1-1-1995

Data Analysis Tools Report On The Scanner Management System (SMS)

Kevin M. Kearns

INSTITUTE FOR SIMULATION AND TRAINING

Contract Number N61339-94-C-0024
CDRL A00R
STRICOM
May 26, 1995

Data Analysis Tools Report on the Scanner Management System (SMS)

Institute for Simulation and Training
3280 Progress Drive
Orlando FL 32826

University of Central Florida
Division of Sponsored Research

IST-TR-95-12

INSTITUTE FOR SIMULATION AND TRAINING

Data Analysis Tools Report on the Scanner Management System (SMS)

IST-TR-95-12
May 26, 1995

Prepared for:
STRICOM
Contract Number #N61339-94-C-0024
CDRL A00R

Prepared by:
Kevin M. Kearns

Kevin M Kearns

Institute for Simulation and Training • 3280 Progress Drive • Orlando, Florida 32826
University of Central Florida • Division of Sponsored Research

TABLE OF CONTENTS

1 Introduction 2
- 1.1 Purpose 2
- 1.2 Acronyms 2

2 Background 2
- 2.1 Design Goals 2
- 2.2 System Design 3

3 Product 4

4 Appendices 4

1 Introduction

1.1 Purpose

This report is a contract deliverable item, CDRL A00R, under subtask 3.2.5.3, entitled "Data Analysis Tools", of the STRICOM contract N61339-94-C-0024, entitled "TRIDIS: A Testbed For Research in Distributed Interactive Simulation".

1.2 Acronyms

DAT	- Digital Audio Tape
DDL	- Data Definition Language
DIS	- Distributed Interactive Simulation
IST	- Institute for Simulation and Training
I/ITSEC	- Interservice/Industry Training Systems and Education Conference
MsDOS	- Mircosoft Disk Operating System
PDU	- Protocol Data Unit
SGI	- Silicon Graphics Inc.
SUT	- System Under Test

2 Background

The Institute for Simulation and Training (IST) has endeavored to perform DIS compliance testing of all simulators participating in DIS interoperability simulations at the Interservice/Industry Training Systems and Education Conferences (I/ITSEC). IST developed several MsDOS-based tools to help in the compliance testing process, to include a Logger, Playback, and a Scanner. The Scanner was designed prior to the 1993 I/ITSEC to be an offline, analysis tool that allowed the user to view packets from a logged binary file. That program had several deficiencies, specifically, it does not perform tests in an automated fashion, results of tests cannot be automatically recorded electronically, and it is not easily reconfigurable. Therefore, in June 1994, an effort was started to build the Scanner Management System to support testing during the 1994 I/ITSEC.

2.1 Design Goals

The Scanner Management System was designed with several goals in mind:

- Develop a tool to automate the DIS-compliance testing process. Also, automate the management of the testing information including the entire test process, company-specific information, tester information, and SUT information. From this information be able to generate reports summarizing the results of the testing process.

- Design a system that is highly flexible and reconfigurable in order to accommodate different versions of the DIS standard, different test specific configurations, and different test procedures.

- Include both manual and automated test modes. Manual mode will be similar to the old DOS-based Scanner, but be augmented to include better filtering, searching and reporting. The Automated mode will allow for on-the-fly configuration of tests and contain code that is capable of automatically determining if the logged data meets the requirements of the compliance tests.

- Develop in an environment that ensures portability to other platforms.

- Be able to read different logged formats.

- User friendly and easily understandable.

- Well documented code, a detailed design document, and a useable users' manual.

2.2 System Design

The Scanner Management System was designed to be similar to a data base management system, in that, all information about a SUT was to be saved within the data base or test suite and could be retrieved later for further analysis. This provides the user with a clean, organized method of managing the test results for each SUT. The program was also designed to be highly configurable at runtime. This was achieved by creating ASCII text, configuration files that defined the PDUs, network protocol layers, entity types, enumerated values, and entire management structure for a test suite. Users can easily add to or modify the values in the configuration files and the program will automatically adapt to the new values. The program also allows the user to specify which configuration files to load at runtime and which configuration files to use to override the default configuration with more specific configuration data.

The system allows the user to filter out unwanted packets and only look at those packets that meet the criteria for a specific test. To ensure the highest level of portability, the program was designed around a X Windows Motif user interface and the code was written to the Unix System V Release 4 standard. The exact, same source code has been compiled and run on both the Motorola and Silicon Graphics environments.

3 Product

The first version of the Scanner Management System meets all of the design goals. The system was used extensively prior to the 1994 I/ITSEC to perform DIS-compliance testing on all of the simulators that were going to participate. Two executable versions of the software are included in the DAT backups, a Motorola version and an SGI version, as well as, the commented source code, configuration files, make files, and readme files.

In order to make the Scanner extremely configurable, a Data Definition Language (DDL) was developed to define the structure of all DIS PDUs, as well as, the different network protocol layers. This language allows users to define a PDU by specifying the format of each field or a group of fields that make up the PDU. Its purpose was limited when it was created for the Scanner, however, it could be the beginning of a more complete, robust language that could be used as a baseline definition of each PDU by many other DIS-oriented software programs.

4 Appendices

APPENDIX A

Scanner System Design Document

SYSTEM DESIGN DESCRIPTION

FOR

IST: SCANNER MANAGEMENT SYSTEM

VERSION 1.0

DATE REVISED: May 24, 1995

Prepared By: The Institute for Simulation and Training

TABLE OF CONTENTS

1 **INTRODUCTION** 1
1.1 Purpose 1
1.2 Scope 1
1.3 Acronyms 3

2 **REFERENCES** 4

3 **DECOMPOSITION DESCRIPTION** 5
3.1 Module Decomposition 5
3.1.1 Graphical User Interface 5
3.1.1.1 Configuration Management Window 5
3.1.1.2 Test Suite Edit Window 6
3.1.1.2.1 System Configuration Window 6
3.1.1.2.2 Network Automated Test Window 7
3.1.1.2.3 PDU Automated Test Window 7
3.1.1.3 Timeline Window 7
3.1.1.4 Packet Count Window 8
3.1.1.5 Packet Display Window 8
3.1.1.6 Packet Filter Window 8
3.1.1.7 Report Results Window 8
3.1.2 Management 9
3.1.3 Testing 9
3.1.4 Reports 9
3.1.5 Packet Identification 9
3.1.6 Data Definition Language 10
3.1.7 Default Network Configuration Table 10
3.1.8 Default PDU Configuration Table 11
3.1.9 Data Mapping 11
3.1.10 Packet Validation 12
3.1.11 Packet Display 12
3.1.12 Database Routines 13
3.2 Data Decomposition 13
3.2.1 Management Structure 13
3.2.2 Indextable Structure 13

4 **DEPENDENCY DESCRIPTION** 15
4.1 Test Suite File Management 15

4.1.1 Create 15
4.1.2 Open 15
4.1.3 Edit 16
4.1.4 Save 16
4.1.5 Close 16
4.1.6 Delete 16
4.2 Edit 17
4.3 Testing 18
4.3.1 Automated Testing 19
4.3.2 Manual Testing 20
4.4 Report Generation 20

5 **INTERFACE DESCRIPTION** 22
5.1 Module Interface 22
5.1.1 **PDU_TBL.C** 22
5.1.1.1 Build Default PDU Configuration Table 22
5.1.1.2 Build Default Network Configuration Table 23
5.1.1.3 Destroy Default PDU Configuration Table 23
5.1.1.4 Destroy Default Network Configuration Table 24
5.1.2 **PDU_UTIL.C** 25
5.1.2.1 Fetch Size of Mapped List 25
5.1.2.2 Map PDU Configuration List to a Data Packet 25
5.1.2.3 Map default PDU Configuration List to Data packet. 26
5.1.2.4 Map default Network Configuration List to Data packet. 27
5.1.2.5 Destroy Mapped Configuration List 27
5.1.3 **PDU_VALID.C** 29
5.1.3.1 Validate the contents of an entire Data Packet 29
5.1.3.2 Validate the contents a Data Field 30
5.1.4 **PDU_DISPLAY.C** 31
5.1.4.1 Display the contents of a data packet in a X Window Packet Display Screen. 31
5.1.5 **INDEXTAB.C** 32
5.1.5.1 Build the Indextable. 32
5.1.5.2 Complete the building of the Indextable 32
5.1.5.3 Identify the contents of a packet 33
5.1.5.4 Return the number of packets in the current binary file. 34
5.1.6 **DB.C** 35
5.1.6.1 Read the next packet from disk. 35

5.1.6.2 Read a specific packet from disk. 35
5.1.6.3 Makes sure a packet is on a long boundary. 36
5.1.6.4 Find the first packet in a binary file that meets a specified criteria 36
5.1.6.5 Find the next packet in a binary file that meets a specified criteria 37
5.1.6.6 Return the number of the current packet being processed. 38
5.1.7 **FILTER.C** 39
5.1.7.1 Initialize Filter structure. 39
5.1.7.2 Determines if current packet meets filtering criteria. 39
5.1.8 **ENTTYPES.C** 41
5.1.8.1 Load the Entity Type Enumerated Table. 41
5.1.8.2 Search the Entity Type list with a string looking for a match. 41
5.1.8.3 Search the Entity Type list with an Entity Type structure looking for a match. 42
5.1.9 **ENUM_CFG.C** 43
5.1.9.1 Build Enumeration Table from Configuration file. 43
5.1.9.2 Search a specific Enumerated Table looking for a value. 43
5.1.10 **MAINWIN.C** 45
5.1.10.1 Main startup routine. 45
5.1.10.2 Initialize default settings. 45
5.1.10.3 Initialize default settings. 46
5.1.11 **MAINMENU.C** 47
5.1.11.1 Get the current test mode (Automated/Manual). . . . 47
5.1.11.2 Set the current test mode flag. 47
5.1.11.3 Create the main menu. 48
5.1.11.4 Disable selection of main menu widgets. 48
5.1.11.5 Enables selection of main menu widgets. 49
5.1.12 **AUTOTEST.C** 50
5.1.12.1 Determines the offset of a specific validation test in the validation test array. 50
5.1.12.2 Generic procedure called to start Automated testing. 50
5.1.12.3 PDU Transmission test procedure. 51
5.1.12.4 Network Transmission test procedure. 52
5.1.12.5 Network Transmission test procedure. 52

5.1.12.6 Network and PDU Reception test procedure. 53
5.1.13 **AUTONETTEST.C** 54
5.1.13.1 Process the Network tests, specific configuration file. 54
5.1.14 **AUTOPDUTEST.C** 55
5.1.14.1 Process the PDU tests, specific configuration file. . . 55
5.1.15 **CO_DIALOG.C** 56
5.1.15.1 Build the Company abbreviation and SUT number dialog. 56
5.1.16 **CONFWIN.C** 57
5.1.16.1 Create/Display the Configuration management window. 57
5.1.16.2 Initialize local management structure window. 57
5.1.16.3 Copy values from managment structure to screen widgets. 58
5.1.17 **FILEMGMT.C** 59
5.1.17.1 Try to open logged binary and test configuration files. 59
5.1.17.2 Close currently used logged binary and test configuration files. 59
5.1.17.3 Tells CreateIndexTable() which binary file to use. 60
5.1.17.4 Read test specific configuration file. 60
5.1.18 **LOG_DIALOG.C** 62
5.1.18.1 Display the logger window. 62
5.1.18.2 Open the pipe between the Scanner and the Logger. 62
5.1.19 **MANAGE.C** 63
5.1.19.1 Rebuild the path to the test suite directory 63
5.1.19.2 Sets the default values in management structure. . . . 63
5.1.19.3 Loads the Management configuration file's contents into memory. 64
5.1.19.4 Fill in the Management structure with any missing tests. 64
5.1.20 **MGR.C** 66
5.1.20.1 Create a new management structure. 66
5.1.20.2 Open an existing management structure. 66
5.1.20.3 Saves the open management structure to disk. 67
5.1.20.4 Closes the opened management structure. 67
5.1.20.5 Deletes an existing management structure. 68
5.1.20.6 Load test suite-specific configuration files. 68

5.1.21 **NETCONF.C** 70
- 5.1.21.1 Destroy Network user edit structure. 70
- 5.1.21.2 Create and display Network user-edit configuration window. 70
- 5.1.21.3 Build the Network user edit list. 71
- 5.1.21.4 Extract PDU header from Network header. 71

5.1.22 **PDUCONFIG.C** 73
- 5.1.22.1 Destroys the PDU user edit configuration screen and widgets. 73
- 5.1.22.2 Creates and displays the PDU user edit window. . . . 73
- 5.1.22.3 Build PDU user edit list. 74
- 5.1.22.4 Free PDU edit list. 74
- 5.1.22.5 Extract PDU data from PDU list. 75

5.1.23 **PC_DIALOG.C** 76
- 5.1.23.1 Create/Display Packet Count Window. 76
- 5.1.23.2 Update packet count values. 76
- 5.1.23.3 Destroy the Packet Count Window. 77

5.1.24 **PF_DIALOG.C** 78
- 5.1.24.1 Create/Display Packet Filter Window. 78
- 5.1.24.2 Initialize contents of Packet Filter Window. 78
- 5.1.24.3 Fetch the contents of the Packet Filter Window. . . . 79
- 5.1.24.4 Fetch the status of selected items in Packet Filter Window. 79
- 5.1.24.5 Destroy the Packet Filter Window. 80

5.1.25 **QRY_DIALOG.C** 81
- 5.1.25.1 Create/Display Query Window. 81
- 5.1.25.2 Initialize contents of Query Window. 81
- 5.1.25.3 Fetch the contents of the Query Window. 82
- 5.1.25.4 Destroy the Query Window. 82

5.1.26 **RP_DIALOG.C** 83
- 5.1.26.1 Create/Display the Report Results Window. 83
- 5.1.26.2 Update Report Results Window with information from management structure. 83
- 5.1.26.3 Add the results of the test to the Report Results Window. 84
- 5.1.26.4 Assign the test status results to the Report Results Window. 84
- 5.1.26.5 Copy information from the Report Results Window to the management structure. 84
- 5.1.26.6 Destroy the Report Results Window. 85

5.1.27 **SC_DIALOG.C** 86

5.1.27.1 Create/Display System Configuration Window. 86
5.1.27.2 Copy information from management structure to System Configuration Screen. 86
5.1.27.3 Destroy the System Configuration Window. 87
5.1.28 **TG_DIALOG.C** 88
5.1.28.1 Create/Display Test Selection Configuration Window. 88
5.1.28.2 Destroy the Test Selection Configuration Window. 88
5.1.29 **TL_DIALOG.C** 90
5.1.29.1 Create/Display the Timeline Window. 90
5.1.29.2 Freshen the text for the current record highlighted in the Timeline 90
5.1.29.3 Delete list of time line entries. 91
5.1.29.4 Destroy the Timeline Window 91
5.1.30 **TS_AUTO_DIALOG.C** 92
5.1.30.1 Create the automated test selection dialog. 92
5.1.30.2 Fetch the Network user edit list. 92
5.1.30.3 Fetch the PDU user edit list. 93
5.1.31 **TS_MAN_DIALOG.C** 94
5.1.31.1 Create manual test selection dialog. 94
5.1.32 **TS_DIALOG.C** 95
5.1.32.1 Create/Display test suite selection dialog. 95
5.1.33 **RPTLOGGER.C** 96
5.1.33.1 Open the report log. 96
5.1.33.2 Close the report log. 96
5.1.33.3 Write the report log header. 97
5.1.33.4 Write the test header to the report log. 97
5.1.33.5 Write report results to the report log. 98
5.1.33.6 Write formatted information to the report log. 98
5.1.34 **RPTRESULTS.C** 99
5.1.34.1 Create Results report. 99
5.1.35 **RPTSTATUS.C** 100
5.1.35.1 Create Status report. 100
5.1.36 **PACKET_ID.C** 101
5.1.36.1 Identify a packet. 101
5.2 Data Interface 102
5.2.1 Data Definition Language 102
5.2.2 Scanner Data Structures 111
5.2.2.1 PDU Data Entry 111
5.2.2.2 Default Group List 115

5.2.2.3 Mapped Group List 117
5.2.2.4 Default Network Configuration Table 117
5.2.2.5 Default PDU Configuration Table 118
5.2.2.6 Logger Header 119
5.2.2.7 Management Structure 120
5.2.2.8 Default Management Structure 125
5.2.2.9 Indextable 126
5.2.2.10 Filter 129
5.2.2.11 Automated Test Structure 131
5.2.3 Scanner Configuration Files 132
5.2.3.1 Scanner Configuration File 132
5.2.3.2 Manager Configuration File 133
5.2.3.3 Network Default Configuration File 135
5.2.3.4 PDU Default Configuration File 135
5.2.3.5 Entity Types Configuration File 136
5.2.3.6 Enumerated Values Configuration File 136
5.2.3.7 Network Test-specific Configuration File 137
5.2.3.8 PDU Test-specific Configuration File 138
5.2.4 Logged Binary File Format 138

6 **DETAILED DESIGN** 140
6.1 PDU/Network 140
6.1.1 PDU Table Building - **PDU_TBL** 140
6.1.2 PDU Utility Routines - **PDU_UTIL** 143
6.1.3 PDU Validation Routines - **PDU_VALID** 147
6.1.4 PDU Display Routines - **PDU_DISPLAY** 152
6.2 Test Suite 163
6.2.1 Load Management Structure - **MANAGE** 163
6.2.2 Create a Test Suite - **CO_DIALOG** and **MGR** 166
6.2.3 Select a Test Suite - **TS_DIALOG** 168
6.2.4 Display contents of a Test Suite - **CONFWIN** 171
6.2.5 Open a Test Suite - **MGR** 173
6.2.6 Save a Test Suite - **MGR** 175
6.2.7 Close a Test Suite - **MGR** 176
6.2.8 Delete a Test Suite - **MGR** 177
6.3 Configuration 178
6.3.1 Scanner Configuration file - **MAINWIN** 178
6.3.2 Entity Types Configuration file - **ENTTYPES** 180
6.3.3 Enumeration Tables Configuration file - **ENUM_CFG** 183
6.3.4 System Configuration window - **SC_DIALOG** 185
6.3.5 Test Selection window - **TG_DIALOG** 187

6.4 Indextable 189
6.4.1 Indextable - **INDEXTAB** 189
6.5 Packet Filtering 192
6.5.1 Packet Filter Routines - **FILTER** 192
6.6 File Management 194
6.6.1 Logged Binary and Configuration files - **FILEMGMT** 194
6.7 Automated Testing 196
6.7.1 Automated test selection - **TS_AUTO_DIALOG** 196
6.7.2 Network User Edit List - **NETCONF** 200
6.7.3 PDU User Edit List - **PDUCONFIG** 205
6.7.4 Common Automated test routines - **AUTOTEST** 208
6.7.5 Network-specific test routines - **AUTONETTEST** 213
6.7.6 PDU-specific test routines - **AUTOPDUTEST** 215
6.8 Manual Testing 219
6.8.1 Manual test selection - **TS_MAN_DIALOG** 219
6.9 Graphical User Interface Windows 223
6.9.1 Packet Count - **PC_DIALOG** 223
6.9.2 Packet Display - **PKT_DISP** 226
6.9.3 Timeline - **TL_DIALOG** 228
6.9.4 Packet Filter - **PF_DIALOG** 232
6.9.5 Report Results - **RP_DIALOG** 235
6.9.6 Query Filter - **QRY_DIALOG** 238
6.10 Logger 241
6.10.1 Interfacing with the IST Logger - **LOG_DIALOG** 241
6.11 Reports 244
6.11.1 Status - **RPTSTATUS** 244
6.11.2 Results - **RPTRESULTS** 245
6.11.3 Report Log - **RPTLOGGER** 246

7 **APPENDIX A** 247
7.1 Scanner Development Environment 247
7.2 Scanner Runtime Environment 247

8 **APPENDIX B** 248
8.1 Hardware Requirements 248

FIGURES

Figure 1: Default Group Listing (Fields only) 115
Figure 2: Default Group Listing (Groups and Fields) 116
Figure 3: Example of Data Mapping 144

1 INTRODUCTION

This Software Design Description (SDD) shows how the Scanner software system was designed to aid in the testing of systems that plan on participating in a Distributed Interactive Simulation (DIS).

1.1 Purpose

The Scanner is a utility program designed to aid in testing the interoperability requirements of systems that will be participating in a Distributed Interactive Simulation. The Scanner creates a test suite (or data base) for each System Under Test (SUT) being tested. This provides the user a place to electronically store the results of each and every test as opposed to having to record all test results on paper. The test suite is saved in it's own, unique directory where all files associated with the test suite will also be kept.

For each test, the Scanner reads a binary (BIN) file logged using IST's Logger and performs an initial range validation on all of the fields within each DIS packet in the file looking for packets within the BIN file that contain erroneous data. The Scanner has two different test modes, Manual and Automated. The Manual mode allows the user to look at any packets in the BIN file based on filter criteria they specify. The user after looking at the desired packets then fills in a report results window with the results they feel are appropriate. In the Automated mode, the program runs the testing and evaluating of the appropriate packets in a BIN file and automatically fills in the report results windows with the correct results. Currently, only a subset of the possible tests to choose from can be run in the Automated mode.

After each test is run and the report results window is correctly filled in, the program will save all of the results within the test suite. Also, all information encountered during the running of the test, to include errors found in a packet, are written to a report log. The report log can be considered a diary of each test that was run and what happened during the running of the test. At any time, the user can generate either of the two report formats supported by the Scanner. The Status and Summary reports, in different levels of specificity, list each test that was selected to be run and the results of each.

1.2 Scope

This document is divided into four separate design views to be used as required. These sections are:

Section 3: Decomposition Description partitions the system into design entities and can be used by designers and maintainers to identify the major design entities of the system. This allows tracing requirements and functions in order to design entities.

Section 4: Dependency Description describes the relationship among entities and system resources and can be used by maintainers to isolate modules causing system failures or resource bottlenecks. It also aids in integration testing.

Section 5: Interface description lists all interfaces and can be used by application programmers, designers, and testers to gain information as to how to use a design entity.

Section 6: Detail Description describes the internal design details of all modules and is used by the programmer during implementation and unit testing.

1.3 Acronyms

DDL	- Data Definition Language
DIS	- Distributed Interactive Simulation
PDU	- Protocol Data Unit
SUT	- System Under Test

2 REFERENCES

1. Technical Report, Test Documents for DIS Interoperability, Institute for Simulation and Training.

2. Standard for Distributed Interactive Simulation -- Application Protocols, Version 2.0, Third Draft.

3. Enumeration and Bit-encoded Values for use with IEEE 1278.1-1994, Distributed Interactive Simulation -- Application Protocols.

4. Motif Reference Manual for OSF/MOtif, Release 1.2, Editions 6A, 6B, O'Rielly & Associates, Inc.

3 DECOMPOSITION DESCRIPTION

This section decomposes the software into major system entities and describes the attributes of each entity.

The software is composed of 10 major entities: Graphical User Interface, Management, Testing, Reports, Packet Identification, Data Mapping, Packet Validation, Packet Display, Database, and Utility Routines.

3.1 Module Decomposition

3.1.1 Graphical User Interface

The Graphical User Interface allows the user to access test suite information, run tests, record test results, view binary log files, and view the contents of the data packets.

3.1.1.1 Configuration Management Window

This window allows the user to enter information about the SUT being tested, the company who designed the SUT, and the Testbed environment being used to perform the tests.

The SUT information includes:

Name:	Name of the site being tested.
Number:	Number associated with the SUT.
IP Address:	IP address of the SUT.
Ethernet Address:	Ethernet address of the SUT.
Version:	Internal company version number of the SUT (filled in automatically from previously entered information).
Platform:	Platform the SUT runs under.

The Company information includes:

Name:	Name of the company performing the test.
Abbreviation:	3-character abbreviation of company name (filled in automatically from previously entered information).
Address 1:	Street address of company.

Address 2:	City, State, Zip Code of company.
Point of Contact:	Person to contact at the company regarding testing information.
E-Mail:	E-mail address.
Phone:	Phone number.

The Testbed information includes:

Tester:	Name of the user running the test.
Date:	The date the test is being run.
Terrain Database:	The name of the terrain database being used.
Data Directory:	The directory where the binary logged files can be found.
IP Address:	The IP address of the testbed.
Ethernet Address:	The ethernet address of the company performing the test.
CGF Version:	The version of the CGF being used for the test.

3.1.1.2 Test Suite Edit Window

From the Test Suite Edit window, the user can configure how the Scanner Management System runs, what configuration files the system uses, what tests are going to be run, and what logged binary files are going to be tested. This drop-down menu window contains several entries, including System Configuration and list of test groups.

The specific tests groups or levels of tests are derived from the Institute for Simulation and Training's *Technical Report, Test Documents for DIS Interoperability.* See that document for specific definitions of what the criteria is for each of these tests. The tests include Network, PDU, Terrain Orientation, Appearance, Interactivity, System Test, Manned Simulator, Protocol Translator, and Capabilities tests. Because of the number of tests, PDU, Appearance, and Interactivity are broken down into separate menus for Reception, Transmission, Adverse, and Erroneous.

3.1.1.2.1 System Configuration Window

The System Configuration window allows the user to change the names of the configuration files that the Scanner uses as well as the default port addresses. The Scanner loads the default configuration files at startup. If the user wants to use different configuration files or ports, go to this screen and change the desired values and the new values will be automatically loaded. The Scanner will also remember to load those same files next time that test suite is loaded.

3.1.1.2.2 Network Automated Test Window

The Network Automated Test window allows the user before performing an automated network test to enter more specific range-checking values than the values specified in the default network configuration file. This window, when it first appears on the screen, is loaded with the MIN/MAX values from the default configuration file for each field from all of the types of network interfaces the Scanner recognizes. The program, when performing the test, will use the values listed on the screen as the range of acceptable values for each field of a network header.

3.1.1.2.3 PDU Automated Test Window

The PDU Automated Test window allows the user before performing an automated PDU test to enter more specific range-checking values than the values specified in the default PDU configuration file. This window when it first appears on the screen is loaded with the MIN/MAX values from the default configuration file for each field of the PDU to be tested. The program, when performing the test, will use the values listed on the screen as the range of acceptable values for each field of the PDU as it is being tested for the specific PDU test.

3.1.1.3 Timeline Window

The Timeline window shows, in a list format, all of the packets that met the requirements specified within the Packet Filter. It appears for both Automated and Manual tests. For Manual tests, the user fills in the Packet filter with the desired criteria. With Automated tests, the Filter is automatically filled in with the appropriate criteria for each test. For each packet that meets the Packet Filter's criteria, it shows the number of the packet within the logged binary file, time the packet was logged, what type of packet it is, what type of DIS PDU the packet is, the source Entity ID number from the packet, and the delta-time from last packet with the same source Entity ID. If the packet does not pass packet validation, the first character in the line will be an '*'. At the bottom of the Timeline window, the number of total packets, the number of the current packet highlighted, the start and end times of the packets, and the time of the current packet highlighted are shown. The user uses the Timeline window to select which packet will be displayed in the Packet Display window by clicking the mouse on the desired packet shown in the Timeline window.

3.1.1.4 Packet Count Window

The Packet Count window displays the number of packet types in the file of the following types: DIS, SIMNET, EAGLE, IST Message, Total, and Unknown. For DIS packets, the count is broken down further for each specific DIS PDU packet type.

3.1.1.5 Packet Display Window

The Packet Display window displays the contents of the packet highlighted in the Timeline window. At the top of the window will be a hex dump of the entire contents of the packet. This is followed by a detailed breakdown of the network header, the transport layer, and the specific DIS PDU type. Each field of the network header and the PDU is displayed on its own line. If the field has an associated enumerated table, the appropriate enumeration is displayed at the end of the line. For groups, such as Entity ID or Entity Type, the group name is displayed and then each element of the group is listed afterwards. If any field or group failed packet validation, the beginning of the line will contain an '*' for quick identification of problem areas within the packet.

3.1.1.6 Packet Filter Window

The Packet Filter window allows the user to define which packets are to be displayed in the Timeline window. The user can select/deselect based on Entity Types, Entity IDs, DIS PDU types, start/end time frames, Exercise number, and Port number. The filter can be set to show DIS only packets, non-DIS only packets (i.e. unrecognizable packets), all packets, and exclude Testbed generated packets.

3.1.1.7 Report Results Window

The Report Results window is where the information is entered concerning whether or not the test passed or failed; if it failed, the reason(s) for failing; and any comments about the test. For manual tests, the user must enter these comments, however, for automated tests, the test status and reason field's are automatically filled in by the program. This window is also the control point for continuing or exiting the testing module. Clicking on the "Quit" button in this window exits the testing mode and removes all related windows. After closing this window, all test information is written to the report log file. Clicking on the "Next" button will close all windows, write the test information to the report log, and return the user to the Test Selection window.

3.1.2 Management

The Scanner Management System is designed to maintain a test suite or data base for each SUT. Inherent in this logic is the concept that the Scanner will have access to all information regarding the current test being conducted or any previous test, as well as future tests. The idea of maintaining a test suite will help users focus there thoughts regarding what and how to test a SUT.

When the user creates a new test suite, a directory is created to hold the test suite, report log, and any other files created by the Scanner. This way the Scanner knows where to find all files necessary to test a SUT.

3.1.3 Testing

The Scanner contains incorporates all of the tests listed in the Institute for Simulation and Training's *Technical Report, Test Documents for DIS Interoperability*. The method of performing those tests have been broken up into two forms, Automated and Manual. All tests can be performed using the Manual method, but only Level 1 - Network tests and Level 2 - PDU tests can be performed using the Automated method.

Manual testing allows the user to look at any packet within a logged binary file and selectively narrow the search criteria to look at specific packets. It is the user's responsibility to determine if the contents of the binary file meet the requirements of the test being run. In Automated testing, the user is shown only those packets that meet the criteria specific to the test being run. The program will automatically filter out the packets that are not relevant to the test.

3.1.4 Reports

The Scanner Management System has two similar reports pre-defined within the program, the Status Summary and the Results Summary. The two reports show basically the same information, however, the Results Summary is more detailed.

3.1.5 Packet Identification

The Packet Identification routines determine all layers of a packet, including the board level (Ethernet or 802.3), Lan and Net layers (AP, UDP, TCP, IP), and the application layer (DIS, SIMNET, IST Eagle, and IST Message). A packet to be analyzed and an

PACKET_IDENTIFICATION_STRUCT are both passed to the packet identification routines. As the packet is analyzed, the PACKET_IDENTIFICATION_STRUCT is filled in with the all of the pertinent information about the packet.

3.1.6 Data Definition Language

The Data Definition Language (DDL) is used to define the contents of a data packet. Each protocol layer of a data packet is defined separately. The current protocol layers are the network layer, transport layer, and application layer.

The reasoning behind the DDL is to allow a generic method for defining the contents of a protocol layer, rather than defining fixed data structures. A generic method allows one set of routines to be used for all the protocol layers. Staticly defined data structures would require a separate function for each protocol layer, and separate functions to process the various types of data structures within the protocol layers (i.e. Separate functions would be needed to process all 27 PDU types). The generic method allows all data types to be handled through a common set of routines. Fields and groups requiring special processing are identified during the parsing of a data packet, and have associated routines to handle the special requirements of these fields or groups.

The DDL allows for defining the attributes of the data field, such as: name, data type, size, minimum value, maximum value, and an associated enumeration table.

The DDL also allows grouping sets of fields into logical groups. Groups are used for clarity, defining variable data sections, and to define sets of fields that should be processed together as a unit (i.e. The entity type structure is a group of fields which have meaning independently, and as a group). Grouping fields together allows for special processing during validation and display of a data packets contents. A variable section is a group that repeats (N) times. Where (N) is defined in another data field within the data packet. Since (N) is defined within the data packet, it may vary from one data packet to the next.

The configuration files containing the DDL data is read at program startup, and default configuration tables are created. Each table contains default lists describing the contents of a protocol layer. These tables are used to build the data mapping lists which are used during packet validation and display. These lists are also used to generate the automated test screens for network and PDU testing.

3.1.7 Default Network Configuration Table

The default network table is a collection of lists that are built at program startup, and contains the default data field definitions used for the network and transport layers. These default lists may then be used to validate and display the data contents of the network and transport layers of a data packet. They are also used to generate the network automated test screen.

A configuration file contains the definition of the default network table, and is read during the creation of the default network table. The data within the configuration file is defined using the Data Definition Language.

3.1.8 Default PDU Configuration Table

The default PDU table is a collection of lists that are built at program startup, and contains the default data field definitions used to define the various PDUs in the DIS PDU application layer. These default lists may then be used to validate and display the data contents of a PDU within the application layer of a data packet. They are also used to generate the PDU automated test screen.

A configuration file contains the definition of the default PDU table, and is read during the creation of the default PDU table. The data within the configuration file is defined using the Data Definition Language.

3.1.9 Data Mapping

This module is used to map either default or user defined configuration lists describing data field information to a binary data packet.

Even though the default configuration lists define the data fields within the various protocol layers of a data packet, they may not directly map to the contents of the data packet. Fixed data structures will map directly to the data within a data packet, but data structures that contain variable sections will not. The default lists describe what one occurrence of a variable section would contain. A field within the data packet defines the number of times the variable section is repeated. This will affect the position of the fields within the variable sections, and any groups or fields following variable sections. For this reason, the default lists are mapped to the data packet before the contents of the data packet is processed for validation or display purposes.

During the data mapping process, a new list is created that overlays the data packet. The new list contains information about each data field, such as: the name of the data field, offset into the data packet, size and type of the data item, minimum and maximum data values, and an associated enumeration list.

This module uses generic routines to decipher the configuration lists allowing one set of routines to support mapping of configuration lists to data packets.

During the mapping phase, groups are updated with the size of their children. After mapping, the top group of each protocol layer contains the size of the mapped layer. The size of the protocol layers within the data packet should add up to be the size of the data packet.

3.1.10 Packet Validation

The packet validation module is used to validate the fields within a packet. This module uses the data mapping routines to obtain access to the fields within the packet.

The packet validation routines allow the caller to validate an entire packets contents, a protocol layer, or a single field within the packet.

The code in this module is designed to traverse a list that has been mapped to a data packet, and validate the fields within the data packet. If a data group requiring special processing is identified, a function designed to handle this group type is called. If a data field requiring special processing is identified, a function designed to handle this field type is called. Otherwise, the individual fields are validated against the minimum and maximum values stored in the mapped configuration list.

If a field or group fails validation, an error message is written to the report log.

3.1.11 Packet Display

The packet display module is used to display the fields within a packet in the X Window Packet Display screen. This module uses the data mapping routines to obtain access to the fields within the packet.

The packet display routines allow the caller to display an entire packet's contents, or a protocol layer within the packet. If the caller is displaying a packet's contents, a hex dump of the packet is displayed, followed by the network and transport layers, and finally the application layer. At this time, only DIS PDU application layers have been defined. If a specified protocol layer is displayed, only the data fields within the configuration list are displayed.

The code in this module is designed to traverse a list that has been mapped to a data packet, and validate the data fields before displaying them. If a data group requiring special processing is identified, a function designed to handle this group type is called. If a data field requiring special processing is identified, a function designed to handle this field type is called. Otherwise, the individual fields are displayed.

If a field fails validation during the display process, the field is flagged with an asterisk. This allows the user to identify fields that fail validation without having to read the report log.

3.1.12 Database Routines

The database routines in the Scanner are controlled by the data stored within the Indextable structure. Since the Indextable contains the offset and size of each packet in the logged binary file, it is straightforward to access any PDU within a binary file.

3.2 Data Decomposition

3.2.1 Management Structure (Test Suite)

The Scanner creates a separate management structure on disk for each SUT. The management structure contains fields for all of the Configuration Management Window information. This includes Company, SUT, and Testbed information, as well as, which tests to run, which binary files to use, if there are specific test configuration files, test results, reasons for failing tests, and much more. The memory reserved for a management structure is allocated at startup of the Scanner. The same structure is used for every management that will be opened by the Scanner.

The management structure is set up as an array of test groups each made up of an array of test entries. These arrays have a finite number of elements, however, what information is stored in each element is driven by the management configuration file, "manage.cfg". That file contains a list of all of the test groups and the specific tests for those groups recognized by the program. The user has the ability to change the names of any of the tests in the configuration file and the program will use the new test name automatically.

3.2.2 Indextable Structure

The Indextable is a array that contains an element for each packet within the logged binary file currently being tested. The Indextable is dynamically created each time a new binary file is read. The Indextable contains pertinent information about each packet within the binary file, including size of packet, offset of packet within binary file, time packet was logged, specific DIS information about the packet, and packet identification and validation information. The Indextable contains enough information so that all searching required to meet a specific filter criteria can be performed on the Indextable rather than searching the binary file on disk.

4 DEPENDENCY DESCRIPTION

This section defines the system dependencies and provides the flow control between the different modules for each functional thread. The functional threads are:

4.1 Test Suite File Management

The Scanner Management System creates a test suite (data base) for each SUT being tested. The test suite provides the user with an electronic means of recording the results of all tests conducted on a SUT. A test suite is a fixed size, therefore, it is possible to perform a block read/write on the data.

4.1.1 Create

From the File drop down menu, choosing New allows the user to create a new test suite. Upon selecting this option, the user will be prompted to input a 3-character company abbreviation and a 1-character SUT number. The Scanner will concatenate the company abbreviation and the SUT number to form a unique name. A subdirectory with that name will be created below the directory where the user is working. In that directory, all Scanner-created files will be stored. By default, the Scanner will create two files, a test suite management file and a report log file, both beginning with the unique name formed by the company abbreviation and the SUT number.

The user can then enter information about the Testbed, Company, and SUT. To select tests, the user goes to the Edit menu where he will see several levels of tests, each made up of numerous individual tests. Each test has a uniquely named file associated with it. This is the default name of the logged binary file for that test. The default name is made up of the company 3-character name and the 1-character SUT number followed by a unique number that has been associated with that test. The user can change any of the default binary filenames and the program will automatically use the new name as the name of the file for that test.

4.1.2 Open

Opening an existing test suite can be accomplished by choosing Open from the File drop down menu. This option permits the user to open a previously created test suite and continue testing with it. A list of all unopened test suites will appear. To select a test suite, the user should left-click on the test suite name and then left-click on the "OK" button. For a test suite to show up in the list, a directory must contain a file with the same name as the directory and a ".mgt" file extension.

The contents of the test suite file are read into a pre-allocated block of memory. If the test suite read in from disk has less tests or test groups than the default listing from the configuration file, the program will automatically add the extra tests from the default list to the test suite.

4.1.3 Edit

For a test suite, a user can edit any of the fields describing the Testbed, Company, or the SUT. The user can also go to the Edit drop down menu and change which tests are to be run and which logged binary files to use for each test.

4.1.4 Save

A test suite is automatically saved, that is written to disk, whenever a test suite is closed, a new one is opened, or after each test has been completed. The user can also force the program to write the management structure to disk by choosing Save on the File drop down window.

4.1.5 Close

Before creating a new test suite, editing an existing test suite, or quitting the program the Scanner will automatically close the currently opened test suite. The user can also manually close a test suite by choosing Close from the File menu. Any changes made to the test suite since the last save will automatically be written to disk before closing the test suite.

4.1.6 Delete

Because there are several files associated with a test suite, the Scanner contains an option to delete an entire test suite and any files stored in the directory containing the test suite. Only closed test suites can be deleted. After selecting the test suite to delete, a warning message will appear asking the user if he is sure whether or not he wants to delete the test suite. If the user chooses Yes, the entire test suite directory will be removed from the hard drive.

4.2 Edit

The Edit option allows the user to configure the Scanner Management System for specific testing purposes. Clicking on the Edit option will pull down a menu with several areas to choose from including editing the system's configuration and choosing which tests to run from the multiple levels of tests.

The System Configuration option allows the user to change the names of the configuration files that the Scanner uses. The Scanner loads the default configuration files at startup. If the user wants to use different configuration files or ports, go to this screen and change the desired values and the new values will be automatically loaded. The Scanner will save these changes and load them every time the test suite is opened.

The Scanner has several configuration files that are read into the program at startup. They are:

scanner.cfg	-	Contains a listing of all of the configuration files the Scanner will use as the default configuration files, as well as, information about the port settings and the logged binary file timestamp.
manage.cfg	-	The file contains all of the data signifying the format of the management structure, to include the names of the test groups, names of the individual tests for each of the groups, and all of the default information about each test.
stdV2D3.pdu.cfg	-	Contains the Data Definition Language description of all 27 DIS PDUs recognized by the Scanner.
stdV2D3.network.cfg	-	Contains the Data Definition Language description of the network and transport protocol layers recognized by the Scanner.
stdV2D3.entity_types.cfg	-	Contains a listing of the entity types recognized by the Scanner.
stdV2D3.enumerations.cfg	-	Contains a listing of the enumerated values recognized by the Scanner.

The following is a list of all of the different levels of tests supported by the Scanner as defined in the Institute for Simulation and Training's *Technical Report, Test Documents for DIS Interoperability*. See that document for specific definitions of what the criteria is for each of these tests. The test categories include Network, PDU, Terrain Orientation, Appearance, Interactivity, System Test, Manned Simulator, Protocol Translator, and Capabilities tests. To choose a test, left-click on the square to the left of the test name to select that test. Left-clicking on the Select All button will turn all of the tests on, left-clicking on the Clear All button turns all of the tests off.

To the right of the test name is the name for the binary file associated with that particular test. The default binary file name for the test is displayed but the user may change this. Whatever name is entered in that field is the name that the Scanner will expect the test's data to be logged to. To the right of the binary filename field is a field to enter the name of a configuration file specific to that test. If a filename is entered here, when Automated testing begins, the program will read the contents of the configuration file, take the values specified and replace the default validation values with the new configuration values. This concept allows the user to overwrite the default values on a test-by-test basis.

4.3 Testing

The Scanner performs all testing and validating of packets by taking an exact mapping of the data to be checked, i.e., network layer or DIS PDU, and doing a field-by-field comparison of the data in the packet against the default values in the mapping. If the packet's values do not fall within the range specified, then an error exists in the packet.

The Testing option allows the user to perform Automated or Manual tests. After selecting either Manual or Automated Testing, the user is presented with a window listing all of the tests selected in the Edit option. If the test has been previously run, the first character on the line will contain the first character of the result of the test, (**P** - Passed, **F** - Failed, or **I** - Incomplete). Also, if any comments were written about the test, a portion of the comments will appear after the name of the test.

After selecting the test to run and clicking the OK button, the IST Logger dialog box appears. If there is a pipe setup between the Scanner and the Logger, all of the buttons in the IST Logger dialog are functional. If the logged binary file exists, the OK button is active. If the binary file does not exist, the user MUST create the binary file. The name of the binary file to create is displayed in the Log File text window. Note that the Scanner looks for all binary files in the directory specified in the Data Directory field of the Testbed information. To either create the binary file necessary for the test or re-log the binary file, choose the Record button. To stop recording, press the Stop button. To begin the test, click on the OK button. Also, if the user changes the filename of the binary file to use, the new name will automatically be changed int he Edit test selection menus.

For all tests, the Scanner performs an initial validation on the logged binary file to be tested checking both the network and PDU information. The validation scans all fields in every packet to see if the values in those fields fall within the default specified parameters. If any errors are found, the field, its value, and the acceptable values are written to the report log. As each test is run, all information about that test (i.e., test results, packet validation results, etc.) is written to a report log file.

4.3.1 Automated Testing

Automated testing provides a way for users to conduct a significant amount of DIS compliance testing with very little manual effort. After choosing the test to be run, the program will fetch the default values for the range checking to be performed in the test. If a test-specific configuration file is specified for this test, the default range values will be overwritten by the configuration file's values. A dialog window will appear showing the user all fields to be tested and the range of acceptable values for that field. At this time, the user can change the values, if so desired.

After selecting the Run button on that dialog window, each packet that meets the criteria of the test is passed to the validation routines to verify that those packets have values in all fields that fall within the specified ranges. If any errors were found in the packet during testing, the program will automatically write all errors to the report log and fill in the Report Dialog with a test status of Failed and give the reason for failure in the reason field. If there were no errors found, the test status field is assigned a value of Passed. Internally, the Indextable is updated with results of the validation procedure.

When the test has completed running, the Packet Display window and the Timeline window will appear. These windows allow the user to look at all packets that meet the criteria of the search. If any of the packets did not pass the test, the first character of the line in the Timeline for that packet will be marked with an "*" for easy recognition. Highlighting that packet will display it in the Packet Display window. For each field that had a value out of range, the first character of the line in the Packet Display window will contain an "*".

After the user verifies that the Scanner assigned the appropriate test result, the user can enter comments into the Report Results window. When the user chooses Quit or Next to leave this test, the program will write all test results to the report log, as well as, updating the test suite structure.

Currently, only two levels of automated testing have been defined within the Scanner, Network header verification and PDU verification. There are spaces listed for multitudinous other tests, but they are not yet defined. However, the user could run the tests in Manual mode and still assign them the appropriate test status result and comments.

4.3.2 Manual Testing

Because not all possible tests have been automated and since a user may want to look at all or any select portion of packets within a binary file, not just the ones appropriate for a test, the Scanner has a manual testing option. After selecting the test to run, the program performs packet validation on each packet in the binary file using the default values specified in the configuration files. Internally, the Indextable is updated with results of the validation procedure. Then the program will open several windows for the user, including the Packet Count, Packet Display, Timeline, Packet Filter, and the Report Results windows. By default, all DIS-only, non-testbed generated packets show up in the Timeline.

By changing the filtering criteria in the Filter screen, the user can narrow or expand the list of items that will appear in the Timeline window. If any of the packets did not pass the initial default validation performed by the Scanner, the first character of the line in the Timeline listing that packet will contain an "*". By highlighting and selecting a packet in the Timeline window, the user can view the contents of the packet in the Packet Display window. The Packet Display window shows all of the information in the packet, including the entire network layer as well as the PDU. If there are text enumerations for specific fields in the packet, the full enumeration is shown. If the packet being displayed had any fields that did not pass initial default validation, the first character of the line containing that field will be marked with an "*".

After the user examines the appropriate packets and determines if the binary file meets the requirements of the test being conducted, the user fills in the Report Results window. When the user chooses Quit or Next to leave this test, the program will write all test results to the report log.

4.4 Report Generation

The Scanner Management System has two pre-defined reports already incorporated within the program, the Status Summary and the Results Summary. All reports are generated as formatted text files in the same directory as the test suite data base. Both reports contain a header that summarizes the Testbed, Company, and SUT information and then lists each level of testing as a discrete section within the report. The main difference between the two reports is the amount of information presented in the report.

The Results Summary report is the more verbose of the two reports. It includes the unique number and name of the test, the test status (Passed/Failed/Incomplete), number of times the test was run, specific type of test (Transmission/Reception/Adverse/Erroneous), name of the binary file tested, reason for failure of test, and comments about test. The Status Summary is a simpler report that includes only the Test status and test name and number.

The Scanner also generates a report log that lists significant information about all tests that have been conducted, to include the result of the test, the number of times run, and user comments. If an error is found in a packet during either initial packet validation or specific packet validation, the erroneous field, it's value, and the correct value range are written to the report log. The report log provides a user with a complete, historical listing of what tests were run, the results of those tests, and any errors that were found while running the tests.

5 INTERFACE DESCRIPTION

This description covers the details of external and internal interfaces.

5.1 Module Interface

This section describes the entry points in each of the scanner's main modules.

NOTE: All references to widgets and shells releate to the xWindows Motif language. This manual will not go into a detailed explanation of these points. Please see Motif Reference Manual for OSF/MOtif, Release 1.2, Editions 6A, 6B for a description/definition of these topics.

5.1.1 PDU_TBL.C

This module is used to build the default configuration tables for network, transport, and application layers of a DIS data packet. Currently, the only application layer defined is the DIS PDU.

5.1.1.1 Build Default PDU Configuration Table

SYNTAX

buildPduConfigTable (filename);

PARAMETERS

char * filename	-	specifies the name of the configuration file containing the PDU data definition.

DESCRIPTION

This function is used to create the default configuration lists for the application layer of the data packet. Specifically, DIS PDUs. This table is used during data mapping, and generation of the PDU automated test window.

RETURNS

TRUE	-	The configuration file was read and the PDU table was built.

FALSE	-	Unable to open the configuration file. The PDU table was not built.

5.1.1.2 Build Default Network Configuration Table

SYNTAX

buildNetConfigTable (filename);

PARAMETERS

char * filename	-	specifies the name of the configuration file containing the network data definition.

DESCRIPTION

This function is used to create the default configuration table for the network layer of the data packet. This table is used during data mapping, and generation of the network automated test window.

RETURNS

TRUE	-	The configuration file was read and the network table was built.
FALSE	-	Unable to open the configuration file. The network table was not built.

5.1.1.3 Destroy Default PDU Configuration Table

SYNTAX

destroyPduConfigTable (void);

PARAMETERS

None.

DESCRIPTION

This function is used to destroy the default PDU configuration table.

RETURNS

None.

5.1.1.4 Destroy Default Network Configuration Table

SYNTAX

destroyNetConfigTable (void);

PARAMETERS

None.

DESCRIPTION

This function is used to destroy the default network configuration table.

RETURNS

None.

5.1.2 **PDU_UTIL.C**

This module is used to map the contents of a configuration list over a binary data packet.

5.1.2.1 Fetch Size of Mapped List

SYNTAX

fetchMappedPduSize (list);

PARAMETERS

LL * list	-	List is a pointer to the first element in the mapped list.

DESCRIPTION

This function is used to fetch the size of a mapped list. The first element of a mapped list is a group structure holding the configuration name, and size in bytes.

RETURNS

Zero	-	The parameter specified does not exist, or there is nothing to map.
Size	-	The size of the mapped list.

5.1.2.2 Map PDU Configuration List to a Data Packet

SYNTAX

mapPduListToData (pduList, packet, length);

PARAMETERS

LL * pduList	-	pduList is a pointer to the first element in the configuration list to be mapped to a data packet.

char * packet	-	The base address of the packet layer within the data packet to be mapped.
u_short length	-	Length of the PDU data section within the data packet.

DESCRIPTION

This function maps a PDU configuration list to the data within a packet. A new list is created containing the mapped data list.

RETURNS

LL * newList	-	A pointer to the newly created list is returned.

5.1.2.3 Map default PDU Configuration List to Data packet.

SYNTAX

mapCfgPduToData (pduType, packet, length);

PARAMETERS

int pduType	-	The pduType field contains the DIS PDU number to be mapped. It is also the index into the default PDU configuration table.
char * packet	-	The base address of the packet layer within the data packet to be mapped.
u_short length	-	Length of the PDU data section within the data packet.

DESCRIPTION

This function maps a default PDU configuration list to the data within a packet. A new list is created containing the mapped data list. The pduType parameter contains a DIS PDU type, and is used to index the default PDU configuration table.

RETURNS

LL * newList	-	A pointer to the newly created list is returned.

5.1.2.4 Map default Network Configuration List to Data packet.

SYNTAX

mapCfgNetToData (netType, packet, length);

PARAMETERS

int netType	-	The netType field contains the index into the default network configuration table.
char * packet	-	The base address of the packet layer within the data packet to be mapped.
u_short length	-	Length of the PDU data section within the data packet.

DESCRIPTION

This function maps a default network configuration list to the data within a packet. A new list is created containing the mapped data list. The netType parameter contains an index into the default network configuration table.

RETURNS

LL * newList	-	A pointer to the newly created list is returned.

5.1.2.5 Destroy Mapped Configuration List

SYNTAX

destroyPduMapping (pduList);

PARAMETERS

LL * pduList	-	pduList is a pointer to the first element of a mapped list.

DESCRIPTION

This function destroys previously created mapped lists.

RETURNS

NULL	-	The mapped list was destroyed.

5.1.3 PDU_VALID.C

This module is used to validate fields within a data packet.

5.1.3.1 Validate the contents of an entire Data Packet

SYNTAX

PacketValidation (packet, netHdr, pduList, pis, length, index);

PARAMETERS

char * packet	-	Pointer to the start of the data packet.
PDU_DATA_HEADER *netHdr	-	Pointer to the network header table to be used during network and transport layer validation. NULL specifies that the default configuration table should be used during validation.
LL * pduList	-	Pointer to the application list to be used during validation. NULL specifies that the default configuration list should be used during validation of the application layer.
PACKET_IDENTIFICATION_STRUCT * pis	-	Pointer to the packet identification structure containing the packet identification information.
u_short length	-	The length of the data packet as defined by the data logger.
u_long index	-	The index of the packet within the binary file of logged data packets.

DESCRIPTION

This function validates all layers of a data packet. The caller may specify the use of the default configuration lists, or a user defined configuration list. If a packet fails any part of the validation process, an error message is written to the report log describing the reason for the failure. The index parameter indicates which packet within the binary log file failed validation.

RETURNS

TRUE	-	The packet passed validation.
FALSE	-	The packet failed validation at some point during the validation thread. The report log has a detailed definition of the failure.

5.1.3.2 Validate the contents a Data Field

SYNTAX

ValidatePduField (packet, pduField);

PARAMETERS

char * packet	-	Pointer to the start of the desired layer within the data packet.
PDU_DATA_ENTRY * pduField	-	Pointer to the data fields entry within the mapped list.

DESCRIPTION

This function is used to validate a single field within the data packet. The field structure defining the data field must come from a mapped list.

RETURNS

TRUE	-	The field passed validation.
FALSE	-	The field failed validation. The report log has a detailed definition of the failure.

5.1.4 **PDU_DISPLAY.C**

Displays the contents of a packet in the X Window Packet Display Screen.

5.1.4.1 Display the contents of a data packet in a X Window Packet Display Screen.

SYNTAX

packetDisplay (packet, netHdr, pduList, pis, length);

PARAMETERS

char * packet	-	Pointer to the start of the data packet.
PDU_DATA_HEADER *netHdr	-	Pointer to the network header table used while displaying the network and transport layer. NULL specifies that the default configuration table should be used.
LL * pduList	-	Pointer to the application list to be used while displaying the application layer. NULL specifies that the default configuration list should be used.
PACKET_IDENTIFICATION_STRUCT * pis	-	Pointer to the packet identification structure containing the packet identification information.
u_short length	-	Length of the packet as defined by the data logger.

DESCRIPTION

This function is used to display the contents of a data packet. The configuration lists are mapped to the data packet and then traversed for display purposes. The hex dump of the data packet is displayed first, followed by the network layer, transport layer, and application layer. Each field is validated before being displayed. If a field fails validation, the display screen indicates this with an asterisk next to the fields name.

RETURNS
None.

5.1.5 **INDEXTAB.C**

This module is used to build the Indextable.

5.1.5.1 Build the Indextable.

SYNTAX

void CreateIndexTable(FILE *bfile);

PARAMETERS

FILE * bfile	-	the bin file to use when creating the Indextable.

DESCRIPTION

The creation of the Indextable is two step process. First, the binary file must be scanned to determine how many packets are in the file. That is what this function does. This function also allocates the required memory for the Indextable and calls IndexTableSecondPass to actually put values into the Indextable. See IndexTableSecondPass() for more information.

RETURNS

TRUE	-	If no errors occurred.
FALSE	-	Not enough memory to create Indextable or an invalid packet was read.

5.1.5.2 Complete the building of the Indextable

SYNTAX

BOOLEAN IndexTableSecondPass(FILE *bfile);

PARAMETERS

FILE *bfile	-	the binary file to use to create the Indextable.

DESCRIPTION

IST: SCANNER

This function is the second part of creating an Indextable. It re-reads the binary file and pulls out select information from the file and stores it within the Indextable. The plan is to store as much of the pertinent information from a packet in the Indextable as possible to avoid having to go to disk when doing queries later in the program.

RETURNS

TRUE	-	If no errors occurred.
FALSE	-	Not enough memory to create Indextable or an invalid packet was read.

5.1.5.3 Identify the contents of a packet

SYNTAX

```
void Identify_Packet(char *packet, long length, TIME time_stamp,
                    PACKET_IDENTIFICATION_STRUCT  *pis);
```

PARAMETERS

char *packet	-	the packet to be identified.
long length	-	length of packet
TIME time_stamp	-	time stamp from logger header
PACKET_IDENTIFICATION_STRUCT *pis	-	the address of the structure to be filled in by the packet identification code

DESCRIPTION

Calls the packet identification code inherited from the DOS CGF to determine what kind of packet the Scanner is looking at. The packet identification information is stored in the Indextable.

RETURNS

Nothing

5.1.5.4 Return the number of packets in the current binary file.

SYNTAX

```
long PacketCount(void);
```

PARAMETERS

None.

DESCRIPTION

Returns the value of a static, local variable containing the number of packets in the current binary file. The number of packets in a binary file directly equates to the number of array elements in the Indextable.

RETURNS

long	-	The number of packets in the current binary file.

5.1.6 **DB.C**

This file contains routines for reading packets from disk and making sure packets are on a long boundary. There are also some "data base" management routines used with the filtering routines to loop through the selected packets looking for packets that meet specific requirements.

5.1.6.1 Read the next packet from disk.

SYNTAX - char * GetNextPacketFromDisk(FILE *bfile, long packet_num)

PARAMETERS

FILE *bfile	-	the binary file to read from.
long packet_num	-	the number of the packet to read. The Indextable contains the offset into binary file where each record begins.

DESCRIPTION

Called to read the next packet from disk into the global variable, "packetBuffer".

RETURNS

char *	-	a pointer to the beginning of a string containing the packet.

5.1.6.2 Read a specific packet from disk.

SYNTAX

char * ReadPacketFromDisk(FILE *bfile, long packet_num)

PARAMETERS

FILE *bfile	-	the binary file to read from.
long packet_num	-	the number of the packet to read. The Indextable contains the offset into binary file where each

record begins.

DESCRIPTION

Called to read a specific packet from disk into the global variable packetBuffer. The first time the function is called it allocates the memory for the largest (MAXSHORT) packet we will read.

RETURNS

char	*	-	a pointer to the static variable packetBuffer.

5.1.6.3 Makes sure a packet is on a long boundary.

SYNTAX

char *AlignPacket(char *packet, short len)

PARAMETERS

char	*packet	-	the address of the packet to be aligned
short	len	-	the number of characters to align

DESCRIPTION

Takes a string containing a packet and copies "len" characters to another string to make sure that the contents start on a long boundary.

RETURNS

char	*	-	the address to the beginning of an aligned packet.

5.1.6.4 Find the first packet in a binary file that meets a specified criteria

SYNOPSIS

long Find_First(DB_FILTER_STRUCT *fs)

PARAMETERS

DB_FILTER_STRUCT * fs	-	the filter structure containing the information the user wants filter the packets on.

DESCRIPTION

Finds the first packet in a binary file that meets the specific requirements specified in the variable (fs). It returns the number of the packet that meets the requirement or -1 if nothing is found. The programmer must load the packet from disk to make a packet the active packet.

RETURNS

long	-	-1 if no packets matched the filter, the Indextable offset if a match was found.

5.1.6.5 Find the next packet in a binary file that meets a specified criteria

SYNOPSIS

long Find_Next(DB_FILTER_STRUCT *fs)

PARAMETERS

DB_FILTER_STRUCT * fs	-	the filter structure containing the information the user wants filter the packets on.

DESCRIPTION

Finds the next packet after the current packet in a binary file that meets the specific requirements specified in the variable (fs). It returns the number of the packet that meets the requirement or -1 if nothing is found. The programmer must load the packet from disk to make a packet the active packet.

RETURNS

long	-	-1 if no packets matched the filter, the Indextable offset if a match was found.

5.1.6.6 Return the number of the current packet being processed.

SYNTAX

```
long CurrentPacket(void)
```

PARAMETERS

None.

DESCRIPTION

Returns the number of the currently active packet within a binary file.

RETURNS

long	-	the current packet read in from disk.

5.1.7 **FILTER.C**

Contains the routines for actually checking to see if a packet meets the requirements specified in a filter structure, DB_FILTER_STRUCT.

5.1.7.1 Initialize Filter structure.

SYNTAX

```
void Initialize_Filter_Struct(DB_FILTER_STRUCT  *fs)
```

PARAMETERS

DB_FILTER_STRUCT		* fs
	-	the filter structure to be initialized.

DESCRIPTION

This function is called before assigning any values to a DB_FILTER_STRUCT to make sure no erroneous values are assigned in the structure. Since the user may only be filling in just a few fields in DB_FILTER_STRUCT, the other fields must be blank.

RETURNS

Nothing.

5.1.7.2 Determines if current packet meets filtering criteria.

SYNTAX

```
short Do_Filtering(DB_FILTER_STRUCT  *fs, long i)
```

PARAMETERS

DB_FILTER_STRUCT		* fs
	-	the filter structure containing the "values" the user wants to search for.
long i	-	the index into the Indextable of the packet to

search.

DESCRIPTION

This function calls several local functions, each of which performs a specific test comparing the filter criteria, specified in the DB_FILTER_STRUCT, against the data saved within the Indextable for packet "i". If the packet meets each of the specific tests, then it passes the filtering and returns TRUE value.

RETURNS

short	-	TRUE if packet passed filtering, FALSE if the packet does not meet the filtering criteria.

5.1.8 ENTTYPES.C

Contains the routines for loading, parsing, and searching the Entity Type Enumerated table. The table contains all Entity Types recognized by the Scanner Management System. The table is created and loaded at runtime.

5.1.8.1 Load the Entity Type Enumerated Table.

SYNTAX

void LoadEntTypeEnumTable(void)

PARAMETERS

char	*filename	-	the name of the Entity Type file to be loaded

DESCRIPTION

This function opens the configuration file containing a listing of the valid DIS entity types. The contents of the file are read into a table that is ordered alphabetically to facilitate faster searching.

RETURNS

BOOLEAN	-	TRUE if no errors were found FALSE if an error opening the file occurred

5.1.8.2 Search the Entity Type list with a string looking for a match.

SYNTAX

char EntTypeBinarySearchStr(char *str, short *ppos, char IsCode)

PARAMETERS

char	*str	-	contains the entity that is to be searched for.
short	*ppos	-	will contain -1 if not found or an index into the Entity Table List if found.
char	IsCode	-	if "str" is in the format "1.1.225.1.1", IsCode is

FALSE (i.e. it needs to be converted to an array of unsigned characters), if "str" is an array of unsigned characters, IsCode is TRUE.

DESCRIPTION

This function searches for "str", a NULL-terminated string, in the Entity Table list by performing a binary search.

RETURNS

char	-	TRUE if "str" was found in the Entity Table List.

5.1.8.3 Search the Entity Type list with an Entity Type structure looking for a match.

SYNTAX

char EntTypeBinarySearchStruct(ENTITY_TYPE *et, short *ppos)

PARAMETERS

ENTITY_TYPE * et	-	the entity type the user wants to search for.
short *ppos	-	will contain -1 if not found or an index into the Entity Table List if found.

DESCRIPTION

This function copies the contents of the ENTITY_TYPE structure into a string of unsigned characters and then calls EntTypeBinarySearchStr().

RETURNS

char	-	TRUE if "et" was found in the Entity Table List.

5.1.9 ENUM_CFG.C

Contains code for reading the enumeration table configuration file into memory. The table is stored in memory as a linked list of tables with each table being a linked list of elements for defining the table. The table element lists are searched to see if a value has a match within the range of acceptable values for that table element.

5.1.9.1 Build Enumeration Table from Configuration file.

SYNTAX

```
int build_enum_list_from_file( char *filename )
```

PARAMETERS

FILE	*input	-	the input file
LL	*head	-	the head of the linked list to be

DESCRIPTION

Reads the input file and builds the table from the file in linked list fashion. NOTE: head MUST be initialized via ll_init() before this function is called

RETURNS

int	-	1 on success, 0 otherwise

5.1.9.2 Search a specific Enumerated Table looking for a value.

SYNTAX

```
char *FindElementFromMinMax( LL *e_head, short sval )
```

PARAMETERS

LL	*e_head	-	head of element list
short	sval	-	search value

DESCRIPTION

IST: SCANNER

Searches for and fetches the enumeration string for the seach value (**sval**).

RETURNS

char * : string of text for sval

5.1.10 **MAINWIN.C**

Contains the main startup routine for the Scanner Management System, as well as, the code for loading the Scanner's main configuration file.

5.1.10.1 Main startup routine.

SYNTAX

int main (int argc, char **argv)

PARAMETERS

int	argc	-	number of command line parameters
char	**argv	-	actual command line parameters

DESCRIPTION

This is the routine that is called at startup of the program. It calls the initialization routines and the screen management routines.

RETURNS

int	-	exit code - 0 always

5.1.10.2 Initialize default settings.

SYNTAX

BOOLEAN Init(void)

PARAMETERS

None.

DESCRIPTION

This function calls functions to load the Entity Type Tables, management configuration defaults, and other configuration tables and values. Also, if a pipe is to be setup between the Scanner and the IST Logger, this function establishes the pipe.

RETURNS

BOOLEAN	-	FALSE if error(s) occurred.

5.1.10.3 Initialize default settings.

SYNTAX

BOOLEAN LoadScannerConfig(void)

PARAMETERS

None.

DESCRIPTION

This function opens the "scanner.cfg" file and parses it, determining what config options/files the user wants to use.

RETURNS

BOOLEAN	-	FALSE if an error occurred

5.1.11 MAINMENU.C

5.1.11.1 Get the current test mode (Automated/Manual).

SYNTAX

int currentTestMode (void)

PARAMETERS

None.

DESCRIPTION

Returns a flag indicating what type of test the user is currently running, either automated or manual.

RETURNS

int	-	1 if automated test; 2 if manual test.

5.1.11.2 Set the current test mode flag.

SYNTAX

void SetTestMode(int tm)

PARAMETERS

int	tm	-	1 if automated test; 2 if manual test.

DESCRIPTION

Sets a local variable to indicate what type of test the user is currently running.

RETURNS

None.

5.1.11.3 Create the main menu.

SYNTAX

```
Widget create_main_win ( Widget top_level )
```

PARAMETERS

top_level	-	Top level shell.

DESCRIPTION

Create the application main window and create a menu bar attached to the main window.

RETURNS

Widget	-	MainWindow widget

5.1.11.4 Disable selection of main menu widgets.

SYNTAX

```
void disable_menu_selections (void)
```

PARAMETERS

None.

DESCRIPTION

Makes inactive the main menu's drop-down menus. The user can not select any menu option when it is inactive.

RETURNS

None.

5.1.11.5 Enables selection of main menu widgets.

SYNTAX

```
void enable_menu_selections (void)
```

PARAMETERS

None.

DESCRIPTION

Makes active the main menu drop-down menus.

RETURNS

None.

5.1.12 **AUTOTEST.C**

Contains the generic functions necessary to perform any of the manual or automated PDU or Network tests. Routines for actually running the test, loading a test-specific, configuration file, and determining which test to run can be found in this file. Because the actual PDU or Network tests are called from a generic routine, the address of the function that actually performs the validation is passed to the generic routine. The definitions of the specific validation routines are listed in this file as well as the data structures containing the address of these functions.

5.1.12.1 Determines the offset of a specific validation test in the validation test array.

SYNTAX

BOOLEAN AutomatedTestFunctionsIndex(char *str, TEST_SUITE_ENTRY *tse)

PARAMETERS

char	*str	-	a string containing which enumeration validation function is to be called.
TEST_SUITE_ENTRY	*tse	-	a pointer to the test suite entry to be modified.

DESCRIPTION

This function is called when reading in the "manage.cfg" file. This function will return the correct offset into the automated test array for the desired test. The offset is entered into the variable **tse**. This offset indicates the address of which test function will be called when the current test is activated.

RETURNS

BOOLEAN	-	returns TRUE if **tse** was properly set, otherwise FALSE is returned.

5.1.12.2 Generic procedure called to start Automated testing.

SYNTAX

void RunAutomatedTest (void * list)

PARAMETERS

void	*list	-	a void pointer to either the Network or PDU linked list.

DESCRIPTION

This function displays the Packet display window and the Timeline window and, depending upon what type of test is to be run, calls the appropriate validation routines. The function is a central point that determines which specific automated test function to invoke.

RETURNS

None.

5.1.12.3 PDU Transmission test procedure.

SYNTAX

BOOLEAN PDUTest_Transmission(int pdu)

PARAMETERS

int	pdu	-	which of the 27 pdus is to be tested

DESCRIPTION

This function is called for all Level 2 PDU Transmission tests. It loops through all packets in a binary file looking for packets that are of the correct PDU type. Each packet is loaded into memory and validated against the values in the user-specified edit list.

RETURNS

BOOLEAN	-	returns the result of the test.

5.1.12.4 Network Transmission test procedure.

SYNTAX

BOOLEAN NetworkTest_Transmission(void)

PARAMETERS

None.

DESCRIPTION

Called to conduct all Network Transmission tests, only packets that are DIS-only, non-testbed generated packets are compared against the min/max values specified in the user's edit list.

RETURNS

BOOLEAN	-	TRUE if file passed test; FALSE if file failed test.

5.1.12.5 Network Transmission test procedure.

SYNTAX

BOOLEAN Network_PDU_Test_Reception(void)

DESCRIPTION

This function is called for all Level 2 PDU Reception, Adverse, and Erroneous tests. It looks for any non-DIS packet, not generated by the testbed, in the binary file. If a packet is found, an error exists in the logged binary file.

PARAMETERS

None.

RETURNS

BOOLEAN	-	returns the result of the test

5.1.12.6 Network and PDU Reception test procedure.

SYNTAX

BOOLEAN Network_PDU_Test_Reception(void)

DESCRIPTION

This function is called for all Network and PDU Reception, Adverse, and Erroneous tests. It looks for any non-DIS packet, not generated by the testbed, in the logged binary file. If a packet is found, an error exists in the binary file.

PARAMETERS

None.

RETURNS

BOOLEAN	-	returns the result of the test

5.1.13 **AUTONETTEST.C**

Contains the routines for conducting all of the Network-specific tests. There is a routine for parsing a Network configuration file as well as routines for performing all of the Network tests.

5.1.13.1 Process the Network tests, specific configuration file.

SYNTAX

void ProcessNetworkTestConfigFile(NET_DISPLAY_HEADER *netList)

PARAMETERS

NET_DISPLAY_HEADER netList	-	a pointer to the list of recognized network headers.

DESCRIPTION

This function reads the configuration file for the current network test and parses the contents of the file. The file contains a list of fields and the new min/max values for those fields. The contents of the file are used to search the NETWORK DISPLAY HEADER looking for a match. If a match is found the new min/max values from the configuration file are used to overwrite the default min/max values.

RETURNS

None.

5.1.14 AUTOPDUTEST.C

Contains the functions for performing all of the specific PDU tests. There is one routine for properly parsing a PDU configuration file and the rest of the routines are used to perform the actual PDU tests.

5.1.14.1 Process the PDU tests, specific configuration file.

SYNTAX

```
void ProcessPduTestConfigFile(LL  *deflist)
```

PARAMETERS

LL	*deflist	-	a linked list of the format of the current PDU to be tested.

DESCRIPTION

This function reads the configuration file for the current PDU test and searches the "defList" looking for all fields specified in the configuration file. If a matching field is found, the min/max values from the configuration file are used to overwrite the contents in the "defList".

RETURNS

None.

5.1.15 **CO_DIALOG.C**

Create and display the dialog screen that allows for input of the 3 character company abbreviation and the 1 character SUT number.

5.1.15.1 Build the Company abbreviation and SUT number dialog.

SYNTAX

co_dialog_display (Widget main_w)

PARAMETERS

Widget	main_w	-	the main window widget

DESCRIPTION

The function creates the dialog in which the user inputs the 3 character company abbreviation and 1 character SUT number for the new test suite being created. If the dialog window is not already created, it will call function "co_dialog_create" to create the dialog widget.

RETURNS

None.

5.1.16 **CONFWIN.C**

This file contains the functions to create and manage the test suite configuration screen. This is part of the main Scanner window. It contains the current test suite information for the Testbed, Company, and SUT being tested.

5.1.16.1 Create/Display the Configuration management window.

SYNTAX

void display_conf_manager (Widget main_window, MANAGEMENT *mgr)

PARAMETERS

Widget	main_window	-	shell to hang objects from
MANAGEMENT	*mgr	-	company manager

DESCRIPTION

Displays the configuration manager and sets up display.

RETURNS

None.

5.1.16.2 Initialize local management structure window.

SYNTAX

MgmtRec * init_mgmt_rec (MANAGEMENT * m)

PARAMETERS

MANAGEMENT	*m	-	a pointer to the management structure

DESCRIPTION

This function copies values from the Management structure to another structure that is used to populate the screen widgets on the main window, where the Testbed, SUT, and Company information is displayed.

RETURNS

MgmtRec	*	-	a pointer to a locally-created structure that is used for holding the values that will go into the widgets used on the main startup screen (admin area).

5.1.16.3 Copy values from managment structure to screen widgets.

SYNTAX

void populate_conf_manager_display (MgmtRec *mgmtrec)

PARAMETERS

MgmtRec	*mgmtrec	-	management record for company being displayed

DESCRIPTION

Puts values in the screen widgets from the management structure.

RETURNS

None.

5.1.17 FILEMGMT.C

This file contains routines that open and close the logged binary file that is to be used for the next test. The variables to hold the pointer to the FILE structure are statically-defined within this file. BuildIndexTable and LoadPacket are interim functions in this file that pass the locally-defined, static variable UserBinFile to the functions that actually build the Indextable and read a packet from disk, respectively.

5.1.17.1 Try to open logged binary and test configuration files.

SYNTAX

BOOLEAN OpenUserFiles(int tg, int te)

PARAMETERS

int	tg	-	specifies which test group is to be used.
int	te	-	specifies which test entry is to be used.

DESCRIPTION

For the test group (tg), test entry (te) specified, the appropriate binary file and, if not empty, the Config file are opened. The function will first look in the default data directory for the binary files. If it cannot be found, it will look in the directory containing the management structure.

RETURNS

BOOLEAN	-	TRUE if all files opened properly, otherwise, FALSE is returned.

5.1.17.2 Close currently used logged binary and test configuration files.

SYNTAX

BOOLEAN CloseUserFiles(void)

PARAMETERS

None.

DESCRIPTION

Closes the currently opened logged binary file and the configuration file associated with the current, active test.

RETURNS

None.

5.1.17.3 Tells CreateIndexTable() which binary file to use.

SYNTAX

BOOLEAN BuildIndexTable(void)

PARAMETERS

None.

DESCRIPTION

This function calls CreateIndexTable() to create the Indextable with the correct binary file. If the binary file is the same as the last binary file tested, the Indextable has already been created, therefore it does not need to be created again. However, the Packet Count window initialization code is called to display the correct packet counts. The Indextable is not freed every time, therefore, it will remember the previous logged binary file's information.

RETURNS

BOOLEAN	-	TRUE if able to create Indextable, else FALSE.

5.1.17.4 Read test specific configuration file.

SYNTAX

LL *ReadTestConfigFile(void)

PARAMETERS

None.

DESCRIPTION

Reads the information contained in the test-specific configuration file into a linked list that can then be passed to a function for processing.

RETURNS

LL	*	-	Linked list of lines of data found in configuration file.

5.1.18 **LOG_DIALOG.C**

This file contains the logger window support routines.

5.1.18.1 Display the logger window.

SYNTAX

```
void logger_dialog_display ( Widget parent, char * name, XtPointer data )
```

PARAMETERS

Widget	main_w	-	main window widget
char	* name	-	label for dialog
XtPointer	data	-	default Xwindows parameters

DESCRIPTION

Creates and displays the logger dialog window.

RETURNS

None.

5.1.18.2 Open the pipe between the Scanner and the Logger.

SYNTAX

```
logger_open ( char * fifo_name )
```

PARAMETERS

char	* name	-	filename to open

DESCRIPTION

Open the named pipe for writing.

RETURNS

None.

5.1.19 **MANAGE.C**

This file contains the support routines for managing, creating, and updating the management structure.

5.1.19.1 Rebuild the path to the test suite directory

SYNTAX

void RebuildTestBedDirectory(void)

PARAMETERS

None.

DESCRIPTION

This function takes the company's 3-character name and the SUT's 1-digit number and concatenates them together to rebuild and "freshen" the management structures internal directory references. This function is called in several places to make sure that if the user changes the values in the company name and SUT number, the program will reference the appropriate directory.

RETURNS

None.

5.1.19.2 Sets the default values in management structure.

SYNTAX

void ReInitManagement(char *cname, char *sutno)

PARAMETERS

char	*cname	-	contains the company's 3-char name
char	*sutno	-	contains a 1-char SUT number

DESCRIPTION

This function resets the contents of the company, testbed, and sut fields to NULL. The other information is selectively reset to NULL. If the two parameters passed in are not empty, then the data directory, default logged binary test filenames, and specific test configuration files are set to values representing the company abbreviation and SUT number. The numbers for the test file names come from the values specified in the management configuration file.

RETURNS

None.

5.1.19.3 Loads the Management configuration file's contents into memory.

SYNTAX

BOOLEAN LoadManagementConfig(char *filetouse)

PARAMETERS

char *filetouse	-	Name of the management config file to use.

DESCRIPTION

This function opens the management config file and calls the appropriate functions to read the file and parse it up and actually build a default management structure based on the contents in the configuration file.

RETURNS

BOOLEAN	-	FALSE if an error occurred.

5.1.19.4 Fill in the Management structure with any missing tests.

SYNTAX

void CopyDefaultManagementConfig(void)

PARAMETERS

None.

DESCRIPTION

This function copies the default values for each test from the default management structure (defMgrPtr) to the actual management structure (mgrPtr). The management structure has room for many tests for each level of testing, however, there are currently not as many tests defined as there are spaces for tests in the management structure. If the user creates a new test and enters it into "manage.cfg", the default management structure will contain it, but any existing management structure saved to disk will not. Therefore, after an existing management structure is read in from disk into the actual management structure, any test that exists in the default management structure but does not exist in the actual structure needs to be copied to the actual management structure. This function does the checking and copying.

RETURNS

None.

5.1.20 **MGR.C**

Contains the support routines for the "File" main menu item, i.e., New, Close, Save, Delete and Open.

5.1.20.1 Create a new management structure.

SYNTAX

BOOLEAN mgr_new(char * co, char * sut)

PARAMETERS

char	*co	-	The three-character company name
char	*sut	-	The one-digit SUT number

DESCRIPTION

This function tries to create a new management structure using the three-character company name and one-digit SUT number. The company name and SUT number are concatenated to form a unique directory name where the management structure will be stored.

RETURNS

BOOLEAN	-	TRUE if the function was able to create a new management structure

5.1.20.2 Open an existing management structure.

SYNTAX

BOOLEAN mgr_open (char dirtouse)

PARAMETERS

char	dirtouse	-	the name of the file/directory to use

DESCRIPTION

This function tries to open an existing management structure. If a management structure cannot be found in a directory of the same name an error occurs. Before opening the file and reading its contents in to the management structure, the file is locked to avoid one user from writing to the file while another is reading.

After the test suite is opened, a function is called to load any configuration files specified in the test suite that are not already loaded.

RETURNS

BOOLEAN	-	TRUE if opening the file and reading its contents was OK

5.1.20.3 Saves the open management structure to disk.

SYNTAX

```
void mgr_save ( MANAGEMENT  mgr )
```

PARAMETERS

MANAGEMENT mgr	-	the management structure to write to disk

DESCRIPTION

This function tries to write the contents of the management structure to disk. Before writing, the file is locked to avoid corruption of data.

RETURNS

None.

5.1.20.4 Closes the opened management structure.

SYNTAX

```
void mgr_close ( MANAGEMENT  mgr )
```

PARAMETERS

MANAGEMENT	mgr	-	the management structure to write to disk

DESCRIPTION

This function saves the contents of the management structure, closes the report log, and resets the management structure variable back to blanks.

RETURNS

None.

5.1.20.5 Deletes an existing management structure.

SYNTAX

void mgr_delete (char dirtodel)

PARAMETERS

char	dirtodel	-	the name of the file/directory to delete

DESCRIPTION

This function physically removes a management structure and the directory it is loaded in from the disk.

RETURNS

None.

5.1.20.6 Load test suite-specific configuration files.

SYNTAX

void LoadManagementsConfigFiles(void)

DESCRIPTION

IST: SCANNER

This function is called after a each test suite is opened to determine if a new set of configuration files needs to be loaded. Since each test suite can contain it's own unique configuration files, the configuration files need to be loaded each time a new test suite is opened.

PARAMETERS

None.

RETURNS

None.

5.1.21 **NETCONF.C**

Screen and management functions for Network header validation.

5.1.21.1 Destroy Network user edit structure.

SYNTAX

```
void net_conf_destroy (void)
```

PARAMETERS

None.

DESCRIPTION

Frees up the memory reserved for the network user-edit list structure and the associated widgets.

RETURNS

None.

5.1.21.2 Create and display Network user-edit configuration window.

SYNTAX

```
void net_conf_display ( Widget w, NET_DISPLAY_HEADER * list,
                        char * label )
```

PARAMETERS

```
Widget              w       -    parent widget
NET_DISPLAY_HEADER
                    *list   -    list to create
char                *label  -    label for shell
```

DESCRIPTION

Creates a shell widget for the network user edit list and then displays it.

RETURNS

None.

5.1.21.3 Build the Network user edit list.

SYNTAX

NET_DISPLAY_HEADER * buildNetEditList (PDU_DATA_HEADER * deflist)

PARAMETERS

LL	*defNetList	-	pointer to the default Network data array

DESCRIPTION

Create a temporary list of net data elements for editing. The calling routine must supply the default Network list which is then copied to create the temporary list.

RETURNS

LL	*	-	pointer to a copy of the PDU data fields for editing

5.1.21.4 Extract PDU header from Network header.

SYNTAX

PDU_DATA_HEADER * extractPduHeader (NET_DISPLAY_HEADER * header)

PARAMETERS

NET_DISPLAY_HEADER	*header	-	network header

DESCRIPTION

IST: SCANNER

This function extracts the network pdu header from a network user edit list.

RETURNS

PDU_DATA_HEADER	*	-	network header LL

5.1.22 PDUCONFIG.C

This file contains functions for handling the PDU (Level 2) automated test screen. Builds the configuration screens for packet validation and editing. This module also contains functions for building and freeing PDU edit lists.

5.1.22.1 Destroys the PDU user edit configuration screen and widgets.

SYNTAX

```
void pdu_conf_destroy ( void )
```

PARAMETERS

None.

DESCRIPTION

This function destroys the PDU configuration screen and frees the PDU edit list that was used by the PDU configuration screen.

RETURNS

None.

5.1.22.2 Creates and displays the PDU user edit window.

SYNTAX

```
pdu_conf_display ( Widget main_w, LL * pduList, char * label )
```

DESCRIPTION

This function will create and display the PDU edit screen if it is not already displayed.

PARAMETERS

Widget	main_w	-	parent widget for this screen
LL	*pduList	-	A linked list of PDU_DISPLAY_FIELDS

XmString	label	-	Window title

RETURNS

None.

5.1.22.3 Build PDU user edit list.

SYNTAX

LL * buildPduEditList (LL * defaultList)

PARAMETERS

LL	* defaultList	-	pointer to the default PDU data fields

DESCRIPTION

Create a temporary list of pdu data elements for editing. The calling routine must supply the default PDU list which is copied to create the temporary list.

RETURNS

LL	*	-	pointer to a copy of the PDU data fields for editing

5.1.22.4 Free PDU edit list.

SYNTAX

void freePduEditList (LL * editPduList)

PARAMETERS

LL	*editPduList	-	pointer to a linked list of PDU_DISPLAY_FIELDs

DESCRIPTION

This routine will free the PDU edit list created by buildPduEditList().

RETURNS

None.

5.1.22.5 Extract PDU data from PDU list.

SYNTAX

LL * extractPduDataList (LL * displayList)

PARAMETERS

LL	*displayList	-	pointer to the PDU display list

DESCRIPTION

This function extracts the PDU data from a PDU display list. The PDU core routines require pointers to linked list (LL *) of PDU data.

RETURNS

LL	*	-	pointer to the newly created PDU data list

5.1.23 PC_DIALOG.C

Contains the support routines for the Packet Count Window.

5.1.23.1 Create/Display Packet Count Window.

SYNTAX

displayPacketCountWindow ()

PARAMETERS

None.

DESCRIPTION

Creates the shell for the packet count window

RETURNS

None.

5.1.23.2 Update packet count values.

SYNTAX

void updatePacketCounts(enum Applications_Recognized ar, int dispdu)

PARAMETERS

enum Applications_Recogized	ar	-	The type of PDU the current packet contains.
int	dispdu	-	If "ar" is a DIS PDU, then it contains one of the valid DIS PDU number, else it contains -1.

DESCRIPTION

Updates the packet counts in the packet count window. Which packet count widget to update is determined by the values in the parameters.

RETURNS

None.

5.1.23.3 Destroy the Packet Count Window.

SYNTAX

void packet_count_destroy(void)

DESCRIPTION

Destroys the packet count shell widget.

PARAMETERS

None.

RETURNS

None.

5.1.24 PF_DIALOG.C

Contains all of the support routines for the Packet Filter dialog window.

5.1.24.1 Create/Display Packet Filter Window.

SYNTAX

void pf_dialog_display (Widget main_w)

PARAMETERS

Widget	main_w	-	main window to pop up / create

DESCRIPTION

Creates and pops up the packet filter window.

RETURNS

None.

5.1.24.2 Initialize contents of Packet Filter Window.

SYNTAX

void pf_initialize_display(void)

PARAMETERS

None.

DESCRIPTION

Initializes the display after the window is first created.

RETURNS

None.

5.1.24.3 Fetch the contents of the Packet Filter Window.

SYNTAX

DB_FILTER_STRUCT *GetManualFilter(void)

PARAMETERS

None.

DESCRIPTION

Returns a pointer to the manual filter structure.

RETURNS

DB_FILTER_STRUCT *	-	A pointer to the locally defined filter structure that represents the contents of the packet filter window.

5.1.24.4 Fetch the status of selected items in Packet Filter Window.

SYNTAX

IsPFTogglePDUSelected(short i)
ISPFSrcIDSelected(XmString item)
ISPFEntityTypeSelected(XmString item)

PARAMETERS

Either item or index to item.

DESCRIPTION

Basic query functions as to the state of the appropriate object in the packet filter window.

RETURNS

BOOLEAN (T/F, yes/no, etc.)

5.1.24.5 Destroy the Packet Filter Window.

SYNTAX

void pf_dialog_destroy (void)

PARAMETERS

None.

DESCRIPTION

Destroys the packet filter dialog.

RETURNS

None.

5.1.25 **QRY_DIALOG.C**

Contains all of the support routines for the Query dialog window.

5.1.25.1 Create/Display Query Window.

SYNTAX

void qry_dialog_display (Widget main_w)

PARAMETERS

Widget	main_w	-	main window to pop up / create

DESCRIPTION

Creates and pops up the query window.

RETURNS

None.

5.1.25.2 Initialize contents of Query Window.

SYNTAX

void qry_initialize_display(void)

PARAMETERS

None.

DESCRIPTION

Initializes the display after the window is first created.

RETURNS

None.

5.1.25.3 Fetch the contents of the Query Window.

SYNTAX

DB_FILTER_STRUCT *GetQueryFilter(void)

PARAMETERS

None.

DESCRIPTION

Returns a pointer to the query windows filter data.

RETURNS

DB_FILTER_STRUCT *	-	A pointer to the locally defined filter structure that represents the contents of the query window.

5.1.25.4 Destroy the Query Window.

SYNTAX

void qry_dialog_destroy (void)

PARAMETERS

None.

DESCRIPTION

Destroys the query dialog.

RETURNS

None.

5.1.26 **RP_DIALOG.C**

Contains all of the support routines for the report results dialog window.

5.1.26.1 Create/Display the Report Results Window.

SYNTAX

void rp_dialog_display (Widget main_w)

PARAMETERS

Widget	main_w	-	main window to pop up / create

DESCRIPTION

Creates and pops up the report results dialog window.

RETURNS

None.

5.1.26.2 Update Report Results Window with information from management structure.

SYNTAX

void UpdateRpDialog(void)

PARAMETERS

None.

DESCRIPTION

Copies appropriate values from the management structure to the report results dialog.

RETURNS

None.

5.1.26.3 Add the results of the test to the Report Results Window.

SYNTAX

```
void AppendReportReasons(char  *s)
```

PARAMETERS

char	*s	-	reason string to insert

DESCRIPTION

Appends reasons for pass/fail to reason field in Report Results dialog.

RETURNS

None.

5.1.26.4 Assign the test status results to the Report Results Window.

SYNTAX

```
void AssignTestStatus(TEST_STATUS  ts)
```

PARAMETERS

TEST_STATUS	ts	-	test status

DESCRIPTION

Assigns the specified test status (Passed, Failed, Incomplete) in the Report Results dialog test status field.

RETURN

None.

5.1.26.5 Copy information from the Report Results Window to the management structure.

SYNTAX

void UpdateManagementFromRpDialog(void)

DESCRIPTION

Updates the management structure with data from the report dialog.

PARAMETERS

None.

RETURNS

None.

5.1.26.6 Destroy the Report Results Window.

SYNTAX

void rp_dialog_destroy (void)

PARAMETERS

None.

DESCRIPTION

Destroys the report results dialog.

RETURNS

None.

5.1.27 **SC_DIALOG.C**

Build the system configuration screen listing the configuration files to use for this test suite. If the user modifies any of the values in the configuration fields, the new configuration files are loaded and used from then on for that test suite.

5.1.27.1 Create/Display System Configuration Window.

SYNTAX

```
int sc_display ( void )
```

PARAMETERS

None.

DESCRIPTION

This is the main entry for creating the system configuration xWindows screen. If the screen has already been created it will not be recreated, but it will be updated and displayed.

RETURNS

None.

5.1.27.2 Copy information from management structure to System Configuration Screen.

SYNTAX

```
int sc_populate_display (void)
```

PARAMETERS

None.

DESCRIPTION

This function fetches the configuration filenames from the management structure and initializes the display.

RETURNS

None.

5.1.27.3 Destroy the System Configuration Window.

SYNTAX

int sc_destroy (void)

PARAMETERS

None.

DESCRIPTION

This function is used to destroy the configuration screen and destroy all xWindows widgets.

RETURNS

None.

5.1.28 **TG_DIALOG.C**

Builds a configuration screen for each test group and allows the user to select which tests to run.

5.1.28.1 Create/Display Test Selection Configuration Window.

SYNTAX

void tg_conf_display (Widget main_w, int group, char * label)

PARAMETERS

Widget	main_w	-	main window for everything
int	group	-	which test group to display
char	* label	-	label for window

DESCRIPTION

This function displays a window for a specific test group and allows the user to select which tests to run from that test group.

RETURNS

None.

5.1.28.2 Destroy the Test Selection Configuration Window.

SYNTAX

void tg_conf_destroy(void)

PARAMETERS

None.

DESCRIPTION:

Destroys the Test Selection Configuration window.

RETURNS

None.

5.1.29 TL_DIALOG.C

Contains the support routines for managing and displaying the Timeline window.

5.1.29.1 Create/Display the Timeline Window.

SYNTAX

void tl_time_line_display(void)

PARAMETERS

None.

DESCRIPTION

Creates and displays the Timeline window.

RETURNS

None.

5.1.29.2 Freshen the text for the current record highlighted in the Timeline.

SYNTAX

void update_time_line_text(void)

PARAMETERS

None.

DESCRIPTION

Makes sure the textual information displayed at the bottom of the Timeline window is up to date.

RETURNS

None.

5.1.29.3 Delete list of time line entries.

SYNTAX

```
void tl_list_delete(void)
```

PARAMETERS

None.

DESCRIPTION

A list of all entries to appear in the Timeline is built during the creation of the Timeline display. That list must be freed up after the Timeline is built. This function frees the memory reserved by that list.

RETURNS

None.

5.1.29.4 Destroy the Timeline Window.

SYNTAX

```
void tl_dialog_destroy(void)
```

PARAMETERS

None.

DESCRIPTION

Destroys the Timeline window's widget.

RETURNS

None.

5.1.30 **TS_AUTO_DIALOG.C**

This file contains the functions to create the Automated testing windows and callback routines for Automated testing functions.

5.1.30.1 Create the automated test selection dialog.

SYNTAX

ts_auto_dialog_display (Widget main_w)

PARAMETERS

Widget	main_w	-	main window widget

DESCRIPTION

This function is the external interface to create and display the automated test selection window.

RETURNS

None.

5.1.30.2 Fetch the Network user edit list.

SYNTAX

NET_DISPLAY_HEADER *GetUserNetList(void)

PARAMETERS

None.

DESCRIPTION

Returns a pointer to the locally-defined, network-tests user edit list.

RETURNS

NET_DISPLAY_HEADER *	-	pointer to network user edit list

5.1.30.3 Fetch the PDU user edit list.

SYNTAX

```
LL *GetUserPduList(void)
```

PARAMETERS

None.

DESCRIPTION

Returns a pointer to the locally-defined, PDU-tests user edit list.

RETURNS

LL	*	-	pointer to PDU user edit list

5.1.31 **TS_MAN_DIALOG.C**

Contains the routines for creating and displaying the manual test selection dialog and beginning the manual test process.

5.1.31.1 Create manual test selection dialog.

SYNTAX

void ts_man_dialog_display (Widget main_w)

PARAMETERS

Widget	main_w	-	main window widget

DESCRIPTION

Creates and displays the manual test selection dialog.

RETURNS

None.

5.1.32 **TS_DIALOG.C**

Contains the routines for creating a selection dialog of all of the test suites existing on the hard drive that the user can choose from.

5.1.32.1 Create/Display test suite selection dialog.

SYNTAX

```
void ts_dialog_display ( Widget main_w, short doingDelete )
```

PARAMETERS

Widget	main_w	-	main window iwdget
short	doingDelete	-	TRUE if this function is called for deleting a test suite; FALSE if not.

DESCRIPTION

Creates and displays a dialog listing all of the test suites created on the hard drive in the directory you are running the Scanner from.

RETURNS

None.

5.1.33 **RPTLOGGER.C**

Contains the support routines for creating and updating the report log file.

5.1.33.1 Open the report log.

SYNTAX

ReportLogOpen (char * fname)

PARAMETERS

char	*fname	-	File name of the report log to be opened.

DESCRIPTION

Open the report log for output. The report log is opened in an append mode.

RETURNS

None.

5.1.33.2 Close the report log.

SYNTAX

void ReportLogClose(void)

PARAMETERS

None

DESCRIPTION

Closes the report log file.

RETURNS

None.

5.1.33.3 Write the report log header.

SYNTAX

void ReportLogHeader(void)

PARAMETERS

None

DESCRIPTION

Output the report log header, which contains the testbed, company, and SUT information from the management structure.

RETURNS

None.

5.1.33.4 Write the test header to the report log.

SYNTAX

void ReportLogTestHeader(int testGroup, int testEntry)

PARAMETERS

int	testGroup	-	Test group from management structure (Network, PDU,...)
int	testEntry	-	Test entry within test group.

DESCRIPTION

Output a header for the individual test specfied by the two parameters.

RETURNS

None.

5.1.33.5 Write report results to the report log.

SYNTAX

```
void ReportLogResults(int  testGroup, int testEntry)
```

PARAMETERS

int	testGroup	-	Test group from management structure (Network, PDU,...)
int	testEntry	-	Test entry within test group.

DESCRIPTION

Output the test results to the report log file.

RETURNS

None.

5.1.33.6 Write formatted information to the report log.

SYNTAX

ReportLogPrintf() - Print a formatted report message.

PARAMETERS

va_alist	-	variable argument list (format and arguments of string to add)

DESCRIPTION

Print a formatted string to the report log file. This function works the same as printf().
Calling convention:

```
        ReportLogPrintf ("format string", args...);
```

RETURNS

None.

5.1.34 **RPTRESULTS.C**

Generates the Results Summary report and writes it to a text file.

5.1.34.1 Create Results report.

SYNTAX

void ReportResultsSummary (void)

PARAMETERS

None.

DESCRIPTION

Creates a report as a text file. For each test selected in the management structure, the results of the report are written to the report.

RETURNS

None.

5.1.35 **RPTSTATUS.C**

Generates the Results Status report and writes it to a text file.

5.1.35.1 Create Status report.

SYNTAX

void ReportStatusSummary (void)

PARAMETERS

None.

DESCRIPTION

Creates a report as a text file. For each test selected in the management structure, the results of the report are written to the report.

RETURNS

None.

5.1.36 **PACKET_ID.C**

Contains the higher level packet identification code.

5.1.36.1 Identify a packet.

SYNTAX

```
void HandlePacket( int screen_output, FILE *text_file, int packet_dump, int
                   verbose, int interp, void *pkt, long plen, TIME
                   packet_time, PACKET_IDENTIFICATION_STRUCT *pis
                   )
```

PARAMETERS

int	screen_output	-	Ignored.
FILE	*text_file	-	Ignored.
int	packet_dump	-	Ignored.
int	verbose	-	Ignored.
int	interp	-	Ignored.
void	*pkt	-	A pointer to a buffer containing a packet
long	plen	-	The length of the packet
TIME	packet_time	-	The time stamp from the Logger header for the packet
PACKET_IDENTIFICATION_STRUCT	*pis	-	A structure to be filled in by the routines called by HandlePacket(). The structure contains information about what kind of packet it is.

DESCRIPTION

Process an incoming packet to determine what values are in the different protocol layers. The PACKET_IDENTIFICATION_STRUCT is filled in by the lower level routines called from this function. The structure contains the information about what type of network headers are on the packet and what type of packet it is.

RETURNS

None.

5.2 Data Interface

5.2.1 Data Definition Language

Rather than defining specific data structures for each PDU and Network type, and the associated code to process the data structures, the Data Definition Language (DDL) was created to allow generic processing of the various data packets.

Certain fields require special processing. These fields are also defined in the default DDL format, but are identified and processed accordingly during the traversal of the packet definition list. The majority of the fields within the PDUs do not require special processing.

The DDL also allows variable sections to be defined. Variable sections must be grouped. The variable group is only defined once. When the data packet is being processed, the variable group fetches its count parameter from the data packet, and is repeated the appropriate number of times. The data field containing the count parameter must appear in the data packet before the variable section. This allows the generic definition and processing to be used for all possible variations of the currently defined PDU types.

Since the data defined using the DDL is kept in ASCII files and read in at runtime, the keywords and associated data are quoted. This allows spaces to be used in the definitions, and parsing of the configuration line is simplified to reading and parsing of strings. This does not limit the data types that may be defined. To keep the parser simple, there is a requirement that a definition must be contained on a single line. Each line must start with a comment character or a keyword. The parameters for a keyword must be on the same line as the keyword. Keywords may be preceded by spaces or tabs for readability. The parser will skip blank lines and comment lines.

The DDL uses the following key words:

*	This character at the start of a line indicates that this is a comment line.

PDU

This keyword defines the start of a PDU definition. This keyword requires two parameters, PDU name, and PDU number. PDU definitions may not be nested. The PDU keyword does require a corresponding **ENDPDU**.

Usage:	PDU	Keyword
	PDU NAME	ASCII name of the PDU
	PDU NUMBER	PDU Number. Must be unique. Used to index the PDU table

Format:	"PDU", "PDU NAME", "PDU NUMBER"
Example:	"PDU", "ENTITY_STATE", "1"

ENDPDU

This keyword defines the end of a PDU definition. Each PDU keyword requires a corresponding ENDPDU keyword.

Usage:	ENDPDU	Keyword
Example:	"ENDPDU"	

NETWORK

This keyword defines the start of a Network level definition. This keyword requires two parameters, Network name, and Network number. Network definitions may not be nested. They do require a corresponding **ENDNETWORK**.

Usage:	NETWORK	Keyword
	NETWORK NAME	ASCII name of the Network
	NETWORK NUMBER	Network Number. Must be unique. Used to index the Network table

Format :	"NETWORK", "NETWORK NAME", "NETWORK NUMBER"
Example:	"NETWORK", "ETHERNET", "1"
Example:	"NETWORK", "UDP/IP", "1"

ENDNETWORK

This keyword defines the end of a Network definition. Each NETWORK keyword requires a corresponding ENDNETWORK keyword.

Usage:	ENDNETWORK	Keyword
Example:	"ENDNETWORK"	

GROUP

This keyword defines the start of a group. The group may be a fixed group or a variable group. A fixed group defines a group of elements that are to be repeated a known number of times, usually once. A variable group is a group that is to be repeated, but the number of times is determined by a data field within the data packet, and is therefore unknown at this time. If a group name contains the keyword **_BITS**, the group elements are treated as a bitfield. Groups may be nested, but each group requires a corresponding **ENDGROUP**.

Fixed groups:

Usage:	GROUP	Keyword
	GROUP NAME	Group Name, must be unique for this PDU or Network definition.
	FIXED	Keyword
	COUNT	Number of iterations

Example: "GROUP", "PDU_HEADER", "FIXED", "1"

Variable groups:

Usage:	GROUP	Keyword
	GROUP NAME	Group Name, must be unique for this PDU or Network definition.
	VARIABLE	Keyword
	CTRL GROUP	Name of group containing the field with the count value.
	CTRL FIELD	Name of field containing the count value specifying this groups number of copies.

Example: "GROUP", "ART_PARMS", "VARIABLE", "ENTITY_STATE", "NUMBER_ART_PARMS"

ENDGROUP

This keyword defines the end of a group definition.

Usage:	ENDGROUP	Keyword
	GROUP NAME	Group Name

Example: "ENDGROUP", "ART_PARMS"

ELEM

This keyword defines an element, whether it is part of the PDU or a sub-group. An element defines the attributes of a data field. Elements are self contained in one line, and do not require an end type keyword.

Usage:	ELEM	Keyword
	PARENT GROUP	The name of this field's immediate parent

	group.
FIELD NAME	The name of this data field.
DATA TYPE	The data type for this data field. See the list below.
DISPLAY TYPE	Display type for this data field. See list below.
MINIMUM VALUE	The minimum value for this data field, or the word "MIN" which will be translated to the minimum value for this fields data type.
MAXIMUM VALUE	The maximum value for this data field, or the word "MAX" which will be translated to the maximum value for this fields data type.
ENUMERATION TABLE	An associated enumeration table name if there is one, otherwise "NONE".
GROUP DIVISOR	The group divisor is used in calculating the iteration count for variable fields. Not all count fields are a number of iterations. Some are total number of bits, therefore they must be divided by this value to obtain a count. The default value is "1". Only a few PDUs require a different value. See the PDU Configuration File section for a more complete definition.

Example: "ELEM", "PDU_HEADER", "PROTOCOL_VERSION", "U8", "DEC", "1", "4", "PROTOCOL_VERSION", "1"

A list of possible display types:

DEC	Decimal
HEX	Hexadecimal
FLOAT	Decimal
ASCII	ASCII text
DATA	Data block

A list of possible element data types:

U8	Unsigned character
S8	Signed character
U16	Unsigned short
S16	Signed short

U32	Unsigned long
S32	Signed long
F32	32-bit Float
U64	Possibly two unsigned longs
F64	64-bit Float
A	Ascii string with length (A#) A15 means 15 ascii characters.
B	Bit fields (bits#mask#shift) B8#F0#4 means its an 8 bit field, the mask is hex F0 and the right shift count used to align the data is 4.
D	Data area with count (D#) D15 means 15 data bytes.

The following is an example of the Network Configuration File.

```
"NETWORK", "ETHERNET", "1"
   "GROUP", "ETHERNET", "FIXED", "1"
      "ELEM", "ETHERNET", "DESTINATION",     "D6",    "HEX",       "MIN", "MAX", "NONE"
      "ELEM", "ETHERNET", "SOURCE",          "D6",    "HEX",       "MIN", "MAX", "NONE"
      "ELEM", "ETHERNET", "ETHER_TYPE",      "U16",   "HEX",       "0",   "MAX", "NONE"
   "ENDGROUP", "ETHERNET"
"ENDNETWORK"

"NETWORK", "UDP/IP", "2"
  "GROUP", "UDP/IP", "FIXED", "1"

   "GROUP", "IP", "FIXED", "1"

     "GROUP", "IP_HEADER_BITS", "FIXED", "1"
       "ELEM", "IP_HEADER_BITS", "IP HEADER LENGTH", "B8#F0#4",   "DEC", "MIN", "MAX", "NONE"
       "ELEM", "IP_HEADER_BITS", "VERSION",          "B8#0F#0",   "DEC", "MIN", "MAX", "NONE"
     "ENDGROUP", "IP_HEADER_BITS"

     "ELEM", "IP", "TYPE OF SERVICE",      "U8",         "DEC", "MIN", "MAX", "NONE"
     "ELEM", "IP", "TOTAL LENGTH",         "U16",        "DEC", "MIN", "MAX", "NONE"
     "ELEM", "IP", "PACKET ID",            "U16",        "DEC", "MIN", "MAX", "NONE"

     "GROUP", "IP_FRAG_BITS", "FIXED", "1"
       "ELEM", "IP_FRAG_BITS", "FRAGMENT OFFSET",   "B16#FFF8#3","DEC", "MIN", "MAX", "NONE"
       "ELEM", "IP_FRAG_BITS", "FLAGS",             "B16#0007#0","HEX", "MIN", "MAX", "NONE"
     "ENDGROUP", "IP_FRAG_BITS"

     "ELEM", "IP", "TIME TO LIVE",           "U8",        "DEC", "MIN", "MAX", "NONE"
     "ELEM", "IP", "PROTOCOL",               "U8",        "DEC", "MIN", "MAX", "NONE"
     "ELEM", "IP", "HEADER CHECKSUM"         "U16",       "DEC", "MIN", "MAX", "NONE"
     "ELEM", "IP", "SOURCE_ADDRESS[0]",      "U8",        "DEC", "MIN", "MAX", "NONE"
     "ELEM", "IP", "SOURCE_ADDRESS[1]",      "U8",        "DEC", "MIN", "MAX", "NONE"
     "ELEM", "IP", "SOURCE_ADDRESS[2]",      "U8",        "DEC", "MIN", "MAX", "NONE"
     "ELEM", "IP", "SOURCE_ADDRESS[3]"       "U8",        "DEC", "MIN", "MAX", "NONE"
     "ELEM", "IP", "DESTINATION_ADDRESS[0]", "U8",        "DEC", "MIN", "MAX", "NONE"
     "ELEM", "IP", "DESTINATION_ADDRESS[1]", "U8",        "DEC", "MIN", "MAX", "NONE"
     "ELEM", "IP", "DESTINATION_ADDRESS[2]", "U8",        "DEC", "MIN", "MAX", "NONE"
```

```
      "ELEM", "IP", "DESTINATION_ADDRESS[3]", "U8",       "DEC", "MIN", "MAX", "NONE"
    "ENDGROUP", "IP"

    "GROUP", "UDP", "FIXED", "1"
      "ELEM", "UDP", "SOURCE PORT",            "U16",       "DEC","MIN", "MAX", "NONE"
      "ELEM", "UDP", "DESTINATION PORT",       "U16",       "DEC","MIN", "MAX", "NONE"
      "ELEM", "UDP", "LENGTH [HDR+TEXT]",      "U16",       "DEC","MIN", "MAX", "NONE"
      "ELEM", "UDP", "CHECKSUM",               "U16",       "DEC","MIN", "MAX", "NONE"
    "ENDGROUP", "UDP"

  "ENDGROUP", "UDP/IP"
"ENDNETWORK"
```

The following is an example of the PDU Configuration File.

```
"PDU", "ENTITY_STATE_PDU"    1

  "GROUP", "ENTITY_STATE", "FIXED", "1"

    "GROUP", "PDU_HEADER",                   "FIXED",     "1"
        "ELEM", "PDU_HEADER",                "PROTOCOL_VERSION",     "U8",   "DEC",   "1",    "4",     "PROTOCOL_VERSION", "1"
        "ELEM", "PDU_HEADER",                "EXERCISE_ID",          "U8",   "DEC",   "MIN",  "MAX",   "NONE", "1"
        "ELEM", "PDU_HEADER",                "PDU_TYPE",             "U8",   "DEC",   "1",    "1",     "PDU_TYPE", "1"
        "ELEM", "PDU_HEADER",                "PADDING",              "U8",   "DEC",   "0",    "0",     "NONE", "1"
        "ELEM", "PDU_HEADER",                "TIME_STAMP",           "U32",  "HEX",   "MIN",  "MAX",   "NONE", "1"
        "ELEM", "PDU_HEADER",                "LENGTH",               "U16",  "DEC",   "144",  "MAX",   "NONE", "1"
        "ELEM", "PDU_HEADER",                "PADDING",              "U16",  "DEC",   "0",    "0",     "NONE", "1"
    "ENDGROUP", "PDU_HEADER"

    "GROUP", "ENTITY_ID",                    "FIXED",     "1"
        "ELEM", "ENTITY_ID",                 "SITE",                 "U16",  "DEC",   "MIN",  "MAX",   "NONE", "1"
        "ELEM", "ENTITY_ID",                 "APPLICATION",          "U16",  "DEC",   "MIN",  "MAX",   "NONE", "1"
        "ELEM", "ENTITY_ID",                 "ENTITY",               "U16",  "DEC",   "MIN",  "MAX",   "NONE", "1"
    "ENDGROUP", "ENTITY_ID"

    "ELEM", "ENTITY_STATE",                  "FORCE_ID",             "U8",  "DEC",   "0",    "3",     "FORCE_ID", "1"
    "ELEM", "ENTITY_STATE",                  "NUMBER_ART_PARAMS",    "U8",  "DEC",   "MIN",  "MAX",   "NONE", "1"

    "GROUP",   "ENTITY_TYPE",                "FIXED",     "1"
        "ELEM", "ENTITY_TYPE",               "ENTITY_KIND",          "U8",   "DEC",   "0",    "7",     "ENTITY_KIND", "1"
        "ELEM", "ENTITY_TYPE",               "DOMAIN",               "U8",   "DEC",   "MIN",  "MAX",   "NONE", "1"
        "ELEM", "ENTITY_TYPE",               "COUNTRY",              "U16",  "DEC",   "0",    "266",   "COUNTRY", "1"
        "ELEM", "ENTITY_TYPE",               "CATEGORY",             "U8",   "DEC",   "MIN",  "MAX",   "NONE", "1"
        "ELEM", "ENTITY_TYPE",               "SUBCATEGORY",          "U8",   "DEC",   "MIN",  "MAX",   "NONE", "1"
        "ELEM", "ENTITY_TYPE",               "SPECIFIC",             "U8",   "DEC",   "MIN",  "MAX",   "NONE", "1"
        "ELEM", "ENTITY_TYPE",               "EXTRA",                "U8",   "DEC",   "MIN",  "MAX",   "NONE", "1"
    "ENDGROUP", "ENTITY_TYPE"

    "GROUP",   "ALT_ENTITY_TYPE",            "FIXED",     "1"
        "ELEM", "ALT_ENTITY_TYPE",           "ENTITY_KIND",          "U8",  "DEC",   "0",    "7",     "ENTITY_KIND", "1"
        "ELEM", "ALT_ENTITY_TYPE",           "DOMAIN",               "U8",  "DEC",   "MIN",  "MAX",   "NONE", "1"
        "ELEM", "ALT_ENTITY_TYPE",           "COUNTRY",              "U16", "DEC",   "0",    "266",   "COUNTRY", "1"
        "ELEM", "ALT_ENTITY_TYPE",           "CATEGORY",             "U8",  "DEC",   "MIN",  "MAX",   "NONE", "1"
        "ELEM", "ALT_ENTITY_TYPE",           "SUBCATEGORY",          "U8",  "DEC",   "MIN",  "MAX",   "NONE", "1"
        "ELEM", "ALT_ENTITY_TYPE",           "SPECIFIC",             "U8",  "DEC",   "MIN",  "MAX",   "NONE", "1"
        "ELEM", "ALT_ENTITY_TYPE",           "EXTRA",                "U8",  "DEC",   "MIN",  "MAX",   "NONE", "1"
    "ENDGROUP", "ALT_ENTITY_TYPE"

    "GROUP",   "ENTITY_LINEAR_VELOCITY",     "FIXED",     "1"
        "ELEM", "ENTITY_LINEAR_VELOCITY",    "X",                    "F32", "FLOAT",   "MIN",  "MAX",   "NONE", "1"
        "ELEM", "ENTITY_LINEAR_VELOCITY",    "Y",                    "F32", "FLOAT",   "MIN",  "MAX",   "NONE", "1"
        "ELEM", "ENTITY_LINEAR_VELOCITY",    "Z",                    "F32", "FLOAT",   "MIN",  "MAX",   "NONE", "1"
    "ENDGROUP", "ENTITY_LINEAR_VELOCITY"

    "GROUP",    "ENTITY_LOCATION",             "FIXED",    "1"
        "ELEM", "ENTITY_LOCATION",             "X",                      "F64", "FLOAT",   "MIN",  "MAX",    "NONE", "1"
        "ELEM", "ENTITY_LOCATION",             "Y",                      "F64", "FLOAT",   "MIN",  "MAX",    "NONE", "1"
        "ELEM", "ENTITY_LOCATION",             "Z",                      "F64", "FLOAT",   "MIN",  "MAX",    "NONE", "1"
    "ENDGROUP", "ENTITY_LOCATION"

    "GROUP",    "ENTITY_ORIENTATION",          "FIXED",    "1"
        "ELEM", "ENTITY_ORIENTATION",          "PSI",                    "F32", "FLOAT",   "MIN",  "MAX",    "NONE", "1"
        "ELEM", "ENTITY_ORIENTATION",          "THETA",                  "F32", "FLOAT",   "MIN",  "MAX",    "NONE", "1"
        "ELEM", "ENTITY_ORIENTATION",          "PHI",                    "F32", "FLOAT",   "MIN",  "MAX",    "NONE", "1"
    "ENDGROUP", "ENTITY_ORIENTATION"

    "ELEM",     "ENTITY_STATE",                "APPEARANCE",             "U32", "HEX",   "MIN",  "MAX",    "NONE", "1"

    "GROUP",    "DEAD_RECK_PARAMS",            "FIXED",    "1"
        "ELEM", "DEAD_RECK_PARAMS",            "DR_ALGORITHM",           "U8",  "DEC",     "0",    "9",      "DEAD_RECKONING_ALGORITHM", "1"
        "ELEM", "DEAD_RECK_PARAMS",            "OTHER",                  "D15", "HEX",     "0",    "0",      "NONE", "1"
        "ELEM", "DEAD_RECK_PARAMS",            "EN_LIN_ACC.X",           "F32", "FLOAT",   "MIN",  "MAX",    "NONE", "1"
        "ELEM", "DEAD_RECK_PARAMS",            "EN_LIN_ACC.Y",           "F32", "FLOAT",   "MIN",  "MAX",    "NONE", "1"
        "ELEM", "DEAD_RECK_PARAMS",            "EN_LIN_ACC.Z",           "F32", "FLOAT",   "MIN",  "MAX",    "NONE", "1"
        "ELEM", "DEAD_RECK_PARAMS",            "EN_ANG_VEL.X",           "F32", "FLOAT",   "MIN",  "MAX",    "NONE", "1"
        "ELEM", "DEAD_RECK_PARAMS",            "EN_ANG_VEL.Y",           "F32", "FLOAT",   "MIN",  "MAX",    "NONE", "1"
        "ELEM", "DEAD_RECK_PARAMS",            "EN_ANG_VEL.Z",           "F32", "FLOAT",   "MIN",  "MAX",    "NONE", "1"
    "ENDGROUP", "DEAD_RECK_PARAMS"

    "GROUP",    "ENTITY_MARKING",              "FIXED",    "1"
        "ELEM", "ENTITY_MARKING",              "CHARACTER_SET",          "U8",  "HEX",  "MIN",  "MAX",    "ENTITY_MARKING_TABLE", "1"
        "ELEM", "ENTITY_MARKING",              "RECORD",                 "A11", "ASCII", "MIN",  "MAX",    "NONE", "1"
    "ENDGROUP", "ENTITY_MARKING"

    "ELEM", "ENTITY_STATE",                    "CAPABILITIES",           "U32", "HEX",   "MIN",  "MAX",    "NONE", "1"

    "GROUP",    "ART_PARAMS",                  "VARIABLE",    "ENTITY_STATE",    "NUMBER_ART_PARAMS"
        "ELEM", "ART_PARAMS",                  "CHANGE",                 "U16", "DEC",   "MIN",  "MAX",    "NONE", "1"
        "ELEM", "ART_PARAMS",                  "ID",                     "U16", "DEC",   "MIN",  "MAX",    "NONE", "1"
        "ELEM", "ART_PARAMS",                  "TYPE",                   "U32", "HEX",   "MIN",  "MAX",    "NONE", "1"
        "ELEM", "ART_PARAMS",                  "VALUE",                  "U64", "HEX",   "MIN",  "MAX",    "NONE", "1"
    "ENDGROUP", "ART_PARMS"

  "ENDGROUP",  "ENTITY_STATE"

"ENDPDU"
```

5.2.2 Scanner Data Structures

The main data structures used by the Scanner are defined in this section.

5.2.2.1 PDU Data Entry

The PDU_DATA_ENTRY structure is the core of the configuration list data definition structures used to identify fields within a data packet. It is used for both default lists, and mapped lists. The data element is used as a link in the linked lists defining the various packet data definition lists.

Each link in the data definition lists is used to describe either a field or a group. Field entries are contained within this structure. Group entries will create a new linked list for their children. The children may be field and/or group definitions. The chain continues until there are no more group definitions.

```
typedef enum
{
   PDL_DATA_GROUP,
   PDL_DATA_ELEM
} PDL_DATA_TYPE;

typedef enum
{
   PDL_CONTROL_FIXED,
   PDL_CONTROL_VARIABLE
} PDL_CONTROL_TYPE;
```

```
/*
**    Field data types
*/
typedef enum
{
    FIELD_NONE = 0,      /* No field descriptor             */
    FIELD_U8,            /* Unsigned character              */
    FIELD_S8,            /* Signed character                */
    FIELD_U16,           /* Unsigned short                  */
    FIELD_S16,           /* Signed short                    */
    FIELD_U32,           /* Unsigned long                   */
    FIELD_S32,           /* Signed long                     */
    FIELD_F32,           /* 32-bit Float                    */
    FIELD_U64,           /* possibly two unsigned longs     */
    FIELD_F64,           /* (2) 32-bit floats               */
    FIELD_A,             /* Ascii string with length        */
    FIELD_B,             /* Bit fields (bits#mask#shift)    */
    FIELD_D              /* Data area with count            */
} FIELD_DATA_TYPE;

/*
**    Field display types
*/
typedef enum
{
    FDT_DECIMAL  = 0,
    FDT_HEX = 1,
    FDT_FLOAT = 2,
    FDT_ASCII = 3,
    FDT_DATA = 4
} FIELD_DISPLAY_TYPE;


#define GROUP_NAME_SIZE    60
#define FIELD_NAME_SIZE    40
#define FIELD_DATA_SIZE    40
```

The following is a definition of the PDU Data Entry structure.

```
typedef struct _pdu_data
{
   PDL_DATA_TYPE       pdlType;                   /* PDL_DATA_GROUP,          */
                                                  /* PDL_ DATA_ELEM           */
   PDL_CONTROL_TYPE    ctrlField;                 /* PDL_CONTROL_(FIXED,      */
                                                  /* VARIABLE)                */
   short               ctrlGrpDiv;                /* group count divisor      */
                                                  /* (def: 1)                 */
   short               ctrlCount;                 /* FIXED: number of groups  */
   char                ctrlGroup[GROUP_NAME_SIZE]; /* Controlling group       */
   char                ctrlName[FIELD_NAME_SIZE]; /* Controlling group field  */
   char                groupName[GROUP_NAME_SIZE];
                                                  /* PDL_DATA_GROUP: Group    */
                                                  /* PDL_DATA_ELEM:           */
                                                  /*   Parent's group         */
   char                fieldName[FIELD_NAME_SIZE]; /* PDU field name          */
   FIELD_DATA_TYPE     dataType;                  /* Data type of PDU field   */
   FIELD_DISPLAY_TYPE dispType;                   /* Display type of PDU      */
                                                  /* field                    */
   short               dataSize;                  /* Size of data field in    */
                                                  /* bytes                    */
   short               maxLength;                 /* Max chars accepted in X  */
   short               display_len;               /* Size of X display field  */
   unsigned short      offset;                    /* PDU field offset         */
   unsigned short      grpSize;                   /* Size of children in      */
                                                  /* bytes                    */
   unsigned short      index;                     /* Current index into       */
                                                  /*   multiple               */
   unsigned short      ofMaxGroups;               /*   copies of this group   */
   short               bits;                      /* Size of bitfield in      */
                                                  /* bytes                    */
   short               shift;                     /* Shift Right count in     */
                                                  /* bits                     */
   unsigned long       mask;                      /* Bit field mask           */
   char                min[FIELD_DATA_SIZE];      /* Minimum field value      */
   char                max[FIELD_DATA_SIZE];      /* Maximum field value      */
   enum Enum_Enum      enumTable;                 /* Table index for          */
                                                  /* validation               */
   LL                * children;                  /* pointer to a groups      */
```

```
                                        /* children                    */
} PDU_DATA_ENTRY;
```

5.2.2.2 Default Group List

The default group lists are defined in configuration files. The Data Definition Language is used to define the groups and there fields.

The default group lists are a direct translation of the contents found within the configuration files. They will define the format of each variable section, but until a packet is actually mapped to this list, the number of variable parameters is unknown, therefore, at this time the program is unable to completely generate the data lists. The count used to define a variable section is located within the data packet. This structure allows us to duplicate the base variable section (N) number of times, when (N) is read from the data packet.

The following illustration is of a group that has only data field definitions as it's children.

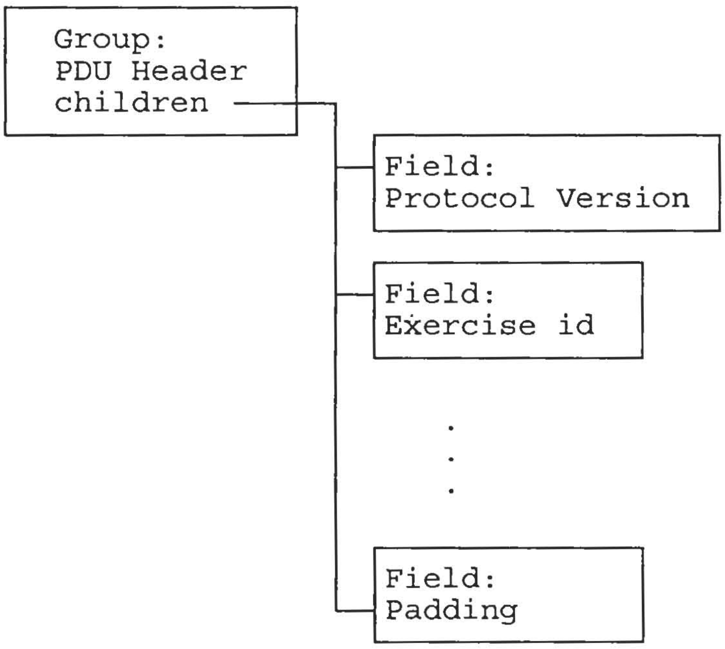


Figure 1: Default Group Listing (Fields only)

The following illustration is of a group that has both groups and data fields as it's children. There is no fixed limit as to how far the group nesting can be defined.

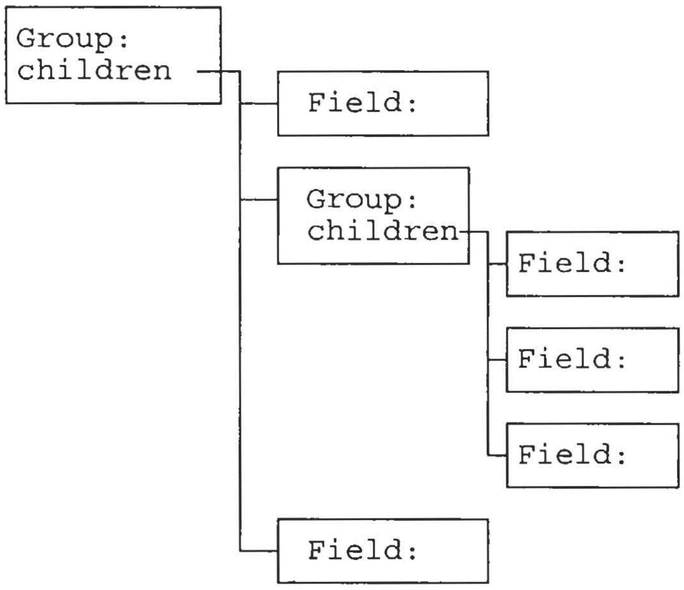


Figure 2: Default Group Listing (Groups and Fields)

5.2.2.3 Mapped Group List

Mapping consists of overlaying either a default configuration list or a user defined configuration list over a section of the data packet. The sections of the data packet are defined by the protocol layers, and their sizes. While mapping the configuration list to the data packet, the values within the data packet are used to resolve variable data section sizes. When mapping a variable data section, the count value is fetched from the data packet, and used to determine how many times to duplicate the variable section in the mapped list. When the data mapping is completed, the newly created mapped list contains a linked list of data fields that cover the data elements within the section of the data packet that was mapped.

The mapped configuration list has the same structure as the default configuration list defined above. The difference is that the variable groups are expanded to meet the requirements of the data packet being mapped to.

The mapped list allows the Scanner to access the fields within a data packet for validation or display purposes.

During mapping, the size of a group is calculated and placed into the group size field of the group structure (PDU_DATA_ENTRY).

5.2.2.4 Default Network Configuration Table

The default network header is defined as an array of PDU_DATA_HEADERs. This allows the same translation code to be used for all packet layers.

The default network table contains both network and transport layer definitions.

The following is a list of supported network and transport layer definitions.

```
enum
{
   NET_8023_TYPE     = 0,
   NET_ETHER_TYPE,
   TL_UDP_IP_TYPE,
   TL_TCP_IP_TYPE,
   TL_ARP_TYPE,
};

/*
```

```
**      Default definition header
*/

typedef struct {
  LL         * children;                      /* pointer to the main PDU group  */
  int          pduNumber;                     /* Number of Network or Transport */
  char         pduName[FIELD_NAME_SIZE];      /* Network enumeration  */
} PDU_DATA_HEADER;
```

5.2.2.5 Default PDU Configuration Table

The default PDU table holds the definitions of DIS PDUs. This is currently the only application layer defined. Other application layer tables may be defined and easily integrated into the Scanner.

The following is a list of possible PDU types used to index the default PDU table.

```
enum
{
    OTHER_PDU_KIND = 0,               /* Types of PDUs */
    ENTITY_STATE_PDU_KIND,
    FIRE_PDU_KIND,
    DETONATION_PDU_KIND,
    COLLISION_PDU_KIND,
    SERVICE_REQUEST_PDU_KIND,
    RESUPPLY_OFFER_PDU_KIND,
    RESUPPLY_RECEIVED_PDU_KIND,
    RESUPPLY_CANCEL_PDU_KIND,
    REPAIR_COMPLETE_PDU_KIND,
    REPAIR_RESPONSE_PDU_KIND,
    CREATE_ENTITY_PDU_KIND,
    REMOVE_ENTITY_PDU_KIND,
    START_RESUME_PDU_KIND,
    STOP_FREEZE_PDU_KIND,
    ACKNOWLEDGE_PDU_KIND,
    ACTION_REQUEST_PDU_KIND,
    ACTION_RESPONSE_PDU_KIND,
    DATA_QUERY_PDU_KIND,
    SET_DATA_PDU_KIND,
    DATA_PDU_KIND,
   EVENT_REPORT_PDU_KIND,
   MESSAGE_PDU_KIND,
   EMISSION_PDU_KIND,
   LASER_PDU_KIND,
   TRANSMITTER_PDU_KIND,
   SIGNAL_PDU_KIND,
   RECEIVER_PDU_KIND
};
```

The default PDU configuration table is defined as an array of PDU_DATA_HEADERs.

```
typedef struct {
 LL        * children;                    /* pointer to the main PDU group  */
 int       pduNumber;                     /* Number of PDU                  */
 char      pduName[FIELD_NAME_SIZE];      /* PDU name ("FIRE...") */
} PDU_DATA_HEADER;
```

5.2.2.6 Logger Header

The logger header is concatenated to the beginning of each logged packet. It contains the time the packet was logged, in seconds and microseconds, and the length of the packet. The following is the definition of the UNIX Logger's header:

```
/*
**      Defines the Timestamp used in the Logger header for the time
**      that each individual packet was created. It is used throughout
**      the program whenever a reference to time is made.
*/
typedef struct
{
        unsigned      long sec;
        unsigned      long usec;
} TIME;


/*
**      The structure of IST's UNIX Logger's header.
*/
typedef struct
```

```
{
        long        size;                   /*    Size of packet to follow        */
        TIME        time;                   /*    Time packet was logged          */
} LOGGER_HEADER;
```

5.2.2.7 Management Structure

The Management structure, also known as a Test Suite, is the structure in which the Scanner holds all of the information about each test and the environment in which the test is run. At the startup of the Scanner, the memory for a Management structure is allocated. The program then reads the default Management configuration file, parses off each line, and copies the data from the line into the appropriate fields in the structure. The configuration file determines what the test group names are and what and how many tests appear in each group. Each line of the configuration file contains a unique test number which, for all Automated tests, is passed to a function that returns the address of the function that will perform the actual test. This way the Management structure knows everything about the test.

The Management structure is also the repository for all test results and comments. That information is input by the user or the program into the report results screen upon completing the test. It is then copied into the appropriate fields in the Management structure. These results are then summarized on either of the two reports created by the Scanner.

To help insure compatibility between versions of the Scanner, when the Management structure is being created, a copy of the data being read from the default configuration file is being saved in a separate default management structure. Then when an existing test suite is opened, if the number of tests and test groups specified in test suite do not correspond with the number in the default management structure the program will automatically add the missing tests and test groups to the existing test suite. This way if the user wants to add a test to an existing test suite, adding a line to the configuration file will solve the problem.

The following is a list of the structures that make up the management structure as well as the management structure itself:

```
/*
*******************************************************************
*   Testbed information structure
*******************************************************************
*/

typedef struct {
        char    tester[80];                 /*    Name of tester                    */
        char    date[9];                    /*    Date of test                      */
        char    terrainDB[80];              /*    Terrain Database name             */
        char    dataDir[FILE_NAME_SIZE];
                                            /*    Directory holding log files       */
        char    testIPA[20];                /*    IP address of test node           */
        char    testETA[28];                /*    Eathernet address of test node    */
        char    ver[20];                    /*    CGF version                       */
} TESTBED;

/*
*****************************************************************
*   Company information structure
*****************************************************************
*/

typedef struct {
        char    name[80];                   /*    Company name                      */
        char    nameAbbr[4];                /*    3 character abbreviation          */
        char    addr1[80];                  /*    Address line 1                    */
        char    addr2[80];                  /*    Address line 2                    */
        char    poc[80];                    /*    Point of Contact                  */
        char    phone[20];                  /*    Phone number of POC               */
        char    email[40];                  /*    Email address                     */
} COMPANY;

/*
*****************************************************************
*   SUT information structure
*****************************************************************
*/

typedef struct {
        char    name[80];                   /*    SUT name                          */
        char    number[2];                  /*    SUT number                        */
        char    sutIPA[20];                 /*    IP address of SUT                 */
        char    sutETA[28];                 /*    Ethernet address of SUT           */
        char    ver[20];                    /*    SUT version                       */
        char    platform[40];               /*    Hardware platform                 */
} SUT;
```

```
/*
*****************************************************************
*    Test Suite Entry information structure
*****************************************************************
*/

#define      REASON_SIZE      322    /*     4x80 plus pad & null termination      */
#define      COMMENT_SIZE     322    /*     4x80 plus pad & null termination      */


typedef enum
{
        N_A           = -1,
        Failed        = 0,
        Passed        = 1,
        Incomplete    = 2
} TEST_STATUS;

typedef struct
{
        char          testNumber[20];                 /*     Test number "1.1.1.1"     */
        char          testDesc[40];                   /*     Test Description          */
        char          testDate[9];                    /*     Date test was last run    */
        char          testType;                       /*     Transmission, Reception   */
        TEST_STATUS
                      testStatus;                     /*     NA  ,  Passed,   Failed,
                                                             Incomplete                */
        char          scheduled;                      /*     Test was selected to be
                                                             run                       */
        short         timesRun;                       /*     Number  of  times  test  was
                                                             run                       */
        short         automated;                      /*     -1 Manual, Index into array
                                                             of automated test
                                                             functions                 */
        int           pduLLoffset;                    /*     offset into linked list
                                                             of default PDU/
                                                             Network values            */
        char          reason[REASON_SIZE];            /*     reason for failure        */
        char          comments[COMMENT_SIZE];         /*     user defined comment      */
        char          binaryLogFile[FILE_NAME_SIZE];
                                                      /*     Logged binary file        */
        char            configFile[FILE_NAME_SIZE];    /*    Configuration file          */
        char            testfileNumber[4];             /*    Unique number in file
                                                             name                        */
} TEST_SUITE_ENTRY;

/*
*******************************************************************
*   Test Group Structure
*******************************************************************
*/

#define         MAX_TEST_ENTRIES        50

typedef struct {
        char                    groupName[40];         /*    Test group name 'PDU'... */
        char                    subgroupName[40];      /*    Name of specific test
                                                             'Transmission'              */
        short                   count;                 /*    Number of test entries      */
        TEST_SUITE_ENTRY        testEntry[MAX_TEST_ENTRIES];
                                                       /*    Each of the specific tests  */
} TEST_GROUP;

/*
*******************************************************************
*   Test Suite Structure
*******************************************************************
*/

#define         MAX_TEST_GROUPS         40

typedef struct {
        int             overall_test_count;            /*    # of tests available to choose
                                                             from                        */
        int             current_tg;                    /*    Current test group being
                                                             tested                      */
        int             current_te;                    /*    Current test entry being
                                                             tested                      */
        short           printed_report_header;
                                                       /*    TRUE if printed report
                                                             header                      */
        short           count;                         /*    Number of test groups       */
        TEST_GROUP          testGroup[MAX_TEST_GROUPS];
                                                        /*      Each of the test group      */
} TEST_SUITE;


#define                 MAX_CONFIG_DIRS         20
#define                 SYSTEM_CONFIG_DIR       0
#define                 NETWORK_DIR             1
#define                 PDU_DIR                 2
#define                 LEVEL_3_DIR             3
#define                 LEVEL_4_DIR             4
#define                 LEVEL_5_DIR             5
#define                 LEVEL_6_DIR             6
#define                 LEVEL_7_DIR             7
#define                 LEVEL_8_DIR             8
#define                 LEVEL_9_DIR             9
#define                 LEVEL_10_DIR            10
#define                 LEVEL_11_DIR            11
#define                 LEVEL_12_DIR            12
#define                 LEVEL_13_DIR            13
#define                 LEVEL_14_DIR            14
#define                 LEVEL_15_DIR            15
#define                 LEVEL_16_DIR            16
#define                 LEVEL_17_DIR            17
#define                 LEVEL_18_DIR            18
#define                 LEVEL_19_DIR            19


/*
**      System configuration information.
*/
typedef struct
{
        char            portNumber[40];
        str80           configDirs[MAX_CONFIG_DIRS];
                                                /*      The name of the directory for the
                                                        configuration file for each level of
                                                        testing.                            */
} System_Config_Struct;

/*
```

```
**      The actual structure of a management configuration file, i.e. test suite.
*/
typedef struct
{
        char                    directory[8];   /*      Name of directory containing test
                                                        suite, i.e., company 3-char name and
                                                        SUT number                          */
        System_Config_Struct    sc;             /*      System configuration directories    */
        long                    tlong1;         /*      Extra field.  Not used.             */
        long                    tlong2;         /*      Extra field.  Not used.             */
        short                   tshort1;        /*      Extra field.  Not used.             */
        short                   tshort2;        /*      Extra field.  Not used.             */
        short                   tshort3;        /*      Extra field.  Not used.             */
        short                   tshort4;        /*      Extra field.  Not used.             */
        char                    str1[20];       /*      Extra field.  Not used.             */
        char                    str2[40];       /*      Extra field.  Not used.             */
        char                    str3[80];       /*      Extra field.  Not used.             */
        TESTBED                 testbed;        /*      Contains the Testbed information.*/
        COMPANY                 company;        /*      Contains the Company
                                                        information.                        */
        SUT                     sut;            /*      Contains the SUT information.       */
        TEST_SUITE              ts;             /*      Contains all test groups and specific
                                                        tests within those groups           */
} MANAGEMENT;
```

5.2.2.8 Default Management Structure

When the Scanner starts up, it creates a management structure from the specified management configuration file. A default management structure is also created. This default management structure is a subset of the overall management structure. This default management structure will be the base line to judge whether any test suite opened in the future is complete or not. If a test suite is opened and it does not contain as many test groups or test entries as the default management structure, the missing elements are copied from the default management structure to the recently opened test suite. This way the user can add tests to an existing management structure by adding new tests or test groups to the management configuration file.

```
typedef struct {
        char            testNumber[20];         /*      Test number "1.1.1.1"               */
        char                testDesc[40];       /*   Test Description                    */
        char                testType;           /*   Transmission, Reception             */
        short               automated;          /*   -1 Manual, Index into array of
                                                     automated test functions            */
        int                 pduLLoffset;        /*   offset into linked list of default  */

                                                /*   PDU/Network values                  */
        char                testfileNumber[4];  /*   Unique number in filename           */
} DEF_TEST_SUITE_ENTRY;


typedef struct {
        char                groupName[40];      /*   Test group name 'PDU'...            */
        char                subgroupName[40];   /*   name of specific test
                                                     'Transmission'                      */
        short               count;              /*   Number of test entries              */
        DEF_TEST_SUITE_ENTRY
                            testEntry[MAX_TEST_ENTRIES];
                                                /*   Each default test entry             */
} DEF_TEST_GROUP;


typedef struct {
        int                 overall_test_count; /*   Number of tests available to
                                                     choose from                         */
        short               count;              /*   Number of test groups               */
        DEF_TEST_GROUP
                            testGroup[MAX_TEST_GROUPS];
                                                /*   Each default test group             */
} DEF_TEST_SUITE;
```

5.2.2.9 Indextable

The Indextable is critical to the speed and functionality of the Scanner Management System. The Indextable is dynamically allocated for each binary file being scanned. The Indextable is an array made up of one record (INDEX_STRUCT) for each packet in the binary file. Each array element of the Indextable contains enough information about it's respective packet in the binary file that when searching or filtering the binary file, only the Indextable needs to be searched. This avoids having to read each packet from disk to do a search, thereby, speeding up the execution of the program. The Indextable is made up of the following structures:

```
/*
**      The format of a DIS standard's version 2.03 PDU header.
*/
typedef struct
{
        unsigned char           version;
        unsigned char           exercise;
        unsigned char           kind;
        unsigned char           unused_8_2;
        unsigned long           time_stamp;
        unsigned short          length;
        unsigned short          unused_16_2;
} DIS_PDU_HEADER;

/*
**      The format of an Entity ID structure used throughout most DIS PDUs.
*/
typedef struct
{
        unsigned short          site;
        unsigned short          application;
        unsigned short          entity;
} entity_ID;

/*
**      The format of an Entity Type structure.
*/
typedef struct                  /* DIS Entity type */
{
        unsigned char           kind;
        unsigned char           domain;
        unsigned short          country;
        unsigned char           category;
        unsigned char           sub_category;
        unsigned char           specific;
        unsigned char           extra;
} ENTITY_TYPE;

/*
```

```
**      Contains information about what comprises the contents of a packet.
*/
typedef struct {
   enum Applications_Recognized  Application;        /* Which App. protocol        */
   enum Networks_Recognized      Network;            /* Which Network protocol     */
   enum Lans_Recognized          Lan;                /* Which LAN protocol         */
   long                          Port;               /* Port # of packet           */
   short                         Brd_Header_Length;
                                                     /* Len. of Brd header         */
   short                         Lan_Header_Length;
                                                     /* Len. of Lan header         */
   short                         Net_Header_Length;
                                                     /* Len. of Net header         */
   short                         DIS_PDU_offset;
                                                     /* Beginning of PDU in packet */
   short                         disType;            /* if DIS PDU, which one      */
   unsigned short                Source_PortNum;
                                                     /* Source's Port #            */
   unsigned short                Dest_PortNum;
                                                     /* Dest.'s Port #             */
   unsigned char                 Source_IP_Addr[4];
                                                     /* Source's IP Address        */

   unsigned char                 Dest_IP_Addr[4];
                                                     /* Destination's IP Address   */
   unsigned char                 Dest_Ether_Addr[6];
                                                     /* Dest.'s Ethernet Address   */
   unsigned short                Ether_Type;         /* Value from Ethernet's Type
                                                            field                  */
   unsigned char                 IP_Protocol;        /* IP Protocol value          */
} PACKET_IDENTIFICATION_STRUCT;


/*
**      Contains the information for one element of the Indextable.
**      The Indextable contains one INDEX_STRUCT for each packet in
**      a binary file.
*/
typedef struct {
        DIS_PDU_HEADER          PDU_Header;         /*     DIS PDU header           */
        ENTITY_TYPE             Entity_Type;        /*     Entity Type for Entity State
```

```
                                                        PDUs                           */
    entity_ID           Entity_ID1;             /*      Entity ID information          */
    TIME                Time_Stamp;             /*      Time packet was logged         */
    unsigned long       Packet_Size;            /*      Length of packet               */
    unsigned long       Offset;                 /*      Offset in binary file of where
                                                        packet begins                  */
    int                 Passed_Initial_Validation;
                                                /*      Indicates whether packet
                                                        passed packet validation       */
    PACKET_IDENTIFICATION_STRUCT
                        Packet_Type;            /*      Misc. information about
                                                        packet                         */
} INDEX_STRUCT;


INDEX_STRUCT            *Indextable;            /*      A dynamically, allocated
                                                        structure containing one
                                                        INDEX_STRUCT record for
                                                        each packet within a binary
                                                        file.                          */
```

5.2.2.10 Filter

The filter structure is used to internally narrow the list of packets the user will see in the Timeline window. For Automated tests, the structure is automatically filled in by the program to only show the packets that are relevant for the test being run. The user cannot change the filter criteria. For Manual tests, the filter structure is initialized to show the user all DIS-only packets not generated by the testbed. The user can change the filter criteria using the Packet Filter window. The Packet Filter windows's data is copied into the filter structure and then the data base functions, FindFirst() and FindNext(), are used to find all packets that meet the criteria specified within the filter.

To determine whether or not a packet meets the criteria specified in the filter, the contents of the filter are compared against the appropriate values in the Indextable. To make the program as fast as possible, the Indextable contains all of the important information necessary to perform the checks between the Indextable and the filter structure.

```
#define         MAX_ENTITY_ID          500
#define         MAX_ENTITY_TYPE        500

/*
**      Used for filtering against the Indextable looking for packets that meet
**      the criteria specified with the DB_FILTER_STRUCT.
*/
typedef struct {
        short             PDU[MAX_PDUS];  /*    Checked if filtering on that specific
                                                PDU                                 */
        short             Search_On_PDU;  /*    TRUE if filtering on PDUs           */
        entity_ID         ID[MAX_ENTITY_ID];
                                          /*    All Entity IDs to filter on         */
        short             Search_On_Ent_ID; /*  TRUE if filtering on Entity IDs     */
        ENTITY_TYPE       Type[MAX_ENTITY_TYPE];
                                          /*    All Entity Types to filter on       */
        short             Search_On_Ent_Type;
                                          /*    TRUE if filtering on Entity
                                                Types                               */
        TIME              Start_Time;     /*    Find all packets after this time    */
        TIME              End_Time;       /*    Find all packets before this time   */
        short             Search_On_Time; /*    TRUE if filtering on Time           */
        short             DIS_Only;       /*    TRUE if only DIS packets pass       */
        short             Excl_Testbed;   /*    TRUE if excluding
                                                testbed-generated packets           */
        short             nonDIS_Only;    /*    TRUE if only non-DIS packets        */
        unsigned short    Exercise_ID;    /*    Find all packets with this Exercise
                                                ID                                  */
        short             Search_On_Exer_ID;
                                          /*    TRUE if filtering on Ex. ID         */
        unsigned short    Version_Num;    /*    Find all packets with this
                                                Version_Num                         */
        short             Search_On_Ver_Num;
                                          /*    TRUE if filtering on Version
                                                Number                              */
        unsigned short    Port_Num;       /*    Find all packets with this Port
                                                Number                              */
        short             Search_On_Port_Num;
                                          /*    TRUE if filtering on Port #         */
} DB_FILTER_STRUCT;
```

5.2.2.11 Automated Test Structure

In order to perform an automated test, there needs to be a function that can be called to perform the actual test. To facilitate this, the addresses of all of the automated test functions have
been loaded into an array of AUTO_TEST_FUNCTIONS structures. Each structure contains an unique identifier that can be compared against the information read in from the configuration files. If the unique identifier read from the configuration file matches with a specific array element, then the offset of that array element is loaded into the management structure. Then when that specific test is activated as an automated test, the program will go to the management structure, fetch that offset, and then activate the automated test defined at that offset.

The AUTO_TEST_FUNCTIONS structure also contains address of a function that will parse the test-specific, configuration file if one is specified, for this test. The "lloffset" field is used for PDU testing to determine which PDU is being tested. The following is a copy of the format of the AUTO_TEST_FUNCTIONS data structure:

```
typedef struct {
        char            uniquestr[8];
        BOOLEAN         (*test)(void);
        void            (*loadconfig)(void *);
        int             lloffset;
} AUTO_TEST_FUNCTIONS;
```

Here are some excerpts from the array that contains all of the automated test definitions:

```
{"T111TA",  Test_T111TA, ProcessNetworkTestConfigFile, 0 },
{"T111RA",  Test_T111RA, NULL, 0 }
{"T111AA",  Test_T111AA, ProcessNetworkTestConfigFile, 0 },
{"T111EA",  Test_T111EA, NULL, 0 },
{"T201TA",  Test_T201TA, NULL, ENTITY_STATE_PDU_KIND},
{"T202TA",  Test_T202TA, NULL, FIRE_PDU_KIND }
```

The first four are the first four Network tests and the last two represent the first two PDU tests. Leaving a NULL in the third parameter is indicating that even though the user may specify a configuration file for that test, the contents of that file will not be parsed or used.

5.2.3 Scanner Configuration Files

The configuration files used by the scanner are defined in this section. The Scanner was designed to be highly configurable at runtime. To do this, there are several text-based configuration files that are loaded when the program starts up. The text-based files allow the user to change the contents of the files, thereby, quickly changing the configuration both in how the program works and how it interprets specific data.

Each test suite can have specific configuration files associated with it different from the default Scanner configuration files. If after opening a test suite the program finds that the names of the configuration files saved within the test suite are different than the default configuration files, the program will automatically destroy the default configuration data and read the data in memory from the appropriate configuration file.

5.2.3.1 Scanner Configuration File

The file contains some of the default information used by the Scanner. The format of the Scanner Configuration file is as follows. Note: The names of the files listed are the current default names used by the Scanner:

```
"MANAGE_CONFIG" = "manage.cfg"

"PORTS" = "6994,3001,3002,3003"

"PDU_CONFIG" = "stdV2Dd.pdu.cfg"

"NETWORK_CONFIG" = "stdV2D3.network.cfg"

"ENTITY_TYPES_CONFIG" = "stdV2D3.entity_types.cfg"

"ENUMERATIONS_CONFIG" = "stdV2D3.enumerations.cfg"

"BINFILE_TIMESTAMP_SIZE" = "0"
```

MANAGE_CONFIG -	Contains the name of the management structure (test suite) configuration file. See Section 5.2.3.2 - Manager Configuration File for format of this file.

PORTS -	Contains a string of four port addresses. The first port is the address of the DIS port. The second, third, and fourth ports are the SIMNET, IST MESSAGE, and EAGLE addresses, respectively. The Scanner uses these numbers internally to determine if a packet is a DIS, SIMNET, IST MESSAGE, or EAGLE packet.
PDU_CONFIG -	Contains the name of the file that defines each DIS PDU using the Data Definition Language. See Section 5.2.1 - Data Definition Language for a description of the format of the configuration file.
NETWORK_CONFIG -	Contains the name of the file that defines the recognized network headers using the Data Definition Language. See Section 5.2.1 - Data Definition Language for a description of the format of the configuration file.
ENTITY_TYPES_CONFIG -	Contains the name of the file holding the list of all of the Entity Types recognized by the Scanner Management System. See Section 5.2.3.5 - Entity Types Configuration File for format of this file.
ENUMERATIONS_CONFIG -	Contains tables of enumerations. Each row of the table contains a min/max range and a text string. The min/max values indicate the range that applies to that enumerated text string.
BINFILE_TIMESTAMP_SIZE -	A timestamp may be written once at the top of a binary file indicating when the binary file was created. This line in the configuration file contains the length of that timestamp. Whether or not a timestamp is included at the beginning of the binary file, this field must have a value that represents the length of the timestamp, inorder for the Scanner to know where the first packet begins.

5.2.3.2 Manager Configuration File

IST: SCANNER

This configuration file is used to fill in the format of a test suite management structure. Initially, the structure has no substance to it. As this configuration file is read in and parsed, it's information is used to determine how many test groups the management structure contains and what the tests are for each group, as well as, specific information about each test. This approach allows the user to re-define a test or group of tests at runtime. Note, however, for automated tests, changes would still need to be made to the executable to actually perform the test differently than the way the test was originally defined. Even this is made relatively easy, since all the user would have to do is just write the new routine and then recompile. Very little or nothing else needs to be done to make the new test functional.

The format of the configuration file is listed and explained below:

```
#GROUP     "NAME"     "SubGroup"
"Number",  "Desc",    "Type",     "Mode",     "Func",     "FileID"
```

The #GROUP moniker is used to distinguish the beginning of a new group of tests. This field is followed by the name of the group, in quotes. The group name is followed by a sub-group name. If there is not a sub-group name, the tag contains "NONE".

Each line that follows lists an individual test for that group. The first field contains the unique test number for that test as defined in IST's document - *Technical Report, Test Documents for DIS Interoperability*. The next column is a textual description of the test. The third field indicates what type of test this test is. The codes translate to "T" for Transmission, "R" for Reception, "A" for Adverse and "E" for Erroneous.

The next column contains either an "A" for Automated tests or a "M" for manual tests. The fifth column is a unique code that is used to get the address of the specific function to perform this test as an automated test. The last column is used during the creation of the test suite. This string, along with the 3-character company name and the SUT number, are combined to come up with a unique logged binary file filename for each test.

The following is a partial sample listing of two groups from the Manager Configuration file:

```
/*****************************************************************
*  Configuration file structure for Management
*
*  GROUP Name : Text name of group ("Network", "PDU",...)
*  Number     : Test number ("1.1.1.1", "1.1.1.2",...)
*  Desc       : Description of test ("Network IP", "Entity State",...)
*   Type      : (T)ransmission, (R)eception, (A)dverse, (E)rroneous
*   Mode       : (A)utomated, (M)anual
*   Func      : Automated test function enum or NONE
*   File ID   : Unique 3 digit file number
*
*   #GROUP NAME
*   Number,   Desc,            Type, Mode, Func
*
*   #GROUP "Network"
*   "1.1.1", "Network IP Test", "T", "A", "T111TA" , "111"
*
**************************************************************/

#GROUP          "Network"             "NONE"
"1.1.1.1.1",    "Broadcast-Transmission",          "T",   "A",   "T111TA",    "111"
"1.1.1.1.2",    "Broadcast-Reception",             "R",   "A",   "T111RA",    "111"
"1.1.2.1.1",    "Broadcast-Adverse",               "A",   "A",   "T111AA",    "111"
"1.1.3.1.1",    "Broadcast-Erroneous",             "E",   "A",   "T111EA",    "111"
...

#GROUP          "PDU"                 "Transmission"
"2.2.1.1",      "Entity State Transmission",       "T",   "A",   "T201TA",    "201"
"2.2.1.2",      "Fire Transmission",               "T",   "A",   "T202TA",    "202"
"2.2.1.3",      "Detonation Transmission",         "T",   "A",   "T203TA",    "203"
...
```

5.2.3.3 Network Default Configuration File

This file contains the format of all of the recognized Network headers defined using the Data Definition Language. See Section 5.2.1 - Data Definition Language for an explanation of the Data Definition Language and the format of the file.

5.2.3.4 PDU Default Configuration File

This file contains the format of all of the DIS PDUs defined using the Data Definition Language. See Section 5.2.1 - Data Definition Language for an explanation of the Data Definition Language and the format of the file.

5.2.3.5 Entity Types Configuration File

The file contains a listing of all Entity Types recognized by the Scanner Management System. Currently, the Scanner only recognizes a fraction of the entire Entity Type list as defined in IST's document - *Enumeration and Bit-encoded Values for use with IEEE 1278.1-1994, Distributed Interactive Simulation -- Application Protocols.* However, the file is easily configurable to include all of the possible Entity Types and the program will automatically adapt to any number of possible Entity Types.

An example of several lines from the configuration file and the format of the file is as follows:

```
{ "1.1.225.1.1.0",  0x2882080c,     "M1*" },
{ "1.1.225.1.1.1",  0x2882080c,     "M1A1" },
{ "1.1.225.1.1.2",  0x2882080c,     "M1A2" },
```

The first column symbolizes the Entity Type code; the second column is the SIMNET code (not used by the Scanner, but is used by some other software programs and was kept for compatibility); and the third column is a text description of the Entity Type code. Each entry was formatted this way so that if it ever needed to be loaded into a static structure within the code, it could be read in as is without having to be changed.

5.2.3.6 Enumerated Values Configuration File

This file contains a listing of all of the enumeration values recognized by the Scanner. The source of this file can be found in IST's document - *Enumeration and Bit-encoded Values for use with IEEE 1278.1-1994, Distributed Interactive Simulation -- Application Protocols*. The file is structure as a series of tables. Each table starts with a line beginning with the "@" character followed by a textual description of the table. The following lines indicate the rows of acceptable values within that table. Each row contains a minimum and maximum value for that row, as well as, the textual description for that range. The min/max range is used for validation checking of an enumerated field value and for finding the correct enumeration text string to return.

The Scanner reads the contents of the file and stores the contents in memory as a linked list of tables, each made up of a linked list of rows from the table. The management structure contains a pointer to the one of the linked list for each field in the management structure that references an enumeration table.

The following is a small subset of the contents of the enuerations configuration file:

```
@PROTOCOL_VERSION
        "1",    "1",    "DIS PDU version "1.0 (May "92)",
        "2",    "2",    "IEEE "1278-1993",
        "3",    "3",    "DIS PDU version "2.0 - third draft (May "93)",
        "4",    "4",    "IEEE "1278.1-1994"
@PDU_FAMILY
        "0",    "0",    "Other",
        "1",    "1",    "Entity Information/Interaction",
        "2",    "2",    "Warfare",
        "3",    "3",    "Logistics",
        "4",    "4",    "Radio Communication",
        "5",    "5",    "Simulation Management",
        "6",    "6",    "Distributed Emission Regeneration"
@DEAD_REACKONING_ALGORITHM
        "0",    "0",    "Other",
        "1",    "1",    "Static (Entity does not move.)",
        "2",    "2",    "DRM(F, P, W)",
        "3",    "3",    "DRM(R, P, W)",
        "4",    "4",    "DRM(R, V, W)",
        "5",    "5",    "DRM(F, V, W)",
        "6",    "6",    "DRM(F, P, B)",
        "7",    "7",    "DRM(R, P, B)",
        "8",    "8",    "DRM(R, V, B)",
        "9",    "9",    "DRM(F, V, B)"
```

5.2.3.7 Network Test-specific Configuration File

The Network Default Configuration file (see Section 5.2.3.3) contains the default range values for each field of all network headers recognized by the Scanner. However, for some tests, the test requires very specific values for certain fields, not just a range of acceptable values. To accomplish this need, the user can specify for each Network test to be run a configuration file that contains values that will override the default values specified in the Network Configuration File.

To facilitate commonality between configuration files, this file is formatted very similarly to the Network Configuration file. See Section 5.2.1 - Data Definition Language for an explanation of the Data Definition Language and the format of the file. Since the program already knows the format, data type, length, etc. about all network fields, this configuration file only needs to specify the group name, field name and the new min/max values for that field.

Note that each Group, Field combination must be entered in the EXACT format that they exist in the "network.cfg" file. Also, the MIN/MAX values must be entered in this file in the same data format as they are to be read from a packet, i.e., if a field is to be read as if it were in hexadecimal format, then it must be entered as "0800" not "2048" to be properly parsed.

The following is an example of a Network Test-specific Configuration File:

```
#Group            Field                        MIN         MAX

"ETHERNET"  "DESTINATION_ADDRESS[0]"       "255"       "255"
"ETHERNET"  "DESTINATION_ADDRESS[1]"       "255"       "255"
"ETHERNET"  "DESTINATION_ADDRESS[2]"       "255"       "255"
"ETHERNET"  "DESTINATION_ADDRESS[3]"       "255"       "255"
"ETHERNET"  "DESTINATION_ADDRESS[4]"       "255"       "255"
"ETHERNET"  "DESTINATION_ADDRESS[5]"       "255"       "255"

"IP"        "DESTINATION_ADDRESS[0]"       "164"       "164"
"IP"        "DESTINATION_ADDRESS[1]"       "217"       "217"
"IP"        "DESTINATION_ADDRESS[2]"       "255"       "255"
"IP"        "DESTINATION_ADDRESS[3]"       "255"       "255"

"ETHERNET"  "ETHER_TYPE"                   "0800"      "0800"

"IP"        "PROTOCOL"                     "17"        "17"

"UDP"       "DESTINATION PORT"             "6994"      "6994"
```

5.2.3.8 PDU Test-specific Configuration File

This configuration file allows the user to specify values for specific fields in specific PDUs that will override the default values for those fields. The format has exactly the same requirements as the Network, test-specific configuration, therefore, see Section 5.2.3.7 - Network Test-specific Configuration File for a detailed description of the format of this file.

5.2.4 Logged Binary File Format

The logged binary (BIN) file discussed below is based on a file logged using IST's UNIX-based logger. The logger captures all packets, DIS and non-DIS, as they are being transferred over the net. Each captured packet is written to a file as is, except that a header is placed at the beginning of the packet. The logger's header is made up of two structures, a long value for holding the length of the packet to follow and two longs to hold the seconds and microseconds of when the packet was logged. See Section 5.2.2.6 for an additional explanation of the logger's header structure.

In one version of IST's logger an additional timestamp is placed once at the beginning of the file when the file is first created. This usually contains the time and date that the file was created. Whether or not that timestamp is included, BINFILE_TIMESTAMP_SIZE parameter in the Scanner Configuration file (Section 5.2.3.1) must contain a value that represents the length of that timestamp.

The packet is read into the Scanner as raw data. After the packet identification code identifies that the packet is a DIS PDU, the DDL (Data Definition Language) structures are "overlayed" onto the data. See Section 5.2.1 for a definition of the Data Definition Language and how it is used.

6 DETAILED DESIGN

6.1 PDU/Network

6.1.1 PDU Table Building - **PDU_TBL**

6.1.1.1 Overview

This module is used to build the default configuration tables for network, transport, and application layers of a DIS data packet.

The following protocol layers are defined:

Network	: Ethernet, IEEE 802.3
Transport	: UDP/IP, TCP/IP
Application	: DIS PDU

6.1.1.2 Data Structures

The default configuration tables used to define the various protocol layers use a common data structure, the PDU_DATA_HEADER. The definition of the PDU_DATA_HEADER can be found in Section 5.2. Each default configuration table is defined as an array of PDU_DATA_HEADERs. Using the same data type allows all tables to be processed using one set of routines.

The PDU_DATA_HEADERs contain pointers to default configuration lists. The default configuration lists are lists of data elements (PDU_DATA_ENTRY) that describe the default structure of a protocol layer. The definition of the PDU_DATA_ENTRY can be found in Section 5.2.

PDU_DATA_ENTRY - This is the base structure used to define each entry in the default configuration lists. The pdlType field is used to specify if the element is a data field, or a group reference. For group references, the fields used to define the groups contents are placed in a new list pointed to by the children field pointer of the group element.

PDU_DATA_HEADER - This is the base structure used to define the default configuration tables. The default configuration tables are arrays of PDU_DATA_HEADERs. Each PDU_DATA_HEADER contains information about a specific protocol layer type, and a pointer to the list of elements describing the protocol layer.

6.1.1.3 Processing

A separate entry point for each table allows for table identification. The parameter to the entry point identifies the configuration file containing the DDL protocol layer definitions.

The following pseudo code describes how a default configuration table is built. The entry point is buildConfigTable():

```
procedure processElem ()
BEGIN
        Initialize a new PDU_DATA_ENTRY structure;
        Parse the input line and fill in the data structure;
        Translate the data type to an enumeration value;
        Translate the display type to an enumeration value;
        Translate the "MIN" and "MAX" strings to numeric values.
        return PDU_DATA_ENTRY structure;
END

procedure processGroup ()
BEGIN
        Allocate a new PDU_DATA_ENTRY;
        Define the PDU_DATA_ENTRY to be a group;
        Parse the input line and initialize the PDU_DATA_ENTRY structure;
        Start a new list for this groups children;
        buildFamilyTree ( PDU_DATA_ENTRY.children );
        return the PDU_DATA_ENTRY structure;
END

procedure buildFamilyTree ( PARENT_PDU_DATA_ENTRY.children )
BEGIN
        WHILE ( NOT EOF ) DO
                Read the next line in the file;
                Parse the line identifying the keyword;
                IF keyword equals ENDPDU     OR
                     keyword equals ENDNETWORK OR
                     keyword equals ENDGROUP   THEN
                        exit while loop;
                ENDIF
                IF keyword equals ELEM THEN
                        element = processElem ();
                        Insert data element information into the linked list;
        ELSEIF keyword equals GROUP THEN
            group = processGroup ();
            Insert group information into linked list;
        ENDIF
    ENDWHILE
END
```

procedure ProcessFile ()

```
BEGIN
    WHILE ( NOT EOF ) DO
        Read the next line in the file;
        Parse the line to identify the keyword PDU or NETWORK;
        IF keyword was found THEN
            Initialize table entry with name and number;
            buildFamilyTree ();
        ENDIF
    ENDWHILE
END
```

procedure buildConfigTable (filename)

```
BEGIN
    Open the configuration file;
    Initialize the default configuration table;
    ProcessFile ();
    Close the configuration file;
END
```

6.1.2 PDU Utility Routines - **PDU_UTIL**

6.1.2.1 Overview

This module allows the default configuration lists to be expanded to overlay the data within the data packet. Configuration lists that do not contain variable sections map directly to a data packet. Configuration lists that contain variable sections must be expanded and then mapped to the data within the data packet.

Before any routine may access the data within the data packet, a configuration list describing the data contents of a protocol layer must be mapped to the protocol layer of the data packet.

6.1.2.2 Data Structures

The data structures used during mapping are the same as the ones used to build a default configuration list. The difference is that the mapped list will duplicate variable sections, giving a complete mapping to the data within a protocol layer. See section 5.2 for a description of the data structures used to define default configuration lists, and mapped configuration lists.

6.1.2.3 Processing

Mapping consists of overlaying either a default configuration list or a user defined configuration list over a section of the data packet. The sections of the data packet are defined by the protocol layers, and their sizes. While mapping the configuration list to the data packet, the values within the data packet are used to resolve variable data section sizes. When mapping a variable data section, the count value is fetched from the data packet, and used to duplicate the variable section in the mapped list. When the data mapping is completed, the newly created mapped list contains a linked list of data fields that cover the data elements within the section of the data packet that was mapped. The offset field of each element in the mapped list contains the elements starting location in the data packet. The group elements of the mapped list are filled in with the size of their children.

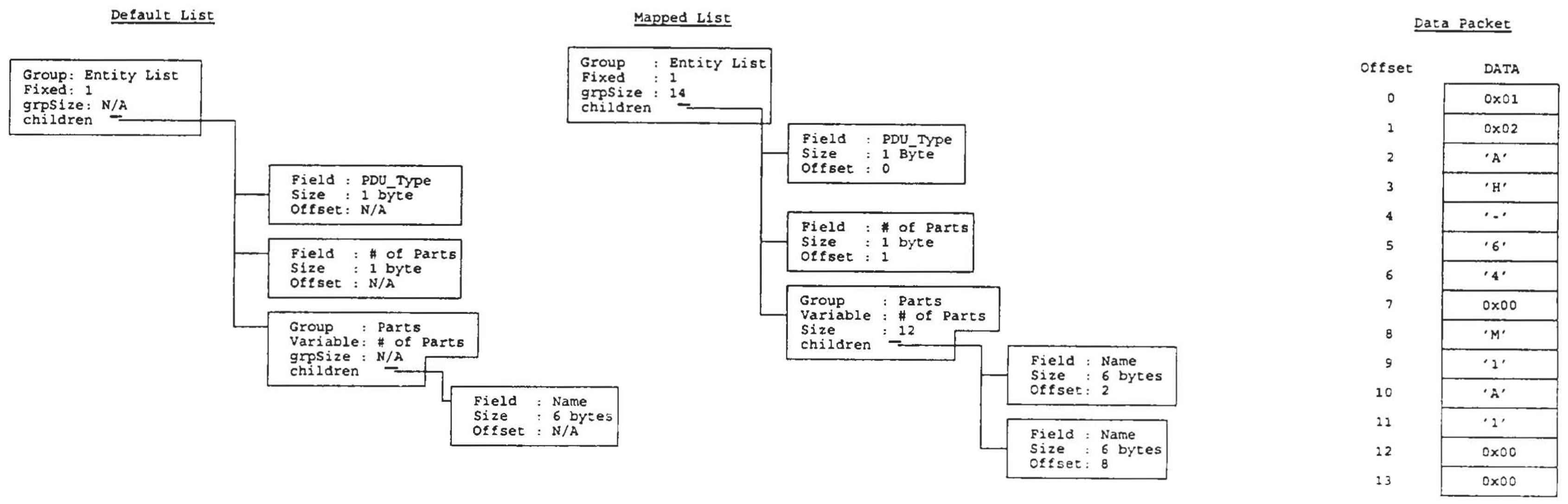


Figure 3: Example of Data Mapping

The above illustration shows the expansion of a default configuration list to a mapped list for a specific section of a data packet. The default list contains a number of fields that are not applicable until the list is mapped to the data packet. During the mapping process, these fields are filled in with the appropriate values. Traversal of the mapped list allows access to the elements of the data packet.

During the traversal of the default configuration lists, when a group entry is encountered, the mapping routines are called recursively to traverse the children of the group.

The following pseudo code describes how a default configuration list is mapped to a data packet.

```
procedure buildMapping ( default list, data packet, mapped_list_size )
BEGIN
    mapped_list = Initialize a new linked list;
    mapped_list_size = 0;
    default_element = Fetch the first element in the default_list;
    WHILE default_element is not equal to NULL DO
        mapped_element = Allocate a new PDU_DATA_ENTRY structure;
        Copy the default_element to the mapped_element;
        IF mapped_element is a GROUP THEN
            IF group type is variable THEN
                ** Note: By the time we hit a variable group definition, the
                ** controlling field has already been defined and mapped
                ** to the data packet.
                Search the mapped_list for the controlling field;
                Fetch the count from the data packet using the offset from the
                controlling field;
            ELSE /* it is a fixed size */
                Fetch the count from the mapped_element;
            ENDIF
            LOOP count TIMES
                mapped_element.offset = current offset;
                mapped_element.children = buildMapping (
                                                default_element.children,
                                                data packet,
                                                size_of_children );
                mapped_element.grpSize = size_of_children;
                mapped_list_size = mapped_list_size + size_of_children;
                Insert the mapped_element into the mapped_list;
            ENDLOOP
        ELSE /* It is a data element */
```

```
                mapped_element.offset = current offset;
                increment current offset by default_element.dataSize;
                increment mapped_list_size by default_element.dataSize;
                Insert the mapped_element into the mapped_list;
            ENDIF
            default_element = Fetch the next element in the default_list;
        ENDWHILE
        return mapped_list, and mapped_list__size;
END
```

6.1.3 PDU Validation Routines - **PDU_VALID**

6.1.3.1 Overview

This module is used to validate the contents of a data packet. Each protocol layer of the packet is validated. Each field within a protocol layer is compared against expected range values for the field. The length of each protocol layer, and the length of the entire packet is also validated.

6.1.3.2 Data Structures

The validation routines map the default lists to mapped lists before validation occurs. There are no new internal structures used by the validation routines.

The following data types may be validated:
- Signed and Unsigned 8-Bit fields
- Signed and Unsigned 16-Bit fields
- Signed and Unsigned 32-Bit fields
- 32-Bit Floating point fields (Single precision)
- 64-Bit Floating point fields (Double precision)
- 8, 16, and 32-Bit fields containing grouped Bit-Fields

Note: The following field types may be defined, but there are no validation routines for these data types:
- Unsigned 64-Bit Fields
- ASCII strings
- Data blocks

6.1.3.3 Processing

Before validation can occur, either the default list or the user edit list must be mapped to the data packet. This allows the validation routines to obtain access to the data values within the data packet.

The following pseudo code describes the basic flow of the validation logic. PacketValidation() is the main entry point for this module.

The compare routines described below are grouped based on similar processing logic.

```
procedure compareUnsigned_8_Bit  ( data_packet, element )
procedure compareSigned_8_Bit     ( data_packet, element )
procedure compareUnsigned_16_Bit  ( data_packet, element )
procedure compareSigned_16_Bit    ( data_packet, element )
BEGIN
        Convert the element.minimum string to a value min;
        Convert the element.maximum string to a value max;
        Fetch the data_value from data_packet[element.offset];

        -- Check the data value against the min and max range.
        -- Then check against the enumeration table.
        -- SearchEnumTable() returns TRUE if the data_value is in the enum table.
        -- SearchEnumTable() returns TRUE if the enum table does not exist.
        IF data_value >= min AND data_value <= max THEN
                return SearchEnumTable ( element.enumTable, data_value );
        ENDIF
        return FALSE;
END

procedure compareUnsigned_32_Bit  ( data_packet, element )
procedure compareSigned_32_Bit    ( data_packet, element )
procedure compareFloat_32_Bit     ( data_packet, element )
procedure compareFloat_64_Bit     ( data_packet, element )
BEGIN
        Convert the element.minimum string to a value min;
        Convert the element.maximum string to a value max;
        Fetch the data_value from data_packet[element.offset];

        -- Check the data value against the min and max range.
        IF data_value >= min AND data_value <= max THEN
                return TRUE;
        ELSE
                return FALSE;
        ENDIF
END

procedure compareGroupedBitFields ( data_packet, element )
BEGIN
        Convert the element.minimum string to a value min;
        Convert the element.maximum string to a value max;

        -- Pull the bit field value from the group element
        Fetch the group data_value from data_packet[element.offset];
        Mask the data_value using element.mask;
        Shift the data_value to the right element.shift times;

        -- Check the data value against the min and max range.
        -- Then check against the enumeration table.
        -- SearchEnumTable() returns TRUE if the data_value is in the enum table.
        -- SearchEnumTable() returns TRUE if the enum table does not exist.
        IF data_value >= min AND data_value <= max THEN
                return SearchEnumTable ( element.enumTable, data_value );
        ENDIF
        return FALSE;
END
```

procedure validateField (data_packet, element)

```
BEGIN
        status = TRUE;
        status = Call the comparison routine for this data type;
        return status;
END
```

procedure groupValidation (data_packet, element)

```
BEGIN
        status = TRUE;
        status = Call the special group validation routine if it is a special group;
        return status;
END
```

procedure validateFields (data_packet, mapped_list)

```
BEGIN
        element = Fetch the first element in mapped_list;
        WHILE element is not equal to NULL DO
                IF element.pdlType equals ELEMENT THEN
                        status = validateField ( data_packet, element );
                        Record any errors to the report log;
                ELSE  -- It is a group
                        -- Check for special group processing
                        status = groupValidation ( data_packet, element );
                        -- Traverse the groups children validation them
                        status = validateFields ( data_packet, element.children );
                ENDIF
        element = Fetch the next element in mapped_list;
    ENDWHILE
    return status;
END
```

procedure packetValidationList (data_packet, default_list, length)

```
BEGIN
    mapped_list = mapListToData ( default_list, data_packet, length );
    status = validateFields ( data_packet, mapped_list );
    destroyMapping ( mapped_list );
    return status;
END
```

procedure packetNetworkValidation (data_packet, network_table, app_list, pkt_id, length)

```
BEGIN
    status = TRUE;
    CASE pkt_id.Lan DO
        ETHERNET    :
          status = packetValidationList ( ...
                      network_table[NET_ETHER_TYPE].children  ...);

        IEEE_8023   :
          status = packetValidationList ( ...
                      network_table[NET_8023_TYPE].children  ...);
    ENDCASE
    return status;
END
```

procedure packetTransportValidation (data_packet, network_table, app_list, pkt_id, length)

```
BEGIN
    status = TRUE;
    CASE pkt_id.Network DO
        TCP_IP      :
          status = packetValidationList ( ...
                      network_table[TL_TCP_IP_TYPE].children  ...);

        UDP_IP      :
          status = packetValidationList ( ...
                      network_table[TL_UDP_IP_TYPE].children  ...);
```

```
            NOVELL      :
                                    -- No default Table defined as of yet
            FILE_TRANSFER:
                                    -- No default Table defined as of yet
            AP_NETWORK  :
                                    -- No default Table defined as of yet
        ENDCASE
        return status;
END
```

procedure packetApplicationValidation (data_packet, app_list, pkt_id, length)

```
BEGIN
        status = TRUE;
        CASE pkt_id.Application DO
            DIS PDU     :
                    Supply the default DIS application list for a NULL parameter;
                    status = packetValidationList ( ... );
                    IF network_length + transport_length + application_length
                      is not equal to data_packet_length THEN
                          status = FALSE;
                    ENDIF
            SIMNET      :
            EAGLE       :
            IST_MESSAGE :
            OTHER_DIS   :       -- No default Table defined as of yet
        ENDCASE
        return status;
END
```

procedure PacketValidation (data_packet, network_table, app_list, pkt_id, length)

```
BEGIN
        Supply the default network table if the parameter network_table is NULL;
        packetNetworkValidation ( ... );
        packetTransportValidation ( ... );
        packetApplicationValidation ( ... );
        IF all layers passed validation THEN
            return TRUE;
        ELSE
            return FALSE;
        ENDIF
END
```

6.1.4 PDU Display Routines - **PDU_DISPLAY**

6.1.4.1 Overview

This module is used to display the contents of a data packet in the Packet Display Window. Each protocol layer of the packet is displayed. Each field within a protocol layer is validated against expected range values for the field. If a field fails validation, an asterisk (*) appears next to the field being displayed.

6.1.4.2 Data Structures

The display routines map the default lists to mapped lists before the packet is displayed. During the display process, all display text is built into a large display buffer. This buffer is written to the Packet Display Window at the end of the display processing. X Windows was visibly too slow when we tried to display each line independently. Displaying the buffer in one X Window call is very fast.

The packet display routines will display the following data types:

- Signed and Unsigned 8-Bit fields
- Signed and Unsigned 16-Bit fields
- Signed and Unsigned 32-Bit fields
- Unsigned 64-Bit fields
- 32-bit Floating point fields (single precision)
- 64-bit Floating point fields (double precision)
- Ascii string with length (A#), A15 means 15 ascii characters.
- Bit fields (bits#mask#shift), B8#F0#4 means its an 8 bit field, the mask is hex F0 and the shift count used to align the data is 4.
- Data area with count (D#), D15 means 15 data bytes.

The display routines will also handle special groups and fields, if they are defined in the special group table, or special field table. They must also have routines for handling them.

6.1.4.3 Processing

Before a packets contents can be displayed, the default lists must be mapped to the data packet. This allows the display routines to obtain access to the data values within the data packet.

The display buffer must be initialized to NULL before the display routines start dumping the elements. After the data packet has been processed, the display buffer is written to the Packet Display Window.

The following pseudo code describes the basic flow of the display logic. PacketDisplay() is the main entry point for this module.

The dump routines described below are grouped based on similar processing logic.

```
procedure hexDumpUnsigned_8_Bit  ( data_packet, element )
procedure hexDumpSigned_8_Bit    ( data_packet, element )
procedure hexDumpUnsigned_16_Bit ( data_packet, element )
procedure hexDumpSigned_16_Bit   ( data_packet, element )
procedure hexDumpUnsigned_32_Bit ( data_packet, element )
procedure hexDumpSigned_32_Bit   ( data_packet, element )
BEGIN
        Fetch the data_value from data_packet[element.offset];
        string = fetchEnumTableStr ( element.enumTable, data_value );

        print to a buffer (element.name, data_value in hex, data_value in decimal, string);
        return buffer and string;
END

procedure hexDumpUnsigned_64_Bit   ( data_packet, element )
BEGIN
        Fetch the data_value from data_packet[element.offset];
        string = NULL;
        print to a buffer ( element.name, first 32-bits of data_value,
                        second 32-bits of data_value );
        return buffer and string;
END

procedure hexDumpFloat_32_Bit    ( data_packet, element )
procedure hexDumpFloat_64_Bit    ( data_packet, element )
BEGIN
        Fetch the data_value from data_packet[element.offset];
        print to a buffer ( element.name, data_value );
        string = NULL;
        return buffer and string;
END

procedure hexDumpAscii ( data_packet, element )
BEGIN
        LOOP element.dataSize TIMES
                Fetch the next ASCII character from
                data_packet[element.offset + LOOP index];
            Add ASCII character to ASCII_string;
        ENDLOOP
        print to a buffer ( element.name, ASCII_string );
        string = NULL;
        return buffer and string;
END
```

procedure hexDumpBitField (data_packet, element)

```
BEGIN
        Fetch data_value from data_packet[element.offset];
        Mask data_value with element.mask;
        Shift data_value right element.shift times;
        string = fetchEnumTableStr ( element.enumTable, data_value );
        print to a buffer ( element.name, data_value in hex,
                data_value in decimal, string );
        return buffer and string;
END
```

procedure hexDumpData (data_packet, element)

```
BEGIN
        LOOP element.dataSize TIMES
            Fetch the next character from
                data_packet[element.offset + LOOP index];
            Add hex representation of character to data_string;
        ENDLOOP
        print to a buffer ( element.name, data_string );
        string = NULL;
        return buffer and string;
END
```

procedure dumpField (data_packet, element)

```
BEGIN
        validStatus = ValidateField ( data_packet, element );
        enumString, buffer = Call hexDump () routine based on element.dataType;
        print to a buffer ( validStatus, buffer, enumString );
        return TRUE;
END
```

procedure groupDisplay (data_packet, element, dumpChildren)

```
BEGIN
```

```
        status = TRUE;
        dumpChildren, status =      Call the special group display routine if it is a
                                    special group;
        return dumpChildren and status;
END
```

procedure dumpFields (data_packet, mapped_list)

```
BEGIN
        element = Fetch the first element in mapped_list;
        WHILE element is not equal to NULL DO
                IF element.pdlType equals ELEMENT THEN
                        -- Check for special field processing
                        IF element requires special processing THEN
                                status = processSpecialField ( data_packet, element );
                        ELSE
                                status = dumpField ( data_packet, element );
                        ENDIF
                        Print the display line that was just built;
                ELSE  -- It is a group
                        -- Check for special group processing
                        status = groupDisplay ( data_packet, element );
                        -- Traverse the groups children displaying them
                        status = dumpFields ( data_packet, element.children );
                ENDIF
                element = Fetch the next element in mapped_list;
        ENDWHILE
        return TRUE;
END
```

procedure disDump (data_packet, default_list, length)

```
BEGIN
        mapped_list = mapListToData ( default_list, data_packet, length );
        status = dumpFields ( data_packet, mapped_list );
        destroyMapping ( mapped_list );
        return status;
END
```

procedure packetNetworkDump (data_packet, network_table, app_list, pkt_id, length)

```
BEGIN
        status = TRUE;
        CASE pkt_id.Lan DO
        ETHERNET    :
           Print header ( " ----- Ethernet Network Layer ----- " );
           status = disDump ( ...
                network_table[NET_ETHER_TYPE].children  ...);

        IEEE_8023   :
           Print header ( " ----- IEEE 802.3 Network Layer ----- " );
           status = disDump ( ...
               network_table[NET_8023_TYPE].children  ...);
    ENDCASE
    return status;
END
```

procedure packetTransportDump (data_packet, network_table, app_list, pkt_id, length)

```
BEGIN
    status = TRUE;
    CASE pkt_id.Network DO
        TCP_IP      :
           Print header ( " ----- TCP/IP Transport Layer ----- " );
           status = disDump ( ...
               network_table[TL_TCP_IP_TYPE].children  ...);

        UDP_IP      :
           Print header ( " ----- UDP/IP Transport Layer ----- " );
           status = disDump ( ...
               network_table[TL_UDP_IP_TYPE].children  ...);

        NOVELL      :
           Print header ( " ----- Novell Transport Layer ----- " );
                       -- No default Table defined as of yet
        FILE_TRANSFER:
           Print header ( " ----- File Transfer Transport Layer ----- " );
                       -- No default Table defined as of yet
        AP_NETWORK  :
           Print header ( " ----- AP Transport Layer ----- " );
                       -- No default Table defined as of yet
    ENDCASE
    return status;
END
```

procedure packetApplicationDump (data_packet, app_list, pkt_id, length)

```
BEGIN
        status = TRUE;
        CASE pkt_id.Application DO
                DIS PDU     :
                        Print header ( " ----- DIS Simulation PDU ----- " );
                        Fetch the default DIS appList if appList is NULL;
                        disDump ( data_packet, appList, pkt_id.appLength );
                SIMNET      :
                        Print header ( " ----- SIMNET Application Layer ----- " );
                                -- No default Table defined as of yet
                EAGLE       :
                        Print header ( " ----- EAGLE Application Layer ----- " );
                                -- No default Table defined as of yet
                IST_MESSAGE :
                        Print header ( " ----- IST Message ----- " );
                                -- No default Table defined as of yet
                OTHER_DIS   :
                        Print header ( " ----- Experimental DIS PDU ----- " );
                                -- No default Table defined as of yet
        ENDCASE
        return status;
END
```

```
procedure PacketDisplay ( data_packet, network_table, app_list, pkt_id, length )
BEGIN
        Supply the default network table if the parameter network_table is NULL;

        Initialize the Packet Display Window;
        Display the hex dump of the data packet;
        packetNetworkDump ( ... );
        packetTransportDump ( ... );
        packetApplicationDump ( ... );
        Display the packet information in the Packet Display Window;
END
```

6.1.4.4 Sample Packet Display Output

The following is a sample from the Packet Display Window. The data being displayed is an Entity State PDU with 2 variable parts.

```
      ----- Hex Dump of Packet -----

  0:  FF FF FF FF FF FF 00 60 8C 55 B1 7B 08 00 45 00 | .......`.U.{..E.
 10:  00 CC 00 00 00 00 3C 11 23 6E A4 D9 11 01 A4 D9 | ......<.#n......
 20:  FF FF 1B 52 1B 52 00 B8 31 D7 03 01 01 00 A1 47 | ...R.R..1......G
 30:  E4 64 00 B0 00 00 11 00 01 00 64 01 02 01 01 01 | .d.........d....
 40:  00 E0 01 03 00 00 00 00 00 00 00 00 00 00 00 00 | ................
 50:  00 00 00 00 00 00 00 00 00 00 C1 44 61 92 06 DB | ...........Da...
 60:  21 18 C1 50 E2 0E F3 7B 89 34 41 4C 69 1E FC D7 | !..P...{.4ALi...
 70:  CE 4F 3F 83 8C D7 BF 79 05 D3 40 46 F4 63 00 00 | .O?....y..@F.c..
 80:  00 00 02 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
 90:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ................
 A0:  00 00 00 00 00 00 00 00 00 00 01 48 31 30 30 00 | ...........H100.
 B0:  00 00 00 00 00 00 00 00 00 03 00 01 00 00 00 00 | ................
 C0:  10 0B 80 00 00 00 00 00 00 00 00 01 00 01 00 00 | ................
 D0:  11 4D 00 00 00 00 00 00 00 00                   | .M........

      ----- Ethernet Network Layer -----

ETHERNET:
      DESTINATION                        : DATA: FF FF FF FF FF FF
      SOURCE                             : DATA: 00 60 8C 55 B1 7B
      ETHER_TYPE                         : x0800       (2048)


      ----- UDP/IP Transport Layer -----

UDP/IP:

IP:

IP_HEADER_BITS:
      IP HEADER LENGTH                   : x04         (4)
      VERSION                            : x05         (5)

      TYPE OF SERVICE                    : x00         (0)
      TOTAL LENGTH                       : x00CC       (204)
      PACKET ID                          : x0000       (0)

IP_FRAG_BITS:
      FRAGMENT OFFSET                    : x0000       (0)
      FLAGS                              : x0000       (0)

      TIME TO LIVE                       : x3C         (60)
      PROTOCOL                           : x11         (17)
      HEADER CHECKSUM                    : x236E       (9070)
      SOURCE_ADDRESS[0]                  : xA4         (164)
      SOURCE_ADDRESS[1]                  : xD9         (217)
      SOURCE_ADDRESS[2]                  : x11         (17)
      SOURCE_ADDRESS[3]                  : x01         (1)
      DESTINATION_ADDRESS[0]             : xA4         (164)
      DESTINATION_ADDRESS[1]             : xD9         (217)
      DESTINATION_ADDRESS[2]             : xFF         (255)
      DESTINATION_ADDRESS[3]             : xFF         (255)


UDP:
      SOURCE PORT                        : x1B52       (6994)
      DESTINATION PORT                   : x1B52       (6994)
      LENGTH [HDR+TEXT]                  : x00B8       (184)
      CHECKSUM                           : x31D7       (12759)


      ----- DIS Simulation PDU -----
```

```
ENTITY_STATE:

PDU_HEADER:
      PROTOCOL_VERSION                  : x03          (3)            DIS PDU version 2.0 - third draft (May 93)
      EXERCISE_ID                       : x01          (1)
      PDU_TYPE                          : x01          (1)            Entity State
      PADDING                           : x00          (0)
      TIME_STAMP                        : xA147E464    (2705843300)
      LENGTH                            : x00B0        (176)
      PADDING                           : x0000        (0)


ENTITY_ID:     17:1:100
      SITE                              : x0011        (17)
      APPLICATION                       : x0001        (1)
      ENTITY                            : x0064        (100)

      FORCE_ID                          : x01          (1)            Friendly
      NUMBER_ART_PARAMS                 : x02          (2)

ENTITY_TYPE:     1.1.224.1.3.0.0     Challenger MBT
      ENTITY_KIND                       : x01          (1)            Platform
      DOMAIN                            : x01          (1)            Land
      COUNTRY                           : x00E0        (224)          United Kingdom
      CATEGORY                          : x01          (1)            Tank
      SUBCATEGORY                       : x03          (3)
      SPECIFIC                          : x00          (0)
      EXTRA                             : x00          (0)


ALT_ENTITY_TYPE:     0.0.0.0.0.0.0     Zero IN, Zero OUT
      ENTITY_KIND                       : x00          (0)            Other
      DOMAIN                            : x00          (0)
      COUNTRY                           : x0000        (0)            Other
      CATEGORY                          : x00          (0)
      SUBCATEGORY                       : x00          (0)
      SPECIFIC                          : x00          (0)
      EXTRA                             : x00          (0)


ENTITY_LINEAR_VELOCITY:     Sqrt(x2 + y2 + z2) = 0.0000000000E+00
      X                                 : 0.0000000000E+00
      Y                                 : 0.0000000000E+00
      Z                                 : 0.0000000000E+00


ENTITY_LOCATION:
      X                                 : -2.6713960535622947E+06
      Y                                 : -4.4257878044150360E+06
      Z                                 : 3.7238379753358732E+06


ENTITY_ORIENTATION:
      PSI                               : 1.0277355909E+00
      THETA                             : -9.7274512052E-01
      PHI                               : 3.1086661816E+00

      APPEARANCE                        : x00000000  (0)
                                          DIS PAINT SCHEME UNIFORM
                                          DIS MOBILITY KILL NONE
                                          DIS FIREPOWER KILL NONE
                                          DIS DAMAGE NONE
                                          DIS SMOKE NONE
                                          DIS TRAILING EFFECTS NONE
                                          DIS HATCH NONE
                                          DIS LIGHTS NONE
                                          DIS FLAMES NONE
                                          DIS ENTITY ACTIVE
                                          DIS LAUNCHER NOT RAISED
                                          DIS DESERT CAMOUFLAGE
                                          DIS NOT CONCEALED
```

```
DEAD_RECK_PARAMS:
      DR_ALGORITHM                        : x02         (2)            DRM(F, P, W)
      OTHER                               : DATA: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      EN_LIN_ACC.X                        : 0.0000000000E+00
      EN_LIN_ACC.Y                        : 0.0000000000E+00
      EN_LIN_ACC.Z                        : 0.0000000000E+00
      EN_ANG_VEL.X                        : 0.0000000000E+00
      EN_ANG_VEL.Y                        : 0.0000000000E+00
      EN_ANG_VEL.Z                        : 0.0000000000E+00


ENTITY_MARKING:
      CHARACTER_SET                       : x01         (1)            ASCII
      RECORD                              : ASCII: H100.......

      CAPABILITIES                        : x00000003   (3)
          Ammo Supply:  True
          Fuel Supply:  True
          Recovery:     False
          Repair:       False

ART_PARAMS [1]:
      CHANGE                              : x0001       (1)
      ID                                  : x0000       (0)
      TYPE                                : x0000100B   (4107)
      VALUE                               : x80000000 x00000000


ART_PARAMS [2]:
      CHANGE                              : x0001       (1)
      ID                                  : x0001       (1)
      TYPE                                : x0000114D   (4429)
      VALUE                               : x00000000 x00000000
```

The following is a sample from the Packet Display Window. The data being displayed is an Emission PDU. The emitter name was out of range according to the data in the default PDU configuration file. The invalid field was marked with an asterisk.

```
     ----- Hex Dump of Packet -----

  0:   FF FF FF FF FF FF 00 60 8C 55 B1 7B 08 00 45 00 | .......'.U.{..E.
 10:   00 80 00 0E 00 00 3C 11 23 AC A4 D9 11 01 A4 D9 | ......<.#.......
 20:   FF FF 1B 52 1B 52 00 6C 5B C6 03 01 17 00 F8 C3 | ...R.R.l[.......
 30:   66 D2 00 64 00 00 00 11 00 01 00 64 00 11 00 01 | f..d.......d....
 40:   00 01 01 01 00 00 12 01 00 00 00 18 03 01 00 00 | ................
 50:   00 00 00 00 00 00 00 00 00 00 0D 01 00 35 4E 8F | .............5N.
 60:   0D 18 00 00 00 00 42 DC 00 00 44 7A 00 00 40 00 | ......B...Dz..@.
 70:   00 00 00 00 00 00 40 49 0F DB 00 00 00 00 00 00 | ......@I........
 80:   00 00 00 00 00 00 01 00 00 00 00 00 00 00       | ..............

     ----- Ethernet Network Layer -----

ETHERNET:
      DESTINATION                      : DATA: FF FF FF FF FF FF
      SOURCE                           : DATA: 00 60 8C 55 B1 7B
      ETHER_TYPE                       : x0800       (2048)


     ----- UDP/IP Transport Layer -----

UDP/IP:

IP:

IP_HEADER_BITS:
      IP HEADER LENGTH                 : x04         (4)
      VERSION                          : x05         (5)

      TYPE OF SERVICE                  : x00         (0)
      TOTAL LENGTH                     : x0080       (128)
      PACKET ID                        : x000E       (14)

IP_FRAG_BITS:
      FRAGMENT OFFSET                  : x0000       (0)
      FLAGS                            : x0000       (0)

      TIME TO LIVE                     : x3C         (60)
      PROTOCOL                         : x11         (17)
      HEADER CHECKSUM                  : x23AC       (9132)
      SOURCE_ADDRESS[0]                : xA4         (164)
      SOURCE_ADDRESS[1]                : xD9         (217)
      SOURCE_ADDRESS[2]                : x11         (17)
      SOURCE_ADDRESS[3]                : x01         (1)
      DESTINATION_ADDRESS[0]           : xA4         (164)
      DESTINATION_ADDRESS[1]           : xD9         (217)
      DESTINATION_ADDRESS[2]           : xFF         (255)
      DESTINATION_ADDRESS[3]           : xFF         (255)


UDP:
      SOURCE PORT                      : x1B52       (6994)
      DESTINATION PORT                 : x1B52       (6994)
      LENGTH [HDR+TEXT]                : x006C       (108)
      CHECKSUM                         : x5BC6       (23494)

     ----- DIS Simulation PDU -----

EMISSION:

PDU_HEADER:
      PROTOCOL_VERSION                 : x03         (3)         DIS PDU version 2.0 - third draft (May 93)
      EXERCISE_ID                      : x01         (1)
      PDU_TYPE                         : x17         (23)        Electromagnetic Emission
```

```
        PADDING                                 : x00         (0)
        TIME_STAMP                              : xF8C366D2   (4173555410)
        LENGTH                                  : x0064       (100)
        PADDING                                 : x0000       (0)


EMITTING_ENTITY_ID:     17:1:100
        SITE                                    : x0011       (17)
        APPLIC                                  : x0001       (1)
        ENTITY                                  : x0064       (100)


EVENT_ID:
        SITE                                    : x0011       (17)
        APPLICAT                                : x0001       (1)
        ENTITY                                  : x0001       (1)

        STATE_UPDATE_INDICATOR                  : x01         (1)
        NUMBER_OF_SYSTEMS                       : x01         (1)
        PADDING                                 : x0000       (0)

SYSTEMS:
        SYSTEM_DATA_LENGTH                      : x12         (18)
        NUMBER_OF_BEAMS                         : x01         (1)
        PADDING                                 : x0000       (0)

EMITTER_SYSTEM:
 *      EMITTER_NAME                            : x0018       (24)
        FUNCTION                                : x03         (3)          ESF Naval Surveillance
        ID_NUMBER                               : x01         (1)


LOCATION:
        X                                       : 0.0000000000E+00
        Y                                       : 0.0000000000E+00
        Z                                       : 0.0000000000E+00


BEAMS:
        BEAM_DATA_LENGTH                        : x0D         (13)
        BEAM_ID_NUMBER                          : x01         (1)
        BEAM_PARAMETER_INDEX                    : x0035       (53)

FUND_PARAM_DATA:
        FREQUENCY                               : 1.2000000000E+09
        FREQ_RANGE                              : 0.0000000000E+00
        ERP                                     : 1.1000000000E+02
        PRF                                     : 1.0000000000E+03
        PULSE_WIDTH                             : 2.0000000000E+00
        BEAM_AZ_CENTR                           : 0.0000000000E+00
        BEAM_AZ_SWEEP                           : 3.1415927410E+00
        BEAM_EL_CENTR                           : 0.0000000000E+00
        BEAM_EL_SWEEP                           : 0.0000000000E+00
        BEAM_SWP_SYNC                           : 0.0000000000E+00

        BEAM_FUNCTION                           : x01         (1)          Search
        NUM_OF_TARGET_IN_TRACK_JAM              : x00         (0)
        HIGH_DENSITY_TRACK/JAM                  : x00         (0)
        PADDING                                 : x00         (0)
        JAMMING_MODE_SEQUENCE                   : x00000000   (0)
```

6.2 Test Suite

6.2.1 Load Management Structure - **MANAGE**

6.2.1.1 Overview

The management structure or test suite is basically an array of arrays. The array of test groups is made up of an array of specific test entries. A test group, for example, is PDU Transmission tests; a specific test entry might be Entity State or Fire PDU. The configuration file determines how many test groups there are and how many individual test entries each test group has. The management structure is accessed by the management pointer, **mgrPtr**. A separate structure is maintained to access the default management structure, **defMgrPtr**.

When a user saves a test suite for a SUT, that test suite knows the name of the configuration files currently active and saves the name of them within the test suite. When that test suite is opened at a later time, the current configuration in memory is removed and replaced with the configuration specified by the configuration files saved within the test suite. Obviously, if the test suite's configuration is the same as the current configuration in memory, the test suite's configuration is not reloaded. However, if the configurations are the same but the test suite after being read in from disk does not contain as many test entries for each test group as the default management configuration file specifies, the test suite is automatically adjusted to include the missing test entries. This is done so that, for example, if a new test is to be added to a test suite, it can be added to the end of the list in the configuration file and program will automatically add it to the test suite. Note that it only applies up to the maximum test groups and test entries allowed. Also note that the test groups could be updated similarly, however, the test selection menu under the Edit menu would not reflect the new test groups.

6.2.1.2 Data Structures

See Section 5.2 for a detailed description of the following data structures:

MANAGEMENT - This is the "grouping" structure that encapsulates the test groups, test entries, configuration information. This is the actual structure that is written to disk when a test suite is written to disk. If the format of this structure changes, special routines will need to be written to read or convert existing test suites.

SYSTEM_CONFIG_STRUCT - Contains the port numbers for recognizing DIS, SIMNET, Eagle, and IST Message packets as well as names of the configuration files used to create this test suite.

TEST_SUITE - Enough space is allocated for MAX_TEST_GROUPS with MAX_TEST_ENTRIES each, however, as the management configuration file is being read in, only as many of the allotted spaces are used as required.

DEF_TEST_SUITE - It is very similar to the TEST SUITE structure but contains only the minimum fields necessary to save the important test group and test entry information.

6.2.1.3 Processing

The following code describes how the management structure configuration file is read in from disk to create a structure within memory.

```
procedure LoadManagementConfig(filetouse)
BEGIN
    Open filetouse;
    DO WHILE NOT EOF()
        Get line from file;
        Process line;
    ENDDO
    Close filetouse;
END

procedure ProcessConfigLine(line)
BEGIN
    IF too many tests or comment line
        RETURN
    ENDIF
    IF line is a GROUP identifier line
        IF too many test groups
            RETURN
        ENDIF
        Increment count of test groups;
        Fetch test group name from line;
        Fetch test sub-group name from line;
        Assign group name to management structure and default management structure;
        Assign sub-group name to management structure and default management
        structure;
    ELSEIF line is a test ENTRY line
        Initialize test ENTRY fields;
        Fetch all of the fields from the line;
        Convert automated test functions index to enumerated value;
```

```
                Assign values or converted values to both mangement and default management
                structure;
        ENDIF
END
```

6.2.2 Create a Test Suite - **CO_DIALOG** and **MGR**

6.2.2.1 Overview

See section 6.2.1 and it's subsections for a detailed explanation of the management and default management structure. Before the actual creation of a test suite management structure, the user needs to supply the program with a three-character company name and a one-character SUT number. These values are used to create a unique identifier for the test suite by contcatenating the two values together. The concatenated string is used as the name of thc test suite, the directory where the test suite is stored, and the beginning characters of the default binary filenames.

6.2.2.2 Data Structures

Same data structures as mentioned in section 6.2.1.1.

6.2.2.3 Processing

The following pseudo code shows the creation of the dialog for inputting the company name and SUT number, as well as, how the new test suite is created.

```
procedure co_dialog_display(main_widget)
BEGIN
    IF company_sut_widget DOES NOT Exist
        Create Widget(main_w);
    ENDIF
    Display Widget;
END

procedure co_dialog_create (main_widget)
BEGIN
    Define widget;
    Create widget;
    Attach fields to widget;
    Attach buttons and callbacks to widget;
    RETURN widget;
END

procedure co_dialog_ok_cb(widget, client_data, call_data)
BEGIN
    IF able to create a new management structure
```

```
                Display Company, SUT, and Testbed data;
                Enable main menu;
        ENDIF
        Destroy widget;
END

procedure mgr_new(co, sut)
BEGIN
        Convert co and sut to uppercase;
        Concatenate co and sut together;
        Try to make a directory with the concatenated string;
        IF NOT able to make directory
                ERROR;
                RETURN FALSE;
        ENDIF
        Create a test suite file named co + sut + ".mgt" in the new directory;
        IF NOT able to create test suite
                ERROR;
                RETURN FALSE;
        ENDIF
        Close new test suite file;
        Reinitialize the management structure in memory to use co and sut;
        Copy Scanner's current config options to test suite structure;
        Create and open Report Log;
        Save test suite to disk;
END
```

6.2.3 Select a Test Suite - **TS_DIALOG**

6.2.3.1 Overview

Selecting a test suite is necessary when opening an existing test suite or deleting an existing test suite. To select a test suite requires building a list of all of the test suites available, adding that list to a selection dialog, and then allowing the user to select one of the tests. The criteria for determining what should be added to the list of test suites is as follows, a directory must exist below the current directory that starts with 3 uppercase characters followed by a number. Also, within that directory, there must be a file with the same 3 character, 1 number name followed by a ".mgt" extension.

6.2.3.2 Data Structures

See Section 5.2 for a detailed description of the following data structures:

MANAGEMENT - This is the "grouping" structure that encapsulates the test groups, test entries, configuration information. This is the actual structure that is written to disk when a test suite is written to disk. If the format of this structure changes, special routines will need to be written to read or convert existing test suites.

SYSTEM_CONFIG_STRUCT - Contains the port numbers for recognizing DIS, SIMNET, Eagle, and IST Message packets as well as names of the configuration files used to create this test suite.

TEST_SUITE - Enough space is allocated for MAX_TEST_GROUPS with MAX_TEST_ENTRIES each, however, as the management configuration file is being read in, only as many of the allotted spaces are used as required.

DEF_TEST_SUITE - It is very similar to the TEST SUITE structure but contains only the minimum fields necessary to save the important test group and test entry information.

6.2.3.3 Processing

The next several routines show how a test suite selection dialog is created.

```
/*      doingDeleteFlag will be FALSE for all calls to this function except when calling from
        the 'Delete a Test Suite' option */
procedure ts_dialog_display (MainWindowWidget, doingDeleteFlag)
BEGIN
        IF test suite selection dialog is not already popped up
```

```
            Set file counter to 0;
            Set list of files to NULL;
            Create test suite selection dialog;
            Pop it up;
        ENDIF
    END
```

```
procedure ts_dialog_create (MainWindowWidget, doingDeleteFlag)
BEGIN
        Create Widget;
        Set parameters within Widget;
        Attach text and button Widgets;
        Create a list of test suites on disk;
        Add contents of test suite list to Widget sorted in alphabetical order;
        RETURN Widget;
END

procedure ts_list_create()
BEGIN
        Scan current directory looking for sub-directories;
        IF FOUND
                Check to see if a file exists in the directory with the same name as the directory
                with a ".mgt" extension;
                IF TRUE
                        Copy that directory name to global list of test suites;
                        Increment file counter;
                ENDIF
        ENDIF
END

/* When user selects/highlights a test suite in the list */
procedure ts_select_cb (Widget, client_data, call_data)
BEGIN
        Get the index of the selected test suite from list;
        Copy the value from the selected test suite to a static global testsuitename variable;
END

procedure ts_dialog_ok_cb(Widget, client_data, call_data)
BEGIN
        IF nothing is SELECTED
                RETURN
```

```
    ENDIF
    Fetch just the test suite name out of the static variable, testsuitename;
    IF able to Open the selected management structure
        Display Company, SUT, and Testbed data;
        Enable main menu;
    ENDIF
    Pop down widget;
    Free memory reserved by Widgets;
    Free memory reserved by test suite list;
END
```

6.2.4 Display contents of a Test Suite - **CONFWIN**

6.2.4.1 Overview

On the main menu screen after a test suite is opened, the testbed, company, and SUT information for that test suite is displayed within it's own framed area. The information is copied from the management structure to a temporary structure containing widgets for the necessary screen interface.

6.2.4.2 Data Structures

The MgmtRec temporary structure used in the following routines is a list of label fields for each field in each section of the test suite and a count of the number entries for each section. The sections include company, testbed, and SUT. The structure is allocated the first time the function is called. Every other time, the data from the test suite being processed overwrites the previous data for each field.

6.2.4.3 Processing

```
procedure display_conf_manager(main_widget, *managementPtr)
BEGIN
    IF temporary structure to hold widgets and text DOES NOT Exist
        Create temporary structure (init_mgmt_rec());
        Create Widgets to hold Company, SUT, and Testbed information;
    ENDIF
    Copy values from temporary structure to widgets;
    Display widgets in main menu window;
END


procedure init_mgmt_rec (managementPtr)
BEGIN
    Allocate a temporary structure to hold the widgets and text that will be displayed in the
    main menu window;
    Copy values from the Management Structure to the temporary structure;
    RETURN a pointer to the temporary structure;
END

procedure populate_conf_manager_display(managementPtr)
BEGIN
    FOR I := 1 TO # of fields in testbed area
```

```
        Copy value from text string to widget;
    NEXT
    FOR I := 1 TO # of fields in company area
        Copy value from text string to widget;
    NEXT
    FOR I := 1 TO # of fields in SUT area
        Copy value from text string to widget;
    NEXT
END
```

6.2.5 Open a Test Suite - **MGR**

6.2.5.1 Overview

Opening a test suite is in reality opening the selected test suite management structure file, reading the contents into memory, and then closing the file.

6.2.5.2 Data Structures

Same data structures as mentioned in Section 6.2.1.2.

6.2.5.3 Processing

Once the test suite is selected (see Section 6.2.3), the following pseudo code shows how the test suite is actually "opened".

```
procedure mgr_open (filetouse)
BEGIN
        Close open test suite, if any;
        IF filetouse DOES NOT Exist
                ERROR;
                RETURN FALSE;
        ENDIF
        IF NOT able to open file
                ERROR;
                RETURN FALSE;
        ENDIF
        Try to Lock file;
        IF NOT able to Lock file
                ERROR;
                RETURN FALSE;
        ENDIF
        Read contents of file into management pointer;
        UnLock file;
        Close file;

        Set Ports to management structures Port settings;
        Open Report Log;
        Verify completeness of management structure (CopyDefaultManagementConfig());
        Load management configuration files;
        RETURN TRUE;
```

```
END

procedure CopyDefaultManagementConfig()
BEGIN
    FOR I :=    management structures test group count TO default management
                structures test group count
        FOR J :=    management structures test count TO default management
                    structures test count
            Copy missing tests from default management structure;
        NEXT
        Copy missing tests groups from default management structure;
    NEXT
END

procedure LoadManagementsConfigFiles()
BEGIN
    Compare currently loaded port settings against the management structures port settings;
    IF different
        Set Ports to management structures Port settings;
    ENDIF
    FOR I := 1 TO MAX_CONFIGURABLE_FILES
        Compare currently loaded config file[I] against the management structures
        config file[I];
        IF different
            IF management config file DOES NOT Exist
                ERROR;
            ELSE
                Copy value from management structure to currently loaded file
                listing;
                Free the memory for the old configuration file and Load the new
                configuration file;
        ENDIF
    NEXT
END
```

6.2.6 Save a Test Suite - **MGR**

6.2.6.1 Overview

See Section 6.2.1 and it's subsections for a detailed explanation of the management and default management structure. Saving a test suite is equivalent to writing the contents of the management structure in memory to a file on disk. The file is locked beforehand to ensure data integrity. If the user has changed the name of the company or the SUT number during the current session, the program will update it's internal reference as to which directory to reference to find files. However, the program will still expect binary and configuration files to be named using the old company name and SUT number combination.

6.2.6.2 Data Structures

Same data structures as mentioned in Section 6.2.1.2.

6.2.6.3 Processing

```
procedure mgr_save (managementPtr)
BEGIN
    IF no management structure open
        RETURN
    ENDIF
    Rebuild reference to Testbed directory;
    IF NOT able to open management structure file
        ERROR;
        RETURN FALSE;
    ENDIF
    IF NOT able to Lock file
        ERROR;
        RETURN FALSE;
    ENDIF
    Write contents of management structure to disk;
    Unlock and Close file;
END
```

```
procedure RebuildTestBedDirectory()
BEGIN
    Replace internal management structure value for directory with the concatenation of the
    current values of the Company and SUT Number fields;
END
```

6.2.7 Close a Test Suite - **MGR**

6.2.7.1 Overview

See Section 6.2.1 and it's subsections for a detailed explanation of the management and default management structure. Closing a test suite also means closing the report log and setting a flag indicating that there are no open test suites.

6.2.7.2 Data Structures

Same data structures as mentioned in Section 6.2.1.2.

6.2.7.3 Processing

```
procedure mgr_close (managementPtr *)
BEGIN
    IF no management structure open
        RETURN
    ENDIF
    Save current test suite to disk;
    Close Report Log;

    /* to indicate not management structures are open */
    Set internal management filename to NULL;
    Clear management structure pointer;
END
```

6.2.8 Delete a Test Suite - **MGR**

6.2.8.1 Overview

See Section 6.2.1 and it's subsections for a detailed explanation of the management and default management structure. To delete a test suite, the program will remove the directory created to store the test suite and also delete <u>ALL</u> files within that directory.

6.2.8.2 Data Structures

Same data structures as mentioned in Section 6.2.1.2.

6.2.8.3 Processing

Once a test suite is selected (see Section 6.2.3), the following pseudo code shows how the selected test suite is deleted.

```
procedure mgr_delete (dirtodel)
BEGIN
        IF trying to delete the open test suite
                RETURN
        ENDIF

        Execute the system command "rm -r " + dirtodel to remove directory and all files in
        directory;
END
```

6.3 Configuration

There are several different configuration files that are used by the Scanner Management System. For a detailed description of the Network or PDU configuration files and how they are used, see Section 6.1 - PDU/Network. For the Management structure configuration file, see Section 6.2 - Test Suite. The following is a listing of the types of configuration files and how they work.

6.3.1 Scanner Configuration file - **MAINWIN**

6.3.1.1 Overview

The Scanner Configuration file lists the management, pdu, network, and entity type default configuration filenames. These are the names of the configuration files that the Scanner will use when creating a new Test Suite. The port numbers for DIS, SIMNET, Eagle, and IST Message can also be found in this configuration file. There is one other field for the binfile timestamp size. This field contains a value indicating how much of the data at the beginning of a binary file is reserved for the time stamp for when the binfile was logged. For example, if this field contains a 0, then the beginning of the file equates to the beginning of the first packet.

6.3.1.2 Data Structures

SYSTEM_CONFIG_STRUCT - The structure contains two fields. One is for the string containing the port numbers for indicating if a packet is DIS, SIMNET, Eagle, or IST Message. The other field is an array containing the filename of the different configuration files the Scanner recognizes. Currently, most of them are ignored, but the first three contain values. The first position contains the name of the Management configuration file, "manage.cfg", the second contains the Network definition configuration file, "network.cfg", and the third contains the PDU definition file, "pdu.cfg".

6.3.1.3 Processing

The pseudo code below summarizes how the Scanner main configuration file is read and loaded.

```
procedure LoadScannerConfig()
BEGIN
    Set the local variable, scandir to the path set by the environmental variable, SCANPROJDIR;
```

```
        Try to open "scanner.cfg" in current directory or SCANDIR directory;
        IF ERROR
                RETURN FALSE;
        ENDIF
        DO WHILE NOT EOF()
                Get line;
                Process line;
        ENDDO
        Close "scanner.cfg";
END
/*
**  See Section 5.2.3.1 - Scanner Configuration File for format of a line.
*/
procedure ProcessScannerLine(line)
BEGIN
        Fetch the config. option from line;
        Fetch the config. value from line;
        Compare config. option against all recognizable options;
        IF match was found
                Copy the config. value into the appropriate SCANNER_CONFIG_STRUCT
                field;
        ENDIF
END
```

6.3.2 Entity Types Configuration file - **ENTTYPES**

6.3.2.1 Overview

To facilitate easy modification of the entity type list as defined in IST's document - *Enumeration and Bit-encoded Values for use with IEEE 1278.1-1994, Distributed Interactive Simulation -- Application Protocols*, the Scanner loads the entity type list from a configuration file. See Section 5.2.3.5 - Entity Types Configuration File for the format of the file. The contents of the file are read into a linked list and stored in alphabetical order. After the contents of the file have been read into the linked list, the contents of the linked list are copied to an array so that they can be binary searched to see if an entity type is valid or not.

6.3.2.2 Data Structures

DIS_ENUMS - Contains the entity type code string, "1.2.225.3.4", the simnet guise as a long, a character description of the entity type, and a conversion of the entity type code string into a string of unsigned characters. An array of these structures is created to hold all of the recognizable entity types.

DIS_ENUMS_LL - Contains a DIS_ENUM structure with two pointers to DIS_ENUM_LL structures, a next and previous pointer. The linked list of recognizable entity types is created with these structures.

6.3.2.3 Processing

Shows how to load the Entity Type default configuration file. Also, the routines to convert the initial format for saving the data in a linked list into an array are listed.

```
procedure LoadEntTypeEnumTable(filename)
BEGIN
    IF NOT able to open filename
        IF NOT able to open filename in Scanner directory
            RETURN FALSE;
        ENDIF
    ENDIF

    Load the contents of the file into a linked list;
    Close file;

    Convert linked list to an array;
END
```

```
procedure LoadTable(file)
BEGIN
    DO WHILE NOT EOF()
        Get a line of text from the file;
        Process the line of text;
    ENDDO
END

procedure ProcessLine(line)
BEGIN
    IF line[0] == "*"  OR line does not start/end with "{ }", respectively
        RETURN
    ENDIF

    Allocate memory for a linked list element;
    Fetch the entity type code as string from line and store in linked list element;
    Fetch the Simnet code from the line and store in linked list element;
    Fetch the entity type description from the line and store in linked list element;
    Convert entity type string into an array of unsigned characters;
    Add linked list element to linked list;
END

procedure ConvertEntityTypeCode(codestr, code)
BEGIN
    /* codestr is in format "1.2.225.3.4"; needs to be changed to something searchable */
    DO WHILE NOT End of String
        Find pos of ".";
        Copy from last "." to next "." to a temporary string;
        Convert temporary string to unsigned char.;
        Move unsigned char. into correct position in code;
    ENDDO
END

procedure AddToLinkedList(DIS_Enums_LL)
BEGIN
    IF global linked list of entity types is NULL
        global linked list = disenum;
        RETURN
    ENDIF
    temp = start of global linked list;
    DO WHILE NOT at end of Linked List
```

```
        IF **disenum's** contents greater than temp's contents
            Insert **disenum** into linked list;
        ELSE
            temp = next linked list element;
            RETURN;
        ENDIF

    ENDDO
    Add **disenum** element to end of global linked list;
END
```

procedure ConvertToArray()

```
BEGIN
    Allocate memory for array of DIS_ENUMs;
    DO WHILE NOT at end of global linked list
        Copy values from linked list element into array element;
        Free memory for linked list element;
    ENDDO
END
```

procedure EntTypeBinarySearchStr(str, ppos, IsCode)

```
BEGIN
    IF str is NOT a string of unsigned characters
        Convert str to a string of unsigned characters
    ENDIF

    Perform a binary search for str in array of entity types;
        Set ppos to position in array of matching entity type;
    RETURN TRUE if found; otherwise return FALSE;
END
```

6.3.3 Enumeration Tables Configuration file - **ENUM_CFG**

6.3.3.1 Overview

To facilitate easy modification of the enumerations list as defined in IST's document - *Enumeration and Bit-encoded Values for use with IEEE 1278.1-1994, Distributed Interactive Simulation -- Application Protocols*, the Scanner loads the enumeration table list from a configuration file. See Section 5.2 for the format of the file. The contents of the file are read into a linked list of "table records". Each "table record" is made up of a linked list of "table elements".

6.3.3.2 Data Structures

The enumeration table is a linked list of TABLE structures. Each TABLE structure has a pointer to the linked list of ELEMENTs that make up that TABLE.

TABLE - Contains the name of the table and a pointer to the list of elements that make up that table.

ELEMENT - Contains the enumeration string and the min/max values for that enumerations string.

6.3.3.3 Processing

```
int LoadEnumerationTable(char  *filename)
BEGIN
    Initialize Enumerated Table Linked List;
    DO WHILE NOT EOF(filename)
        Read line;
        IF line is start of a new table
            Allocate new table pointer;
            Attach new table pointer to table linked list;
            Copy table name to new pointer;
            Initialize element pointer for new table pointer;
        ELSE /* element for a table */
            Allocate new element pointer;
            Fetch values from line and copy into element pointer;
            Add element pointer to element linked list;
        ENDIF
    ENDDO
END
```

```
LL *FindElementListFromTable( char *table_reference )
BEGIN
    DO WHILE NO more Tables to search
        IF TablePtr->TableName == table_reference
            RETURN TablePtr->ElementPtr;
        ENDIF
    ENDDO
    RETURN NULL;
END

char *FindElementFromMinMax( LL* elementptr, short searchval)
BEGIN
    DO WHILE NO more Elements in list to search
        IF searchval is between MIN/MAX values for element
            RETURN elementptr->enumeration_string;
        ENDIF
    ENDDO
END

void destroyEnumLL(void)
BEGIN
    DO WHILE more Tables
        Free elementptr;
    ENDDO
    Free tableptr;
    Set tableptr to NULL;
END
```

6.3.4 System Configuration window - **SC_DIALOG**

6.3.4.1 Overview

The System Configuration window lists the management, pdu, and network default configuration filenames. The port numbers for DIS, SIMNET, Eagle, and IST Message can also be found on this configuration screen. The default values for these fields come from the Scanner Configuration file (see Section 6.3.1). These values are saved within the test suite. When the test suite is opened, the configuration of the new test suite is compared against the current configuration and if there are differences the new configuration is loaded in place of the old configuration.

6.3.4.2 Data Structures

SYSTEM_CONFIG_STRUCT - The structure contains two sections. One is for the string containing the port numbers for indicating if a packet is DIS, SIMNET, Eagle, or IST Message. The other field is an array containing the filename of the different configuration files the Scanner recognizes. Currently, most of them are ignored, but the first three contain values. The first position contains the name of the Management configuration file, "manage.cfg", the second contains the Network definition configuration file, "network.cfg", and the third contains the PDU definition file, "pdu.cfg".

6.3.4.3 Processing

procedure sc_display()
```
BEGIN
    Disable main menu options;
    IF System Config dialog does NOT exist
        Create it;
    ENDIF
    Fill dialog with appropriate values;
    Display dialog;
END
```

procedure sc_create(mainWindowWidget)
```
BEGIN
    Create popup shell widget;
    Create a pane for popup shell widget;
    Add controlling buttons and fields;
    Manage pane widget;
```

```
END

procedure FieldExit_cb(value_widget, config.data field pointer, not used)
BEGIN
    IF value in value_widget is NOT NULL
        Copy value in value_widget to config.data field pointer;
    ENDIF
END

procedure sc_ok_cb(not used)
BEGIN
    Update the port numbers variable if the port numbers were changed and let the packet
    ID code use the new port number;

    FOR I := 1 to # of configuration directories
        IF config file[I] NOT the same as management structure's config file
            IF config file[I] exists
                Copy config file[I] to management structure;
            ELSE
                ERROR;
            ENDIF
            Destroy appropriate configuration information;
            Load newly specified configuration information;
        ENDIF
    NEXT

    Destroy system configuration widget;
    Enable main menu options;
END
```

6.3.5 Test Selection window - **TG_DIALOG**

6.3.5.1 Overview

This dialog window is a generic window showing the tests selected from within a specified test group. From the edit test window, the user is presented with a list of several different test groups to choose from. When the user picks one, the program automatically fetches the offset for that test group within the management structure. This code works generically for all test groups. It will adapt to show the number test entries for the specified test group.

The test group information for the selected test group is copied to a temporary variable. The address of each of the test entries for the test group is also "copied" to the temporary variable. The contents of the temporary variable are then copied into the test selection dialog. When the user selects/deselects a test or changes the binary filename or configuration file name, the new values are automatically copied into the appropriate fields in the management structure.

6.3.5.2 Data Structures

The TestGroupRec and TestEntryRec structures are used for performing these routines. The TestGroupRec contains a count of the number of TestEntryRecs contained in the TestGroupRec and the name of the test group. The TestEntryRec contains three widgets for binary filename, configuration filename, and test selection as well as a field pointing to the address of the associated test entry in the management structure. These structures are used with temporary variables for holding the information displayed and entered on the screen.

6.3.5.3 Processing

```
procedure tg_dialog_display(mainWindowWidget,  test group, label)
BEGIN
        IF test group dialog does NOT exist
                Reset local variables;
                Copy test group information to local variable;
                Create test group dialog;
        ENDIF

        Populate dialog widget;
        Display dialog widget;
        Disable main menu options;
END
```

```
procedure tg_conf_create(mainWindowWidget, label)
BEGIN
    Create popup shell widget;
    Create a pane for popup shell widget;
    Add controlling buttons;
    Add test group information to widget;
        Add test entry information for each group;
    Manage pane widget;
END

procedure init_test_group(local test group variable, management structure's test group)
BEGIN
    Copy all information about management structure's test group to local test group
    variable;
    Copy the address of test entry information from management structure to local test
    group variable;
END

procedure populate_test_group(local test group variable)
BEGIN
    FOR I := 1 TO # of test entries for test group
        Set appropriate selection toggle state for each test entry;
        Copy binary filename to widget;
        Copy configuration filename to widget;
    NEXT
END

procedure cnfFileCB(value widget, pointer to test suite entry, not used)
procedure binFileCB(value widget, pointer to test suite entry, not used)
BEGIN
    Copy value from widget into pointer to test suite entry field;
END
```

6.4 Indextable

6.4.1 Indextable - **INDEXTAB**

6.4.1.1 Overview

This module is used to create the Indextable. The Indextable is a list with one element for each packet within a logged binary file. The Indextable was designed to hold, for each packet in the binary file, all of the information necessary to perform very fast filtering. This was done to avoid having to go to disk to perform the filtering. A new Indextable is created for each binary file tested. However, if the next test to be run uses the same file as the last, the Indextable is not destroyed, but rather reused.

6.4.1.2 Data Structures

DIS_PDU_HEADER - Contains the DIS PDU header information as defined in the document *Standard for Distributed Interactive Simulation -- Application Protocols, Version 2.0, Third Draft*. The version, exercise and time_stamp fields from this structure are used as possible criteria on which to filter packets.

ENTITY_TYPE - This structure is only filled in for Entity State PDUs. It contains the information about the entity in a format defined in the Institute for Simulation and Training's *Enumeration and Bit-encoded Values for use with IEEE 1278.1-1994, Distributed Interactive Simulation -- Application Protocols*. The contents of this structure are used when the user wants to filter on entity type values.

ENTITY_ID - Contains the identifier for the source entity in the PDU.

PACKET_IDENTIFICATION_STRUCT - This structure is filled in by the packet identification code. It contains significant information about the contents of the packet, including what specific type of Network, Transport, and Application protocol layers appear in the packet, the size of each layer, port number the packet was logged on, and IP and Ethernet address.

6.4.1.3 Processing

The Indextable is built as a two-step process. The first step scans the binary file to determine how many packets are within the file, in order to, know how large of an Indextable to allocate from memory. The second pass actually reads data from the binary file into the appropriate cell of the Indextable. The following pseudo code describes how the process is completed:

```
**procedure CreateIndexTable(file)**
BEGIN
        Free memory of previously created Indextable, if necessary;
        DO WHILE NOT EOF()
                Read Logger header;
                IF full packet is there
                        (Number of Packets) ++;
                ENDIF
                Skip packet to position file pointer at next Logger header;
        ENDDO

        IF > 0 packets found
                Allocate memory for (Number of Packets);
        ELSE
                Return FALSE;
        ENDIF

        Get TestBed Address from management structure;
        IndexTableSecondPass(file, Testbed Address);
        RETURN TRUE;
END

procedure IndexTableSecondPass(file, Testbed Address)
BEGIN
        DO WHILE NOT EOF()
                Read Logger Header;
                Read Packet;
                Identify Packet;
                Add Packet and Logger Header information to Indextable;
                Update Packet Count window;
                IF packet is NOT from Testbed
                        Perform Packet Validation;
                ENDIF
                Packet Count++;
        ENDDO
        Make sure Packet Count window is up to date;
END

procedure Add_PacketInfo_to_IndexTable(packet #, Packet)
```

IST: SCANNER

```
BEGIN
    IF Packet is DIS PDU
        Move PDU header from Packet to Indextable;
        Move Entity ID from Packet to Indextable;
        Set field indicating what type of DIS PDU the packet is;
        IF Packet is ENTITY_STATE_PDU
            Move Entity Type from Packet to Indextable;
        ENDIF
    ENDIF
END
```

6.5 Packet Filtering

6.5.1 Packet Filter Routines - **FILTER**

6.5.1.1 Overview

Packet filtering is used to narrow the number of packets the user sees in the Timeline window to only those packets that meet the filter criteria. The user can filter on a number of criteria, specifically, PDU type, entity id, entity type, time frame, exercise id, exercise number, port number, if DIS only, if not DIS only, if testbed-generated, and if not testbed-generated. All of the data necessary to determine if a packet meets one of the filter criteria is stored within the Indextable to allow for very fast searching.

6.5.1.2 Data Structures

DB_FILTER_STRUCT - This structure contains all of the fields the user can filter a packet upon. It was defined to allow the user to filter on as many as 500 different Entity Ids and Entity Types. Each of those values is stored in an array. The user can also check of any or all of the 27 different DIS PDUs to filter on, as well as, other unknown DIS PDUs. With one structure to hold all of the values to be filtered on, it makes the filtering process very generic.

6.5.1.3 Processing

Once the DB_FILTER_STRUCT is filled in it is passed to the Do_Filtering procedure to check each of the possible filter criteria to see which ones are to be tested.

```
procedure Do_Filtering(filter structure, packet #)
BEGIN
    Filter on PDU;
    Filter on Entity ID;
    Filter on Entity Type;
    Filter on Time;
    Filter on DIS only packets;
    Filter on Excluding Testbed generated packets;
    Filter on Exercise ID;
    Filter on Version Number;
    Filter on Port Number;
    Filter on Non-DIS only packets;

    IF all of the above filtering passed
```

```
            RETURN TRUE
        ELSE
            RETURN FALSE
        ENDIF
END
```

procedure Filter_On_PDU(filter structure, packet #)

```
BEGIN
        IF Search_On_PDU field is FALSE OR NOT a DIS packet
            RETURN TRUE
        ENDIF
        IF DB_FILTER_STRUCT.PDU[1] is checked off AND
            Indextable[packet #] is a ENTITY_STATE_PDU
            RETURN TRUE
        ENDIF
        ...
        IF DB_FILTER_STRUCT.PDU[27] is checked off AND
            Indextable[packet #] is a RECEIVER_PDU
            RETURN TRUE
        ENDIF

        RETURN FALSE
END
```

procedure Filter_On_Entity_Type(filter structure, packet #)

```
BEGIN
        IF NOT searching on Entity Type
            RETURN TRUE
        ENDIF
        IF NOT Indextable[packet #] NOT an ENTITY_STATE_PDU
            RETURN TRUE
        ENDIF

        FOR I := 1 TO MAX_ENTITY_TYPES
            IF DB_FILTER_STRUCT[I].Entity_Type == Indextable[packet #].Entity_Type
                RETURN TRUE
            ENDIF
        NEXT

        RETURN FALSE
END
```

6.6 File Management

6.6.1 Logged Binary and Configuration files - **FILEMGMT**

6.6.1.1 Overview

The routines in this file provide a central point for opening and closing the logged binary files and the test-specific configuration files. Several other functions, like building the Indextable and reading a specific packet in from disk require the name of the logged binary file to use, therefore, function calls to perform those tasks are made from within this file.

6.6.1.2 Data Structures

Static variables to hold the name of the last binary file processed which can be used to determine if the program can use the existing Indextable or not. There are also static variables to hold a file pointer to the currently opened logged binary file and test-specific configuration file.

6.6.1.3 Processing

procedure OpenUserFiles(test_group, specific_test)

```
BEGIN
    Search the test suite's data directory for the appropriate binary file;
    IF NOT FOUND
        Search the current directory for the appropriate binary file;
        IF NOT FOUND
            ERROR;
            RETURN FALSE;
        ENDIF
    ENDIF

    Set flags to remember name of opened binary file;

    Search the test suite's data directory for the appropriate test-specific configuration file;
    IF NOT FOUND
        Search the current directory for the appropriate test-specific configuration file;
        IF NOT FOUND
            ERROR;
            RETURN FALSE;
        ENDIF
    ENDIF
```

```
        RETURN TRUE
END

procedure CloseUserFiles()
BEGIN
        IF binary file open
                Close binary file;
        ENDIF
        IF test-specific configuration file open
                Close configuration file;
        ENDIF
END

procedure BuildIndexTable()
BEGIN
        IF binary file currently being processed is same as last binary file
                Freshen Packet count window;
        ELSE
                Create a new Indextable;
        ENDIF
END

procedure ReadTestConfigFile()
BEGIN
        IF no test-specific configuration file is open
                RETURN NULL
        ENDIF

        Start a new linked list;
        DO WHILE not EOF() of configuration file
                Get a line from configuration file;
                IF line is not blank, comments, etc.
                        Add line to linked list;
                ENDIF
        ENDDO

        RETURN linked list;
END
```

6.7 Automated Testing

6.7.1 Automated test selection - **TS_AUTO_DIALOG**

6.7.1.1 Overview

This module is used to build the list of pre-selected tests from which the user can choose which one will be executed. The names of all tests chosen earlier under the Edit configuration window by the user are copied to a list. At the same time, a second list is created that contains the test group and test entry offset from within the management structure for each test in the test selection window. This second list is necessary in translating from the list of tests to choose from to the management structures list of tests.

6.7.1.2 Data Structures

TEST_OFFSETS - A structure that holds the test group and test entry offsets for a specific test in the management structure. An array of these structures is created to hold the test group and test entry offsets for all tests that the user can choose from on the automated test selection window. When a test is chosen, the selected test's offset into the array indicates which specific test was actually chosen.

```
typedef struct
{
        short   tg_index;
        short   te_index;
} Test_Offsets;
```

6.7.1.3 Processing

The following pseudo code shows the creation and display of the list of automated tests. Once the test is selected from the list, a function is called to display the appropriate windows and call the automated test procedures.

```
procedure ts_auto_dialog_display(main_window_widget)
BEGIN
    Set the number of tests to choose from counter to 0;
    Clear all test selection structures;
    Create automated test selection dialog;
    Display automated test selection dialog;
END
```

```
procedure ts_auto_dialog_create(main_window_widget)
BEGIN
        Create a pop-up shell widget;
        Create list widget and attach to shell widget;
        Add "OK"/"Cancel" buttons to shell widget;
        Create a list of automated tests that can be conducted.
        Copy contents of list to list widget;
        Delete automated test list;
END

procedure auto_test_list_create(startpos)
BEGIN
        FOR i := 1 to # of test groups
                FOR j := 1 to # of test entries for test group i
                        IF test[i,j] is NOT scheduled .OR.
                           test[i,j] is NOT automated
                                LOOP
                        ENDIF
                        Copy test status (Passed/Failed, etc) to temp. string;
                        Concatenate test number and test description to temp. string;
                        Concatenate test comments to end of temp. string;
                        Convert temp. string to XmString;
                        Append XmString to automated test list;

                        IF have not picked a default test yet .AND.
                           test status of test[i,j] is N/A
                                Assign startpos number of tests counter;
                        ENDIF
                        Increment number of tests counter;
                        Add test[i,j] test group, test entry offsets to TestOffset array;
                NEXT
        NEXT
END

procedure auto_test_list_delete()
BEGIN
        FOR I := 1 to number of tests counter
                IF automated test list[i] is NOT NULL
                        Free XmString;
                ENDIF
        NEXT
```

```
END

procedure ts_auto_select_cb (widget, not used, automated test selection list )
BEGIN
        Fetch the selected item position in automated test selection list and store in a global
        variable;
END

procedure ts_auto_dialog_ok_cb(not used)
BEGIN
        IF nothing is selected in test selection window
                RETURN
        ENDIF

        Fetch the test group and test entry offsets for the selected test using the variable set in
        function ts_auto_select_cb();
        Set the management structures internal pointer to the same test group/entry offsets;

        Fetch the name of the binary file required for that test from the management structure;
        Display the Logger dialog;

        Close test selection window;
        Destroy the test selection widget;
END

procedure ts_auto_exec(name_of_binary_file)
BEGIN
        /*
        ** User could have changed the name_of_binary_file in the logger dialog. Just making
        ** sure management structure is up to date.
        */
        Copy name_of_binary_file to management structure position pointed to by selected
        test's test group and entry offsets;

        IF test suite has NOT had it's report log header printed
                Print it;
                Set internal flag in management structure to TRUE to avoid printing again;
        ENDIF

        Try to open appropriate logged binary file and associated configuration file;
        IF able to
```

```
                IF Automated NETWORK test
                        Fetch the default network edit list;
                ELSE IF Automated PDU test
                        Fetch the default PDU edit list;
                ELSE
                        ERROR
                        RETURN;
                ENDIF

                Display Packet Count window;

                IF NETWORK test
                        Build Network Edit List;
                        Load Network test-specific configuration file;
                        Display Network Edit List;
                ELSE IF PDU test
                        Build PDU Edit List;
                        Load PDU test-specific configuration file;
                        Display PDU Edit List;
                ENDIF

                Display Report Results window;
                Try to build Indextable;
                IF able to
                        Display the binary filename in Report Results dialog;
                        Update Report Results dialog;
                        IF no packets were found in logged binary file
                                Set Report Results dialog reason to "BIN file contains 0 records"
                        ENDIF
                        Disable main menu widgets;
                ELSE
                        Destroy Packet Count window;
                        Destroy Report Results window;
                        Destroy Network test Edit window;
                        Destroy PDU test Edit window;
                        Enable main menu widgets;
                ENDIF
        ENDIF
END
```

6.7.2 Network User Edit List - **NETCONF**

6.7.2.1 Overview

This file contains functions for handling the Network header (Level 1) automated test screen. Network testing is made up of testing both the network layer and the transport layer. It builds the configuration screen that allows the user to edit the min/max values that will be used when performing packet validation on each matching packet within a binary file.

Network validation is a little different than PDU validation in that in PDU validation, the user specifies which PDU is to be tested. All other PDU types are ignored. With network validation, the network layer could be Ethernet or IEEE 802.3 and the transport layer could be UDP/IP or TCP/IP (these are the only types the Scanner recognizes). Therefore, we need to allow the user to enter min/max values for any combination of network and transport layers that could appear within a packet in a binary file.

6.7.2.2 Data Structures

The default network configuration table uses a common data structure, the PDU_DATA_HEADER. The definition of the PDU_DATA_HEADER can be found in Section 5.2. The network configuration table is defined as an array of PDU_DATA_HEADERs. Using the same data type allows all tables to be processed using one set of routines.

The PDU_DATA_HEADERs contain pointers to default, field-definition, configuration lists. The default configuration lists are lists of data elements (PDU_DATA_ENTRY) that describe the default structure of a protocol layer. The definition of the PDU_DATA_ENTRY can be found in Section 5.2.

PDU_DATA_ENTRY - This is the base structure used to define each entry in the default configuration lists. The pdlType field is used to specify if the element is a data field, or a group reference. For group references, the fields used to define the groups contents are placed in a new list pointed to by the children field of the group element.

PDU_DATA_HEADER - This is the base structure used to define the default configuration tables. The default configuration tables are arrays of PDU_DATA_HEADERs. Each PDU_DATA_HEADER contains information about a specific protocol layer type, and a pointer to the list of elements describing the protocol layer.

6.7.2.3 Processing

The network configuration user edit list pseudo code is listed below. The list is built, displayed, edited and then passed to the automated test procedures.

```
procedure buildNetEditList(default_Network_List)
BEGIN
    /*  Copy the default Network linked list into a structure that contains the same
        information as well as widgets for screen management.  */
    Allocate as much memory as the array of recognizable network headers;
    FOR I := 1 TO # of recognizable network headers
        Copy information from default list to new display list;
        IF children for default_Network_List current element
            Copy the network children's values to new display list;
        ENDIF
        Set display list's widget to NULL;
    NEXT
    RETURN newly created display list;
END

procedure copyNetList(linked_list_ptr)
BEGIN
    Create a new list for editing;
    Set a pointer to the beginning of linked_list_ptr;
    DO WHILE not at end of linked_list_ptr
        Copy values from current element to temp. variable;
        Set temp. variable's children pointer to NULL;
        IF current element has children
            Call copyNetList() with pointer to children;
        ENDIF
        Insert temp. variable's data in new list;
        Get next element in linked_list_ptr;
    ENDDO
    RETURN new list for editing;
END

procedure net_conf_display(mainWindowWidget, NetworkDefaultListPtr, window label)
BEGIN
    IF local Network Edit list already exists
        Free local Network Edit List;
    ENDIF
    Set local Network Edit List to NetworkDefaultListPtr;
    IF NOT created already
```

```
        Create Network Edit window;
    ENDIF
    Display Network Edit window;
END

procedure freeNetEditList(NetLinkedList)
BEGIN
    Set pointer to beginning of NetLinkedList;
    DO WHILE NOT end of NetLinkedList
        IF element has children
            Call freeNetEditList() with pointer to children;
            Free children's linked list;
        ENDIF
        Get next element in list;
    ENDDO

END

procedure net_conf_run_cb(not used)
BEGIN
    Create a copy of the user edit list the user just finished editing; (extractPduHeader())
    Call the central point for running automated tests;
    Free up memory reserved by copy of user edit list;
END

procedure extractPduHeader(Network_Display_Header_List)
BEGIN
    Allocate as much memory as the array of recognizable network headers;
    Set a pointer to the beginning of the allocated memory;
    Set a pointer to the beginning of Network_Display_Header_List;
    FOR I := 1 TO # of recognizable network headers
        Copy just the data not widget values from Network_Display_Header_List;
        IF Network_Display_Header_List has children
            Copy the list of children; (extractPduList())
        ENDIF
        Move pointers to next element;
    NEXT

    RETURN newly allocated network header list;
END
```

```
procedure extractPduList(List_of_children)
BEGIN
    Create a new list;
    DO WHILE NOT end of List_of_children
        Copy values from List_of_children to temp. variable;
        IF current element in List_of_children has children
            Copy the list of children to temp. variable;
        ENDIF
        Insert contents of temp. variable into new list;
        Get next element in List_of_children;
    ENDDO
    RETURN new list;
END

procedure create_net_fields(mainWindowWidget, NetworkDisplayHeaderList)
BEGIN
    Create a frame widget and save within NetworkDisplayHeaderList;
    Create a row/column widget using frame widget;
    IF NetworkDisplayHeaderList has children
        Get first child;
        DO WHILE more children
            IF child data type is DATA_ELEMENT
                Create Network Field; (createNetField())
            ELSE IF child data type is DATA_GROUP
                Create Network Group; (createNetGroup())
            Get next child;
        ENDDO
    ENDIF
    Manage widgets;
END

procedure createNetGroup(parentWidget, NetDisplayField)
BEGIN
    Create frame widget and save within parentWidget;
    Create row/column widget using frame widget;
    IF parentWidget has children
        Get first child;
        DO WHILE more children
            IF child data type is DATA_ELEMENT
                Create Network Field; (createNetField())
            ELSE IF child data type is DATA_GROUP
```

```
                    Create Network Group; (createNetGroup())
                Get next child;
            ENDDO
        ENDIF
        Manage widgets;
END
```

procedure createNetField(parentWidget, NetDisplayField)

```
BEGIN
        Create a form widget from parentWidget;
        Create a label widget from form widget;
        Set parameters of label widget;
        Set min/max text widgets, fields, and callbacks;
END
```

6.7.3 PDU User Edit List - **PDUCONFIG**

6.7.3.1 Overview

This file contains functions for handling the PDU (Level 2) automated test screen. It builds the configuration screen that allows the user to edit the min/max values that will be used when performing packet validation on each matching packet within a binary file. This module also contains functions for building and freeing PDU edit lists.

6.7.3.2 Data Structures

An array of PDU_DATA_HEADERS exists, each with a linked list of data elements (PDU_DATA_ENTRYs) that contain the information necessary to determine the format of each PDU. The array contains 27 elements, each representing one of the 27 DIS PDUs. The program knows which list in the array is the correct one to pass for processing. See Section 5.2 for more detail.

PDU_DATA_ENTRY - This is the base structure used to define each entry in the default configuration lists. The pdlType field is used to specify if the element is a data field, or a group reference. For group references, the fields used to define the groups contents are placed in a new list pointed to by the children field of the group element.

6.7.3.3 Processing

The following pseudo code shows the creation of the user's PDU edit list for specific min/max range validation during automated tests.

```
procedure buildPduEditList(defaultPDUList)
BEGIN
    Create a new list of PDU DISPLAY FIELDS for editing;
    Set a pointer to beginning of defaultPDUList
    DO WHILE NOT END OF defaultPDUList
        Copy defaultPDUList element's data to temporary variable;
        Set temporary variable's children pointer to NULL;
        IF defaultPDUList has children
            Call buildPduEditList() with pointer to children;
        ENDIF
        Insert copy of temporary variable into new list;
        Get next element in defaultPDUList;
    ENDDO
    RETURN new PDU DISPLAY FIELDS list;
```

```
END

procedure pdu_conf_display(mainWidget, PDUEditList, screen label)
BEGIN
    IF local PDU Edit list already exists
        Free local PDU Edit List;
    ENDIF
    Set local PDU Edit List to PDUEditList;
    IF NOT created already
        Create PDU Edit window;
    ENDIF
    Display PDU Edit window;
END

procedure freePduEditList(pduLinkedList)
BEGIN
    IF pduLinkedList != NULL
        Set pointer to beginning of pduLinkedList;
        DO WHILE NOT end of pduLinkedList
            IF element has children
                Call freePduEditList() with pointer to children;
            ENDIF
        ENDDO
        Free element;
    ENDIF
END

procedure displayPduList(screenWidget, PDU_Display_Fields_Linked_List)
BEGIN
    /* Traverse recursive linked list building entries for each PDU field */
    DO WHILE NOT END OF PDU_Display_Fields_Linked_List
        IF current element in list is a DATA ELEMENT
            Display individual PDU field;
        ELSE IF element is DATA GROUP
            IF element has children
                Create a frame widget and save in current element;
                Label the widget with the group name;
                Create a row/column widget for frame widget;
                Call displayPduList() with row/col widget and pointer to children;
                Manage all widgets;
            ENDIF
        ENDIF
    ENDDO
END
```

procedure pdu_conf_run_cb(not used)

```
BEGIN
    Run the automated test using the user edit list just edited;
END
```

6.7.4 Common Automated test routines - **AUTOTEST**

6.7.4.1 Overview

Similar functionality is used to perform both automated network and PDU tests. This file contains the generic functions necessary to perform the actual tests. The central automated test function, RunAutomatedTests() is located in this file and it provides a single function to call all automated testing. Routines for actually running the test, loading test-specific configuration files and determining which type of test to run can also be found in this file.

Testing is broken up into four types, Transmission, Reception, Adverse, and Erroneous. Transmission requires the SUT to create and send a specific type of packet which will be analyzed by the Scanner to determine if the data in the packet is in the correct format and has values between the specified min/max ranges. The other three types are all reception tests. These tests require the testbed to generate packets and send them to the SUT to see if the SUT properly "handles" them without crashing. The regular Reception test requires sending good data, Adverse requires sending packets with data outside of the norm, and Erroneous requires sending packets with completely out of range or missing data. The Scanner performs the reception tests by searching all packets in the logged binary file not generated by the testbed. If any packets are found that are not DIS PDUs, an error has occurred and the test fails.

6.7.4.2 Data Structures

The automated test functionality is made possible by creating a central function that can be called for automated testing that is "smart" enough to know which specific automated test to call. In the Scanner, this is made possible by loading addresses of each of the specific automated test functions into a array. Then, using the appropriate offset, the Scanner knows exactly which test function to execute. The definition for this internal structure can be found in Section 5.2.2.11 of the manual.

The functionality within this file also uses the DB_FILTER_STRUCT to allow the program to "filter" the Indextable to locate packets that meet the criteria of the test.

6.7.4.3 Processing

Each test listed in the management configuration file (manage.cfg) has a unique identifier associated with it. When the management configuration file is read in, the unique identifier is passed to a function that searches a list to find that unique identifier. When it is found, the offset of the identifier is returned and loaded into the management structure for that test. When that test is to be run in the automated mode, the offset is used to determine which automated test to execute from the array of pointers to automated test functions.

```
procedure AutomatedTestFunctionsIndex(unique_id, test_suite_entry_ptr)
BEGIN
        IF unique_id == "NONE"
                Set test_suite_entry_ptr automated flag to -1;
                Set test_suite_entry_ptr pdu offset to 0;
                RETURN FALSE
        ENDIF

        FOR I := 1 TO # of automated tests
                IF unique_id == automated test array[I]'s unique_id
                        Set test_suite_entry_ptr automated flag to I;
                        Set test_suite_entry_ptr pdu offset to automated test array[I]'s pdu
                        offset;
                        RETURN TRUE
                ENDIF
        NEXT

        ERROR;
        RETURN FALSE;
END

procedure RunAutomatedTest(user_edit_list)
BEGIN
        IF performing a network test
                Set local pointer to user_edit_list;
                Call Network validation procedure;
        ELSE IF performing a PDU test
                Set local pointer to user_edit_list;
                Call PDU validation procedure;
        ELSE
                ERROR;
                RETURN;
        ENDIF

        Display packet display window;
        Display Timeline window;
END

procedure NetworkValidation()
procedure PDUValidation()
BEGIN
```

```
        Call the function pointed to in the automated test array with the offset from the
        management structure for the currently selected test;
        IF the return value is TRUE
                Assign PASSED to the test result;
        ELSEIF the return value is FALSE
                Assign FAILED to the test result;
        ENDIF
END
```

Each automated test initializes the DB_FILTER_STRUCT with the appropriate values to narrow the selection of packets for testing to only those packets that meet the specified criteria for that test. For network and PDU tests, the filter is initialized to find only DIS packets and they must not be generated by the testbed. A further qualification for PDU testing is that the matching packets must be DIS PDUs of the appropriate type for that test. Actually, once the filter structure is initialized, one generic function can be called to perform all reception tests since the Scanner is only looking for "junk" within the logged binary file. A separate function for network and PDU transmission testing is required because they are looking for slightly different information.

The following pseudo code shows the loading of matching packets and the actual test procedures.

```
procedure LoadPacketForTest(Filter, firsttimerun)
BEGIN
        IF this is the first time to call this function
                Find FIRST packet in Indextable that meets the Filter;
                IF one is FOUND
                        Load the Packet from disk;
                        Reset Validation flag to TRUE;
                        RETURN TRUE;
                ENDIF
        ELSE
                Find NEXT packet in Indextable that meets the Filter;
                IF one is FOUND
                        Load the Packet from disk;
                        Reset Validation flag to TRUE;
                        RETURN TRUE;
                ENDIF
        ENDIF
        RETURN FALSE;
END
```

```
procedure PDUTest_Transmission(whichpdu)
BEGIN
    Clear contents of filter structure;
    Set filter to filter for PDU whichpdu, DIS only, and non-testbed generated;
    DO WHILE TRUE
        Find the "next" packet that meets the filter;
        IF none where ever found
            RETURN FALSE;
        ELSE IF no more where found
            EXIT;
        ENDIF
        Call packet validation with user edit list for this test;
        IF failed packet validation
            Write to report dialog failure message;
            Set Indextable's validation flag to FALSE;
        ENDIF
    ENDDO
    RETURN TRUE if no errors were found, otherwise return FALSE;
END

procedure NetworkTest_Transmission()
BEGIN
    Clear contents of filter structure;
    Set filter to filter for DIS only, and non-testbed generated packets;
    DO WHILE TRUE
        Find the "next" packet that meets the filter;
        IF none where ever found
            RETURN FALSE;
        ELSE IF no more where found
            EXIT;
        ENDIF
        Call packet validation with user edit list for this test;
        IF failed packet validation
            Write to report dialog failure message;
            Set Indextable's validation flag to FALSE;
        ENDIF
    ENDDO
    RETURN TRUE if no errors were found, otherwise return FALSE;
END
```

```
procedure Network_PDU_Test_Reception()
BEGIN
    Clear contents of filter structure;
    Set filter to filter for DIS only, and non-testbed generated packets;
    Find any packet that meets the filter;
    IF one was found
        Write to report dialog failure message;
        Set Indextable's validation flag to FALSE;
        RETURN FALSE;
    ENDIF
    RETURN TRUE;
END
```

As mentioned above, the addresses of all automated test functions are saved in an array. That array also contains the unique identifier for that test and the address to a function for loading the test-specific configuration file, if one is specified. The test-specific configuration file allows the user to specify values to override the defaults on a test-by-test basis. The function for loading the configuration file is passed the appropriate default user-edit list for that type of test. When the configuration file is read and parsed, the values from the default user-edit list are overwritten by the matching values from the test-specific configuration file.

```
procedure LoadPduTestConfigFile( UserEditListPtr )
procedure LoadNetTestConfigFile( UserEditListPtr )
BEGIN
    IF there is a configuration file load routine for this test
        Call configuration file load function with UserEditListPtr;
    ENDIF
END
```

6.7.5 Network-specific test routines - **AUTONETTEST**

6.7.5.1 Overview

This file contains the specific, unique network tests. For Network and PDU testing, the requirements of testing are very similar. See Section 6.7.4 for description of the types of tests. Because the program can use the common functions mentioned in Section 6.7.4 to perform the actual tests, the specific network tests only need to call the common functions, passing along the appropriate parameters.

6.7.5.2 Data Structures

The linked list structure is used to contain all items read from the test-specific configuration file that are to be changed in the default user edit list.

6.7.5.3 Processing

The following routine highlights the concept of reading a network, test-specific configuration file and using the contents of that file to overwrite the contents of the Network default user edit list.

```
procedure ProcessNetworkTestConfigFile( default_user_edit_list )
BEGIN
    Read the appropriate test-specific configuration file and return a linked list of the
    contents of the file;

    LOOP through the linked list processing each element;
        Fetch the group and field names from the current element;
        Fetch the min/max values from the current element;
        FOR I := 1 TO # of recognizable Network headers
            Try to find group name in default_user_edit_list[I];
            IF FOUND
                Replace min/max values with values from configuration file;
            ENDIF
        NEXT
        Get next element in linked list;
    ENDLOOP
    Free memory for linked list;
END
```

The format for each of the specific network configuration tests is very similar. The functions call either the Reception or Transmission generic validation routine depending upon the requirements of the test. The structure of the test name is made up of the 6 unique characters associated with each test listed in the management configuration file.

```
procedure Test_T111TA()
procedure Test_T111RA()
procedure Test_T111AA()
procedure Test_T111EA()
procedure Test_T112TA()
procedure Test_T112RA()
procedure Test_T112AA()
procedure Test_T112EA()
procedure Test_T113TA()
procedure Test_T113RA()
procedure Test_T113AA()
procedure Test_T113EA()
procedure Test_T121TA()
procedure Test_T121RA()
procedure Test_T121AA()
procedure Test_T121EA()
procedure Test_T131TA()
procedure Test_T131RA()
BEGIN
    IF test is a TRANSMISSION test
        Call Network Transmission test;
    ELSEIF test is any kind of RECEPTION test
        Call Network/PDU Reception test;
    ENDIF
END
```

6.7.6 PDU-specific test routines - **AUTOPDUTEST**

6.7.6.1 Overview

This file contains the specific PDU tests. For Network and PDU testing, the requirements of testing are very similar. See Section 6.7.4 for description of the types of tests. Because the program can use the common functions mentioned in Section 6.7.4 to perform the actual tests, the specific PDU tests only need to call the common functions and pass along the appropriate parameters.

6.7.6.2 Data Structures

The linked list structure is used to contain all items read from the test-specific configuration file that are to be changed in the default user edit list.

6.7.6.3 Processing

The following routine highlights the concept of reading a PDU, test-specific configuration file and using the contents of that file to overwrite the contents of the PDU default user edit list.

```
procedure ProcessPduTestConfigFile( default_user_edit_list )
BEGIN
    Read the appropriate test-specific configuration file and return a linked list of the
    contents of the file;

    LOOP through the linked list processing each element;
        Fetch the group and field names from the current element;
        Fetch the min/max values from the current element;
        Try to find group name in default_user_edit_list;
        IF FOUND
            Replace min/max values with values from configuration file;
        ENDIF
        Get next element in linked list;
    ENDLOOP
    Free memory for linked list;
END
```

The format for each of the specific PDU configuration tests is very similar. They call either the Reception or Transmission generic validation routine depending upon the requirements of the test. The structure of the test name is made up of the 6 unique characters associated with each test listed in the management configuration file.

procedure Test_T201TA()
procedure Test_T202TA()
procedure Test_T203TA()
procedure Test_T204TA()
procedure Test_T205TA()
procedure Test_T206TA()
procedure Test_T207TA()
procedure Test_T208TA()
procedure Test_T209TA()
procedure Test_T210TA()
procedure Test_T211TA()
procedure Test_T212TA()
procedure Test_T213TA()
procedure Test_T214TA()
procedure Test_T215TA()
procedure Test_T216TA()
procedure Test_T217TA()
procedure Test_T218TA()
procedure Test_T219TA()
procedure Test_T220TA()
procedure Test_T221TA()
procedure Test_T222TA()
procedure Test_T223TA()
procedure Test_T224TA()
procedure Test_T225TA()
procedure Test_T226TA()
procedure Test_T227TA()
procedure Test_T201RA()
procedure Test_T202RA()
procedure Test_T203RA()
procedure Test_T204RA()
procedure Test_T205RA()
procedure Test_T206RA()
procedure Test_T207RA()
procedure Test_T208RA()
procedure Test_T209RA()
procedure Test_T210RA()
procedure Test_T211RA()
procedure Test_T212RA()

```
procedure Test_T213RA()
procedure Test_T214RA()
procedure Test_T215RA()
procedure Test_T216RA()
procedure Test_T217RA()
procedure Test_T218RA()
procedure Test_T219RA()
procedure Test_T220RA()
procedure Test_T221RA()
procedure Test_T222RA()
procedure Test_T223RA()
procedure Test_T224RA()
procedure Test_T225RA()
procedure Test_T226RA()
procedure Test_T227RA()
procedure Test_T201AA()
procedure Test_T202AA()
procedure Test_T203AA()
procedure Test_T204AA()
procedure Test_T205AA()
procedure Test_T206AA()
procedure Test_T207AA()
procedure Test_T208AA()
procedure Test_T209AA()
procedure Test_T210AA()
procedure Test_T211AA()
procedure Test_T212AA()
procedure Test_T213AA()
procedure Test_T214AA()
procedure Test_T215AA()
procedure Test_T216AA()
procedure Test_T217AA()
procedure Test_T218AA()
procedure Test_T219AA()
procedure Test_T220AA()
procedure Test_T221AA()
procedure Test_T222AA()
procedure Test_T223AA()
procedure Test_T224AA()
procedure Test_T225AA()
procedure Test_T226AA()
procedure Test_T227AA()
procedure Test_T201EA()
procedure Test_T202EA()
procedure Test_T203EA()
procedure Test_T204EA()
procedure Test_T205EA()
procedure Test_T206EA()
procedure Test_T207EA()
procedure Test_T208EA()
procedure Test_T209EA()
procedure Test_T210EA()
procedure Test_T211EA()
procedure Test_T212EA()
procedure Test_T213EA()
procedure Test_T214EA()
procedure Test_T215EA()
procedure Test_T216EA()
procedure Test_T217EA()
procedure Test_T218EA()
procedure Test_T219EA()
procedure Test_T220EA()
procedure Test_T221EA()
procedure Test_T222EA()
procedure Test_T223EA()
procedure Test_T224EA()
procedure Test_T225EA()
procedure Test_T226EA()
procedure Test_T227EA()
BEGIN
        IF test is a TRANSMISSION test
                Call PDU Transmission test;
        ELSEIF test is any kind of RECEPTION test
                Call Network/PDU Reception test;
        ENDIF
END
```

6.8 Manual Testing

6.8.1 Manual test selection - **TS_MAN_DIALOG**

6.8.1.1 Overview

This module is used to build the list of previously-selected tests from which the user can choose which one will be executed. The names of all tests chosen by the user are copied to a list. At the same time, a second list is created that contains the test group and test entry offset from with the management structure for each test in the test selection window. This second list is necessary in translating from the list of tests to choose from to the management structures list of tests.

6.8.1.2 Data Structures

TEST_OFFSETS - A structure that holds the test group and test entry offsets for a specific test in the management structure. An array of these structures is created to hold the test group and test entry offsets for all tests that the user can choose from on the manual test selection window. When a test is chosen, the selected test's offset into the array indicates which specific test was actually chosen.

```
typedef struct
{
        short   tg_index;
        short   te_index;
} Test_Offsets;
```

6.8.1.3 Processing

The following code shows how the manual test selection dialog is created and displayed. Once the user selects a test to run, the program will open the appropriate logged binary file and display the contents in appropriate gui-windows.

```
procedure ts_man_dialog_display(main_window_widget)
BEGIN
    Set the number of tests to choose from counter to 0;
    Clear all test selection structures;
    Create manual test selection dialog;
    Display manual test selection dialog;
END
```

```
procedure ts_man_dialog_create(main_window_widget)
BEGIN
    Create a pop-up shell widget;
    Create list widget and attach to shell widget;
    Add "OK"/"Cancel" buttons to shell widget;
    Create a list of manual tests that can be conducted.
    Copy contents of list to list widget;
    Delete manual test list;
END

procedure man_test_list_create(startpos)
BEGIN
    FOR i := 1 to # of test groups
        FOR j := 1 to # of test entries for test group i
            IF test[i,j] is NOT scheduled
                LOOP
            ENDIF
            Copy test status (Passed/Failed, etc) to temp. string;
            Concatenate test number and test description to temp. string;
            Concatenate test comments to end of temp. string;
            Convert temp. string to XmString;
            Append XmString to manual test list;

            IF have not picked a default test yet .AND.
               test status of test[i,j] is N/A
                Assign startpos number of tests counter;
            ENDIF
            Increment number of tests counter;
            Add test[i,j] test group, test entry offsets to TestOffset array;
        NEXT
    NEXT
END

procedure man_test_list_delete()
BEGIN
    FOR I := 1 to number of tests counter
        IF manual test list[i] is NOT NULL
            Free XmString;
        ENDIF
    NEXT
END
```

```
procedure ts_man_select_cb (widget, not used, man. test selection list )
BEGIN
        Fetch the selected item position in man. test selection list and store in a global variable;
END

procedure ts_man_dialog_ok_cb(not used)
BEGIN
        IF nothing is selected in test selection window
                RETURN
        ENDIF

        Fetch the test group and test entry offsets for the selected test using the variable set in
        function ts_man_select_cb();
        Set the management structures internal pointer to the same test group/entry offsets;

        Fetch the name of the binary file required for that test from the management structure;
        Display the Logger dialog;

        Close test selection window;
        Destroy the test selection widget;
END

procedure ts_man_exec (name_of_binary_file)
BEGIN
        /*
        ** User could have changed the name_of_binary_file in the logger dialog. Just making
        ** sure management structure is up to date.
        */
        Copy name_of_binary_file to management structure position pointed to by selected
        tests test group and entry offsets;

        IF test suite has NOT had it's report log header printed
                Print it;
                Set internal flag in management structure to TRUE to avoid printing again;
        ENDIF

        Try to open appropriate logged binary file and associated configuration file;
        IF able to
                Display Packet Count window;
                Display Packet Display window;
                Display Report Results window;
        Display Packet Filter window;
        Display Timeline window;

        Set cursor to watch while waiting;
        Try to build Indextable;
        IF able to
            Display the binary filename in Report Results dialog;
            Initialize packet filter;
            Update Report Results dialog;
            IF no packets were found in logged binary file
                Set Report Results dialog reason to "BIN file contains 0 records"
            ENDIF
            Update Timeline dialog;
            Disable main menu widgets;
        ELSE
            Destroy Packet Count window;
            Destroy Packet Display window;
            Destroy Report Results window;
            Destroy Packet Filter window;
            Destroy Timeline window;
            Enable main menu widgets;
        ENDIF
        Reset cursor;
    ENDIF
END
```

6.9 Graphical User Interface Windows

6.9.1 Packet Count - **PC_DIALOG**

6.9.1.1 Overview

The packet count dialog window shows the user the number of recognizable protocol application layers found in the packets in a logged binary file. The Scanner recognizes several different protocol application layers including, DIS PDU, SIMNET, IST Message, and EAGLE. The Scanner also keeps a count of unrecognizable protocol application layers and a total count of all values. If the application layer is a DIS PDU, the program also keeps a count of which specific DIS PDUs were found.

As each packet is being read from the logged binary file during the creation of the Indextable, the packet is passed to the packet identification code. The application type and DIS PDU information are passed to a function to update the appropriate packet count widgets. Too speed up the processing, since xWindows screen output is relatively slow, only after every 100 packets are processed is the screen actually updated with new counts. After the entire binary file is processed, a routine is called to make the final screen update.

6.9.1.2 Data Structures

For each type of application layer from a packet that the Scanner recognizes, the program needs to keep a count of the number of occurrences within the logged binary file. To do this, a structure is defined to save the textual name of the application layer, a count of the number times the application layer was found in the binary file, a widget for the xWindows interface, and the length of the field in the xWindows display. This structure is defined in DisplayCountField. This same structure is used for recording the number of specific DIS PDUs that appeared within the binary file. The following is the definition of the DisplayCountField structure:

```
typedef struct
{
        char        * label;
        int         width;
        Widget      widget;
        int         value;
} DisplayCountField;
```

6.9.1.3 Processing

The following is the pseudo code for the creation and destruction of the packet count window:

```
procedure displayPacketCountWindow()
BEGIN
    IF window does not already exist
        Create a popup shell widget, i.e. packet count window widget;
        Create a scrolled window child of the shell widget;
        Create and attach the binary filename widget;
        Create and attach the packet types widgets;
        Create and attach the DIS packet types widgets;
    ENDIF
    Display packet count window;
END

procedure packet_count_destroy()
BEGIN
    IF packet count window exists
        Destroy packet count window widget;
        Reset all packet type widgets to NULL and counts to 0;
        Reset all DIS packet type widgets to NULL and counts to 0;
    ENDIF
END
```

The pseudo code below shows the routines for updating the contents of the packet count window:

```
procedure updatePacketCounts(Application, DISpdu)
BEGIN
    IF Application is recognizable
        Increment counter for that type of application;
        Increment TOTAL counter;
        IF another 100 packets have been processed
            Convert the counter values and store them in the associated widgets;
        ENDIF
    ENDIF
    IF Application is DIS
        Increment counter for that type of DIS application;
        IF another 100 DIS packets have been processed
            Convert the counter values and store them in the associated widgets;
        ENDIF
        ENDIF
END
```

procedure finalUpdatePacketCounts()

```
BEGIN
        For all application types
                Convert the counter values and store them in the associated widgets;
        For all DIS application types
                Convert the counter values and store them in the associated widgets;
END
```

6.9.2 Packet Display - **PKT_DISP**

6.9.2.1 Overview

The Packet Display code creates a text widget and displays the contents of the widget in a scrollable window. The screen displays the entire contents of the packet, broken down into the different protocol layers. See Section 6.1.4 - PDU Display Routines for a detailed description of how and what text is written to the screen.

A text string is allocated to hold all of the text to display in the text window. The text string is initially set to NULL. Then each text string describing more of the contents of the packet is concatenated to the end of the string. Finally, the entire string is copied to the Packet Display widget, thereby, having it appear on the screen. Having each text string concatenated to one master string as opposed to having each text string copied into the widget was done to speed up the update time of the xWindows interface.

6.9.2.2 Data Structures

A large, static text string is used to hold the text to be displayed within the window.

6.9.2.3 Processing

The following functions represent in pseudo code the creation and destruction routines for the Packet Display window.

procedure PktDispWinInit(mainWindowWidget)
```
BEGIN
    Set the local flag to indicate the window is open;
    Set the cursor_position flag to 0;
    IF Packet Display widget does not exist
        Create a top level shell for the text window;
        Create a scrollable text window widget;
        Add a callback to update current cursor position;
        Allow window manager to manage and display the text window;
    ENDIF
    Display text window;
END
```

procedure PktDispWinClose()
```
BEGIN
    IF Packet Display window is active
```

```
                Destroy window;
                Set local flag to indicate the window is open to FALSE;
                Set all widgets to NULL;
        ENDIF
END
```

The following functions show how the text string displayed in the packet display window is created and displayed.

```
procedure PktDispInitMsg()
BEGIN
        Set internal message string to NULL;
END

procedure PktDispBuildTextMsg( text_to_add )
BEGIN
        IF text_to_add plus current length of internal message string is not too long
                Concatenate text_to_add to internal message string;
        ENDIF
END

procedure PktDispWriteMsg()
BEGIN
        IF packet display window is active
                Set packet display window widget with the internal message string;
                Reset the text window to the previous text windows position;
        ENDIF
END
```

6.9.3 Timeline - **TL_DIALOG**

6.9.3.1 Overview

The Timeline shows the user a list of packets that can be selected and then have the contents of that packet appear in the Packet Display window. The contents of the Timeline is driven by different sources depending upon the type of testing being conducted. If the test being run is a manual test, the items selected in the Packet Filter window drive what packets will appear in the Timeline. If the test is an automated test, the attributes of the specific test being run are set in a DB_FILTER_STRUCT, and those values drive which packets will appear in the Timeline. Very similar in functionality, however, in manual tests, the user has the ability to change the filter criteria; in the automated tests the user does not. The contents of the Timeline can be furthered narrowed by entering information into the query filter dialog.

6.9.3.2 Data Structures

A list of values is created for every packet the meets the criteria specified. The values are actually Motif XmString structures. An array of XmStrings is allocated during the creation of the creation of the Timeline and the array contents are added to the xWindows scroll widget used for the Timeline.

6.9.3.3 Processing

The creation and destruction of the Timeline widget (window) is described in the pseudo-code below:

```
procedure tl_dialog_display(mainWindowWidget)
BEGIN
        IF Timeline widget does NOT exist
                Create Timeline widget
        ENDIF
        Display Timeline widget (window)
END

procedure tl_dialog_create(mainWindowWidget)
BEGIN
        Create popup shell widget for Timeline;
        Create pane for dialog;
        Create each row for dialog;
        Attach a label to the dialog;
        Create the Timeline display window;
        Attach button bar button widgets;
        Attach help line widget;
        Manage widgets;
        RETURN popup shell widget;
END
```

procedure tl_time_line_display()

```
BEGIN
        Build an array of the packets that should be displayed in the Timeline;
        IF Timeline list widget exists
                Delete all items in the Timeline list widget;
                Add to the Timeline list widget the contents of the array;
                Select the first item in the list;
        ENDIF
        Delete the contents of the array;
END
```

procedure build_time_line()

```
BEGIN
        Set counters;
        Allocate array of XmStrings;
        IF manual test
                Get the manual test filter structure;
        ELSEIF automated test
                Get the automated test filter structure;
        ENDIF
        Get the query filter structure;

        FOR I := 1 TO # of packets in binary file
                IF packet[I] passes test filter
                        IF packet[I] passes query filter
                                Build the contents of string to be displayed in the Timeline;
                                Convert string to XmString and add to XmString array;
                                Increment counter of array elements;
                        ENDIF
                ENDIF
        NEXT
END
```

procedure tl_list_delete()

```
BEGIN
```

```
        IF list of XmStrings to appear in Timeline is not EMPTY
                FOR I := 1 TO # of elements in XmString array
                        Free each element's XmString;
                NEXT
                Free pointer to array of XmStrings;
        ENDIF
END
```

procedure tl_dialog_destroy()

```
BEGIN
        IF Timeline widget exists
                Destroy widget;
                Reset all flags;
                Free memory of local variables;
        ENDIF
END
```

The following are the callback routines executed when the user selects a specific entry in the Timeline or one of the buttons below the Timeline window:

procedure tl_list_select_cb(not used, not used, pointer to line selected in Timeline)

```
BEGIN
        Get actual text for selected line in Timeline window;
        Get the packet number for the selected line;
        Load packet from disk;
        Update the Timeline's text window;
        IF running an automated test
                Fetch the network test's user edit list;
                Fetch the PDU test's user edit list;
        ENDIF
        Freshen Packet Display window;
        IF running automated test
                Free locally defined network header pointer;
        ENDIF
END
```

procedure tl_query_cb(not used)

```
BEGIN
        IF Query window is not already open
                Set Query-window-open flag to TRUE;
                Open Query window;
        Initialize Query window contents;
    ENDIF
END
```

procedure tl_clear_cb(not used)

```
BEGIN
    Set Query-window-open flag to FALSE;
    Destroy Query window;
    Call Packet Filter callback routine to freshen Timeline window;
END
```

6.9.4 Packet Filter - **PF_DIALOG**

6.9.4.1 Overview

The Packet Filter dialog window only appears during manual testing. During automated testing the underlying DB_FILTER_STRUCT is automatically filled in with the appropriate values in the specific automated test function. For manual testing, however, the user can change any of the criteria they want to filter the packets. All packets that passedthe filter appear in the Timeline window.

6.9.4.2 Data Structures

The DB_FILTER_STRUCT structure is integral to the Packet Filter dialog. After the user selects the appropriate criteria in the Packet Filter window for manual testing, those choices are translated and summarized in one place, a DB_FILTER_STRUCT structure. See Section 5.2.2.10 for a detailed description of the DB_FILTER_STRUCT.

6.9.4.3 Processing

The following pseudo code shows how the Packet Filter window is created:

```
procedure pf_dialog_display(mainWindowWidget)
BEGIN
    IF packet filter widget does NOT exist
        Create the widget (window);
        Initialize the widget with default values;
    ENDIF
    Display packet filter widget (window);
END

procedure pf_dialog_create(mainWindowWidget)
BEGIN
    Create popup shell widget for packet filter;
    Create pane for dialog;
    Create pdu types, source id, entity types, and packet info. panes (widgets) and attach
    to popup shell widget;
    Create the buttons;
    Create the packet filter window;
    Manage widgets;
    RETURN popup shell widget;
END
```

```
procedure pf_initialize_display()
BEGIN
    Build the source ID list for displaying in window;
    Build the entity type list for displaying in window;
    Set toggle widgets to appropriate default settings;
    Apply the contents of filter window and show results in Timeline window;
END

procedure pf_Build_Src_Id_List()
BEGIN
    IF source ID list widget does not exist
        RETURN
    ENDIF

    Delete contents of source ID list widget;
    FOR I := 1 TO # of packets
        IF packet[I] is DIS packet
            Check to see if packet[I]'s Source ID is in list widget already;
            IF NOT
                Convert packet[I]'s Source ID to XmString;
                Add XmString to source ID list widget;
                Free XmString;
            ENDIF
        ENDIF
    NEXT
END

procedure pf_Build_Entity_Type_List()
BEGIN
    IF entity type list widget does not exist
        RETURN
    ENDIF

    Delete contents of entity type list widget;
    FOR I := 1 TO # of packets
        IF packet[I] is Entity State PDU packet
            IF packet[I]'s entity type is in Scanner's Entity Type list
                Fetch the textual description of entity type;
            ENDIF
            Check to see if packet[I]'s Entity Type is in list widget already;
            IF NOT
```

```
                        Convert entity type information to XmString;
                        Add XmString to entity type list widget;
                        Free XmString;
                    ENDIF
                ENDIF
        NEXT
END
```

The following is the callback routine called when the user wants to apply the contents of the packet filter window as a filter and show the results in the Timeline window:

procedure pf_dialog_apply_cb(not used)

```
BEGIN
    Clear the local manual filter structure;
    Set the default filter criteria;
    Select all pdu types in manual filter that are selected in packet filter window;
    Convert start/end times from screen values to filterable values;
    Copy values from filter window and set flags within manual filter;
    Fetch all selected entity types and source entity IDs from scroll list and add them to
    manual filter;
    Freshen Timeline display using new filter values;
END
```

6.9.5 Report Results - **RP_DIALOG**

6.9.5.1 Overview

The Report Results window provides the user a means of entering comments and test status information about the current test being processed and having those values saved within the management structure so that the can be printed on a summary report at a later date. This window is a controlling point for both automated and manual testing. The only way to return to the main menu after starting a test is to choose either the "Quit" or "Next" buttons on this window. Those two buttons control the resetting of the main menu as well as the destruction of the currently opened windows.

6.9.5.2 Data Structures

There is a structure defined to contain all "fields" that can be filled in on the screen. The structure contains the name of the field, the displayable length of the field, the total length of the field, a widget, and the string that contains the value input into the "field".

6.9.5.3 Processing

```
procedure rp_dialog_display(mainWindowWidget)
BEGIN
        IF report dialog widget does NOT exist
                Create the widget (window);
                Initialize the widget with default values;
        ENDIF
        Display report dialog widget (window);
END

procedure rp_dialog_create(mainWindowWidget)
BEGIN
        Create popup shell widget for packet filter;
        Create pane for dialog;
        Create and attach each of the "fields" to the pane;
        Create and attach the buttons;
        Manage widgets;
        RETURN popup shell widget;
END

procedure UpdateRpDialog()
```

```
BEGIN
        Copy information from the management structure to the "fields" in the report dialog
        window;
        Write the Report Log test header;
END
```

After the completion of an automated test, the program determines if the binary file being tested passed the criteria of the test or not. The functions called to assign the resulting test status and reasons for failure are listed below:

procedure AssignTestStatus(test_status)

```
BEGIN
        Assign test_status to widget for result field on report dialog window;
        Copy the string value in test_status to the local test status string variable;
END
```

procedure AppendReportReasons(new_reason_str)

```
BEGIN
        Fetch the string from the reason widget;
        IF the length of that string + length of new_reason_str < max. string length
                Concatenate the two strings.
                Put resultant string back into widget;
        ENDIF
END
```

There are several callback routines that are called either upon exiting a field or when the user has selected either the "Quit" or "Next" buttons.

procedure rp_result_exit_cb(results_field_widget, field_structure)

```
BEGIN
        Fetch the field_structure data;
        Fetch the text from the results_field_widget;
        Make sure the result (PASSED, FAILED, ...) is completely spelled out in the text;
        Replace the text back into the results_field_widget;
        Copy the text into the results text string;
        Free text;
END
```

procedure rp_dialog_next_cb()

```
BEGIN
        Update management structure with information in Report dialog window;
        Write the results to the report log;
        Destroy all open windows except main menu;
        Close binary and test-specific configuration files;
        Write the management structure to disk;
        IF manual test mode
                Call manual test selection dialog function;
        ELSEIF automated test mode
                Call automated test selection dialog function;
        ENDIF
END
```

procedure rp_dialog_quit_cb()

```
BEGIN
        Update management structure with information in Report dialog window;
        Write the results to the report log;
        Clear flag remembering last binary file processed;
        Destroy all open windows except main menu;
        Close binary and test-specific configuration files;
        Write the management structure to disk;
        Enable the main menu choices;
END
```

procedure UpdateManagementFromRpDialog()

```
BEGIN
        Convert and copy all pertinent information from the report dialog window to the
        management structure for the appropriate test;
END
```

6.9.6 Query Filter - **QRY_DIALOG**

6.9.6.1 Overview

The Query Filter window is almost exactly the same as the Packet Filter window. The purpose of this window is to allow the user to further narrow the selected set of packets without having to change the baseline filter, set in the Packet Filter window. The only substantial difference between the two is that the Query window will only allow the user to choose from a subset of items that were selected in the Packet Filter window.

This window can only be selected from the Timeline window by selecting the "Query" button on that window. A window very similar to the Packet Filter window will appear with only the Packet-Filter-selected items selectable. To exit the Query window, the user needs to choose the "Clear Query" button on the Timeline.

6.9.6.2 Data Structures

The DB_FILTER_STRUCT structure is integral to the Query Filter dialog. After the user selects the appropriate criteria in the Query Filter window, those choices are translated and summarized in one place, a DB_FILTER_STRUCT structure. See Section 5.2.2.10 for a detailed description of the DB_FILTER_STRUCT.

6.9.6.3 Processing

The following pseudo code shows how the Query Filter window is created:

```
procedure qry_dialog_display(mainWindowWidget)
BEGIN
    IF query filter widget does NOT exist
        Create the widget (window);
        Initialize the widget with default values;
    ENDIF
    Display query filter widget (window);
END

procedure qry_dialog_create(mainWindowWidget)
BEGIN
    Create popup shell widget for query filter;
    Create pane for dialog;
    Create pdu types, source id, entity types, and packet info. panes (widgets) and attach
    to popup shell widget;
```

```
        Create the buttons;
        Create the query filter window;
        Manage widgets;
        RETURN popup shell widget;
END
```

procedure qry_initialize_display()

```
BEGIN
        Build the source ID list for displaying in window;
        Build the entity type list for displaying in window;
        Set toggle widgets to appropriate default settings;
        Apply the contents of filter window and show results in Timeline window;
END
```

procedure qry_Build_Src_Id_List()

```
BEGIN
        IF source ID list widget does not exist
                RETURN
        ENDIF

        Delete contents of source ID list widget;
        FOR I := 1 TO # of packets
                IF packet[I] is DIS packet
                        IF source ID selected in Packet Filter
                                Check to see if packet[I]'s Source ID is in list widget already;
                                IF NOT
                                        Convert packet[I]'s Source ID to XmString;
                                        Add XmString to source ID list widget;
                                        Free XmString;
                                ENDIF
                        ENDIF
                ENDIF
        NEXT
END
```

procedure qry_Build_Entity_Type_List()

```
BEGIN
        IF entity type list widget does not exist
                RETURN
        ENDIF
        Delete contents of entity type list widget;
        FOR I := 1 TO # of packets
                IF packet[I] is Entity State PDU packet
                        IF packet[I]'s entity type is in Scanner's Entity Type list
                                Fetch the textual description of entity type;
                        ENDIF
                        IF entity type is selected in packet filter window
                                Check to see if packet[I]'s Entity Type is in list widget already;
                                IF NOT
                                        Convert entity type information to XmString;
                                        Add XmString to entity type list widget;
                                        Free XmString;
                                ENDIF
                        ENDIF
                ENDIF
        NEXT
END
```

This procedure is called when the user presses the "Clear Query" button on the Timeline window:

procedure ClearQueryFilter()

```
BEGIN
        Set query filter to NULL;
        Destroy the Query Filter window;
END
```

The following is the callback routine called when the user wants to apply the contents of the query filter window as a filter and show the results in the Timeline window:

procedure qry_dialog_apply_cb(not used)

```
BEGIN
        Clear the local query filter structure;
        Set the default filter criteria;
        Select all pdu types in query filter that are selected in query filter window;
        Convert start/end times from screen values to filterable values;
        Copy values from filter window and set flags within query filter;
        Fetch all selected entity types and source entity IDs from scroll list and add them to
        query filter;
        Freshen Timeline display using new filter values;
END
```

6.10 Logger

6.10.1 Interfacing with the IST Logger - **LOG_DIALOG**

6.10.1.1 Overview

To facilitate the scanning process, the Scanner was designed to interface directly with IST's Logger. This is done through a FIFO or pipe. A pipe is actually a file that is opened by both programs and is used as a conduit to send information from one program to another. The Scanner opens the pipe as a write-only file and the Logger opens the file as a read-only file. IST's Logger must open the pipe first before the Scanner can attach to the pipe. This is done to assure that there is a complete connection between the two programs.

Once the pipe is established, the Scanner can send messages from the Logger Dialog window to the Logger. The Logger is set in a loop reading information from the pipe and then acting upon the information. The Scanner sends specific messages based upon the buttons pressed on the dialog window. If the use presses the "RECORD" button, a message is passed to the Logger to begin logging data and saving it into the file specified in the Logger dialog's filename window. The Logger will continue logging until the "STOP" button is pressed.

6.10.1.2 Data Structures

A file pointer is maintained to allow access to the pipe.

6.10.1.3 Processing

The following pseudo code shows the routines to build the logger dialog:

```
procedure logger_dialog_display(mainWindowWidget,  testname, FuncToCallOnExit)
BEGIN
    Create logger dialog window;
    Display logger dialog window;
END

procedure logger_dialog_create(mainWindowWidget,  testname, FuncToCallOnExit)
BEGIN
    Create the application shell widget;
    Create the pane;
    Create a single text field widget for test name;
    Create and add the buttons;
    Manage widgets;
```

```
END
```

The following are the callbacks for the binary file text widget and the buttons:

```
procedure log_file_cb(text_widget, not used)
BEGIN
        Fetch binary filename from text_widget;
        IF EMPTY(binary filename)
                Set temp. variable to "default.bin";
                Set text_widget to "default.bin";
        ELSE
                Copy binary filename to temp. variable
        ENDIF
        IF temp. variable is an invalid filename
                ERROR;
                RETURN;
        ENDIF

        Copy the temp. variable back into the management structure binary filename for the
        current test;

        IF that temp. variable filename already exists
                Set "OK" button to TRUE;
        ENDIF
END

procedure logger_ok_cb(button_widget, pointer_to_func_to_call, not used)
BEGIN
        IF pointer_to_func_to_call NOT NULL
                Call function pointed to by pointer_to_func_to_call;
        ENDIF
        Destroy logger dialog;
END

procedure logger_stop_cb(not used)
BEGIN
        Write to pipe "STOP\n";
        Set sensitivity of STOP button to FALSE;
        Set sensitivity of OK button to TRUE if binary file exists;
        Set sensitivity of RECORD button to TRUE;
END
```

```
procedure logger_record_cb(not used)
BEGIN
        Write to pipe "RECORD " + binary filename + "\n";
        Set sensitivity of STOP button to TRUE;
        Set sensitivity of OK button to FALSE;
        Set sensitivity of RECORD button to FALSE;
        Reset flag remembering last binary file tested;
END
```

The next two routines represent the code necessary to communicate between the Scanner and the IST Logger via the pipe:

```
procedure logger_open(filename)
BEGIN
        IF EMPTY(filename)
                RETURN;
        ENDIF
        IF !FileExists(filename)
                RETURN
        ENDIF

        Open a pipe for write only access;
        IF pipe has NOT been previously opened in read only access from the Logger
                RETURN;
        ENDIF

        Change the pointer to the pipe to a stream pointer so that the stream can be flushed
        after a write;
END
```

```
procedure logger_write( string_to_write )
BEGIN
        IF pipe is open
                Write string to pipe;
                Flush pipe;
        ELSE
                Print string to screen;
        ENDIF
END
```

6.11 Reports

6.11.1 Status - **RPTSTATUS**

6.11.1.1 Overview

The Status report includes a one line summation of the information about each scheduled test. As mentioned in Section 5, the Scanner creates all reports as a text file. The files are created in the management structures directory in order to keep everything together.

6.11.1.2 Data Structures

The management structure is used as the source for the information that is printed on the report.

6.11.1.3 Processing

```
procedure ReportStatusSummary()
BEGIN
    Create a report text file in the management structure directory;
    Write the testbed information to the file;
    Write the company information to the file;
    Write the SUT information to the file;
    FOR I := 1 TO # of tests groups
        Write header for group to file;
        FOR J := 1 TO # of test entries
            IF test[I,J] is scheduled
                Write test number, description, type, and status to the file;
            ENDIF
        NEXT

    NEXT
    Close the report file;
END
```

6.11.2 Results - **RPTRESULTS**

6.11.2.1 Overview

The Results report includes all of the information about each scheduled test. As mentioned in Section 5, the Scanner creates all reports as a text file. The files are created in the management structures directory in order to keep everything together.

6.11.2.2 Data Structures

The management structure is used as the source for the information that is printed on the report.

6.11.2.3 Processing

```
procedure ReportResultsSummary()
BEGIN
    Create a report text file in the management structure directory;
    Write the testbed information to the file;
    Write the company information to the file;
    Write the SUT information to the file;
    FOR I := 1 TO # of tests groups
        Write header for group to file;
        FOR J := 1 TO # of test entries
            IF test[I,J] is scheduled
                Write test number, description, type, and status to the file;
                Write # of times run, binary filename, test-specific configuration
                filename, reason field, and comments to file;
            ENDIF
        NEXT

    NEXT
    Close the report file;
END
```

6.11.3 Report Log - **RPTLOGGER**

6.11.3.1 Overview

The report log is a text file stored in the same directory as the management structure. The file is a record of each test that was run within that management structure. A summary of the tester, test name, times run is written for each test that is run. Also, if any errors occur during initial validation or specific validation, an error message is written to the log, listing the packet number in question, the specific field, it's value, and the min/max validation values.

As each test is run, information about each test is appended to the end of this file. The report log could grow to a very large size if many errors occur during testing and a large number of tests are run. The Scanner was designed in such a way that if that file does not exist, the program will automatically create it. Therefore, if the file gets too large, it can be erased and the Scanner will still function properly.

6.11.3.2 Data Structures

The management structure is used as the source for the information that is written to the report log.

6.11.3.3 Processing

```
procedure ReportLogOpen(filename)
BEGIN
    Open filename as a text file in read/write append mode;
END

procedure ReportLogTestHeader( testGroup, testEntry )
BEGIN
    IF report log is open
        Write testbed and test information to the report log for the current test;
    ENDIF
END

procedure ReportLogResults( testGroup, testEntry )
BEGIN
    IF report log is open
        Write test result, reason and comments to the report log;
    ENDIF
END
```

7 APPENDIX A

7.1 Scanner Development Environment

Described below are the environment variables that need to be set up for the Scanner development environment to work correctly. These are normally placed in the file, .profile

```
#
# Allow the developer and the Scanner to create files that are
# accessible by everyone in the development team.
#
umask 002

#
# Define the base directory for the Scanner development
# environment. The Makefiles use this environment variable
# to locate files.
#
SCANDIR=/usr/tridis/proj/scanner
```

7.2 Scanner Runtime Environment

Defined below are the environment variables that need to be set up for the Scanner runtime environment to work correctly. These are normally placed in the file, .profile

```
#
# Allow the Scanner to create files that are
# accessible by everyone in the test team.
#
umask 002

#
# Define the Scanner runtime configuration directory.
# If the config files are not in the current directory, the
# Scanner will look in this directory for the configuration files.
#
SCANPROJDIR=/usr/tridis/proj/scanner/config
```

8 APPENDIX B

8.1 Hardware Requirements

The Scanner requires a Unix System based on a big endian architecture, and running X Windows Version X11R5, Motif Version 1.2, and System V. The X Window and Motif versions are minimum versions. The Scanner should not have a problem with later versions of X Windows or Motif. Currently, hardware based on the little endian architecture is not supported. The Scanner could be modified to run on both environments with a minimal set of changes. The Motorola system that the Scanner has been running on, has 64 megabytes of system ram. The Scanner does not require this much memory. We have not officially tested for the minimal memory requirements. The variable part of the equation is the size of the binary file you wish to support without having to page swap. Keep in mind that X Windows consumes a large chunk of memory while running.

The current development environment for the Scanner requires approximately 50 megabytes of disk space. This includes all source and object files, configuration files, libraries, logged binary files for testing purposes, two management directories for testing purposes, the Scanner executable, and a few tar files holding the last few versions of the Scanner source code.

The current runtime environment is based on the number of test suites supported. The Scanner and it's configuration files are under 8 megabytes in size. The actual size depends on whether or not the Scanner is built with the debug options turned off. The Scanner currently has all debug options turned on. Each test suite requires it's own directory. The test suite directory contains a 2 megabyte management structure, report logs, and binary log files. The number of test suites supported is based on available disk space.

A Silicon Graphics version has been built and tested. The code is exactly the same as the Motorola's version except that a "SGI_VERSION" #define is passed to the compiler to handle a difference in include file declarations between the two environments.

APPENDIX B

Scanner Users' Manual

Scanner Management System

Users' Manual

Institute for Simulation and Training
3280 Progress Drive
Orlando, Florida 32826

University of Central Florida
Division of Sponsored Research

1 Introduction

This Users' Manual shows how to use the Scanner Management System to aid in the testing of systems that plan on participating in a Distributed Interactive Simulation (DIS). The Scanner is a utility program designed to aid in testing the interoperability requirements of systems that will be participating in a Distributed Interactive Simulation. The Scanner creates a test suite (or data base) for each System Under Test (SUT) being tested. This provides the user a place to electronically store the results of each and every test as opposed to having to record all test results on paper. The test suite is saved in its own, unique directory where all files associated with the test suite will also be kept.

For each test, the Scanner reads a binary (BIN) file logged using the Institute for Simulation and Training's (IST) Logger and performs an initial range validation on all of the fields within each DIS packet in the file looking for packets within the BIN file that contain erroneous data. The Scanner has two different test modes, Manual and Automated. The Manual mode allows the user to look at any packets in the BIN file based on filter criteria they specify. The user after looking at the desired packets then fills in a report results window with the results they feel are appropriate. In the Automated mode, the program runs the testing and evaluating of the appropriate packets in a BIN file and automatically fills in the report results windows with the correct results. Currently, only a subset of the possible tests to choose from can be run in the Automated mode.

After each test is run and the report results window is correctly filled in, the program will save all of the results within the test suite. Also, all information encountered during the running of the test, to include errors found in a packet, are written to a report log. The report log can be considered a diary of each test that was run and what happened during the running of the test. At any time, the user can generate either of the two report formats supported by the Scanner. The Status and Summary reports, in different levels of specificity, list each test that was selected to be run and the results of each.

2 Scanner Menu Options

From the main menu bar the user can access the features of the *Scanner Management System*. There are four options available on the main menu bar: *File, Edit, Testing,* and *Report*. Each of these features is described in the sections below.

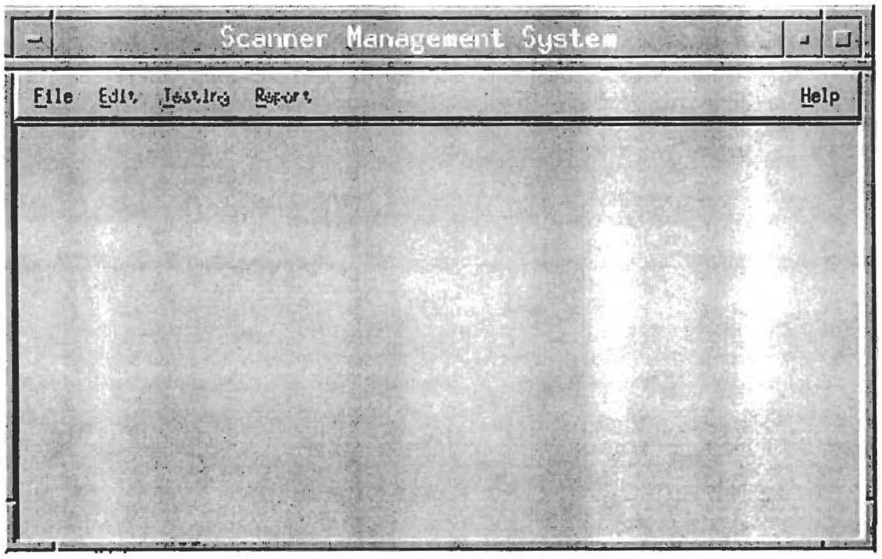


Figure 1 - Scanner Main Menu

2.1 The **File** Option

The *File* option allows the user to perform various storage/retrieval functions on Scanner test suites. A test suite or data base is a structured file that contains all of the information about the System under Test (SUT), to include, which tests to run and the results of those tests. The test suite can be found in a directory below the current directory with a .mgt extension. The report log, with a ".log" extension, can also be found in that directory. The binary logged files should also be logged to this directory.

The File functions are:

New:

This option permits the user to create a new test suite. Information about the Testbed, Company, and System Under Test (SUT) is entered here.

Upon selecting the **New** option, the user will be prompted to input a 3-character company abbreviation and a 1-character SUT number.

NOTE: The Scanner will concatenate the company abbreviation and the SUT number to form a unique name. A subdirectory with that name will be created below the directory where the user is working. All files created by the Scanner to build a test suite will be stored in that directory. By default, the Scanner creates both a test suite management file and a report log file using the unique name. Do not rename any of these files or the subdirectory, otherwise, the Scanner will not work properly!

Also, this directory is where the Scanner will default to looking for all logged binary files when testing commences. The user should either make sure all BIN files are in the test suite directory or change the data directory value in the testbed section to the correct default directory.

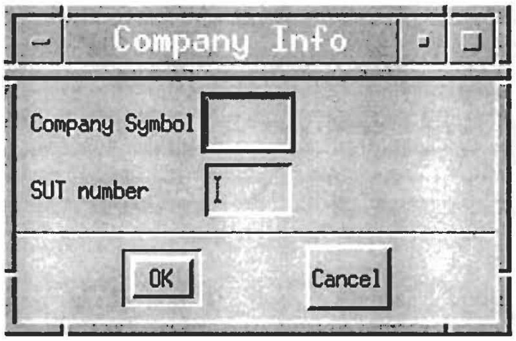
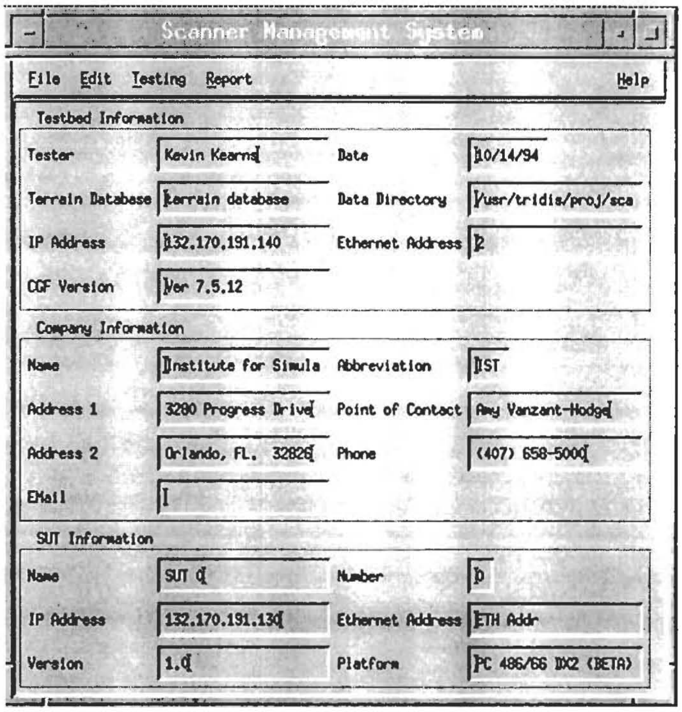


Figure 2 - New Test Suite Dialog



Testbed Information

Tester	Kevin Kearns	Date	10/14/94
Terrain Database	terrain database	Data Directory	/usr/tridis/proj/sca
IP Address	132.170.191.140	Ethernet Address	2
CGF Version	Ver 7.5.12		

Company Information

Name	Institute for Simula	Abbreviation	IST
Address 1	3280 Progress Drive	Point of Contact	Amy Vanzant-Hodge
Address 2	Orlando, FL. 32826	Phone	(407) 658-5000
EMail			

SUT Information

Name	SUT 0	Number	0
IP Address	132.170.191.130	Ethernet Address	ETH Addr
Version	1.0	Platform	PC 486/66 DX2 (BETA)

Figure 3 - Scanner SUT-Testbed-Company Info

Once the test suite is created, the user will need to enter information on the following:

Testbed Information

Tester:	Name of the user running the test.
Date:	The date the test suite was created.
Terrain Database:	The name of the terrain database being used.
Data Directory:	The directory where the binary logged files can be found. (*The default is the directory created by the Scanner that holds the test suite management file, however, there is no requirement to keep the binary data files in this directory.*)
IP Address:	The IP address of the testbed. The proper format for this field is ###.###.###.###. (*Very important that this field is properly filled in because the Scanner uses this field to determine which packets were generated by the Testbed and which packets were generated by the SUT.*)
Ethernet Address:	The ethernet address of the company performing the test.
CGF Version:	The version of the CGF being used for the test.

Company Information

Name:	Name of the company who designed the SUT.
Abbreviation:	3-character abbreviation of company name (inserted automatically from previously entered information). (**NOTE:** *If the user changes this field or the SUT number field, the program will try to reference the directory created by concatenating the company name and SUT number when creating reports.*)
Address 1:	Street address of company.
Address 2:	City, State, Zip Code of company.
Point of Contact:	Person to contact at the company regarding testing information.

E-Mail:	E-mail address.
Phone:	Phone number.

SUT Information

Name:	Name of the simulator being tested.
Number:	Number associated with the SUT.
IP Address:	IP address of the SUT.
Ethernet Address:	Ethernet address of the SUT.
Version:	Internal company version number of the SUT.
Platform:	Platform the SUT runs under.

Open:

This option permits the user to open a previously created test suite. To select a test suite left-click on the suite name and then left-click on the "OK" button. (*Note, if the user has a test suite open and happens to choose this option again, he will notice that the file selection window will not show the open test suite in the list.*)

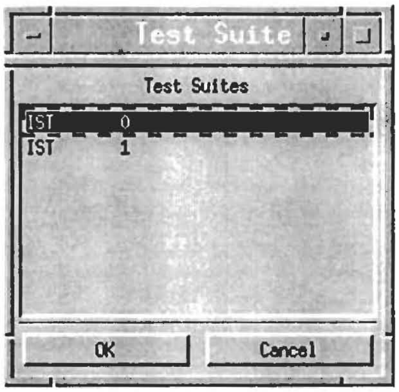


Figure 4 - Test Suite Selection Dialog

Close:

This option closes the currently open test suite saving any changes made to the test suite.

Save:

This option saves the current test suite.

Delete:

This option deletes a test suite and all related data files. The user is presented with a list of all unopened test suites. After selecting the test suite to delete, this option will physically remove ***ALL*** data stored in the directory where the test suite is saved. It will also remove the directory. Note that the user can not delete a currently opened test suite.

> NOTE: The user may want to back-up all of the data before executing this option. Once a test suite is deleted, it cannot be recovered!

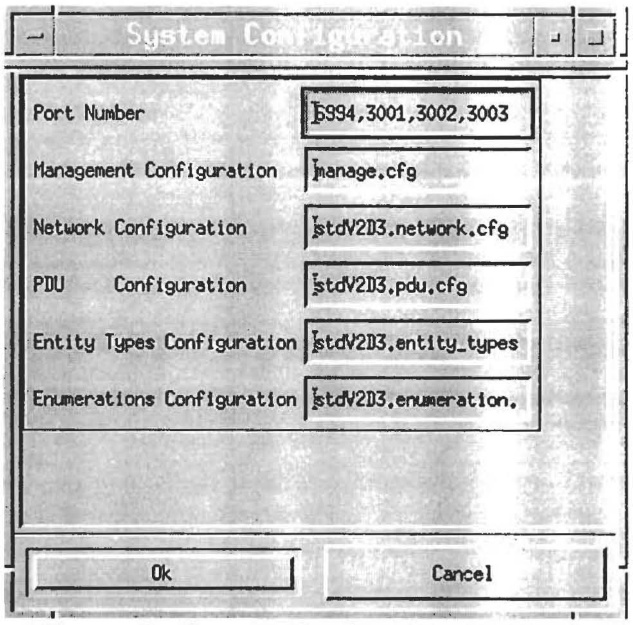
Quit:

This option exits the *Scanner Management System* and returns the user to the operating system. The system will automatically save any changes before exiting.

2.2 The **Edit** Option

The *Edit* option allows the user to configure the *Scanner Management System* for specific testing purposes. Clicking on the *Edit* option will pull down a menu with several areas to choose from including editing System Configuration or any of the multiple levels of tests.

The **System Configuration** option allows the user to change the names of the configuration files that the Scanner uses. The Scanner loads the default configuration files at startup. If the user wants to use different configuration files or ports, go to this screen and change the desired values and the new values will be automatically loaded each time the test suite is opened.



Port Number	5994,3001,3002,3003
Management Configuration	manage.cfg
Network Configuration	stdV2D3.network.cfg
PDU Configuration	stdV2D3.pdu.cfg
Entity Types Configuration	stdV2D3.entity_types
Enumerations Configuration	stdV2D3.enumeration.

Figure 5 - System Configuration Window

The following is a list of all of the different levels of tests defined in the Institute for Simulation and Training's *Technical Report, Test Documents for DIS Interoperability*. See that document for specific definitions of what the criteria is for each of these tests. The tests include Network, PDU, Terrain Orientation, Appearance, Interactivity, System Test, Manned Simulator, Protocol Translator, and Capabilities tests.

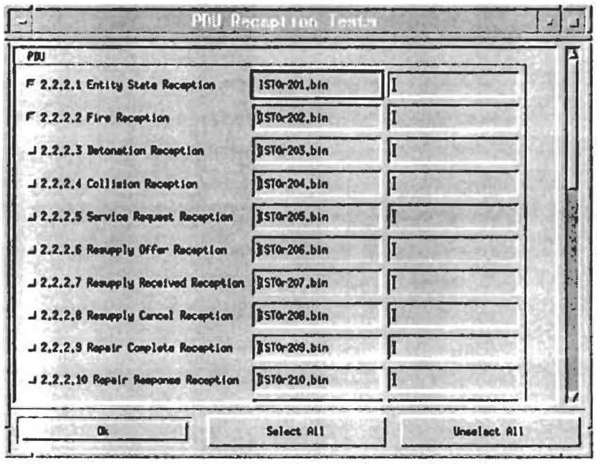
To the right of the test name is the name for the BIN file associated with that particular test. The default name for the test is displayed but the user may change this. The default name is made up of the 3-character company name and the 1-character SUT number as well as the letter t,r,a, or e followed by a 3-character unique test number. The letters t,r,a, or e reflect Transmission, Reception, Adverse, or Erroneous tests, respectively.

After the BIN file name is a field to enter a specific configuration file for that test. Most tests use the default configuration values for the acceptable ranges for each field in a packet, however, some tests need to have more specific values than the defaults. This field allows the user to specify the name of a properly formatted configuration file to that has values to override the defaults. The format of this file is very similar to PDU or NETWORK configuration files, in that the user specifies the group name, field name and MIN/MAX values for the field. The following is an example of a Network Test-specific Configuration File:

#Group	Field	MIN	MAX
"ETHERNET"	"DESTINATION_ADDRESS[0]"	"255"	"255"
"ETHERNET"	"DESTINATION_ADDRESS[1]"	"255"	"255"
"ETHERNET"	"DESTINATION_ADDRESS[2]"	"255"	"255"
"ETHERNET"	"DESTINATION_ADDRESS[3]"	"255"	"255"
"ETHERNET"	"DESTINATION_ADDRESS[4]"	"255"	"255"
"ETHERNET"	"DESTINATION_ADDRESS[5]"	"255"	"255"
"IP"	"DESTINATION_ADDRESS[0]"	"164"	"164"
"IP"	"DESTINATION_ADDRESS[1]"	"217"	"217"
"IP"	"DESTINATION_ADDRESS[2]"	"255"	"255"
"IP"	"DESTINATION_ADDRESS[3]"	"255"	"255"
"ETHERNET"	"ETHER_TYPE"	"0800"	"0800"
"IP"	"PROTOCOL"	"17"	"17"
"UDP"	"DESTINATION PORT"	"6994"	"6994"

NOTE: Each Group, Field combination must be entered in the EXACT format that they exist in the NETWORK.CFG file. Also, the MIN/MAX values must be entered in this file in the same data format as they are to be read from a packet, i.e., if a field is to be read as if it were in hexadecimal format, then it must be entered as "0800" not "2048" to be properly parsed.

To the left of the test name is a box (button). If the button is pushed in, the test is selected. Left-click on the button to the left of the test name to select that test. Left-clicking on the **Select All** button will select all tests in the list, left-clicking on the **Clear All** button will unselect all tests.

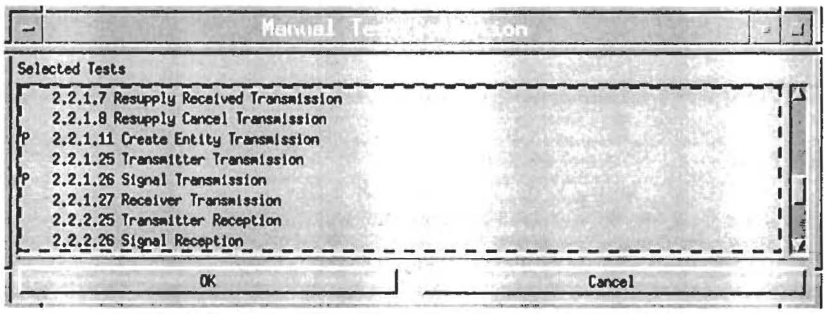


[✓] 2.2.2.1 Entity State Reception	ISTOr201.bin	I
[✓] 2.2.2.2 Fire Reception	ISTOr202.bin	I
2.2.2.3 Detonation Reception	ISTOr203.bin	I
2.2.2.4 Collision Reception	ISTOr204.bin	I
2.2.2.5 Service Request Reception	ISTOr205.bin	I
2.2.2.6 Resupply Offer Reception	ISTOr206.bin	I
2.2.2.7 Resupply Received Reception	ISTOr207.bin	I
2.2.2.8 Resupply Cancel Reception	ISTOr208.bin	I
2.2.2.9 Repair Complete Reception	ISTOr209.bin	I
2.2.2.10 Repair Response Reception	ISTOr210.bin	I

Figure 6 - Test Selection Window

2.3 The **Testing** Option

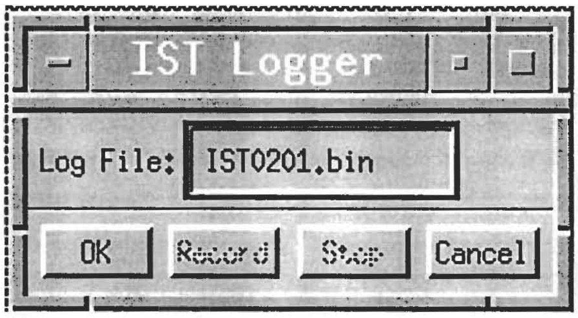
The *Testing* option allows the user to perform **Automated** or **Manual** tests. After selecting either *Manual* or *Automated* Testing, the user is presented with a window listing all of the tests selected in the **Edit** option. If the test has been previously run, the first character on the line will contain the first character of the result of the test, (**P** - Passed, **F** - Failed, or **I** - Incomplete). Also, if any comments were written about the test, a portion of them will appear after the name of the test.



```
 2.2.1.7 Resupply Received Transmission
 2.2.1.8 Resupply Cancel Transmission
P 2.2.1.11 Create Entity Transmission
 2.2.1.25 Transmitter Transmission
P 2.2.1.26 Signal Transmission
 2.2.1.27 Receiver Transmission
 2.2.2.25 Transmitter Reception
 2.2.2.26 Signal Reception
```

Figure 7 - Test Selection Dialog

After selecting the test to run and clicking the *OK* button, the **IST Logger Dialog** appears. If there is a pipe setup between the Scanner and the Logger, all of the buttons in the **IST Logger Dialog** are functional. Otherwise, only the *OK* button is active. To begin testing, click on the *OK* button.



Log File: IST0201.bin

Figure 8 - IST Logger Dialog

If the BIN file exists, the *OK* button is active. If the BIN file does not exist, the user MUST create the BIN file. The name of the BIN file to create is displayed in the Log File text window and is the same name as was listed in the **Edit** test option. If the user types a new name in the Log File text window, that name will automatically overwrite the default name in the **Edit** test option BIN file name field for the current test. ***(NOTE:** The Scanner looks for all BIN files in the directory specified in the **Testbed - Data Directory** field.*) To either create the BIN file necessary for the test or re-log the BIN file, choose the *Record* button. To stop recording, press the *Stop* button.

For all tests, the Scanner performs an initial validation on the BIN file to be tested checking both the network and PDU information. The validation scans all fields in every packet to see if the values in those fields fall within specified parameters. If any errors are found, the field, its value, and the acceptable values are written to the report log. As each test is run, all information about that test (i.e., test results, packet validation results, etc.) is written to the report log file.

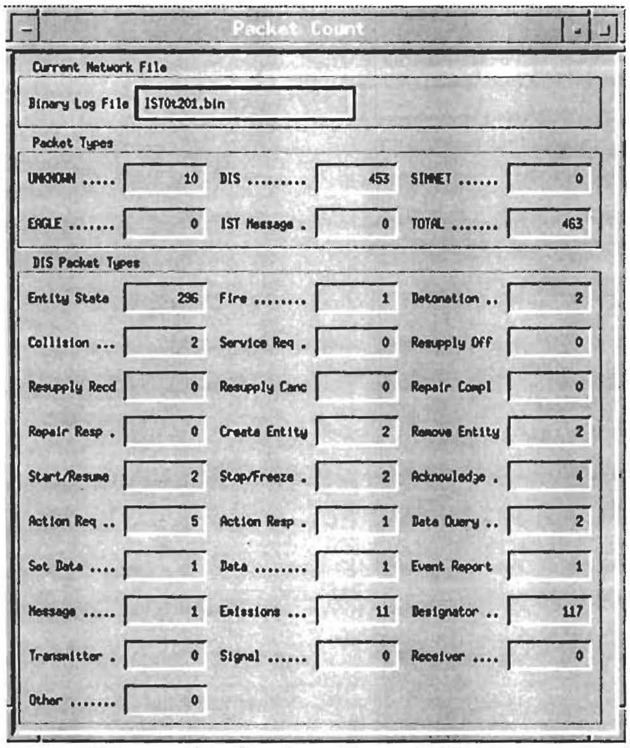
2.3.1 Automated Testing

After selecting the test to run and clicking the "OK" button, three windows will appear: Packet Count, Report, and the User Edit List specific for the desired test.

NOTE: Only Level 1 - Network and Level 2- PDU tests are currently active as Automated tests.

NOTE: In the Automated test mode, the Network and PDU Reception, Adverse, and Erroneous tests used the following logic to determine failure: If a packet in the BIN file was found to be non-Testbed-generated or contain non-DIS information, the BIN file failed that specific test.

Packet Count: The Packet Count window displays the number of packet types in the file of the following types: DIS, SIMNET, EAGLE, IST Message, Total, and Unknown. For DIS packets, the count is broken down further for each specific DIS PDU packet type.



Current Network File

Binary Log File: IST0t201.bin

Packet Types

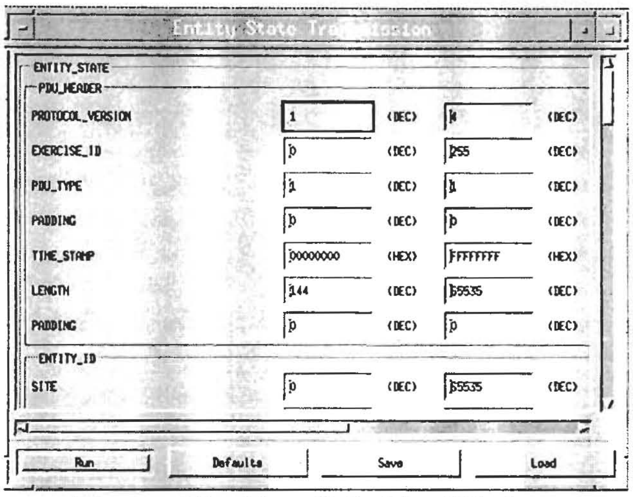
UNKNOWN	10	DIS	453	SIMNET	0
EAGLE	0	IST Message .	0	TOTAL	463

DIS Packet Types

Entity State	296	Fire	1	Detonation ..	2
Collision ...	2	Service Req .	0	Resupply Off	0
Resupply Recd	0	Resupply Canc	0	Repair Compl	0
Repair Resp .	0	Create Entity	2	Remove Entity	2
Start/Resume	2	Stop/Freeze .	2	Acknowledge .	4
Action Req ..	5	Action Resp .	1	Data Query ..	2
Set Data	1	Data	1	Event Report	1
Message	1	Emissions ...	11	Designator ..	117
Transmitter .	0	Signal	0	Receiver	0
Other	0				

Figure 9 - Packet Count Window

User Edit List: The User Edit List window can only be found in the Automated test mode. For Network tests, this window contains all of the different types of network headers recognized by the Scanner and the fields that make up each header. For PDU tests, this window contains a listing of all of the fields of the PDU type being tested.



ENTITY_STATE

PDU_HEADER

PROTOCOL_VERSION	1	(DEC)	4	(DEC)
EXERCISE_ID	0	(DEC)	255	(DEC)
PDU_TYPE	1	(DEC)	1	(DEC)
PADDING	0	(DEC)	0	(DEC)
TIME_STAMP	00000000	(HEX)	FFFFFFFF	(HEX)
LENGTH	144	(DEC)	65535	(DEC)
PADDING	0	(DEC)	0	(DEC)

ENTITY_ID

SITE	0	(DEC)	65535	(DEC)

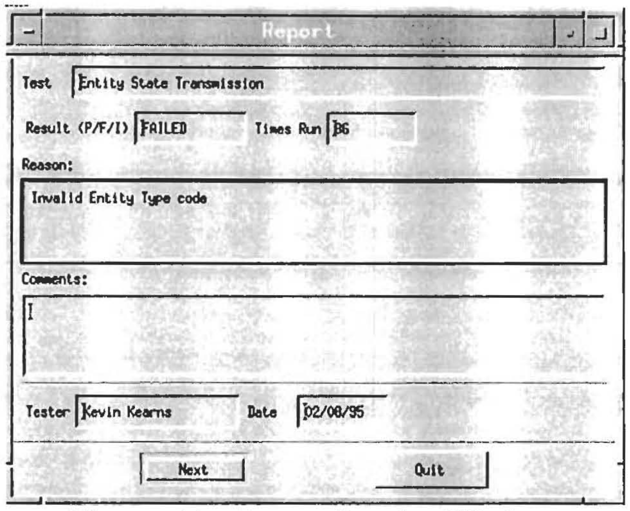
Figure 10 - User Edit List Window

When the Scanner performs its initial packet validation it uses default values for the range checking. The User Edit List allows the user to change the default values to be more specific and then have the Scanner use the new values to perform the packet validation. For example, the **Exercise ID** field will allow any value between 0 and 255 as its default. If the user wants to make sure all packets contain an **Exercise ID** of 1, he just needs to change the 0 to 255 range to be 1 to 1 and then perform the test.

After changing the desired values in the **User Edit List** window, choose the *Run* button to start the Automated test. The program will use the values specified in the **User Edit List** and compare them against all packets matching the requirements of the specific test being run. If any errors were found that would cause the test to fail, they are written to the report log. The report window's test status will automatically be filled in with the appropriate value of **Passed** or **Failed**. If the test failed, the report window's reason field will contain text stating that "Specific Validation failed".

Report: The Report window is where the results of the test are entered before they are saved to the test suite. In Automated testing, the test result field is automatically filled by the program after the test is completed. If any errors occurred, the reason field is also filled in. If so desired, the user can change any of the filled-in values or add comments after the test is complete.

Clicking on the *Quit* button in this window exits the automated testing and removes all related windows. After closing this window, all test information is written to the report log file. Clicking on the *Next* button will close all windows, write the test information to the report log, and return the user to the Automated Test Selection window.


Report

Test: Entity State Transmission

Result (P/F/I): FAILED   Times Run: 86

Reason:

Invalid Entity Type code

Comments:

Tester: Kevin Kearns   Date: 02/08/95

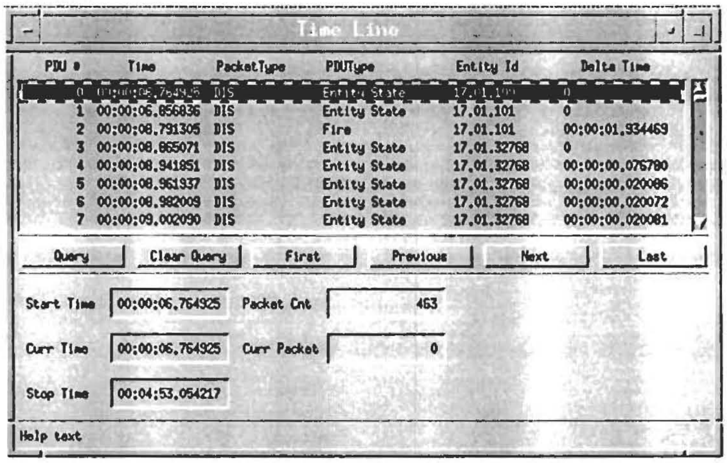
Next   Quit

Figure 11 - Report Results Window

To facilitate quickly determining where specific errors occurred and in which packets, after the test is run two more windows will appear, Time Line and Packet Display.

Timeline: The Timeline window shows, in a list format, all of the packets that met the requirements of the specific test being run. The window shows the number of the packet within the BIN file; time the packet was logged; what type of packet it is; if it is a DIS PDU, what type of DIS PDU the packet is; the source Entity ID number from the packet; and the delta-time from last packet with the same source Entity ID. If the packet does not pass the specific packet validation using the **User Edit List**, the first character in the line will be an '*'.

The Timeline window also displays the number of total packets, the number of the current packet highlighted, the start and end times of the packets, and the time of the current packet highlighted.


Time Line

PDU #	Time	PacketType	PDUType	Entity Id	Delta Time
0	00:00:06.764925	DIS	Entity State	17.01.101	0
1	00:00:06.856836	DIS	Entity State	17.01.101	0
2	00:00:08.791305	DIS	Fire	17.01.101	00:00:01.934469
3	00:00:08.865071	DIS	Entity State	17.01.32768	0
4	00:00:08.941851	DIS	Entity State	17.01.32768	00:00:00.076780
5	00:00:08.961937	DIS	Entity State	17.01.32768	00:00:00.020086
6	00:00:08.982009	DIS	Entity State	17.01.32768	00:00:00.020072
7	00:00:09.002090	DIS	Entity State	17.01.32768	00:00:00.020081

Start Time: 00:00:06.764925   Packet Cnt: 463

Curr Time: 00:00:06.764925   Curr Packet: 0

Stop Time: 00:04:53.054217

Help text

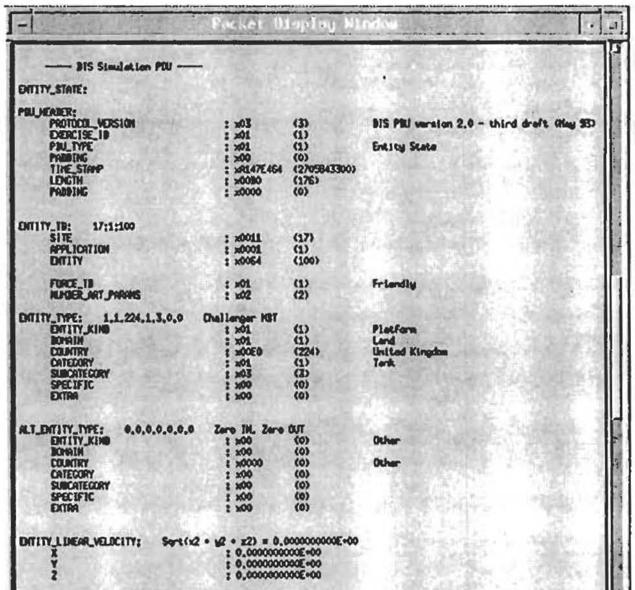
Figure 12 - Timeline Window

The user can select a packet to be displayed in the Packet Display window by clicking the left mouse button on the desired packet in the Timeline window. Below the Timeline list there are six buttons. Query allows the user to display a query window, that allows the user to further narrow what items will show up in the Timeline. Clear Query removes the Query window and resets the Timeline window back to all packets initially shown in the Time Line window. The First, Next, Last, and Previous buttons will take the user to the first record in the list, next record in the list, last record in the list and the previous record in the list, respectively.

Note: Use Clear Query to close the Query window as opposed to choosing the Close in the upper left hand corner of the window to avoid crashing the program.

Packet Display: The Packet Display window displays the contents of the packet highlighted in the Timeline window. At the top of the window will be a hex dump of the entire contents of the packet. This is followed by the network header and the specific DIS PDU type. Each field of the network header and the PDU is displayed on its own line. If the field has an associated enumerated table, the appropriate enumeration is displayed at the end of the line. For groups, such as Entity ID or Entity Type, the group name is displayed and then each element of the group is listed afterwards. If any field or group failed specific validation, the beginning of the line will contain an '*' for quick identification of problem areas within the packet.

Note: A useful trick for quickly viewing items in the packet display window, is to position the screen where all of the desired information is displayed and then click the left mouse button with the mouse in the bottom right corner of the Packet Display window. Then when the user moves to the next packet, the window will automatically scroll down and position itself at the same position that the screen was at when the user clicked the mouse.



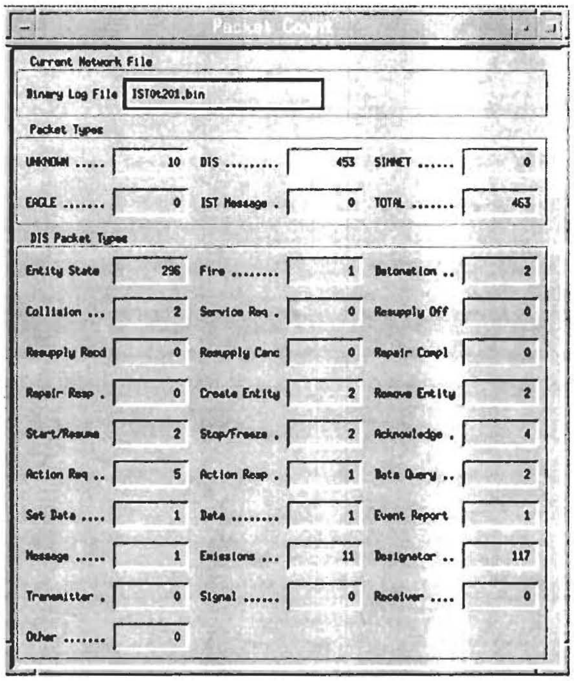
```
    ---- DIS Simulation PDU ----

ENTITY_STATE:

PDU_HEADER:
      PROTOCOL_VERSION          : x03        (3)           DIS PDU version 2.0 - third draft (May 93)
      EXERCISE_ID               : x01        (1)
      PDU_TYPE                  : x01        (1)           Entity State
      PADDING                   : x00        (0)
      TIME_STAMP                : xA147E464  (2705843300)
      LENGTH                    : x00B0      (176)
      PADDING                   : x0000      (0)


ENTITY_ID:    17:1:100
      SITE                      : x0011      (17)
      APPLICATION               : x0001      (1)
      ENTITY                    : x0064      (100)

      FORCE_ID                  : x01        (1)           Friendly
      NUMBER_ART_PARAMS         : x02        (2)

ENTITY_TYPE:    1.1.224.1.3.0.0    Challenger MBT
      ENTITY_KIND               : x01        (1)           Platform
      DOMAIN                    : x01        (1)           Land
      COUNTRY                   : x00E0      (224)         United Kingdom
      CATEGORY                  : x01        (1)           Tank
      SUBCATEGORY               : x03        (3)
      SPECIFIC                  : x00        (0)
      EXTRA                     : x00        (0)


ALT_ENTITY_TYPE:    0.0.0.0.0.0.0    Zero IN. Zero OUT
      ENTITY_KIND               : x00        (0)           Other
      DOMAIN                    : x00        (0)
      COUNTRY                   : x0000      (0)           Other
      CATEGORY                  : x00        (0)
      SUBCATEGORY               : x00        (0)
      SPECIFIC                  : x00        (0)
      EXTRA                     : x00        (0)


ENTITY_LINEAR_VELOCITY:    Sqrt(x2 + y2 + z2) = 0.0000000000E+00
      X                         : 0.0000000000E+00
      Y                         : 0.0000000000E+00
      Z                         : 0.0000000000E+00
```

Figure 13 - Packet Display Window

2.3.2 Manual Testing

After selecting the test to run and clicking the "OK" button, the five manual test screens are displayed. These screens are: Packet Count, Timeline, Filter, Packet Display, and Report.

Packet Count: The Packet Count window displays the number of packet types in the file of the following types: DIS, SIMNET, EAGLE, IST Message, Total, and Unknown. For DIS packets, the count is broken down further for each specific DIS PDU packet type.


Packet Count

Current Network File

Binary Log File: IST0t201.bin

Packet Types

UNKNOWN	10	DIS	453	SIMNET	0
EAGLE	0	IST Message .	0	TOTAL	463

DIS Packet Types

Entity State	296	Fire	1	Detonation ..	2
Collision ...	2	Service Req .	0	Resupply Off	0
Resupply Recd	0	Resupply Canc	0	Repair Compl	0
Repair Resp .	0	Create Entity	2	Remove Entity	2
Start/Resume	2	Stop/Freeze .	2	Acknowledge .	4
Action Req ..	5	Action Resp .	1	Data Query ..	2
Set Data	1	Data	1	Event Report	1
Message	1	Emissions ...	11	Designator ..	117
Transmitter .	0	Signal	0	Receiver	0
Other	0				

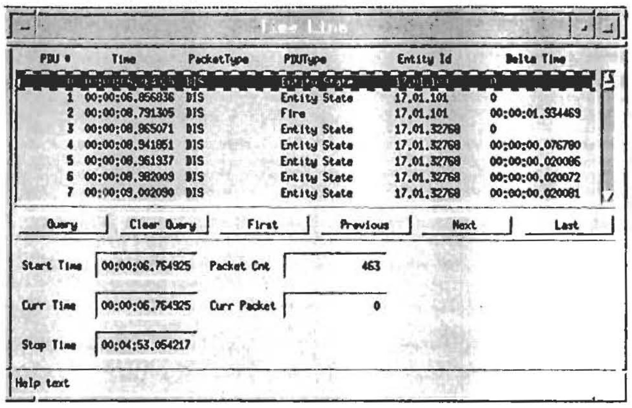
Figure 14 - Packet Count Window

Timeline: The Timeline window shows, in a list format, select information about each packet that meets the packet filter criteria. The window shows the number of the packet within the BIN file, time the packet was logged, what type of packet it is, what type of DIS PDU the packet is, the source Entity ID number from the packet, and the delta-time from last packet with the same source Entity ID. If the packet does not pass the initial packet validation, the first character in the line will be an '*.'

The Timeline window also displays the number of total packets, the number of the current packet highlighted, the start and end times of the packets, and the time of the current packet highlighted. Initially, the Timeline will default to showing all DIS-only, non-testbed generated packets.

The user can select a packet to be displayed in the Packet Display window by clicking the left mouse on the desired packet in the Timeline window. Below the Timeline list there are six buttons. Query allows the user to display a query window, very similar to the Packet Filter window, that allows the user to further narrow what items will show up in the Timeline. Clear Query removes the Query window and resets the Timeline window back to all packets that pass the Packet filter window. The First, Next, Last, and Previous buttons will take the user to the first record in the list, next record in the list, last record in the list and the previous record in the list, respectively.

Note: Use Clear Query to close the Query window as opposed to choosing the Close in the upper left hand corner of the window to avoid crashing the program.



```
PDU #  Time              PacketType  PDUType        Entity Id     Delta Time
    0  00:00:06.764925   DIS         Entity State   17.01.101     0
    1  00:00:06.856836   DIS         Entity State   17.01.101     0
    2  00:00:08.791305   DIS         Fire           17.01.101     00:00:01.934469
    3  00:00:08.865071   DIS         Entity State   17.01.32768   0
    4  00:00:08.941851   DIS         Entity State   17.01.32768   00:00:00.076780
    5  00:00:08.961937   DIS         Entity State   17.01.32768   00:00:00.020086
    6  00:00:08.982009   DIS         Entity State   17.01.32768   00:00:00.020072
    7  00:00:09.002090   DIS         Entity State   17.01.32768   00:00:00.020081
```

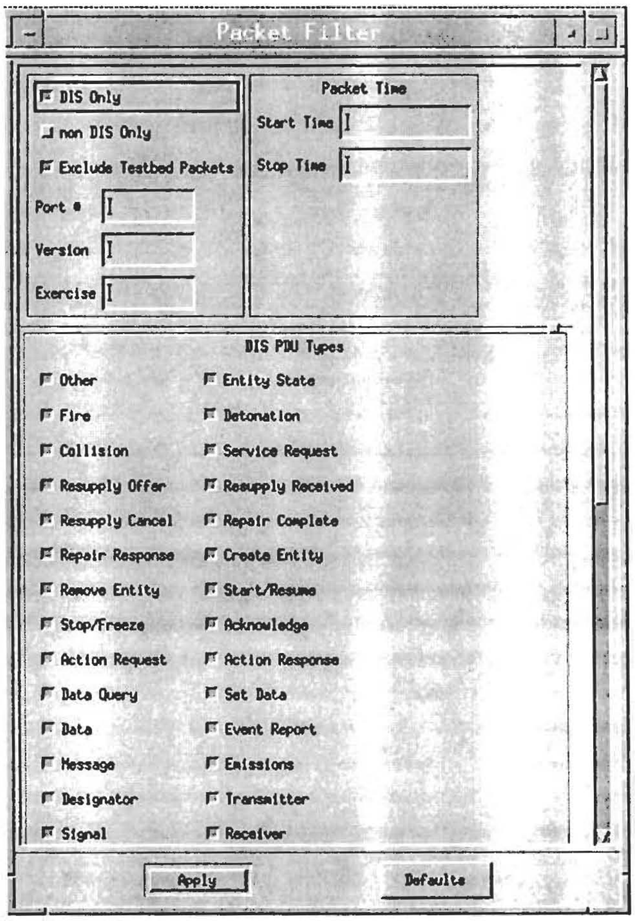
Start Time 00:00:06.764925 Packet Cnt 463

Curr Time 00:00:06.764925 Curr Packet 0

Stop Time 00:04:53.054217

Figure 15 - Timeline Window

Packet Filter: The Packet Filter allows the user to define which packets are to be displayed in the Timeline window. The user can select/deselect based on Entity Types, Entity IDs, DIS PDU types, start/end time frames, Exercise number, and Port number. The filter can be set to show DIS only packets, non-DIS only packets (i.e. unrecognizable packets), all packets, and exclude Testbed generated packets.



[✓ DIS Only]
non DIS Only
[✓ Exclude Testbed Packets]
Port #
Version
Exercise

Packet Time
Start Time
Stop Time

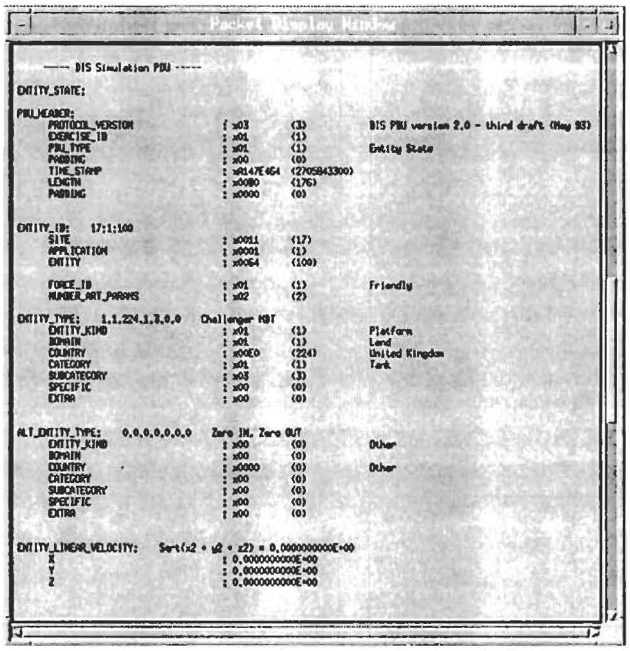
DIS PDU Types

[✓ Other]	[✓ Entity State]
[✓ Fire]	[✓ Detonation]
[✓ Collision]	[✓ Service Request]
[✓ Resupply Offer]	[✓ Resupply Received]
[✓ Resupply Cancel]	[✓ Repair Complete]
[✓ Repair Response]	[✓ Create Entity]
[✓ Remove Entity]	[✓ Start/Resume]
[✓ Stop/Freeze]	[✓ Acknowledge]
[✓ Action Request]	[✓ Action Response]
[✓ Data Query]	[✓ Set Data]
[✓ Data]	[✓ Event Report]
[✓ Message]	[✓ Emissions]
[✓ Designator]	[✓ Transmitter]
[✓ Signal]	[✓ Receiver]

Figure 16 - Packet Filter Window

Packet Display: The Packet Display window displays the contents of the packet highlighted in the Timeline window. At the top of the window will be a hex dump of the entire contents of the packet. This is followed by the network header and the specific DIS PDU type. Each field of the network header and the PDU is displayed on its own line. If the field has an associated enumerated table, the appropriate enumeration is displayed at the end of the line. For groups, such as Entity ID or Entity Type, the group name is displayed and then each element of the group is listed afterwards. If any field or group failed specific packet validation, the beginning of the line will contain an '*' for quick identification of problem areas within the packet.

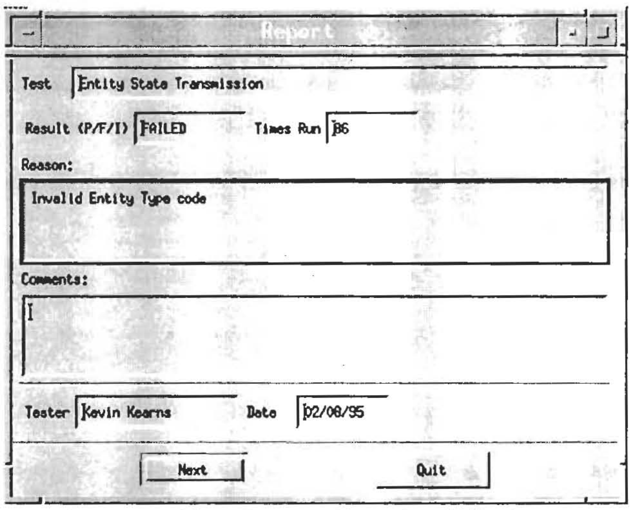
Note: A useful trick for quickly viewing items in the packet display window, is to position the screen where all of the desired information is displayed and then click the left mouse button with the mouse in the bottom right corner of the Packet Display window. Then when the user moves to the next packet, the window will automatically scroll down and position itself at the same position that the screen was at when the user clicked the mouse.



```
    ---- DIS Simulation PDU -----

ENTITY_STATE:

PDU_HEADER:
     PROTOCOL_VERSION             : x03        (3)            DIS PDU version 2.0 - third draft (May 93)
     EXERCISE_ID                  : x01        (1)
     PDU_TYPE                     : x01        (1)            Entity State
     PADDING                      : x00        (0)
     TIME_STAMP                   : xA147E464  (2705843300)
     LENGTH                       : x00B0      (176)
     PADDING                      : x0000      (0)


ENTITY_ID:    17;1;100
     SITE                         : x0011      (17)
     APPLICATION                  : x0001      (1)
     ENTITY                       : x0064      (100)

     FORCE_ID                     : x01        (1)            Friendly
     NUMBER_ART_PARAMS            : x02        (2)

ENTITY_TYPE:    1.1.224.1.3.0.0    Challenger MBT
     ENTITY_KIND                  : x01        (1)            Platform
     DOMAIN                       : x01        (1)            Land
     COUNTRY                      : x00E0      (224)          United Kingdom
     CATEGORY                     : x01        (1)            Tank
     SUBCATEGORY                  : x03        (3)
     SPECIFIC                     : x00        (0)
     EXTRA                        : x00        (0)


ALT_ENTITY_TYPE:    0.0.0.0.0.0.0    Zero IN, Zero OUT
     ENTITY_KIND                  : x00        (0)            Other
     DOMAIN                       : x00        (0)
     COUNTRY                      : x0000      (0)            Other
     CATEGORY                     : x00        (0)
     SUBCATEGORY                  : x00        (0)
     SPECIFIC                     : x00        (0)
     EXTRA                        : x00        (0)


ENTITY_LINEAR_VELOCITY:    Sqrt(x2 + y2 + z2) = 0.0000000000E+00
     X                            : 0.0000000000E+00
     Y                            : 0.0000000000E+00
     Z                            : 0.0000000000E+00
```

Figure 17 - Packet Display Window

Report: The Report window is where the user manually enters whether or not the test passed or failed; if it failed, the reason(s) for failing; and any comments about the test. Clicking on the "Quit" button in this window exits the manual testing and removes all related windows. After closing this window, all test information is written to the report log file. Clicking on the "Next" button will close all windows, write the test information to the report log, and return the user to the Manual Test Selection window.



Test Entity State Transmission

Result (P/F/I) FAILED Times Run 86

Reason:

Invalid Entity Type code

Comments:

Tester Kevin Kearns Date 02/08/95

Figure 18 - Report Results Window

2.4 The **Report** Option

Selecting the Report option brings up two choices: Results Summary and Status Summary. All reports are generated as formatted text files in the same directory as the report log. The files will be named using the following scheme: 3-character company name + 1-character SUT number + "." + either "status" or "results" + ".rpt". Both reports contain a header that summarizes the Testbed, Company, and SUT information and then lists each level of testing as a discrete section within the report. The main difference between the two reports is the amount of information presented in the report.

The Results Summary report is the more verbose of the two reports. It includes the unique number and name of the test, the test status (Passed/Failed/Incomplete), number of times run, specific type of test (Transmission/Reception/Adverse/Erroneous), name of the BIN file, reason, and comments. The Status Summary is a simplier report that includes the Test status and test name and number.

3 Scanner Configuration

The Scanner was designed to be highly configurable at runtime. To do this, there are several text-based configuration files that are loaded when the program starts up. The text-based files allows the user to change the contents of the files, thereby, quickly changing the configuration both in how the program works and how it interprets specific data. The configuration files include the Scanner, management, pdu definition, network definition, entity type, enumerations, and test-specific.

The Scanner defaults to looking for all configuration files in the current directory. If the files cannot be found in that directory it will search the environment for the variable "SCANPROJDIR". If that variable is found, the directory specified after it will be used to search for the files. For example, using the following setting:

```
SCANPROJDIR=/usr/tridis/proj/scanner/config
```

will tell the Scanner to look in the "/usr/tridis/proj/scanner/config" directory for the configuration files if they cannot be found in the current directory.

Each test suite can have specific configuration files associated with it. If after opening a test suite the program finds that the names of the configuration files saved within the test suite are different than the default configuration files, the program will automatically free up the memory reserved by the default configuration data and read the configuration data from the appropriate configuration file.

3.1 Scanner Configuration File

The file contains some of the default information used by the Scanner. The format of the Scanner Configuration file is as follows. Note: The following file names are the default names used by the Scanner.

```
"MANAGE_CONFIG" = "manage.cfg"

"PORTS" = "6994,3001,3002,3003"

"PDU_CONFIG" = "stdV2D3.pdu.cfg"

"NETWORK_CONFIG" = "stdV2D3.network.cfg"

"ENTITY_TYPES_CONFIG" = "stdV2D3.entity_types.cfg"

"BINFILE_TIMESTAMP_SIZE" = "0"
```

MANAGE_CONFIG -	Contains the name of the management structure (test suite) configuration file. See section 5.2.3.2 - Manager Configuration File for format of this file.
ENTITY_TYPES_CONFIG -	Contains the name of the file holding the list of all of the Entity Types recognized by the Scanner Management System. See section 5.2.3.5 - Entity Types Configuration File for format of this file.
PORTS -	Contains a string of four port addresses. The first port is the address of the DIS port. The second, third, and fourth ports are the SIMNET, IST MESSAGE, and EAGLE addresses, respectively. The Scanner

uses these numbers internally to determine if a packet is a DIS, SIMNET, IST MESSAGE, or EAGLE packet. For the program to recognize that a packet is DIS, for example, the packet must be logged to the port specfied as the first parameter of this string. If the DIS packet is logged to a different port, change the value of the the first parameter in the "PORTS" string. NOTE: If you are using an existing test suite and you need to change the ports, see section 3.6 for more information.

PDU_CONFIG -	Contains the name of the file that defines each DIS PDU using the Data Definition Language. See section 5.2.1 - Data Definition Language for a description of the format of the configuration file.
NETWORK_CONFIG -	Contains the name of the file that defines the recognized network headers using the Data Definition Language. See section 5.2.1 - Data Definition Language for a description of the format of the configuration file.
BINFILE_TIMESTAMP_SIZE -	A timestamp may be written once at the top of a binary file indicating when the binary file was created. This line in the configuration file contains the length of that timestamp. Whether or not a timestamp is included at the beginning of the binary file, this field must have a value that represents the length of the timestamp.

3.2 Management Configuration File

This configuration file is used to fill in the format of a test suite management structure. The structure has no substance to it by default. As this configuration file is read in and parsed, its information is used to determine how many groups the management structure contains and what the tests are for each group, as well as, specific information about each test. This approach allows the user to re-define a test or group of tests at runtime. Note, however, for automated tests, changes would still need to be made to the executable to actually perform the test differently than the way the test was originally defined. Even this is made relatively easy, since all the user has to do is just write the new routine and then recompile. Very little or nothing else need be done to make the new test functional.

The format of the configuration file is listed and explained below:

```
#GROUP       NAME  "SubGroup
"Number",    "Desc", "Type", "Mode",       "Func", "FileID"
```

The #GROUP moniker is used to distinguish the beginning of a new group of tests. This field is followed by the name of the group, in quotes. The group name is followed by a sub-group name. If there is not a sub-group name, the tag contains "NONE".

Each line that follows lists an individual test for that group. The first field contains the unique test number as defined in IST's document - *Technical Report, Test Documents for DIS Interoperability*. The next column is a textual description of the test. The third field indicates what type of test this test is. The codes translate to "T" for Transmission, "R" for Reception, "A" for Adverse and "E" for Erroneous.

The next column contains either an "A" for Automated tests or a "M" for manual tests. The fifth column is a unique code that is used to get the address of the specific function to perform this test as an automated test. The last column is used during the creation of the test suite. This string, along with the 3-character company name and the SUT number, are combined to come up with a unique logged binary file filename for each test.

The following is a partial sample listing of two groups from the Manager Configuration file:

```
#GROUP          "Network"                   "NONE"
"1.1.1.1.1",    "Broadcast-Transmission",   "T",   "A",   "T111TA",   "111"
"1.1.1.1.2",    "Broadcast-Reception",      "R",   "A",   "T111RA",   "111"
"1.1.2.1.1",    "Broadcast-Adverse",        "A",   "A",   "T111AA",   "111"
"1.1.3.1.1",    "Broadcast-Erroneous",      "E",   "A",   "T111EA",   "111"
...


#GROUP          "PDU"       "Transmission"
"2.2.1.1",      "Entity State Transmission", "T",  "A",   "T201TA",   "201"
"2.2.1.2",      "Fire Transmission",         "T",  "A",   "T202TA",   "202"
...
```

3.3 PDU and Network Default Configuration File

3.3.1 Data Definition Language

Rather than defining specific data structures for each PDU and Network type, and the associated code to process the data structures, the Data Definition Language (DDL) was created to allow generic processing of the various data packets.

Certain fields require special processing. These fields are also defined in the default DDL format, but are identified and processed accordingly during the traversal of the packet definition list. The majority of the fields within the PDUs do not require special processing.

The DDL also allows variable sections to be defined. Variable sections must be grouped. The variable group is only defined once. When the data packet is being processed, the variable group fetches its count parameter from the data packet, and is repeated the appropriate number of times. The data field containing the count parameter must appear in the data packet before the variable section. This allows the generic definition and processing to be used for all possible variations of the currently defined PDU types.

Since the data defined using the DDL is kept in ASCII files and read in at runtime, the keywords and associated data are quoted. This allows spaces to be used in the definitions, and parsing of the configuration line is simplified to reading and parsing of strings. This does not limit the data types that may be defined. To keep the parser simple, there is a requirement that a definition must be contained on a single line. Each line must start with a comment character or a keyword. The parameters for a keyword must be on the same line as the keyword. Keywords may be preceded by spaces or tabs for readability. The parser will skip blank lines and comment lines.

The DDL uses the following key words:

* This character at the start of a line indicates that this is a comment line.

PDU

This keyword defines the start of a PDU definition. This keyword requires two parameters, PDU name, and PDU number. PDU definitions may not be nested. The PDU keyword does require a corresponding **ENDPDU**.

Options: PDU Keyword

	NAME	ASCII name of the PDU
	NUMBER	PDU Number. Must be unique. Used to index the PDU table

Usage: PDU, NAME, NUMBER

Example: "PDU", "ENTITY_STATE", "1"

ENDPDU

This keyword defines the end of a PDU definition. Each PDU keyword requires a corresponding ENDPDU keyword.

Options:	ENDPDU	Keyword

Example: "ENDPDU"

NETWORK

This keyword defines the start of a Network level definition. This keyword requires two parameters, Network name, and Network number. Network definitions may not be nested. They do require a corresponding **ENDNETWORK**.

Options:	NETWORK	Keyword
	NAME	ASCII name of the Network
	NUMBER	Network Number. Must be unique. Used to index the Network table

Usage: NETWORK, NAME, NUMBER

Example: "NETWORK", "ETHERNET", "1"

Example: "NETWORK", "UDP/IP", "1"

ENDNETWORK

This keyword defines the end of a Network definition. Each NETWORK keyword requires a corresponding ENDNETWORK keyword.

Options:	ENDNETWORK	Keyword

Example: "ENDNETWORK"

GROUP

This keyword defines the start of a group. The group may be a fixed group or a variable group. A fixed group defines a group of elements that are to be repeated a known number of times. A variable group is a group that is to be repeated, but the number of times is determined by a data field within the data packet, and is therefore unknown at this time. If a group name contains the keyword **_BITS**, the group elements are treated as a bitfield. Groups may be nested, but each group requires a corresponding **ENDGROUP**.

Fixed groups:

Options:	GROUP	Keyword
	NAME	Group Name, must be unique for this PDU or Network definition.
	FIXED	Keyword

COUNT	Number of iterations

Usage: GROUP, NAME, FIXED, COUNT

Example: "GROUP", "PDU_HEADER", "FIXED", "1"

Variable groups:

Options:		
	GROUP	Keyword
	NAME	Group Name, must be unique for this PDU or Network definition.
	VARIABLE	Keyword
	CTRL GROUP	Name of group containing the field with the count value.
	CTRL FIELD	Name of field containing the count value specifying this groups number of copies.

Usage: GROUP, NAME, VARIABLE, CTRL GROUP, CTRL FIELD

Example: "GROUP", "ART_PARMS", "VARIABLE", "ENTITY_STATE", "NUMBER_ART_PARMS"

ENDGROUP

This keyword defines the end of a group definition.

Options:		
	ENDGROUP	Keyword
	GROUP NAME	Group Name

Example: "ENDGROUP", "ART_PARMS"

ELEM

This keyword defines an element, whether it is part of the PDU or a sub-group. An element defines the attributes of a data field. Elements are self contained in one line, and do not require an end type keyword.

Options:		
	ELEM	Keyword
	PARENT GROUP	The name of this fields immediate parent group.
	FIELD NAME	The name of this data field.
	DATA TYPE	The data type for this data field. See the list below.
	DISPLAY TYPE	Display type for this data field. See list below.
	MINIMUM VALUE	The minimum value for this data field, or the word "MIN" which will be translated to the minimum value for this fields data type.
	MAXIMUM VALUE	The maximum value for this data field, or the word "MAX" which will be translated to the maximum value for this fields data type.
	ENUMERATION TABLE	An associated enumeration table name if there is one, otherwise "NONE".
	GROUP DIVISOR	The group divisor is used in calculating the iteration count for variable fields. Not all count fields are a number of iterations. Some are total number of bits, therefore they must be divided by this value to obtain a count. The default value is "1". Only a few PDUs require a different value. See the PDU Configuration File section for a more complete

definition.

Usage: ELEM, PARENT GROUP, FIELD NAME, DATA TYPE, DISPLAY TYPE, MIN. VALUE, MAX. VALUE, ENUMERATION TABLE, GROUP DIVISOR

Example: "ELEM", "PDU_HEADER", "PROTOCOL_VERSION", "U8", "DEC", "1", "4", "PROTOCOL_VERSION", "1"

A list of possible display types:

DEC	Decimal
HEX	Hexadecimal
FLOAT	Decimal
ASCII	ASCII text
DATA	Data block

A list of possible element data types:

U8	Unsigned character
S8	Signed character
U16	Unsigned short
S16	Signed short
U32	Unsigned long
S32	Signed long
F32	32-bit Float
U64	Possibly two unsigned longs
F64	64-bit Float
A	Ascii string with length (A#) A15 means 15 ascii characters.
B	Bit fields (bits#mask#shift) B8#F0#4 means its an 8 bit field, the mask is hex F0 and the shift count used to align the data is 4.
D	Data area with count (D#) D15 means 15 data bytes.

The following is an example of the Network Configuration File.

```
"NETWORK", "ETHERNET", "1"
   "GROUP", "ETHERNET", "FIXED", "1"
      "ELEM", "ETHERNET", "DESTINATION",    "D6",    "HEX",      "MIN", "MAX", "NONE"
      "ELEM", "ETHERNET", "SOURCE",         "D6",    "HEX",      "MIN", "MAX", "NONE"
      "ELEM", "ETHERNET", "ETHER_TYPE",     "U16",   "HEX",      "0",   "MAX", "NONE"
   "ENDGROUP", "ETHERNET"
"ENDNETWORK"
```

The following is an example of the PDU Configuration File.

```
"PDU", "ENTITY_STATE_PDU"    1

  "GROUP", "ENTITY_STATE", "FIXED", "1"

    "GROUP", "PDU_HEADER",                   "FIXED",    "1"
        "ELEM", "PDU_HEADER",                "PROTOCOL_VERSION", "U8",   "DEC",   "1",    "4",     "PROTOCOL_VERSION", "1"
        "ELEM", "PDU_HEADER",                "EXERCISE_ID",      "U8",   "DEC",   "MIN",  "MAX",   "NONE", "1"
        "ELEM", "PDU_HEADER",                "PDU_TYPE",         "U8",   "DEC",   "1",    "1",     "PDU_TYPE", "1"
        "ELEM", "PDU_HEADER",                "PADDING",          "U8",   "DEC",   "0",    "0",     "NONE", "1"
        "ELEM", "PDU_HEADER",                "TIME_STAMP",       "U32",  "HEX",   "MIN",  "MAX",   "NONE", "1"
        "ELEM", "PDU_HEADER",                "LENGTH",           "U16",  "DEC",   "144",  "MAX",   "NONE", "1"
        "ELEM", "PDU_HEADER",                "PADDING",          "U16",  "DEC",   "0",    "0",     "NONE", "1"
    "ENDGROUP", "PDU_HEADER"

    "GROUP", "ENTITY_ID",                    "FIXED",    "1"
        "ELEM", "ENTITY_ID",                 "SITE",             "U16",  "DEC",   "MIN",  "MAX",   "NONE", "1"
        "ELEM", "ENTITY_ID",                 "APPLICATION",      "U16",  "DEC",   "MIN",  "MAX",   "NONE", "1"
        "ELEM", "ENTITY_ID",                 "ENTITY",           "U16",  "DEC",   "MIN",  "MAX",   "NONE", "1"
    "ENDGROUP", "ENTITY_ID"

    "ELEM", "ENTITY_STATE",                  "FORCE_ID",          "U8",  "DEC",   "0",    "3",     "FORCE_ID", "1"
    "ELEM", "ENTITY_STATE",                  "NUMBER_ART_PARAMS", "U8",  "DEC",   "MIN",  "MAX",   "NONE", "1"

    "GROUP",   "ENTITY_TYPE",                "FIXED",    "1"
        "ELEM", "ENTITY_TYPE",               "ENTITY_KIND",         "U8",   "DEC",   "0",    "7",     "ENTITY_KIND", "1"
        "ELEM", "ENTITY_TYPE",               "DOMAIN",              "U8",   "DEC",   "MIN",  "MAX",   "NONE", "1"
        "ELEM", "ENTITY_TYPE",               "COUNTRY",             "U16",  "DEC",   "0",    "266",   "COUNTRY", "1"
        "ELEM", "ENTITY_TYPE",               "CATEGORY",            "U8",   "DEC",   "MIN",  "MAX",   "NONE", "1"
        "ELEM", "ENTITY_TYPE",               "SUBCATEGORY",         "U8",   "DEC",   "MIN",  "MAX",   "NONE", "1"
        "ELEM", "ENTITY_TYPE",               "SPECIFIC",            "U8",   "DEC",   "MIN",  "MAX",   "NONE", "1"
        "ELEM", "ENTITY_TYPE",               "EXTRA",               "U8",   "DEC",   "MIN",  "MAX",   "NONE", "1"
    "ENDGROUP", "ENTITY_TYPE"

    "GROUP",   "ALT_ENTITY_TYPE",            "FIXED",    "1"
        "ELEM", "ALT_ENTITY_TYPE",           "ENTITY_KIND",         "U8",  "DEC",   "0",    "7",     "ENTITY_KIND", "1"
        "ELEM", "ALT_ENTITY_TYPE",           "DOMAIN",              "U8",  "DEC",   "MIN",  "MAX",   "NONE", "1"
        "ELEM", "ALT_ENTITY_TYPE",           "COUNTRY",             "U16", "DEC",   "0",    "266",   "COUNTRY", "1"
        "ELEM", "ALT_ENTITY_TYPE",           "CATEGORY",            "U8",  "DEC",   "MIN",  "MAX",   "NONE", "1"
        "ELEM", "ALT_ENTITY_TYPE",           "SUBCATEGORY",         "U8",  "DEC",   "MIN",  "MAX",   "NONE", "1"
        "ELEM", "ALT_ENTITY_TYPE",           "SPECIFIC",            "U8",  "DEC",   "MIN",  "MAX",   "NONE", "1"
        "ELEM", "ALT_ENTITY_TYPE",           "EXTRA",               "U8",  "DEC",   "MIN",  "MAX",   "NONE", "1"
    "ENDGROUP", "ALT_ENTITY_TYPE"

    "GROUP",   "ENTITY_LINEAR_VELOCITY",     "FIXED",    "1"
```

```
        "ELEM", "ENTITY_LINEAR_VELOCITY",    "X",                        "F32", "FLOAT",   "MIN",  "MAX",    "NONE", "1"
        "ELEM", "ENTITY_LINEAR_VELOCITY",    "Y",                        "F32", "FLOAT",   "MIN",  "MAX",    "NONE", "1"
        "ELEM", "ENTITY_LINEAR_VELOCITY",    "Z",                        "F32", "FLOAT",   "MIN",  "MAX",    "NONE", "1"
    "ENDGROUP", "ENTITY_LINEAR_VELOCITY"

    "GROUP",   "ENTITY_LOCATION",            "FIXED",    "1"
        "ELEM", "ENTITY_LOCATION",           "X",                        "F64", "FLOAT",   "MIN",  "MAX",    "NONE", "1"
        "ELEM", "ENTITY_LOCATION",           "Y",                        "F64", "FLOAT",   "MIN",  "MAX",    "NONE", "1"
        "ELEM", "ENTITY_LOCATION",           "Z",                        "F64", "FLOAT",   "MIN",  "MAX",    "NONE", "1"
    "ENDGROUP", "ENTITY_LOCATION"

    "GROUP",   "ENTITY_ORIENTATION",         "FIXED",    "1"
        "ELEM", "ENTITY_ORIENTATION",        "PSI",                      "F32",  "FLOAT",   "MIN",  "MAX",    "NONE", "1"
        "ELEM", "ENTITY_ORIENTATION",        "THETA",                    "F32",  "FLOAT",   "MIN",  "MAX",    "NONE", "1"
        "ELEM", "ENTITY_ORIENTATION",        "PHI",                      "F32",  "FLOAT",   "MIN",  "MAX",    "NONE", "1"
    "ENDGROUP", "ENTITY_ORIENTATION"

    "ELEM",     "ENTITY_STATE",              "APPEARANCE",               "U32",  "HEX",   "MIN",  "MAX",    "NONE", "1"

    "GROUP",   "DEAD_RECK_PARAMS",           "FIXED",    "1"
        "ELEM", "DEAD_RECK_PARAMS",          "DR_ALGORITHM",             "U8",   "DEC",     "0",    "9",     "DEAD_RECKONING_ALGORITHM", "1"
        "ELEM", "DEAD_RECK_PARAMS",          "OTHER",                    "D15",  "HEX",     "0",    "0",      "NONE", "1"
        "ELEM", "DEAD_RECK_PARAMS",          "EN_LIN_ACC.X",             "F32",  "FLOAT",   "MIN",  "MAX",    "NONE", "1"
        "ELEM", "DEAD_RECK_PARAMS",          "EN_LIN_ACC.Y",             "F32",  "FLOAT",   "MIN",  "MAX",    "NONE", "1"
        "ELEM", "DEAD_RECK_PARAMS",          "EN_LIN_ACC.Z",             "F32",  "FLOAT",   "MIN",  "MAX",    "NONE", "1"
        "ELEM", "DEAD_RECK_PARAMS",          "EN_ANG_VEL.X",             "F32",  "FLOAT",   "MIN",  "MAX",    "NONE", "1"
        "ELEM", "DEAD_RECK_PARAMS",          "EN_ANG_VEL.Y",             "F32",  "FLOAT",   "MIN",  "MAX",    "NONE", "1"
        "ELEM", "DEAD_RECK_PARAMS",          "EN_ANG_VEL.Z",             "F32",  "FLOAT",   "MIN",  "MAX",    "NONE", "1"
    "ENDGROUP", "DEAD_RECK_PARAMS"

    "GROUP",   "ENTITY_MARKING",             "FIXED",    "1"
        "ELEM", "ENTITY_MARKING",            "CHARACTER_SET",            "U8",   "HEX",  "MIN",  "MAX",   "ENTITY_MARKING_TABLE", "1"
        "ELEM", "ENTITY_MARKING",            "RECORD",                   "A11",  "ASCII", "MIN",  "MAX",    "NONE", "1"
    "ENDGROUP", "ENTITY_MARKING"

    "ELEM", "ENTITY_STATE",                  "CAPABILITIES",      "U32",  "HEX",   "MIN",  "MAX",    "NONE", "1"

    "GROUP",   "ART_PARAMS",                 "VARIABLE",    "ENTITY_STATE",    "NUMBER_ART_PARAMS"
        "ELEM", "ART_PARAMS",                "CHANGE",            "U16",  "DEC",   "MIN",  "MAX",    "NONE", "1"
        "ELEM", "ART_PARAMS",                "ID",                "U16",  "DEC",   "MIN",  "MAX",    "NONE", "1"
        "ELEM", "ART_PARAMS",                "TYPE",              "U32",  "HEX",   "MIN",  "MAX",    "NONE", "1"
        "ELEM", "ART_PARAMS",                "VALUE",             "U64",  "HEX",   "MIN",  "MAX",    "NONE", "1"
    "ENDGROUP", "ART_PARMS"

  "ENDGROUP",  "ENTITY_STATE"

"ENDPDU"
```

3.4 Entity Types Configuration File

This file contains a listing of all Entity Types recognized by the Scanner Management System. Currently, the Scanner only recognizes a fraction of the entire Entity Type list as defined in IST's document - *Enumeration and Bit-encoded Values for use with IEEE 1278.1-1994, Distributed Interactive Simulation -- Application Protocols.* However, the file is easily configurable to include all of the possible Entity Types and the program will automatically adapt to any number of possible Entity Types.

An example of several lines from the configuration file and the format of the file is as follows:

```
{ "1.1.225.1.1.0",  0x2882080c,   "M1*" },
{ "1.1.225.1.1.1",  0x2882080c,   "M1A1" },
{ "1.1.225.1.1.2",  0x2882080c,   "M1A2" },
```

The first column symbolizes the Entity Type code; the second column is the SIMNET code (not used by the Scanner, but is used by some other software and was kept for compatibility); and the third column is a text description of the Entity Type code. Each entry was formatted this way so that if it ever needed to be loaded into a static structure within the code, it could be read in as is without having to be changed.

3.5 Enumerated Values Configuration File

This file contains a listing of all of the enumeration values recognized by the Scanner. The source of this file can be found in IST's document - *Enumeration and Bit-encoded Values for use with IEEE 1278.1-1994, Distributed Interactive Simulation -- Application Protocols.* The file is structure as a series of tables. Each table starts with a line beginning with the "@" character followed by a textual description of the table. The following lines indicate the rows of acceptable values within that table. Each row contains a minimum and maximum value for that row, as well as, the textual description for that range. The min/max range is used for validation checking of an enumerated field value and for finding the correct enumeration text string to return.

The Scanner reads the contents of the file and stores the contents in memory as a linked list of tables, each made up of a linked list of rows from the table. The management structure contains a pointer to the one of the linked list for each field in the management structure that references an enumeration table.

The following is a small subset of the contents of the enuerations configuration file:

```
@PROTOCOL_VERSION
      "1",     "1",     "DIS PDU version "1.0 (May "92)",
      "2",     "2",     "IEEE "1278-1993",
      "3",     "3",     "DIS PDU version "2.0 - third draft (May "93)",
      "4",     "4",     "IEEE "1278.1-1994"
@PDU_FAMILY
      "0",     "0",     "Other",
      "1",     "1",     "Entity Information/Interaction",
      "2",     "2",     "Warfare",
      "3",     "3",     "Logistics",
      "4",     "4",     "Radio Communication",
      "5",     "5",     "Simulation Management",
      "6",     "6",     "Distributed Emission Regeneration"
@DEAD_REACKONING_ALGORITHM
      "0",     "0",     "Other",
      "1",     "1",     "Static (Entity does not move.)",
      "2",     "2",     "DRM(F, P, W)",
      "3",     "3",     "DRM(R, P, W)",
"4",        "4",        "DRM(R, V, W)",
"5",        "5",        "DRM(F, V, W)",
"6",        "6",        "DRM(F, P, B)",
"7",        "7",        "DRM(R, P, B)",
"8",        "8",        "DRM(R, V, B)",
"9",        "9",        "DRM(F, V, B)"
```

3.6 PDU and Network Test-specific Configuration File

The PDU and Network Default Configuration files contain the default range values for each field of all DIS PDUs and network headers, respectively, recognized by the Scanner. However, for some tests, the test requires very specific values for certain fields, not just a range of acceptable values. To accomplish this need, the user can specify for each Network test to be run a user can specify a configuration file that contains values that will override the default values specified in the Network Configuration File.

To facilitate commonality between configuration files, this file is formatted very similarly to either the PDU or Network Configuration files. See section 3.3 - PDU and Network Default Configuration Files for an explanation of the Data Definition Language and the format of the file. Since the program already knows the format, data type, length, etc. about all fields, test-specific configuration file only needs to specify the group name, field name and the new min/max values for that field.

Note that each Group, Field combination must be entered in the EXACT format that they exist in the pdu.cfg or network.cfg files. Also, the MIN/MAX values must be entered in this file in the same data format as they are to be read from a packet, i.e., if a field is to be read as if it were in hexadecimal format, then it must be entered as "0800" not "2048" to be properly parsed.

The following is an example of a Network Test-specific Configuration File:

```
#Group            Field                          MIN        MAX

"ETHERNET"   "DESTINATION_ADDRESS[0]"      "255"      "255"
"ETHERNET"   "DESTINATION_ADDRESS[1]"      "255"      "255"
"ETHERNET"   "DESTINATION_ADDRESS[2]"      "255"      "255"
"ETHERNET"   "DESTINATION_ADDRESS[3]"      "255"      "255"
"ETHERNET"   "DESTINATION_ADDRESS[4]"      "255"      "255"
"ETHERNET"   "DESTINATION_ADDRESS[5]"      "255"      "255"

"IP"         "DESTINATION_ADDRESS[0]"      "164"      "164"
"IP"         "DESTINATION_ADDRESS[1]"      "217"      "217"
"IP"         "DESTINATION_ADDRESS[2]"      "255"      "255"
"IP"         "DESTINATION_ADDRESS[3]"      "255"      "255"

"ETHERNET"   "ETHER_TYPE"                  "0800"     "0800"

"IP"         "PROTOCOL"                    "17"       "17"

"UDP"        "DESTINATION PORT"            "6994"     "6994"
```

3.7 Edit existing test suite configuration

The Edit main menu option allows the user to configure the Scanner Management System for specific testing purposes. Clicking on the Edit option will pull down a menu with several areas to choose from including editing System Configuration or any of the multiple levels of tests.

The System Configuration option allows the user to change the names of the configuration files that the Scanner uses. The Scanner loads the default configuration files at startup. If the user wants to use different configuration files or ports for this test suite, go to this screen and change the desired values and the new values will be automatically loaded. For example, the default string for port ids is "6994,3001,3002,3003". If the DIS data your are logging is coming addressed to port 3000, for example, go to this screen and change the port string to "3000,3001,3002,3003". This will tell the Scanner that for this test suite only, all DIS traffic is logged to port 3000. See section 3.1 for a detailed discussion of how to change default values for all new test suites created.

4 For Your Information

The following is a list of items observed by previous users of the Scanner that should be noted but did not "fit" into the above text:

- Due to a programming limitation, it is recommended that users iconize unwanted windows rather than clicking on the upper left hand corner and closing them. Closing certain windows, for example the Packet Count, Packet Display, Timeline windows while testing, will cause the program to crash upon exiting the testing mode.
- IST has developed both Motorola and Silicon Graphics versions of the Scanner. The Motorola used for testing and development was a MVME197 single board computer system using System V R4 UNIX.

0000174