "EVENT TYPE 1= PERIODIC
 4= DATA TO EES
 5= DATA FROM EES"

 ITEM RESEXT INTEGER ; "CURRENT SIMULATION TIME"

 ITEM RESEXT FLOAT ; "RETURN CODE 0= DO NOTHING
 1= READ FLIGHT PROFILE TAPE"

 ITEM RESEXT INTEGER ; "RESEXT LOGICAL PRESET FALSE"

 ITEM RESEXT INTEGER ; "INDEX FOR RESEXT TABLE"

 ITEM RESEXT INTEGER ; "NUMBER OF KEYS TO BE USED FROM RESEXT"

 ITEM RESEXT INTEGER ; "23"

 TABLE RESEXT (0:RESEXT) 1 ;
 BEGIN
 ITEM RESEXTTYPE INDEX (0,0) ;
 ITEM RESEXT INDEX (18,0) ;
 ITEM RESEXTC INDEX (10,0) ;
 ITEM RESEXTVAL " "
 END

 TABLE RESEXT (1:50) 2 ;
 BEGIN
 ITEM RESEXTMODEL CHART 6 (0,0) PRESET
 'FLR', 'RADALT', 'ADS',
 'ILS', '4('', 'MPSM', 'SCUSM',
 'VSDSM', 'PCPSM', 'AFSAS', 'MPDIP',
 'LCP2S', 'LCPUS', 'IMFKR', '2('',
 'IMFKS', 'MFSBS', 'MPD2P',
 '9('', 'PROPEL', 'GROUND', 'EARTH',
 'ATMO', 'AFI', 'HELGUM',
 'DUPLER', 'HELGUM' ;
 ITEM RESEXTINDEX INDEX (18,1) PRESET 0 ;
 END
 *****SCPEES *****
 END

Figure 12

(SCPEES Compool) (continued)

d. Software Development Standards

This section describes the SDVS software development standards for development of the SDVS. The standards were predicated on the use of structured programming techniques and a high order language that supports these techniques.

(1) Structured Programming

Structured programming is based on the mathematically proven Structure Theorem which states that any proper program (a program with one entry and one exit) is equivalent to a program that contains as logic structures only:

- o sequences of two or more operations
- o conditional branch to one of two operations and return (IF a THEN b ELSE c)
- o repetition of an operation while a condition is true (DO WHILE)

Each of the three structures itself represents a proper program. Using combinations of these basic structures, any program can be built. Even though any program can be built with these basic structures, it is desirable to include additional structures which provide more readable and self documenting programs, more efficient programming, and programmer conveniences

As a tool to aid in designing the SDVS software using a set of programming constructs, structured flowcharts were used. These flowcharts are different than conventional flowcharts in that each programming structure has a unique flowchart representation. The programmer builds his program by combining the basic structures in constructing a program. The flowcharts simplify the arrangement of program logic to a process like that used in engineering where logic circuits are constructed from a basic set of logic functions.

(a) SDVS Structured Programming Constructs

The structured programming techniques employed included the definition of six basic structures (IF, WHILE, FOR, IF-THEN-ELSE, IF-ANY-1, and CASE) and standards defining the J73/I language constructs to be used for each structure. Two additional structures were added to accommodate situations

where the six basic structures would be inefficient and would lead to software that would contain unnecessary logic and therefore be harder to understand, verify, and maintain. These structures include an ESCAPE construct allowing an error exit from a routine without having to use a number of extra IF-THEN-ELSE constructs testing for the error conditions, and an EXPANSION construct which provides the programmer a strictly controlled GO-TO capability to execute logical functions without having to use subroutine calls.

Figure 13 illustrates the flow chart standards and JOVIAL J73/I source code for each control structure.

(b) Evaluation of Structured Programming

Use of structured programming techniques proved to be an extremely beneficial in the development of the SDVS software. By carefully defining the basic structures to be used in the context of the J73/I language, structured programming facilitated the on-schedule development, debugging, testing, and delivery of a highly reliable system.

The use of structured flowcharts is a natural way to represent program logic in a language independent notation. Originally SDVS was developed in JOVIAL J70 prior to the availability of J73/I. During the J70 development, the structured flowchart constructs were represented by J70 syntax instead of J73 as shown in Figure 13. The conversion from J70 to J73 was therefore a "cookbook" operation, i.e., substituting one set of syntactical statements for another. It would even be possible to write FORTRAN statements corresponding to each basic structure, although the structures are more suited to a JOVIAL like language. In other words, the flowcharts allow the programmer a way of representing the abstractions of a problem independent of the language and computer to be implemented.

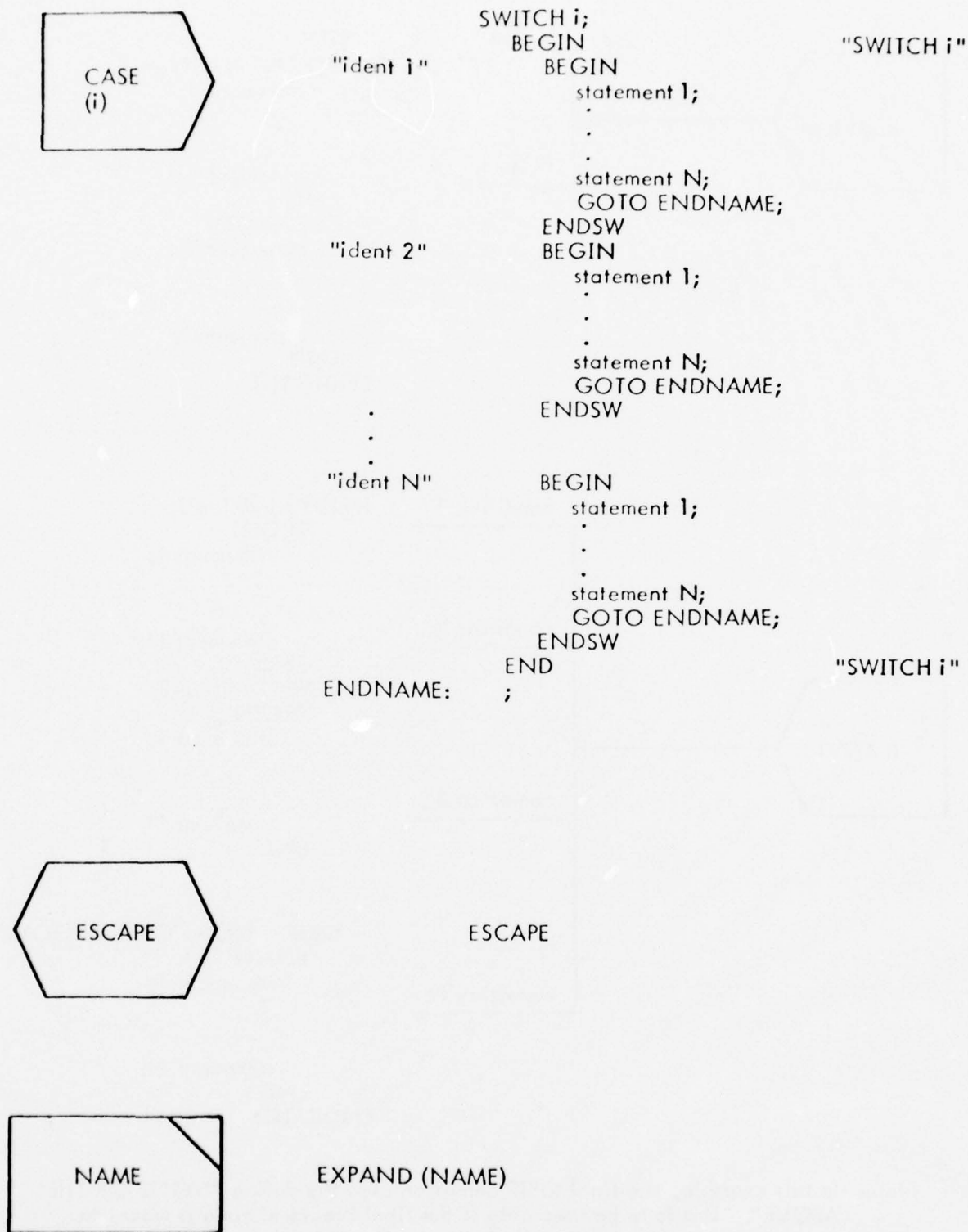
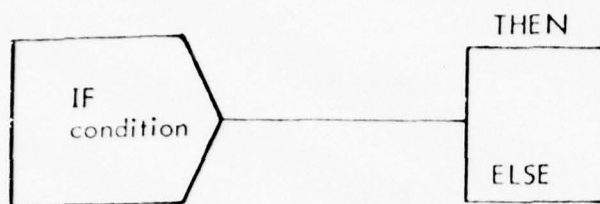


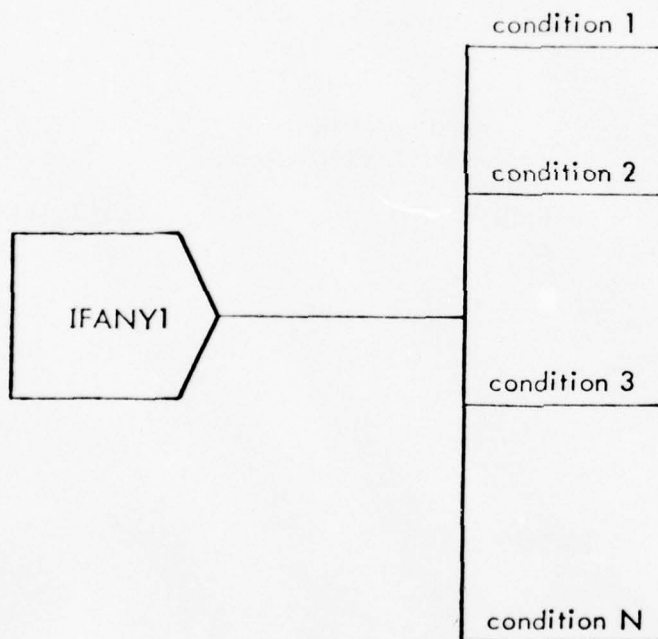
Figure 13 SDVS/J73 Control Structures



```

IFEITH
  "THEN" BEGIN
    statement 1;
    .
    .
    statement N;
  END
  ELSE BEGIN
    statement 1;
    .
    .
    statement N;
  END
ENDIFEITH

```



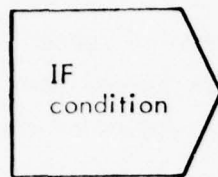
```

IFEITH condition 1;
  BEGIN
    statement 1;
    .
    .
    statement N;
  END
  ORIF condition 2;
  BEGIN
    statement 1;
    .
    .
    statement N;
  END
  .
  .
  ORIF NONE'OF'THE'ABOVE;
  BEGIN
    statement 1;
    .
    .
    statement N;
  END
ENDIFEITH

```

Note: In this example, the final ORIF condition used the define "NONE'OF'THE'ABOVE". This is to be used only if the final blocks of code is meant to process any situation not handled above. If nothing is to be done, if all conditions fail, the ORIF NONE'OF'THE'ABOVE block should be omitted.

Figure 13 SDVS/J73 Control Structures (Continued)



```

IF condition;
  BEGIN
    statement 1;
    .
    .
    statement N;
  END

```



```

WHILE condition;
  BEGIN
    statement 1;
    .
    .
    statement N;
  END

```



```

FOR i : j BY k WHILE i LQ m;
  BEGIN
    statement 1;
    .
    .
    statement N;
  END

```

where:

i is a variable name.
j, *k*, *m* are arithmetic expressions

Figure 13 SDVS/J73 Control Structures (Concluded)

Additional advantages of using structured programming include:

- o The use of structured flow charts was judged to be the most useful tool, with respect to structured programming concepts, for designing software. All of the programmers agreed that it provided the means to clearly represent the structure of their programs on paper. Abstract ideas could be easily written down using these flowcharting technique. It also proved very easy for one programmer to study another's flowcharts and readily understand them. All of the programmers agreed that they prefer this technique to those of the past, and would continue to use it on future projects.
- o Readable source code that can easily be cross-referenced with flowcharts. The code itself was very self documenting.
- o Facilitates debugging in that program structure is more apparent and that there is only one entry and exit point.
- o As a by-product, it somewhat standardized programming style across the project. This made it easy for project members to work on programs other than their own and facilitated open communication among programmers about the design and debugging of one's code. This open communication concept is sometimes referred to as "Egoless Programming" in the literature.
- o The use of the structured flowcharts seemed to automatically enforce software modularity. The process of breaking a problem into successively simpler parts and piecing these parts together naturally fell out of using these flowcharting techniques.

By and large, SDVS benefitted from the use of structured programming. However, structured programming is not a cure-all for poor programming techniques - in fact, structured programs may lead to worse programs than non-structured ones because it is more difficult to do things which probably shouldn't be done!

(2) Program Modularity

The goal of program modularity is to separate complex operations into several well defined functions each of which in itself can be declared to be "obviously correct" by inspection. To a large degree this goal was achieved in SDVS. Modularity requires a more disciplined design approach in that more procedure calls are required and each procedure must include several overhead statements defining local data items, etc. This tediousness and human nature are the greatest obstacles to overcome in producing modular code and the reason that no large system is ever modularized as completely as it could be.

Modularization does introduce some overhead to a system, but this overhead is greatly reduced if data can be accessed by the procedures without having to pass parameters. This is achieved in SDVS by nesting of procedures and by defining data in compools global to all procedures within a given program. The second drawback of modularity is in error checking. If an error can occur several procedure calls deep, each calling procedure must have a check and separate branch for the occurrence of that error. This situation was often avoided in SDVS by using the Expansion Block and Escape constructs. This provided the functional separation of modularity while allowing errors to escape to a much higher logical level.

The true value of modularity is the ease of isolating and correcting logic errors. The short time in which SDVS was actually debugged evidences this benefit. Furthermore, the task of enhancing or modifying code was found to be much easier in the more modularized areas of SDVS.

(3) Use of a High Level Language

Without the use of the JOVIAL language, the implementation of SDVS software would undoubtedly have taken many extra months. The power of using a high level language to develop a system such as SDVS is even more evident when one considers that the SDVS development was burdened with conversion from a J70 dialect to the J73 dialect had to use a newly developed, first time used compiler, and still the SDVS development team felt that the use of JOVIAL was essential to meeting the delivery date.

One reason JOVIAL was so helpful was the ease with which it can be learned. To master the DEC MACRO assembly language would probably have taken much longer than mastering JOVIAL. Just the reduction in lines of code between a JOVIAL and a MACRO implementation of SDVS saved countless hours of coding and debugging time.

The JOVIAL control structures (WHILE, IF-ELSE IF, etc.) lent themselves very well to implementation of our structured flowcharts. By expanding these structures with DEFINE's, TRW was able to develop code templates corresponding to each of the logical constructs allowed by the structured programming standards. This made coding from flowcharts extremely straightforward.

The data structures allowed by JOVIAL were also beneficial to SDVS development. Com pools were used to control program interfaces. Processing of cumbersome data was eased by multiple word items (e.g., long character strings) and multiple item table entries. Specified tables allowed efficient use of partial word items such as links, easy implementation of variable length table entries, and overlaying of data items. These capabilities were used to define and manipulate complex linked lists and to develop templates for externally defined data descriptions (i.e. Link Item Types, UUO data blocks, etc.). Processing of data input to buffers was greatly facilitated by making these templates based. Specified tables and the BIT and BYTE functions allowed JOVIAL code to perform tasks often done in assembly code. In fact JOVIAL was so powerful that only 10% instead of 90% of the SDVS Control Program had to be written in MACRO code.

Undeniably the use of JOVIAL did introduce inefficiencies and make the SDVS system slower. However TRW feels that by careful timing analysis these bottlenecks can be found and corrected (by recoding key routines in MACRO, if

necessary). Moreover these bottlenecks could not have been identified during the design phase and many hours would have been wasted developing "efficient" and most likely tricky MACRO code in routines where optimization was not really needed.

The unique JOVIAL J73 capabilities most useful in SDVS implementation were: the DEFINE strings, specified and unspecified TABLES, BIT and BYTE manipulations, and COMPOOLS.

4. Programming Standards

In addition to using structured programming techniques, there were other very important standards that facilitated the development process. The following paragraphs highlight some of these.

Use of DEFINE Strings

Perhaps the most valuable standard was the use of the DEFINE capability in JOVIAL which allows the programmer the capability to substitute one character string for another. The DEFINE feature is very similar to the MACRO feature found in many assemblers, i.e., it allows the programmer to define new language constructs for his application.

The DEFINE capability was used to greatly simplify the conversion of SDVS from JOVIAL J70 to J73/I. Table 1 outlines a list of SDVS DEFINES which were contained in a file used by all SDVS programs. Note, that the defined names are much more natural, self documenting descriptions than either the J70 or J73/I syntax. In converting from J70 to J73/I this DEFINE file was modified to reflect the change in language syntax. Note, also, that if SDVS is rehosted on a machine with a 32 bit word size, only the DEFINES specifying word size must be changed instead of every item declaration appearing in the source code.

The DEFINE feature was also used to create the ESCAPE and EXPAND structured programming constructs shown in Figure 13. Note, from Table 1 that the use of a DEFINE provides a controlled use of the GOTO statement for the EXPAND macro.

Interprogram Data

A fundamental SDVS programming standard was that there was to be only one COMPOOL between any two programs for the purposes of sharing data. The hierarchy shown in Figure 10 was the basis for defining the sharing of inter-program data. Organizing shared data this way proved beneficial during testing in that there were only very minor problems found with respect to SDVS shared data interfaces. Usually, testing uncovers a plethora of data interface problems.

	<u>J73</u>	<u>J70 DIFFERENCE</u>
DEFINE FLOAT	"F";	Same
DEFINE INTEGER	"S 35";	"I 36 S"
DEFINE FULLWORD	"U 36";	"I 36 S"
DEFINE INDEX	"U 18";	"I 18 U"
DEFINE ADDRESS	"U 18";	"I 18 U"
DEFINE FLAG	"U 18";	"I 18 U"
DEFINE CHAR7	"C";	Same
DEFINE OCTAL(A)	"3B!A";	"A!"
DEFINE PRESET	"=";	"p"
DEFINE ENDSW	"END";	Same
DEFINE ENDIFEITH	" ";	Not used
DEFINE IFEITH	"IF";	Not used
DEFINE ORIF	"ELSE IF";	Not used
DEFINE NONE 'OF 'THE 'ABOVE	"1";	Not used
DEFINE EQ	"=";	Not used
DEFINE NQ	"<>";	Not used
DEFINE LS	"<";	Not used
DEFINE LQ	"<=";	Not used
DEFINE GR	">";	Not used
DEFINE GQ	">=";	Not used
DEFINE CALL	" ";	Not used
DEFINE ESCAPE	"RETURN";	Not used

```

DEFINE EXPAND(E"XPANSION NAME") " ""INVOKED AN EXPANSION""
  BEGIN ""SO THAT EXPAND WILL BE A SINGLE STATEMENT""
    GOTO !E; SS!E; ;
    END ""END THE STATEMENT""
""END THE DEFINE STRING --->"" ";

DEFINE BEGIN'EXPANSION(F"XPANSION NAME") " ""EXPANSION DEFINITION""
  RETURN; ""SO A RETURN IS NOT NEEDED FOLLOWING MAIN PROC""
  BEGIN !F;
""END THE DEFINE STRING --->"" ";

DEFINE END'EXPANSION(F"XPANSION NAME") " ""END EXPANSION DEFINITION""
  $$$!E: ""DEF A LABEL TO PREVENT ERRORS""
  GOTO $$$!E; ""RETURN TO EXPANSION INVOKATION""
  END
""END THE DEFINE STRING --->"" ";

```

Table 18 Example of SDVS DEFINE Strings

Coding Standards

As is common among all programming projects, a number of standards were defined dealing with the following topics:

- o Naming conventions
- o Data specifications
- o Calling sequences
- o Program annotation

Figure 13 is an example of a SDVS module which illustrates the use of structured programming and good program annotation. This procedure is part of the Simulation Control Language translator and is used to generate commands for evaluation of arithmetic expressions during a SDVS simulation. This routine is called after the arithmetic expression has been parsed and the appropriate elements stored in operator and operand stacks. It "pops" the stacks and based on the operator generates the necessary commands. The following points should be made concerning this example:

- o The indentation of the program structures (IF-THEN-ELSE and CASE) facilitate the readability of the code. The code itself is effectively self documenting.
- o The use of meaningful variable names and comments defining them.
- o The procedure is short (less than one page) and therefore it is easy to grasp its overall function.
- o Expansion block structures are used for each of the operator types (assignment, unary, binary). If error conditions are found in an expansion block, an ESCAPE structure can be used to terminate processing of this routine instead of redundant code to check for errors if subroutines were used.
- o All sequence structures are delimited by BEGIN-END blocks which define the function performed.

	<u>J73</u>	<u>J70 DIFFERENCE</u>
DEFINE FLOAT	"F";	Same
DEFINE INTEGER	"S 35";	"I 36 S"
DEFINE FULLWORD	"U 36";	"I 36 S"
DEFINE INDEX	"U 18";	"I 18 U"
DEFINE ADDRESS	"U 18";	"I 18 U"
DEFINE FLAG	"U 18";	"I 18 U"
DEFINE CHAR7	"C";	Same
DEFINE OCTAL(A)	"3B!A";	"A!"
DEFINE PRESET	"=";	"P"
DEFINE ENDSW	"END";	Same
DEFINE ENDIFEITH	" ";	Not used
DEFINE IFEITH	"IF";	Not used
DEFINE ORIF	"ELSE IF";	Not used
DEFINE NONE 'OF 'THE 'ABOVE	"1";	Not used
DEFINE EQ	"=";	Not used
DEFINE NQ	"<>";	Not used
DEFINE LS	"<";	Not used
DEFINE LQ	"<=";	Not used
DEFINE GR	">";	Not used
DEFINE GQ	">=";	Not used
DEFINE CALL	" ";	Not used
DEFINE ESCAPE	"RETURN";	Not used

```

DEFINE EXPAND(E"XPANSION NAME") " " "INVOKE AN EXPANSION"
  BEGIN "SO THAT EXPAND WILL BE A SINGLE STATEMENT"
  GOTO !E; $$$E: ;
  END "END THE STATEMENT"
  ""END THE DEFINE STRING --->"" ";

DEFINE BEGIN'EXPANSION(F"XPANSION NAME") " " "EXPANSION DEFINITION"
  RETURN; "SO A RETURN IS NOT NEEDED FOLLOWING MAIN PROC"
  BEGIN !F;
  ""END THE DEFINE STRING --->"" ";

DEFINE END'EXPANSION(F"XPANSION NAME") " " "END EXPANSION DEFINITION"
  $$$E: "DEF A LABEL TO PREVENT ERRORS"
  GOTO $$$E; "RETURN TO EXPANSION INVOKATION"
  END
  ""END THE DEFINE STRING --->"" ";

```

Table 18 Example of SDVS DEFINE Strings

```

764. 09780      "***** POPSTACK POPS AN OPERATOR AND ONE OR TWO *****"
764. 09790      "***** OPERANDS FROM THE OPERATOR AND OPERAND *****"
764. 09794      "***** STACKS RESPECTIVELY, AND OUTPUTS THEM *****"
764. 09798      "***** TO THE COMMAND BUFFER. *****"
764. 09802
764. 09806
764. 09810      DEF PROC POPSTACK;
765. 09814      BEGIN
765. 09818      ITEM NOWDS1      "POP OPERATOR AND OPERAND STACKS"
766. 09822      ITEM NOWDS2      INDEX;      "NUMBER OF WORDS IN OPERAND 1"
767. 09826      INDEX;      "NUMBER OF WORDS IN OPERAND 2"
767. 09830      IF EITH JJ < 1;
768. 09834      "THEN" BEGIN      "INDEX UNDERFLOW"
768. 09838      ERRFL = 36;
769. 09842      END
770. 09846      ELSE BEGIN      "INDEX UNDERFLOW"
771. 09850      IF EITH OP(JJ) = 17;      "STACK NOT EMPTY"
772. 09854      BEGIN      "ASSIGNMENT OPERATOR"
773. 09858      EXPAND(ASSIGNMENT'OPERATOR);
777. 09862      END      "ASSIGNMENT OPERATOR"
778. 09866      OR IF OP(JJ) = 1 OR OP(JJ) = 2 OR OP(JJ) = 14;
780. 09870      BEGIN      "UNARY OPERATORS"
781. 09874      EXPAND(UNARY'OPERATORS);
785. 09878      END      "UNARY OPERATORS"
786. 09882      OR IF 1;
788. 09886      BEGIN      "BINARY OPERATORS"
789. 09890      EXPAND(BINARY'OPERATORS);
793. 09894      END      "BINARY OPERATORS"
794. 09898      END IF EITH
794. 09902      END
795. 09906      END IF EITH
795. 09910      RETURN;
796. 09914
796. 09918      !EJECT;

```

e. Software Testing

Detailed testing of software is crucial to the delivery of a useful system. Yet, testing is generally the first area to be cut back when deadlines approach too rapidly. To insure against this, TRM planned for three levels of test planning with strong configuration control enforced after completion of the first testing phase.

(1) Test Planning

Testing was performed on SDVS software in three phases:

- o separate programs were tested individually and then interfaced with their controlling and subordinate programs
- o after a reasonable working system was obtained from interface testing, Preliminary Qualification Tests (PQT) were developed to test functional requirements
- o a large all-inclusive Formal Qualification Test (FQT) was then run to demonstrate SDVS's ability to run in a realistic environment.

These three levels provided a comprehensive and disciplined testing approach.

Individual Program Testing

As a program neared completion, each programmer would test his own program. Because of the clearly defined program interfaces in SDVS, it was often easy for the individual programmer to provide dummy input to his program. This was done either manually or in the case of complex structures, via separately developed test programs. This allowed testing of lower level programs prior to completion of their controlling program. In addition, when two programs were being interfaced and the lower level program had already undergone extensive testing with dummy inputs it became much easier to pinpoint generation of improper data by the controlling program. This early bottom up testing often exercised many of the error handling routines while correcting gross logical errors.

PQT Testing

Once the programmers were satisfied that a sufficiently stable system had been put together, a series of PQT tests were developed. In SDVS this turned out to be a two step process. SDVS really contains two systems, the Interactive System and the Simulation System. The high level Interactive System in which the user created files, translated test cases etc., had to be working before the lower level simulations could be tested. Thus, PQT was a top-down process in which the Interactive System was tested long before the Simulation System.

The process of developing the PQT tests was also top-down in nature. The first step was to divide SDVS into 8 separate configurations (see figure 15). Then a detailed list of functional requirements was drawn up for each program in each configuration. For a given configuration these lists were combined into a single list describing all functions to be tested for that configuration (see figure 16). Finally, a set of PQT tests incorporating each of these test objectives was developed (see figure 17). As each PQT test was successfully completed the corresponding test objective was checked off the list. This process allowed for detailed testing with a minimum amount of overlap, yet the test results for each test were clearly defined and easily documented.

FQT Testing

After PQT testing had shown that each functional requirement was working properly, the FQT test was used to show SDVS ability to operate in a realistic environment. Generally the FQT test was a large comprehensive program developed at least in part by AFAL. Thus, it exercised SDVS in situations not necessarily covered by the smaller, more specialized PQT tests. Furthermore, the fact that the FQT tests were developed by non-TRI/ personnel introduced new programming styles in the mission software or new approaches to entering and compiling files which uncovered some errors in both the Interactive and the Simulation System.

PHASE III SDVS TESTING CONFIGURATION

NO.	CONFIGURATION DESCRIPTION	SDVS PROGRAMS												
		CP	SMP	FG	SCG PRE	SCG TCD	SCP	HBCL	RB	SLS	PRE	DBS	EES	ICS
1	Interactive Conversational SDVS	X	X	X										
2	Post Run Edit Directives File Translation	X	X		X									
3	Test Case Directives File Translation	X	X			X								
4	Standalone SLS Simulation	X					X	X	X	X				
5	Post Run Editor Execution	X	X							X				
6	Complete SLS Simulation Multi SLS, DBS, EES	X					X	X	X	X	X	X	X	
7	Standalone ICS Simulation	X					X	X	X					X
8	Complete ICS Simulation Multi ICS, DBS, EES	X					X	X	X		X	X	X	X

Figure 15

BEST AVAILABLE COPY

SDVS PHASE III CONFIGURATION TESTING

CONFIGURATION NO.	CONFIGURATION DESCRIPTION	Page 1
4	STANDALONE SLS SIMULATION	
SDVS PROGRAMS TESTED.		
SDVSCP, HBCL, RB, SCP, SLS, EES		
EXTERNAL SOFTWARE REQUIRED (BY)		TEST SCHEDULE
Mission Software files		3/29 to 4/2
PRIMARY FUNCTIONS TO BE TESTED.		
- SDVSCP-SLS coordination of error-trap handling. - SCP End-of-JOB execution for fatal errors/traps. - SCP End-of-JOB execution when simulation times out. - Variable trace of MSW variable set from SCL and MSW. - Repeated execution of MSW modules (EFS). - Repeated overlapping execution of MSW modules (EFS error) - EFS trace output. - User Referencable SDVS variables (SCP, HBC) in conditions. - Selective Variable Trace (only from SIMT A to SIMT B) - Evaluating conditions - when a variable gets changed, verify all of its conditions get checked. - Variable Trace/Condition evaluation on all elements of an array. - As a result of evaluating a condition, change another monitored variable and verify that it also is checked. - Move a variable from its location to a ROT block. - Set value of variable from SCL (all data types, multiple words) (setvalue and initialize) - Multiple monitored variables (MSW) changed in one SLS statement. - SLS trace(statement trace with and without exec call). - SLS Trace (Transfer trace).		

Figure 16

SDVS PHASE III CONFIGURATION TESTING

CONFIGURATION NO.	CONFIGURATION DESCRIPTION	Page 2
4	STANDALONE SLS SIMULATION	
SDVS PROGRAMS TESTED.		
SDVSCP, SCP, HBCL, RB, SLS, EES		
EXTERNAL SOFTWARE REQUIRED (BY)		TEST SCHEDULE
Mission Software files		3/29 to 4/2
PRIMARY FUNCTIONS TO BE TESTED.		
18. Access PALEFAC object file from PALEFAC PPN. Verify file gets loaded correctly.		
19. Activate ROT and EES blocks from 2 performs, deallocate one, see that correct one is deallocated. (One even, one odd).		
20. MET trace output correctly		
21. Test TRACE, BLK set correctly at ACTIVATE and during simulation.		
22. One simple rollback with new test case, verify snapshot time selected.		
23. Verify that ROT is copied correctly thru rollback point.		
24. Add Variables (MSW)		
25. Verify SNAPSHOT's taken as requested.		
26. Verify execution of standalone mode with EES. Periodic execution, EES TRACE data, Fly Mission profile. (interface through SCL statements)		
27. Add new ROT block in Rollback with old and new variable.		
28. Verify at least one special jump instruction is included in MSW.		
29. Legal UO0 trapping (halts and exec calls)		
30. Block mode execution		
31. Several PROCS in MSW, test calling and returns, Transfer trace.		
32. DONE statement ending the simulation.		
33. Verify Master Trace Capability (*,*) each applicable type STMT, VAR, TRAN, NONE		
34. Block mode specified across proc boundary using qualifiers		
35. Test Default HBC no. in Trace and Block.		

Figure 16 (continued)

SDVS PHASE III CONFIGURATION TESTING

CONFIGURATION NO.	CONFIGURATION DESCRIPTION	Page 3
4	STANDALONE SLS SIMULATION	
SDVS PROGRAMS TESTED.		
SDVSCP, SCP, HBCL, RB, SLS, IES		
EXTERNAL SOFTWARE REQUIRED (BY)		TEST SCHEDULE
Mission Software files		3/29 to 4/2
PRIMARY FUNCTIONS TO BE TESTED.		
36. Use labels, statement numbers and offsets for trace or Block.		
37. Set Trace or Block in mid-simulation.		
38. Verify error handling for trace or block		
A. no final /		
B. $ST_1 > ST_2$		
C. illegal character		
D. nonexistent HBC, PROC, LABEL, STMT NO.		
E. Set STDCOD to 'WARN' or 'FATAL'		
39. Link to J73 runtime routines, AFAL math routines, MSW procs, MSW COMPOOLS, MSW REL's and DEF's.		
40. Verify Load Map Generation.		
41. Error test in loader, undefined and multiple externals.		

CONFIGURATION 4 STANDALONE SLS & SCP

FILES	TEST OBJECTIVES
1. TC-PQT-CON-4-1 TESTAD-PROCEDURE/1/0	4A5,4A6,4-4,10,12,13,15,17,18,19
2. TC-PQT-CON-4-2 ARITHMETIC-TEST	4A7,4-8 "CONDITION TEST"
3. TC-PQT-CON-4-3 TESTAD-PROCEDURE/2/0 LINK-COM LINK-PROCEDURE	4-9,28,31,32,34,39
4. TC-PQT-CON-4-4 TESTAD-PROCEDURE	4-21,35,36,37,38,41 "STDCOD TESTS"
5. TC-PQT-CON-4-5 TESTAD-PROCEDURE	4-30 "BLOCK MODE"
6. TC-PQT-CON-4-6 SPARSE-DATA	4-11
7. TC-PQT-CON-4-11 ARITHMETIC-TEST	4-14 "ARITHMETIC TEST"
8. TC-PQT-CON-4-7 TC-PQT-CON-4-8 TC-PQT-CON-4-9 TC-PQT-CON-4-10 EES-EFS-TEST	
9. SDVS-TEST-FILE-DIR-5/2/1 SDVS-TEST-FILE-DIR-5/3/1	4-5,7,16,20 4-6
10. TEST-ROLLBACK-DIR-5/1/LAST	4-22,23
11. SDVS-TEST-FILE-DIR-6/1/LAST PARTIAL-TEST-CASE-1/1/0	4-16,25,27,29,40
12. SDVS-TEST-FILE-DIR-6/2/1 PARTIAL-TEST-CASE-1/2/1	4-16,

Figure 17

(2) Configuration Control

Once the individual testing had been completed and PQT testing was initiated, all SDVS programs were placed under configuration control. All source files were copied to a file area whose password was known only to the configuration control manager. Whenever, a PQT test exhibited errors in SDVS, the person running the test would submit a problem report stating the test run and the observed error symptoms. This problem report would be forwarded to the appropriate SDVS analyst who would analyze the problem and complete a software change authorization form detailing the necessary programming modifications. If the change was approved by the SDVS manager, the analyst would copy the affected files from the configuration control file area, make the changes, and rerun the specific PQT test causing the problem. The configuration control manager would then copy the modified program back onto the configuration control file area (these files were write protected from all other file areas) and rerun a set of previously working PQT tests checking for side effects from the program change.

This configuration control did create some time consuming paperwork, but all involved agreed it was worth the effect. It is somewhat of a programming law that fixing an error has a high probability of introducing additional errors. These new error were usually caught by the re-testing procedures and if not, the problem reports and software change forms were often useful in tracking down errors caught at a later time which had worked in previous PQT tests. In addition, the configuration control procedures allowed two people to work on different errors simultaneously without interference.

When SDVS was delivered the problem report/software change forms provided a good base for user problem reports and a good discipline for updating a piece of software while it is being used.

2. Program Plan

a. Overall Philosophy

The program plan for the development and delivery of SDVS was based on a three phased development approach. The philosophy was to provide interim SDVS capabilities to permit early use of SDVS and to facilitate quality assurance through early use. Both of these objectives were achieved with this approach. An added benefit was the flexibility provided by being able to respond to user feedback and changing requirements in early phases and incorporate necessary changes prior to the final delivery. This flexibility was especially important for SDVS due to the evolutionary nature of DAIS and therefore SDVS. The early SDVS versions could be viewed as SDVS software prototypes. This insured quality assurance at an early date and provided time for evaluation of SDVS capabilities via user feedback.

b. Development Concept

The development of the SDVS software followed a top-down procedure according to the program hierarchy structure shown in Figure 10. This approach is based on the overall top-down development techniques described in section 5.1. The following paragraphs outline the SDVS programs developed in each phase and summarized in Table 2.

(1) Phase I SDVS

Even though a working J73 compiler was not yet available in August of 1975, TRW felt that a basic usable SDVS system would provide invaluable information in determining good final system requirements and design. To meet this goal, TRW began implementation of SDVS in the JOVIAL J70 dialect. The Phase I SDVS was geared to providing a basic interactive SDVS capability for the user in the File Generation mode to build SDVS files, and to provide a basic simulation capability with the SLS. Only a basic subset of test case directives was provided by the original Simulation Control Language. Phase I involved development and testing of SDVS software from each of the hierarchical levels shown in Figure 10, thereby validating the overall SDVS structure and demonstrating the feasibility of SDVS as a tool for development of DAIS mission software.

SDVS Program	Phase I (8 Oct. 75)	Phase II (23 Jan. 76)	Phase III (14 May 76)
SDVS Control Program	X	X*	X*
Software Management	X	X* (Multiple Compoils and Copy)	X
File Generator	X	X*	X* (new interactive commands)
Scenario Generator			
Simulation Control Language	X (timed-keyed events)	X	X* (conditional events)
Data Processing Language	-	-	X
Simulation Control Program	X (standalone mode only)	X	X* (full capabilities)
Loader			
Statement Level Simulator	X	X	X* (Variable Trace)
Interpretive Computer Simulation	-	-	X
Hot Bench	-	-	X
Post Run Editor	-	-	X
Interpretive Computer Simulation	-	-	X
Statement Level Simulator	X	X	X
Data Bus Simulation	-	-	X
External Environment Simulation	-	-	X
Snapshots/Rollback	-	-	X

*Program Enhanced

Note: Phase I-A (Phase I SDVS converted to J73) delivered 15 December 1975

Table 2 SDVS Capabilities by Phase

Integration of the SDVS Control Program with the DEC10 operating system and the second level Software Management, Scenario Generator, and Simulation Control Programs validated the top-down control and data interfaces between these programs. Integration of the Software Management Program involved a critical interface with the DEC Data Base Management System (DBMS) software. Integrating this software early, validated the complex integration of this software with the SDVS and identified SDVS/DBMS interfaces inefficiencies involving catalog searches. The lead time gained on this type of problem enabled TRW to do an extensive analysis of DBMS cataloging operations and optimized the necessary SDVS interfaces.

Analysis of the Phase I interfaces between the translated Simulation Control Language directives to the Simulation Control Program also resulted in some re-design of this interface which resulted in a more powerful design delivered with the final version of SDVS.

In summary, the Phase I SDVS demonstrated the SDVS concept, validated the SDVS interfaces with the host DEC10, and provided feedback on the design resulted in an improved final product.

(2) Phase II SDVS

Phase II SDVS was geared toward enhancing SDVS capabilities concerning the structure of the Software Management Program mission software files, and new File Generation commands based on changes in requirements for the enforcement of mission software development standards by SDVS. In addition, the SDVS Phase I software was converted from JOVIAL J70 to J73/I and delivered as the Phase I-A SDVS.

Originally, the Phase II software was to include the Interpretive Computer Simulation, the Data Bus Simulation, and the External Environment Simulation. All of these programs were dependent on DAIS specifications which were not available for a Phase II delivery and were rescheduled for Phase III.

BEST AVAILABLE COPY

(3) Phase III SDVS

The major objectives of the Phase III SDVS were to integrate the CAIS simulator programs (level 3 in Figure 10) into the basic SDVS framework provided in Phase II. A comprehensive simulation Snapshot Roll-back capability was developed and the second level Post Run Editor Program was also included. High powered tracing and monitoring capabilities (WHEN, WHENEVER, etc.) were added to the Simulation Control Program in response to needs demonstrated during usage of the Phase I and Phase II systems. This experience gained from the previous systems concerning translating and executing simulation scenarios resulted in a very powerful, efficient, and sophisticated simulation facility. In addition Phase I and Phase II usage allowed TRW to identify some of the detrimental impacts of SDVS on the DEC10 system. Modifications to the configuration of SDVS on the machine (multiple high segments, restoration of SDVS low segment, placing some code in the low segment, etc.) alleviated this impact. This can be attributed to the three phase development approach in which the first phase demonstrates all the basic capabilities and is used to evaluate the design and fine-tune the system accordingly. The demonstration of a basic framework early in the SDVS program played a key role in the on-time delivery of a system that has been proven to exceed original performance requirements and operate reliably.

3. Productivity Using SDVS Development Techniques

As described in section 5, 1. the design, development and testing of the SDVS software applied many state of the art software engineering techniques, i.e., top-down design, use of structured programming, etc. Overall, application of these techniques proved instrumental in all phases of the software development process as discussed in section 5, 1.

To provide a quantitative measure of the productivity associated with the development of the SDVS software, the size of each of the SDVS programs shown in Figure 10 was tabulated and is shown in Table 2. For each program the number of JOVIAL J73/I statements and the corresponding size of the generated object code is presented for both program data and code. The total number of JOVIAL statements for both program data and code for the SDVS software is 82,965. The total number of engineering man-months over a 24 month period for the development was 227 man-months. The engineering effort included the following:

- o Requirement Definition
- o Program Specification development including preliminary and critical design reviews for each SDVS program.
- o Coding
- o Preliminary and formal qualification testing for the Phase I, II, and III versions of SDVS
- o Delivery of all documentation including program specifications (over 5000 pages total), user manuals for each phase, test result documents, etc.
- o Providing SDVS user support since the original Phase I delivery
- o Analysis and fine tuning of SDVS performance
- o One person as a full time manager
- o One person full time responsible for configuration management and testing

The effective number of JOVIAL statements developed per man-month is therefore:

$$82965/227 \approx 365 \text{ statements/man-month.}$$

SDVS Program

	<u>Program Data</u>		<u>Program Code</u>		<u>Comments</u>	<u>Total</u>	
	Statements	Core	Statements	Core		Statements	Core
SDVS Control Program	4550	15.0K	10200	30.5K	3000	17750	45.5K
File Generator	650	2.5K	5700	16.5K	750	7100	19.0K
Software Management							
JOVIAL	650						
COBOL	590	17.5K					
Post Run Editor	1000	28.5K	4400	9.5K	770	5820	27.0K
Scenario Generator			2600		2100	5290	
SCL Translator			9025	18.5K	2100	12125	47.0K
DPL Translator	1065	36.0K					
Simulation Control	1000	10.5K	11010	20.0K	1900	13975	56.0K
Program			5900	12.0K	1000	7900	22.5K
Snapshot/Rollback	2040	10.0K	14280	27.5K	2840	19160	37.5K
Program							
Statement Level							
Simulation	175	4.0K	400	1.0K		925	5.0K
Interpretive Computer							
Simulation	500	1.75K	2090	6.0K	350	3290	7.75K
Data Bus Simulation	740	1.0K	2000	5.5K	700	3940	6.5K
External Environment							
Simulation	100	6.5K	600	7.0K*	200	900	13.5K
Loader	300	3.25K	1400	3.75K	460	2160	7.0K
	13360	136.5K	69605	157.75K	17370	100335	294.25K

*Including AFAL supplied Fortran models

SDVS Program	Program Data		Program Code		Comments	Total	
	Statements	Core	Statements	Core		Statements	Core
SDVS Control Program	4550	15.0K	10200	30.5K	3000	17750	45.5K
File Generator	650	2.5K	5700	16.5K	750	7100	19.0K
Software Management JOVIAL COBOL	650	17.5K	4400	9.5K	770	5820	27.0K
	590		2600		2100	5290	
Post Run Editor	1000	28.5K	9025	18.5K	2100	12125	47.0K
Scenario Generator SCL Translator DPL Translator	1065	36.0K	11010	20.0K	1900	13975	56.0K
	1000	10.5K	5900	12.0K	1000	7900	22.5K
Simulation Control Program	2040	10.0K	14280	27.5K	2840	19160	37.5K
Snapshot/Rollback Program							
Statement Level Simulation	175	4.0K	400	1.0K	350	925	5.0K
Interpretive Computer Simulation	500	1.75K	2090	6.0K	700	3290	7.75K
Data Bus Simulation	740	1.0K	2000	5.5K	1200	3940	6.5K
External Environment Simulation Loader	100	6.5K	600	7.0K*	200	900	13.5K
	300	3.25K	1400	3.75K	460	2160	7.0K
	13360	136.5K	69605	157.75K	17370	100335	294.25K

*Including AFAL supplied Fortran models

Table 3 Size of SDVS Programs

This number represents the development rate over the entire life of the contract and includes all engineering tasks associated with the SDVS development.

From Table 2, it can be seen that the total number of JOVIAL J73/I executable statements (69605) resulted in the generation of 157.75K machine instructions. The ratio of machine instructions to source statements is

$$\frac{157750}{69605} = 2.27 \frac{\text{machine instructions}}{\text{JOVIAL statement}}$$

This expansion rate demonstrates the efficiency of code generated by the J73/I compiler and therefore justifies the use of a high level language compiler like JOVIAL for software development. This low ratio of machine instructions per source statement implies that the code generated by the JOVIAL compiler was as efficient as if the coding was performed in assembly language.

It should be noted that not all the SDVS programs (294.25K) are not core resident at any one time. SDVS employs several different schemes to segment the SDVS functions such that only the necessary programs required are loaded in core at any point in time.

SECTION VI

CONCLUSIONS AND RECOMMENDATIONS

The development of the SDVS provided a wealth of experience relating to the merits of top-down design and structured programming techniques. Since these concepts are relatively new, their application to a large scale software development project like SDVS has presented the opportunity to evaluate their effectiveness in a real-world situation. The following paragraphs summarize this experience and the main points of this report.

Utility of the SDVS

- o SDVS is an integrated collection of software tools to aid in the design, development, coding and validation of DAIS mission software.
- o SDVS provides the user a powerful High Level Language (Simulation Control Language) to specify mission scenarios and the recording of simulation data. The SDVS Data Processing Language provides a user tool to specify the data reduction and analysis of simulation data.
- o SDVS provides an automated configuration and file management system for controlling mission software, simulation scenario software, post run processing software, and data generated by a simulation.
- o SDVS software has proven to be flexible enough to accommodate changing requirements.
- o Although SDVS response time is very good during low use periods on the DEC10, competition with many interactive jobs results in a significant slowdown in response time. Timing evaluations and efficiency optimizations are warranted.
- o Given that a reasonably compatible JOVIAL compiler and data base management system are available, rehosting of SDVS appears viable due to the successful containment of monitor interaction in only one program.

Top-Down Design and Structured Programming

- o Top down hierarchical design proved beneficial during initial design phases but proved somewhat inadequate during detailed design. One should not be afraid to modify initial design to accommodate implementation details.
- o Structured programming provided a basis for developing software
 - quickly
 - which was often error free
 - in which errors were easily corrected
 - which was easily understood by others
- o Structured flowcharts were liked by everyone. By associating coding constructs with each flowchart construct, coding from the flowcharts was straightforward. The same flowcharts were used for both the J70 and J73 versions of SDVS.
- o SDVS was not as modular as it should have been. The expansion block concept provides the benefits of modular procedures without inflicting the adverse overhead and error processing side effects.
- o Programming standards imposed enough similarity in programming styles that programmers found it easy to familiarize themselves with programs written by others. The conversion from J70 to J73 was a "cookbook" operation due to the programming standards. The control of interprogram communications resulted in quick and easy integration of the several main SDVS program modules.
- o The standard of producing completely detailed flowcharts prior to the start of coding each SDVS program resulted in increased productivity.
- o Flexible working hours were essential to completing the SDVS on schedule. Use of the DEC10 in off-hours was often two to three times as productive as using the machine during normal working hours.

Use of a High Level Language

- o The low ratio (2.3 to 1) of assembly language instructions per JOVIAL statements shows the viability of using JOVIAL to develop large software systems. The specified tables, bit and byte manipulations, and multiple word items allowed many functions typically written in assembly language to be implemented in JOVIAL. The Define capability, nested procedures, and compools lead JOVIAL to be easily used for structured, self-documented code.

Phased Development and Testing

- o Phased development of software provides the flexibility to accommodate deficiencies encountered when using initial system deliveries. The program developers must however guard against making gross design simplifications in order to meet delivery dates of early phases. These simplifications only result in major reprogramming efforts.
- o Comprehensive testing and strict configuration control are essential to developing a stable usable software system. The top-down development of PQT tests produced such comprehensive testing.

LIST OF ACRONYMS

AFAL	Air Force Avionics Laboratory
BCIU	Bus Control Interface Unit
CIU	Console Interface Unit
CMP	COMPOOL
CP	Control Point
DAIS	Digital Avionics Information System
DBMS	Data Base Management System
DEC10	DEC System 10
DPL	Data Processing Language
EES	External Environment Simulation
EFS	Executive Functional Simulation
FG	File Generator
FQT	Formal Qualification Testing
ICS	Interpretive Computer Simulation
IDR	Internal Directives Record
ITB	Integrated Test Bed
J73 or J73/I	JOVIAL Version 73 Level One
MSW	Mission Software
OFP	Operational Flight Program
PMC	Performance Monitor and Control
PQT	Preliminary Qualification Testing
PRE	Post Run Editor
ROT	Rough Output Tape
RT	Remote Terminal
SCADU	Super Control and Display Unit
SCG	Scenario Generator
SCL	Simulation Control Language
SCP	Simulation Control Program
SDVS	Software Design and Verification System
SDVS-CP	SDVS Control Program
SLS	Statement Level Simulation
SMP	Software Management Program
SSDF	Simulated Subsystems Data Formatter
TECO	Text Editor and Corrector Program
UUO	Undefined User Operator
VCC	Video Control Center

REFERENCES

1. "Software Design and Verification System (SDVS) Requirements Document", TRW Defense and Space Systems Group, 10 October 1974.
2. "DAIS: The First Step", Ltc. John C. Ruth, AFAL, 1973.
3. "The Software Design and Verification System (SDVS) An Integrated Set of Software Development and Management Tools", M. E. Hollowich, F. Borasz, NAECON 1976 proceedings, May 1976.
4. "Top-Down, Bottom-Up Structured Programming and Program Structuring", M. Hamilton, et al, Massachusetts Institute of Technology, Cambridge, Massachusetts, December 1972.
5. "Management Concepts for Top Down Structured Programming", R. C. McHenry, IBM, Gaithersburg, Maryland.
6. "The DAIS Software Design and Verification System (SDVS)", Charles Stark Draper Laboratory, Inc. F33615-73-C-1335, April 1974.
7. "Digital Avionics Information System Prototype", T. R. Price, M. Thullen, NAECON 1976 proceeding, May 1976.
8. "SDVS Software Development Standards", TRW Defense and Space Systems Group, 3 February 1975.