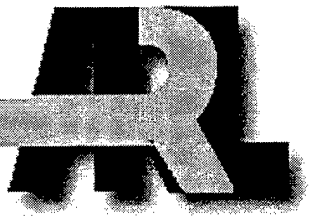
ARMY RESEARCH LABORATORY



The Distributed Interactive Simulation (DIS) Lethality Communication Server Volume II: User and Programmer's Manual

Geoffrey C. Sauerborn

ARL-TR-1775 FEBRUARY 1999

IRIX™ and Open GL™ are trademarks of of Silicon Graphics, Inc.

Linux® is a registered trademark of Linus Torvalds.

POSIX® is a registered trademark of the Institute of Electrical and Electronics Engineers, Inc.

Silicon Graphics® and IRIS® are registered trademarks of Silicon Graphics, Inc.

Sun Solaris® is a registered trademark of Sun Microsystems Compter Company.

VR Link® is a registered trademark of MäK.

UNIX™ is a trademark of Bell Laboratories.

Windows NT® is a registered trademark of Microsoft Corporation.

Abstract

Volume 1 presented the distributed interactive simulation lethality communication server, a client-server approach to handling battle simulation lethality. Although Volume 1 explained the approach and its benefits and limitations, it presented no information about how to set up, run, or modify the server. In this volume, these vital (yet sometimes tedious) details are provided.

TABLE OF CONTENTS

INTENTIONALLY LEFT BLANK

LIST OF FIGURES

INTENTIONALLY LEFT BLANK

LIST OF TABLES

INTENTIONALLY LEFT BLANK

THE DISTRIBUTED INTERACTIVE SIMULATION (DIS) LETHALITY COMMUNICATION SERVER VOLUME II: USER AND PROGRAMMER'S MANUAL

1. PURPOSE

This report is a user and programmer's manual for the distributed interactive simulation (DIS) lethality communication server (the server). The report can be used to learn how to initialize, operate, or modify the server. Instructions for modifying the server are written for an expert level audience; therefore, only experienced "C" programmers should attempt this.

2. INTRODUCTION

The ARL DIS lethality communication server is a combination of application program interface (API) libraries and utility programs that make it possible to allow multiple applications to access a single lethality data source. The server is designed for the DIS environment. As such, the server returns lethality results as described by (the DIS) Institute of Electrical and Electronics Engineers (IEEE) Standard 1278.1 **[1,2]**. Furthermore, the server expects input in DIS standard protocol data unit (PDU) format (although the equivalent input may be greatly condensed at more abstracted layers within the APIs). The DIS lethality server has demonstrated a data latency of less than 1/100th of a second and thus may be useful for a wide variety of applications, including real time **[3,4]**. This project was jointly sponsored by the Army Modeling & Simulation Office (as a 1997 Army modeling improvement program project [AMIP]) and by the U.S. Army Research Laboratory (ARL).

3. QUICK START INSTALLATION

The server is designed to run in the UNIX™ environment but might be portable to most POSIX® systems with an American National Standards Institute (ANSI) C compiler and the "csh" or "tcsh" command line interpreter shell.[1] This includes most UNIX™-like systems and Windows™ NT®. So far, the server has only been tested under IRIX™, Linux®, and Sun Solaris® operating systems.

---

[1]The server requires an environmental variable to be set (***VLS_HOME***). This is best accomplished by running the server or client application under the "csh" or "tcsh" command line interfaces.

3.1 Unpacking and Installing

The server comes packaged in a "tar" format archive. This archive needs to be unpacked in a convenient location accessible by server users, to be used by those who actually wish to run the server and by those who merely wish to have access to its libraries to create client applications. In the following examples, we assume that the location will be */usr/local/DIS/Lserver*, but the actual location chosen is not significant.

Change to the directory where you wish to install the server and extract the tar archive there. For example, suppose the tar archive is in the file */home/mystuff/lserver_v123.tar*. To install the server in */usr/local/DIS/Lserver*,

```
mkdir /usr/local/DIS/
mkdir /usr/local/DIS/Lserver
cd /usr/local/DIS/Lserver
tar -xf  /home/mystuff/lserver_v123.tar
```

3.2 Compiling

Assuming no errors were encountered when the tar archive was unpacked, we are now ready to compile the server. Change to the server's "home" (installation) directory and type "./compile.sh":

```
cd /usr/local/DIS/Lserver
./compile.sh
```

By default, the compiling script uses "cc" to compile source code. This may be changed by using a CC=*compiler* argument to the script. For example, to compile using */usr/gnu/bin/gcc*, type

```
./compile.sh CC=/usr/bin/gnu/gcc
```

Output for this procedure should appear similar to that shown in Appendix A.

3.3 Test Programs

Before running a client application, it will be necessary to define the environmental variable **VLS_HOME**. This is set to the server's installation directory. In the C shell (csh), this is accomplished by typing the command

```
set VLS_HOME=/usr/local/DIS/Lserver
setenv VLS_HOME /usr/local/DIS/Lserver
```

These same commands may be added to your *$home/.cshrc* (C-shell initialization "run command") file so that you will not have to retype these commands every time you run the csh. Once **VLS_HOME** is set, the shell replaces the string "$VLS_HOME" with the argument to which it was assigned.

The first test program simply tests that the server is able to communicate with a simple client application. It is executed with the command

```
$VLS_HOME/bin/test_Xsimple.csh
```

The output should be similar to the following:

```
Tests connection to the simple server.
Uses Xwindows xterm (xterm must be in the current path).
Enter key when ready. . .

sleeping for 5 seconds...
4 ...
3 ...
2 ...
1 ...
starting client program...
-----------------------------------------------
Client seems to be speaking with vlserver.    -  OK.
-----------------------------------------------
```

The next test program is more complicated since it requires a number of processes to be sequentially or asynchronously executed. As with the simple test program, connectivity between the server and client is verified. This time, however, the client will also query the server for the results to a specific fire/detonation event. In order for the server to know about the event, the ARL DIS manager is launched, DIS PDUs are broadcast to the DIS network, and the DIS monitor provided with the server is run to monitor the PDUs. If errors are reported, it might be because one or more of these programs was not running when it should have because of simplistic "sleep" delays built into the test program. If this is the case, you might try to run the test program again or change the amount of time the test program "sleeps" between launch times of the various modules. This test program is executed with the following command:

```
$VLS_HOME/bin/test_Xall.csh
```

In addition to several windows opening for the various modules, the output should be similar to the following:

```
-----------------------------------------------------
Tests connection to the VLserver attached to the DIS monitor
Will also run ARL DIS Manager in order to do so.
Uses Xwindows xterm (xterm must be in the current path).
-----------------------------------------------------
Enter key when ready. . .
sleeping for 5 seconds...
4 ...
3 ...
2 ...
1 ...
sleeping for 5 seconds...
4 ...
3 ...
2 ...
1 ...
sleeping for 20 seconds...
starting client program...
############# T H E query  W A S ###############
QUERY TYPE_mfkDIS_Result ARGS_mfkDIS_IDS 135 2 1005 135 2 12
############# T H E answer W A S ###############
"5 Received from server:1: 4 0  "
################################################

This means that: 4 and 0 are the RESULT and FLAG codes,
     respectively, returned by the server
RESULT code:  4
RESULT Meaning:PS_MFK_NODAMAGE - No Damage

FLAG code:  0 = Success.
FLAG Meaning:
  0     Success.

   The pkh source for the referenced entity and threat
   munition (as defined in the DAMAGE_SOURCE_META_DATA_FILE)
   was successfully found, interpreted, and used in
   the calculation of the returned (VL_Result) value.
-----------------------------------------------
DIS Monitor seems to be speaking with vlserver.   -  OK.
 -----------------------------------------------
```

This concludes the execution of the test programs. The following section explains the different modules and data flow that occur during the execution of these test programs (and through the server in general).

4. THE SERVER'S ARCHITECTURE

Figure 1 displays a view of the server's architectural layout. Boxes enclosed by solid lines represent independent processes. Each of these processes may be run on separate computers. The one exception is the **DIS lethality server** and **DIS monitor**; these two processes must reside on the same host machine, as indicated by the dotted box. Dashed lines separating the

vulnerability/lethality (**VL**) **API** and **Data Manager** indicate that these represent DIS lethality server service layers (APIs) that reside within a parent process.

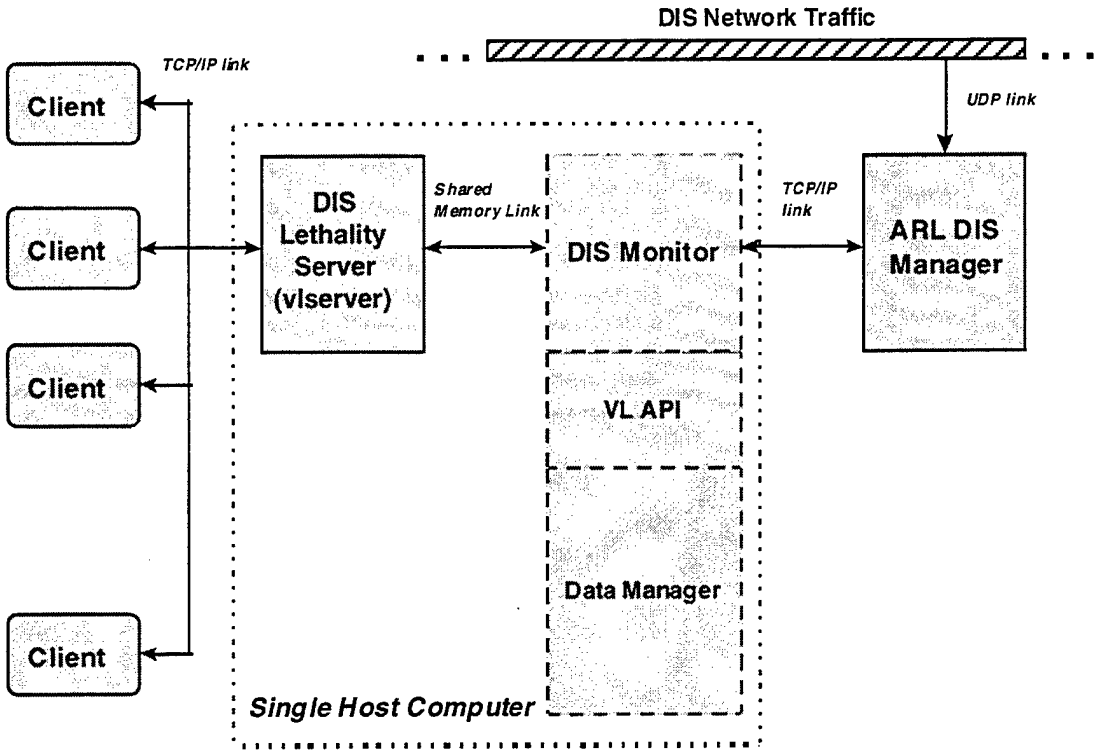


Figure 1. DIS Lethality Server Architecture.

Not shown in Figure 1 is the clients' connectivity to the DIS network. To connect to the DIS network, clients may choose to use the **ARL DIS Manager** (which is freely provided with the lethality server), a commercially available product (such as VR Link®), or their own in-house DIS networking library. It is not the responsibility of or within the scope of the lethality server to decide how clients connect with the DIS network.

An explanation of the components follows:

- The **ARL DIS Manager** monitors DIS PDUs and sends them its own clients. In this case, the DIS manager has one client (the **DIS Monitor**). Because the DIS monitor is currently only concerned with MFK[2] vulnerability resulting from munitions, it only requests to receive (from the ARL DIS manager) entity state, fire, and detonation PDUs (since these are the only

[2]MFK – system damage in terms of mobility, fire power, and catastrophic damage; see Table 1 (Section 5.1) for a further explanation of MFK.

PDUs necessary to calculate MFK results). The DIS monitor may request other PDU types from the DIS manager as necessary.

- The **DIS Monitor** monitors all fire/detonation events (along with information concerning any entities involved). It maintains cached records of these events. In this way, the parameters involved will be available when the **DIS Lethality Server** queries it for the results of a particular detonation event.

Upon receipt of a query from the DIS lethality server, the DIS monitor calls the **VL API** which sets the appropriate parameters that describe the conditions at the time of the detonation (e.g., munition type, velocity, etc.). (The API function *vlp_print_all_params()* may be called to show where the parameter values were set; see **vlparam(3)** in Appendix B.) The VL API then calls the **Data Manager** API which provides data (presumably those data are the vulnerability analysis results). The **VL API** layer then returns these data in a format appropriate to query.

The **Data Manager** API manages many types of low level data. It maintains records of where to find data sources for each entity and threatening munition. It keeps track of which functions are used to initialize (or read each type of data source into memory) and (once initialized) which function to use to extract results (from the cached memory data structures). It is also responsible for maintaining which DIS enumerations are used to describe a particular vehicle, munition, or other item.

The job of the **DIS Lethality Server** (vlserver) component is relatively simple. This component merely passes client queries to the **DIS Monitor** and returns the DIS monitor's results to the client.

The blocks in Figure 1 labeled as "Client" represent clients of the DIS lethality server. The current maximum number of clients that the server will accept is 32. This arbitrary limit may be changed by modifying the value of the variable *Max_Num_Clients* in the source file *$VLS_HOME/src/Server/vlserver.c*. In the case of our test programs (from Section 3.3), just one client was active. These test programs were simply shell scripts that launch the various server applications programs. (These programs are shown as separate processes in Figure 1.) When these applications are not being launched from a shell script, the proper order of execution must be followed. When started, the server components should be executed in the following order:

```
1st  (or 2nd)  ARL DIS manager.
2nd  (or 1st)  DIS lethality server.
3rd  DIS monitor.
4th  client(s) (to DIS lethality server).
```

The ARL DIS manager must be running before the DIS monitor is running. This is because the DIS monitor is a client of the DIS manager. The DIS lethality server must be running before the DIS monitor because the server creates a common shared memory. Furthermore, administrative details concerning how to connect to this shared memory location are communicated to the DIS monitor through a transmission control protocol/internet protocol (TCP/IP) link, of which the DIS monitor is a client. After this initial network "hand shaking," the remainder of the communication occurs through shared memory. Finally, clients of the DIS lethality server may join and leave as they wish. In the following three sections, we explain how to execute and use these applications manually.

4.1 Server Application—ARL DIS Manager

The ARL DIS manager must be running before the DIS monitor is started. The DIS manager may be started by typing

```
$VLS_HOME/bin/dis_mgr.exe  -x off
```

The `-x off` option turns off the DIS exercise identification (ID) number filtering. (This allows multiple DIS exercises to be monitored.) If you would like to monitor only one exercise, use the `-x` option followed by the exercise number. Other command line options may be seen by using the `-help` command line option or by viewing the **dis_mgr(1)** "man" page in Appendix B. DIS manager source code and documentation are presented in $VLS_HOME/src/Libs/DIS.

4.2 Server Application—The DIS Server

The DIS server must be started before the DIS monitor. To run the server, type the following command line:

```
$VLS_HOME/bin/vlserver.exe
```

If you receive an error message similar to

```
pkg_permserver: bind: Address already in use
init_server(): Failed.
another vulnerability/lethality (V/L) server is probably already
using the Port: 4976.  Use the "-P" flag to specify a different
server port.
```

This most likely means that the server is still running (perhaps as a background process or in another window). Command line options and more details about the server are given in the **vlserver(1)** manual page in Appendix B. In order for the server to respond to DIS vulnerability

queries, the DIS monitor must also be running. Starting the DIS monitor is explained in the next section.

4.3 Server Application—The DIS monitor

To run the DIS monitor, type the following command line:

```
$VLS_HOME/bin/dis_mon.exe
```

You may receive an error message that includes information similar to

```
Connecting to DIS manager on YOUR_HOST_NAME...
pkg_open: client connect: Connection refused
Unable to connect to DIS manager on YOUR_HOST_NAME
cleaning up.
```

The DIS monitor needs to connect to the ARL DIS manager. This error message most likely means that the DIS manager was not started or has stopped or that a path (network route) to the computer where it is running could not be found. There are command line options that allow the DIS monitor to look for the ARL DIS server at other computer IP addresses or sites. For information about these and other command line options and details about the DIS monitor, see the **dis_mon(1)** "man" page in Appendix B.

5. INITIALIZING THE SERVER

This section explicitly notes server starting options, location and formats of initialization files, and other preparatory information required to start the server.

5.1 Server Initialization Files

Recall that the environmental variable VLS_HOME set from Section 3.3 is set to the "home" directory where the DIS lethality server was installed. Initialization files are located in the Data/Init subdirectory relative to VLS_HOME. That is, initialization data files are located in the directory

```
${VLS_HOME}/Data/Init/
```

The main initialization file in this directory is **vls_db_init.ini**. This file tells the server where to find all the other initialization files. Only three initialization files are identified by **vls_db_init.ini**:

1. A DIS enumeration file—these are the names and equivalent DIS numerical representation for entities, munitions, etc. More than 6,000 IEEE standard enumerations are provided [2].

2. A DIS auxiliary enumeration file—intended for "additional" entities added for a particular exercise.

3. A lethality "***meta data***" file—this tells the server all it needs to know about the lethality data to be delivered upon demand. The meta data file contains meta data records.

A lethality meta data record identifies several items for the server. First, it specifies which type of vulnerability/lethality (V/L) analysis method is used when a particular threat attacks a certain target. Then it identifies where the data are given that describe the damage state outcomes (with respect to the type of vulnerability analysis method in question). Finally, the meta data record identifies which library functions are used to read the data source. (Identifying a library function allows flexibility in how data are stored and retrieved. Vulnerability data need not be just static "look-up" tables. They may be a reference to a network connection or even a separately running application that calculates results "on the fly".)

It was stated that the lethality meta data file identifies the "V/L analysis method". One such example of an analysis method is the mobility, firepower, catastrophic (MFK) method for describing damage state outcomes (as seen in Table 1).

Table 1. MFK "Probability" Space

Outcome	Outcome Explanation
MKILL	Mobility and only mobility kill.
FKILL	Firepower and only firepower kill.
MFKILL	Mobility and firepower kills.
KKILL	Catastrophic kill.
NoDamage	No Additional damage inflicted.

In the MFK method, the set of all outcomes of a target-threat interaction are defined in terms of these conditions. Since these sets are normally treated as probabilistic events, it is necessary that the complete set of outcomes contain the universe of all possible events (so that their probabilities may sum to one). Any number of analysis methods are possible, provided that mathematical and probabilistic rules are adhered to and a reasonable V/L taxonomy is applied. It is the responsibility of higher level applications (e.g., war games) to know what these V/L results

mean and to treat them in an appropriate manner. V/L server technology has potentially powerful implications to the analysis community, provided the V/L metrics and applications that use them (e.g., war games) are properly coupled **[5]**. Currently, the server just implements the MFK method that is an "end game" description of kill probability given a hit ("PKH"). How another method is incorporated into the server is explained in Section 7. Other specifics concerning the formats for the **vls_db_init.ini**, DIS enumerations, and the meta data records are presented in the **vls_db_init(5)** manual page in Appendix B.

6. COMMUNICATING WITH THE SERVER

This section shows in a general sense how application programs may communicate with an initialized and running server. For the explicit details, see the manual pages for **vlserver(1)** and **vlsclient(3)** in Appendix B.

The V/L server has a group of API calls specifically designed for high level applications (such as war games and simulators). (That is, these applications are high level as viewed from the perspective of executing high fidelity vulnerability calculations.) This API group is called the VLSClient (or **vlsclient(3)**) library. For war games and other high level applications, this interface to the V/L server provides the functionality and fidelity needed for detailed vulnerability analysis; yet, this is accomplished with a relatively simple interface. These functions communicate directly with a running DIS VL server module (**vlserver**) as shown in Figure 2 (a modified view of the server architecture that was displayed in Figure 1).

To avoid confusion, the **VLSclient** library was not shown in Figure 1. The **VLSclient** calls are actually compiled in a client's application. This is shown in Figure 2 where the font size of the dashed lines separates **Client** from **VLSclient**. While the API may appear large in this figure, the interface itself is quite simple, comprising only the four functions shown in Table 2.

Client applications need only open *(vls_open())*, a connection to the server. They may send *(vls_send())* and read *(vls_read())* the answer to as many queries as they like and may close *(vls_close())* the connection when appropriate. The syntax for sending and receiving answers to queries is explained in the **vlserver(1)** and **vlsclient(3)** manual pages of Appendix B.

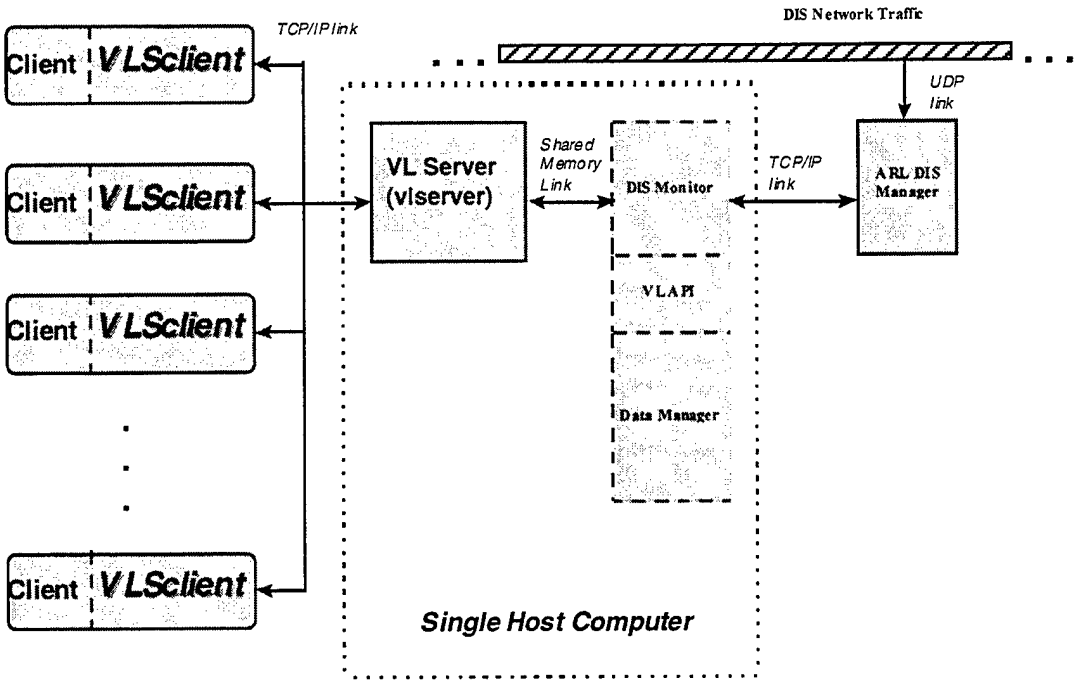


Figure 2. A Modified View of the Server Architecture.

Table 2. Application's Interface to the Server (VLSclient)

API	Purpose
vls_open()	open a connection with the vl server.
vls_close()	close a connection.
vls_send()	send a message (usually a query) to the server.
vls_read()	read data (usually an answered query) from the server.

7. EXPANDING THE SERVER

From a programming point of view, the server is designed to be expandable. However, many extensions can be accomplished without programming (by manipulating system parameters and initialization data; see Section 5). Other enhancements require additional software. This section focuses on modifications that require changes in the software.

7.1 Adding a New Vulnerability Taxonomy Description (vulnerability method)

In Section 5, we described a vulnerability method implemented in the DIS V/L server (the MFK method, Table 1). The server's overall architecture is designed to accommodate other

vulnerability descriptions upon demand by following the approach outlined in this section (some assembly is required).

7.1.1 *What is a Vulnerability Description?*

In the MFK method, all outcomes of a target-threat interaction are defined in a finite set. Since outcomes in these sets are normally treated as probabilistic events, the complete set of outcomes must contain the universe of all possible events (so that their probabilities may sum to one). Other methods of describing a system's vulnerable state may be defined (with more or less fidelity) in the same manner (e.g., a simple binary methodology with two states, "dead" or "alive"). That is, the outcome of a V/L analysis will result in the subject entity being classified as either "dead" or "alive". Both the MFK and the dead/alive taxonomies are "vulnerability descriptions". They describe a finite (yet comprehensive) set of outcomes that describe a system's performance capabilities following the occurrence of some event. However, as far as the V/L server is concerned, it is not necessary for probabilities to be associated with each outcome. For instance, another vulnerability description could be a list of components. These components could be identified as functional or nonfunctional. It would then be the responsibility of the calling application to simulate the system's behavior when only certain specified components were working. The process just described follows a very high fidelity vulnerability methodology known as "degraded states" **[6, 7, 8]**.

7.1.2 *Why the Server Needs to Know Which Vulnerability Description is Used*

The server needs to know which data to deliver to a client (and in what format). If a client simulator is designed to operate using an MFK method, it would be meaningless to send this simulator degraded states or any other V/L description. Secondly, the server needs to know which battlefield environmental parameters to monitor (in order to initialize conditions for the vulnerability calculation).

How then does the server distinguish between vulnerability methods and how does a client communicate its wishes to the server? The short answer is that we first incorporate the vulnerability method into the server, then select a protocol so that clients may query according to that vulnerability description. In the next section, we follow the steps for "folding" a new vulnerability method into the server. This involves adding new APIs to the V/L API layer. Later, in Section 7.3, we see how to establish a query-answer protocol (between a client and the DIS server), which will allow remote access to these new APIs.

7.1.3 *Incorporating a New Vulnerability Method Into the Server*

The first step is to edit data structures in the **vl_meth.h** "include" file. By way of example, suppose we wish to add a new vulnerability method that describes a vehicle or system as strictly "alive" or "dead". Let us call this a BINARY method. First, we will edit the file **$VLS_HOME/src/Db/vl_meth.h** and add the lines shown as bold in Figure 3. (The code in Figure 3 that is not bold was already present before any changes were made.)

Starting on line 32 of **vl_meth.h** of Figure 3, we see that base enumerations are created for the new (BINARY) vulnerability description. The names of these base enumerations are preceded by a double underscore "__". The reason behind this is to force the final revision of these names (for subsets within the vulnerability description) to start with an enumeration of zero (0). The result is that the first element name will have an internal value of zero (0), the second one (1), the third two (2), and so on. In this way, when probabilities are returned by a (newly created) lethality server API for every possible outcome in a vulnerability description set, they may all be returned in a single array. The elements of that array may be referenced (in order) by using the names defined for each outcome in the vulnerability description. (Look ahead to the final revision of the names defined on lines 42 and 43.)

In Section 7.1.2, we noted that the server needs to know which data to monitor on the virtual battlefield in order to have the proper parameters available for the lethality calculation. The next section of code we turn our attention to (on line 104) is modified so that this may occur. Here, we are adding a new enumeration for "collision" type interactions. The enumerations already defined (in the data type *VLSetParam_t*), starting on line 86, are used to inform the server's V/L APIs which parameters are significant for a particular calculation. These parameters are applicable for any type of vulnerability methodology (e.g., MFK or BINARY). On line 104, we define the internal enumeration **VL_PARAM_SET_COLLISION** to inform server APIs to prepare data parameters for damage resulting from collision. Damage resulting from munition threats (both direct and indirect fire) were already defined on lines 89-90. Later (in Figure 5), we shall see how server APIs use this information to prepare initial conditions for a lethality calculation.

The *VLSetParam_t* enumerations (defined between lines 86 and 107) are internal values and only have meaning within the V/L server code itself. It is also necessary for the server to be able to associate these internal values with external representations. This association is made in the *VL_Meth_List[]* array defined in Figure 3 on lines 119 through 140. The character string "DIS Collision" is associated with our newly defined vulnerability parameter type on line 136 of **vl_meth.h** (Figure 3). The server looks for these string representations when it reads the external

```
/* $Id: vl_meth.h,v 0.6 1997/08/21 17:08:58 geoffs Exp geoffs $ */
#ifndef VL_METH_H
#define VL_METH_H

typedef enum _mfk_result_enums {
       _PS_LOWER_BOUND = -3,

         PS_ERROR = -2,
         __PS_MFK_LOWER_BOUND = -1,
         PS_MFK_M = 0, /* start at zero so it can be 1st element in an array*/
         PS_MFK_F,
         PS_MFK_MF,
         PS_MFK_K,
         PS_MFK_NODAMAGE,

       /* if more Probability spaces are added, then
        * we will have to add make the _mfk_result_enums
        *  hidden enumerations (like: " __PS_MFK_M ")
        * and add upper an lower bounds for that result
        * type (like: __PS_MFK_LOWER_BOUND,)
        * then we do this:
        * # define PS_MFK_M (__PS_MFK_M-__PS_MFK_LOWER_BOUND+1)    (* 0 *)
        * # define PS_MFK_F (__PS_MFK_F-__PS_MFK_LOWER_BOUND+1)    (* 1 *)
        * # define PS_MFK_MF (__PS_MFK_MF-__PS_MFK_LOWER_BOUND+1) (* 2 *)
        * ...
        *
        *  an array is dimensioned:
        *     all_types_ps_mfk[ PS_MFK_UPPER_BOUND ]
        */
        __PS_MFK_UPPER_BOUND,

             __PS_BINARY_LOWER_BOUND,

         __PS_BINARY_DEAD,
         __PS_BINARY_ALIVE,

             __PS_BINARY_UPPER_BOUND,

       _PS_UPPER_BOUND
} VL_Result;

#define  PS_BINARY_DEAD  (__PS_BINARY_DEAD-  1  -  __PS_BINARY_LOWER_BOUND)/*  0  */
#define  PS_BINARY_ALIVE  (__PS_BINARY_ALIVE-  1-__PS_BINARY_LOWER_BOUND)/*  1  */

#ifdef VL_METH_C
struct __VL_Result_strings_t {
     VL_Result id;
     char *string;
};
static struct __VL_Result_strings_t __VL_Result_strings[]= {
       { _PS_LOWER_BOUND, "_PS_LOWER_BOUND" },
       {   PS_ERROR, "PS_ERROR" },

         {__PS_MFK_LOWER_BOUND, "__PS_MFK_LOWER_BOUND"},

       {   PS_MFK_M, "PS_MFK_M" },
       {   PS_MFK_F, "PS_MFK_F" },
       {   PS_MFK_MF, "PS_MFK_MF" },
       {   PS_MFK_K, "PS_MFK_K" },
       {   PS_MFK_NODAMAGE, "PS_MFK_NODAMAGE" },

         {__PS_MFK_LOWER_BOUND, "__PS_MFK_UPPER_BOUND"},

       { _PS_UPPER_BOUND, "_PS_UPPER_BOUND" }
```

```
65
66          /*  add  a  BINARY  Vulnerability  Methodology  */
67
68              {__PS_BINARY_LOWER_BOUND,  "__PS_BINARY_LOWER_BOUND"},
69
70          {     PS_BINARY_DEAD,  "PS_BINARY_DEAD"  },
71          {     PS_BINARY_ALIVE,  "PS_BINARY_ALIVE"  },
72
73              {__PS_BINARY_LOWER_BOUND,  "__PS_BINARY_UPPER_BOUND"},
74
75          { _PS_UPPER_BOUND, "_PS_UPPER_BOUND" }
76      };
77      #endif
78
79      /*
80       ~ VL_Meth data type.
81       *
82       * type used to indicate which data sources (inputs) are sufficient
83       * to set the VL parameters in order to be able to return the
84       *   correct result from the lookup table (or other data source).
85       */
86      typedef enum {
87             _VL_INPUT_ENUMS_BEGIN    = 0 /* below lowest boundary */
88
89             , VL_PARAM_SET_METH_DIS_HitToKill
90             , VL_PARAM_SET_METH_DIS_ProxKill
91
92             /*
93              * VLSetParam_t == VL_PARAM_SET_METH_DIS_HitToKill
94              *      (or  VL_PARAM_SET_METH_DIS_ProxKill)
95              * Indicates that passing the DIS PDUS
96              *             Enity State (target)
97              *             Enity State (firer)
98              *             FirePDU
99              *             DetonationPDU
100             *   shall be sufficient to set the VL parameters to return the
101             *   correct result from the lookup table (or other data source).
102             */
103
104            ,  VL_PARAM_SET_COLLISION
105
106             , _VL_INPUT_ENUMS_END          /* upper boundary */
107     } VLSetParam_t ;
108
109     typedef struct _vl_meth_struc {
110            VLSetParam_t  id;  /*Analysis input Parameter Methodology Identifier */
111            char *name;    /* String Identifier for this method
112                            *  ( used in the Meta V/L Table list )
113                            *    "DAMAGE_SOURCE_META_DATA_FILE"
114                            */
115     } VL_Meth;
116
117
118     #ifdef VL_METH_C
119     /*
120      *  VL_Meth_List[]    identify the which inputs are needed
121      *                    (e.g. for DIS - which PDUs are needed)
122      *                    and it also is used to identify what
123      *                    special procedures or processes are
124      *                    required handling to handle the inputs
125      *                    vulnerability calculation (e.g. when
126      *                    "DIS HitToKill" is being used, then
127      *                    the munition *MUST* hit the target to
128      *                    have ANY effect.
```

```
129     */
130     static VL_Meth VL_Meth_List[] =  {
131               { _VL_INPUT_ENUMS_BEGIN, NULL}
132
133             , { VL_PARAM_SET_METH_DIS_HitToKill , "DIS HitToKill" }
134             , { VL_PARAM_SET_METH_DIS_ProxKill ,  "DIS ProxKill" }
135
136             , { VL_PARAM_SET_COLLISION,  "DIS  Collision"  }
137
138              , { _VL_INPUT_ENUMS_END, NULL}          /* upper boundary */
139     };
140
141     #endif
```

Figure 3. vl_meth.h Code Changes—Adding a New Vulnerability Method.

lethality "**meta data**" records (described in Section 5.1). A sample lethality meta data file (the **DAMAGE_SOURCE_META_DATA_FILE**) is shown on the **vls_db_init(5)** manual page of Appendix B. Specifically, the third field of a **DAMAGE_SOURCE_META_DATA_FILE** contains the text string that associates a set of (initial condition) parameters with a vulnerability data source that requires those parameters. In the meta data file excerpt (shown near the end of the **vls_db_init(5)** manual page and repeated in Figure 7), "DIS HitToKill" is displayed as the string identifying the vulnerability initial condition parameter requirements. On line 133 of Figure 3, we can see that this external string is associated with the internal enumeration **VL_PARAM_SET_METH_DIS_HitToKill**.

Next, we show how the server internally uses the enumerations (to pass the proper parameters to server V/L APIs) and how multiple vulnerability methodologies are accommodated.

7.1.3.1 How the Server Accommodates Multiple Vulnerability Methodologies and Multiple Types of Parameters

When a new vulnerability method is created, new API routines also have to be created to deliver the new type of data.

These routines accomplish the following objectives:

1. They set the appropriate parameters that describe the conditions at the time a lethal event occurs (e.g., munition type, terminal velocity, etc.);

2. Once these parameters are set, the delivery routine must then call the appropriate lethality analysis algorithm (this could be as simple as a table look-up function); and

3. They finally return the data (in a form and format that is appropriate for that vulnerability method) to the calling function. This architecture is depicted in Figure 4 where we see a data delivery layer, a lethality data reader (table look-up) layer, and a vulnerability parameter layer between them. The layers seen in Figure 4 are actually sub-layers that fall within the larger **V/L API** layer, which was shown in Figure 1.

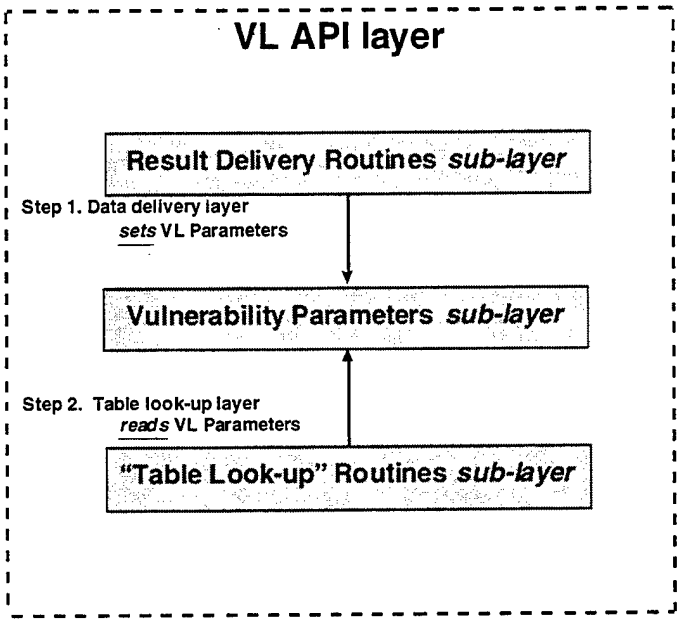


Figure 4. V/L API: Lethality Data Delivery, Parameters, and Reader Layers.

Because the **Result Delivery sub-layer** needs to set values in the **Vulnerability Parameters sub-layer**, routines in the **Result Delivery sub-layer** must have prior knowledge of all the environmental information necessary to complete lethality calculation for the vulnerability method in use. For example, in order to describe the results of a munition impact, the MFK methodology requires information from the DIS fire, detonation, and entity state PDUs. Therefore, these PDUs are passed to all **Result Delivery sub-layer** routines that service the MFK methodology. We may examine prototypes of some implemented MFK delivery routines (shown in Table 3). (These APIs are documented in detail in the **vl(3)** manual page of Appendix B.) Notice that each routine has a *VLSetParam_t* enumeration as its first argument. This first argument (VLSetParam_t itype) informs the called API in what form the environmental variables will appear. That is, it tells the function which arguments will be substituted for the "..." seen in Table 3.

Table 3. Some Result Delivery APIs (for the MFK methodology)

ANSI C Prototype Declaration (for the MFK Methodology).
`float*  vl_mfk_ArlDIS_ProbAll_NoNet( VLSetParam_t itype, ... );`
`double  _vl_mfk_ArlDIS_ProbM_NoNet(VLSetParam_t itype, ...);`
`double  _vl_mfk_ArlDIS_ProbMF_NoNet(VLSetParam_t itype, ...);`
`double  _vl_mfk_ArlDIS_ProbF_NoNet(VLSetParam_t itype, ...);`
`double  _vl_mfk_ArlDIS_ProbK_NoNet(VLSetParam_t itype, ...);`
`double  _vl_mfk_ArlDIS_ProbNoDamage_NoNet(VLSetParam_t itype, ...);`
`VL_Result  vl_mfk_ArlDIS_Result_NoNet(int*flg, VLSetParam_t itype, ...);`

Thus far, the server only has two possible values of type *VLSetParam_t* (namely, VL_PARAM_SET_METH_DIS_HitToKill and VL_PARAM_SET_METH_DIS_ProxKill). Each of these parameter setting indicators informs the server to expect DIS fire, detonation, and entity state PDU arguments to follow as the remaining arguments to the function call. They also inform the server that a "munition" is the damage-causing mechanism. (The delivery routines will then proceed to set "munition" type variables in the **Vulnerability Parameters sub-layer** by using these passed PDU arguments.)

However, we could easily define a new *VLSetParam_t* type that tells the delivery routines to expect some other type of arguments (in order to return an MFK result based on different input parameter formats). For example, non-munition damage (such as damage caused by a collision between moving vehicles) could be accommodated by adding a new *VLSetParam_t* type (e.g., VL_PARAM_SET_METH_DIS_COLLISION) in which the resulting delivery routines would now expect "collision" type variables (as the remaining arguments). In the DIS environment, a combination of collision and entity state PDUs would suffice as arguments. The delivery routines could then return MFK results based on these different damage-causing mechanisms (provided that valid data sources existed in the "**Table Look-Up" sub-layer** that described MFK damage resulting from those mechanisms [e.g., vehicular collisions]).

7.1.3.2 Adding V/L Layer API Routines for a New Vulnerability Method

We now return to the sample task—adding to the V/L API's **Result Delivery sub-layer** a new vulnerability method that describes an entity's vulnerability state as strictly "alive" or "dead" (our "BINARY" vulnerability method).

First, we need to decide what values are useful to be returned by the BINARY method APIs. These values are then returned by the new V/L API functions we will write. In this case, we shall have several returned types. A function will be written for each type. Using this approach, we outline a new set of APIs in Table 4. We can follow the function-naming pattern already used in the MFK APIs (shown in Table 3).

Table 4. Newly Defined Result Delivery APIs (for the BINARY methodology)

ANSI C Prototype Declaration (for the BINARY Methodology)
`float*  vl_binary_ArlDIS_ProbAll_NoNet(VLSetParam_t itype, ... );`
`double  _vl_binary_ArlDIS_ProbDEAD_NoNet(VLSetParam_t itype, ...);`
`double  _vl_binary_ArlDIS_ProbALIVE_NoNet(VLSetParam_t itype, ...);`
`VL_Result  vl_binary_ArlDIS_Result_NoNet(int*flg, VLSetParam_t itype, ...);`

Briefly, the specific purpose of each API is as follows:

```
double    _vl_binary_ArlDIS_ProbDEAD_NoNet(VLSetParam_t itype, ...);
```

returns the probability that the outcome of the event results in a "DEAD" state for the system in question.

```
double    _vl_binary_ArlDIS_ProbALIVE_NoNet(VLSetParam_t itype, ...);
```

returns the probability that the resulting outcome of the event is an "ALIVE" (or non-DEAD) state of the system in question.

```
float*    vl_binary_ArlDIS_ProbAll_NoNet( VLSetParam_t itype, ... );
```

returns an array containing the probabilities of all possible outcomes occurring. The array elements are indexed according to the internal definitions we established in the vl_meth.c file (Figure 3, lines 42 and 43). Namely, the "[PS_BINARY_ALIVE]" element of the array contains the probability that the outcome of the event results in an "ALIVE" (or non-DEAD) state of the system in question. Similarly, "[PS_BINARY_DEAD]" indexes the probability of a "DEAD" state.

```
VL_Result  vl_binary_ArlDIS_Result_NoNet(int*flg, VLSetParam_t itype, ...);
```

determines the probability of each event occurring, then randomly draws an outcome from the set of possible events. The outcomes are drawn according to the distribution established by the

probabilities. The answer returned is of type VL_Result. Therefore, the only allowed results returned by this API are __PS_BINARY_ALIVE and __PS_BINARY_DEAD, as we established in lines 34 and 35 of Figure 3. In fact, any result not falling between __PS_BINARY_LOWER_BOUND and __PS_BINARY_UPPER_BOUND should be considered invalid. For example, if P(PS_BINARY_ALIVE)=.75 and P(PS_BINARY_DEAD) = .25, then about 75% of the time, a VL_Result of __PS_BINARY_ALIVE will be returned (and __PS_BINARY_DEAD will be returned 25% of the time).

Each of these APIs will read the passed parameters, use those parameters to set initial conditions (in the **Vulnerability Parameter sub-layer**), call the vulnerability analysis routine (in the **Table Look-up sub-layer**), and return the result. By way of example, we will concentrate on the API `vl_binary_ArlDIS_ProbAll_NoNet()`. The other APIs will follow a similar pattern.

`vl_binary_ArlDIS_ProbAll_NoNet()` will return an array of floating point numbers that represent the probabilities of achieving the two kill levels (dead or alive). When called, this function's first argument **(itype)** could be any of the **VLSetParam_t** enumerations we defined on lines 86 through 107 of Figure 3. Figure 5 displays a sample ANSI C function showing how `vl_binary_ArlDIS_ProbAll_NoNet()` could be implemented.

On line 9 of Figure 5, we define a default outcome (`binaryPS_HasNoEffect`) that is returned when an exception occurs in which we know that there will be no additional damage to the entity or component being threatened. Later (on line 140), we shall see how this default outcome shall be used to prevent an erroneous result from being returned during certain conditions.

The next significant portion of the code we note is on line 59 where we determine what input parameters are required in order to establish the proper initial conditions for the vulnerability calculation. From lines 61 through 74, the "collision" initial condition parameter is handled. We see that when "collision" is the damage mechanism being evaluated, the collision PDU and the entity state PDUs must be provided as arguments to the API. The entity state PDUs that are provided are for both the entity whose vulnerability is being evaluated (shown as "tgt" on line 66) and the entity that is colliding with it (`colliding_entity`). The order in which these arguments are provided is significant. Following retrieval of the arguments (on lines 66 through 68), these PDUs are used to set parameters in the **Vulnerability Parameter sub-layer** "VLParam". (The VLParam layer is shown on Figure 4 and documented in the manual page **VLParam(3)** in Appendix B.) Source code for the function (`vlp_setp_all_Collision_Frm_DIS()`) shown on line 71 is not provided. Its purpose is to decompose the PDUs passed to it, extract applicable information from them, and use that information to set the appropriate variables in the VLParam layer. It is assumed

```
#include <stdlib.h>
#include <stdarg.h>

#include "vl.h"
#include "vl_meth.h"
#include "vlparam.h"
#include "metatbls.h"

static float binaryPS_HasNoEffect[]={ 0., 1.};
     /*
      * recall that PS_BINARY_DEAD  = 0
      *             PS_BINARY_ALIVE = 1
      *  therefore binaryPS_HasNoEffect[]={ 0., 1.};
      *  is structured so that the first (zero'th element)
      *  may be indexed by PS_BINARY_DEAD (i.e.
      *  binaryPS_HasNoEffect[PS_BINARY_DEAD] ).
      */

/*
 ~ vl_binary_ArlDIS_ProbAll_NoNet()
 *
 *  float * vl_binary_ArlDIS_ProbAll_NoNet( VLSetParam_t itype,  ... )
 *
 * This function returns a static array containing probabilities of
 * certain kill levels.
 *
 * The first parameter argument is of type VLSetParam_t.
 *
 * This type is used to indicate which data sources (inputs)
 * are sufficient to set the VL parameters in order to be able
 * to return the correct result from the lookup table
 * (or other data source).   These indicated data sources (inputs)
 * shall then be the 2nd, 3rd, 4th, ... etc. parameter arguments
 * to the function.
 *
 *  RETURNS:
 *
 *       An array containing the probability of all possible outcomes.
 *   The array elements are defined as follows:
 *
 *
 * Array    Element (index)
 * Element  Value                      Value  Meaning
 * -------  ---------------     ---------------------------------
 *  0       PS_BINARY_DEAD      Probability that the subject is dead
 *  1       PS_BINARY_ALIVE     Probability of not being dead.
 */
float * vl_binary_ArlDIS_ProbAll_NoNet( VLSetParam_t itype,  ... )
{  va_list ap;
   static char *whoami="vl_binary_ArlDIS_ProbAll_NoNet()";
   float *ret;
   int missed_me, error, do_vl_calc,  ok_to_query;

   ok_to_query = 0; /* false */
   ret = NULL;
   error = 0;
   va_start( ap, itype );

   switch ( itype ) {

     case  VL_PARAM_SET_COLLISION:
      EntityStatePDU  *tgt,  *colliding_entity;
      CollisionPDU  *collision;
```

```
                /* extract the 2nd, 3rd, and 4th arguments */
              tgt                 = va_arg( ap, EntityStatePDU * );
               colliding_entity = va_arg( ap, EntityStatePDU * );
              collision           = va_arg( ap, CollisionPDU *);

        vlp_zero_all_params(); /* initialize paramters */
        vlp_setp_all_Collision_Frm_DIS(tgt, colliding_entity, collision);
        ok_to_query = 1;

    break;

      case VL_PARAM_SET_METH_DIS_HitToKill:
      case VL_PARAM_SET_METH_DIS_ProxKill:
{
        EntityStatePDU *tgt, *shooter;
        FirePDU *fire;
        DetonationPDU *det;

              /* extract the 2nd, 3rd, 4th, and 5th arguments */
              tgt     = va_arg( ap, EntityStatePDU * );
              shooter = va_arg( ap, EntityStatePDU * );
              fire    = va_arg( ap, FirePDU *);
              det     = va_arg( ap, DetonationPDU * );

             /* test to see that we know what type of target is present */
        if (tgt==NULL) {
           _rpt_error(RE_TGT_UKNOWN ,whoami);
           ++error;
        } else if (det==NULL) {
           _rpt_error(RE_THREAT_UKNOWN ,whoami);
           ++error;
        }

        if (error==0) {
              missed_me = FALSE;
         if (itype == VL_PARAM_SET_METH_DIS_HitToKill ) {
                /*
                 * See if we can
           *  ignore the detonation based on the result field.
           *
           */
           if (TRUE == vl_mfk_directFireIsAHit(det->detonation_result))
               missed_me = FALSE;
                 else
               missed_me = TRUE;
         }

               do_vl_calc=FALSE;
         if (error == 0) {
            if (missed_me == TRUE) {
               /* NOT a direct entity impact */
               switch(itype) {
                  case VL_PARAM_SET_METH_DIS_HitToKill:
                    do_vl_calc=FALSE;/* leave as false */
                    /*
                     * we know we missed with a hit-to-kill
                     *  threat. So we attempt to lookup
                     *  the (wrong) answer according
                     *  to the vl parameters.
                     *  But we do return a result.
                     */
                    ret = mfkPS_HasNoEffect;
                    break;
                 case VL_PARAM_SET_METH_DIS_ProxKill:
```

```
                        /* we don't care if it did miss - calc anyway. */
                    do_vl_calc=TRUE;
                    break;
                default:
                    cprint(CH_WARN,
                        "%s: switch missing case for method type %d\n",itype);
                    break;
                }
            } else {
                do_vl_calc=TRUE;/* no errors an we hit tgt! */
            }
        }
        if (do_vl_calc) {  /* tgt is hit */
          vlp_zero_all_params(); /* initialize paramters */
          vlp_setp_all_Munition_Frm_DIS(tgt, shooter, fire ,det);

        /*
         * vlp_setp_all_Munition_Frm_DIS()
         *  will have set VLP_target_id
         *              VLP_threat_id
         *    and other VLP_*  parameters.
         */
          ok_to_query = 1;
            }
       } /* end if error==0 */
          } /* end case: stmt. */
          break;

      default:
          cprint(CH_ERR,"%s: passed unknown VL methodology (%d)\n"
           ,whoami,itype);
          break;
  }



 /*
  * What this final code segment does:
  *
  * At this point the parameters have been added to the
  * the VLParam layer (see VLParam(3) manual page
  * If no errors occurred, then we are ready to
  * look for a meta record that matches the tgt, threat.
  * AND vulnerability method (namely "BINARY").
  * Once we have that record, we can retrieve a the
  * data source (URL) and the vulnerability calculation
  * function.
  * Finally we call that function and return its results.
  */
 if (ok_to_query == 1)
 { MetaTable_t mquery;
   MetaTable_t *mrec;
   extern MetaTable_t *MetaTable_get_rec();
   float *f, *(*funcptr)(void *);
   VL_Meth *mptr;


   /* zero meta Table data structure */
   memset((void*)&mquery, (int) 0, sizeof(MetaTable_t) );

   mquery.tgt = (dbEntityType*)  &VLP_target_type;
   mquery.threat = (dbEntityType*) &VLP_threat_type;
   mptr = vl_meth_get_FromID( itype );
   if (mptr==NULL) {
```

```
193         cprint(CH_ERR,"%s: internal error\n",whoami);
194         break;
195       }
196       mquery.vl_meth= mptr->name;
197
198       mrec=MetaTable_get_rec(mquery.tgt,mquery.threat,itype);
199       if (mrec==NULL) {
200            /* if mrec == NULL then no record found */
201            _rpt_error(RE_NO_META_REC ,whoami);
202            ++error;
203       } else {
204            funcptr = db_tbl_result_func(mrec);
205            if ( funcptr != NULL ) {
206            f = funcptr( db_tbl_retrieve( mrec ) );
207            if (f==NULL) {
208              /* error reading tbl result */
209                 _rpt_error(RE_VLSOURCE_INTERP ,whoami);
210                 ++error;
211            }
212            ret = f;
213         } /* endif ( funcptr != NULL ) */
214       } /* end if (mrec == NULL)  else */
215     } /* end if (ok_to_query == 1)
216
217     va_end(ap);
218
219     return ret;
220   }
```

Figure 5. Adding an API for a New Vulnerability Method vl_binary_ArlDIS_ProbAll_NoNet().

that additional parameters needed to execute a collision damage assessment have been added to the VLParam layer (such as the masses of the colliding entities, etc.).

If no errors occurred in setting the VLParam parameters, then the vulnerability assessment may proceed (on lines 178 through 215).

Meanwhile, we note that the case for "munition" type damage is handled between lines 76 and 161. The first thing we note about this section of code is that it is much longer than the "collision" damage section; yet, it does essentially the same thing (initializes the VLParam layer's variables). The difference is that this code section performs robust and proper error checking throughout. Errors are recorded to rpt_error APIs (see **rpt_error** on the **cprint(3)** manual page in Appendix B). Rpt_error APIs store important information about what went wrong when errors occur; this information may be extracted by other routines. (This is useful because the application-calling server routines may be removed by several layers from the APIs where the error occurred. By the time the program returns to the application level, the nature of the error may be lost.) A second thing this section of code adds is that it tests for exception conditions. In this example, test for a direct hit against the queried target. If we determine that the munition requires a direct impact on the target to initiate any significant damage but we

"missed" the target, then a special exception value is returned (on line 126). This is the value we defined on line 9.

We now return our attention to lines 178 through 215. At this point, the parameters have been added to the VLParam layer. If no errors or exceptions occurred, then we are ready to look for a meta record that matches the target, threat, and vulnerability method used. Lines 189 through 196 gather the threat, target, and V/L method identifier. Notice that the target and the type of threat that is threatening it are extracted from the VLParam layer on lines 189 and 190. The variables shown here are members of the VLParam layer (`VLP_target_type` and `VLP_threat_type`). The external form of the vulnerability method is required to retrieve the meta data record. (Remember that we defined external representations for these types on lines 120 through 140 in Figure 3.) On line 196 of Figure 5, the external American standard code for information exchange (ASCII) identifier of this vulnerability method is used to fill the V/L method field of the meta record being queried. These data items are added to a blank meta data structure. We then search for the meta record that matches these parameters on line 198. (Recall that we explained how and where vulnerability meta data records were read in Section 5.1 and in Appendix B, **vls_db_init(5)**.) Once we have that meta data record, we may retrieve the location of the appropriate lethality data (in uniform resource locator [URL] format) and the function that calculates the system's vulnerability. On line 204, we obtain a pointer to that function from the data manager API `db_tbl_result_func()`. This referenced function operates on a data set to return the lethality calculation for a set of initial conditions that are provided by the **VLParam sub-layer**. We call this function the lethality "data look-up function" because it often is just parsing a look-up table. However, it may actually do the lethality calculation itself or initiate other processes that do the calculation. How it gets the results is not a matter of concern as long as it returns the results appropriate to the vulnerability method in use (the "BINARY" method in our sample case). The API `db_tbl_retrieve()` that we see used on line 206 returns that data set to be operated upon by the "data look-up function". Normally, `db_tbl_retrieve()` returns a table of vulnerability results that are then used as the look-up table for our table look-up function. However, `db_tbl_retrieve()` need not return a look-up table; it could return a URL, a password, or any data structure or value. As with the data look-up function, it does not matter, as long as whatever it returns will be usable by the table look-up function to return the correct results for the present vulnerability calculation. The APIs, `db_tbl_result_func()` and `db_tbl_retrieve()`, are documented in the **db(3)** manual page of Appendix B. How and where the data look-up functions are placed into the server is the subject of Section 7.2.

Finally, on line 206, we call our data look-up function and return its results (on line 219) to complete the vulnerability calculation.

7.2 Adding a New (look-up) Table Format

Every lethality data set type is required to have a data reader (also called data look-up function) and a data loader function. The loader function initializes the lethality data set. This may simply involve reading a static table into memory or may involve slightly more complicated initialization procedures such as opening network connections, etc. There is really no limit as to what the loader function does. (However, the data loader functions provided with this initial release of the DIS lethality server only read and load static "look-up" tables into memory.) The second required function, the data reader (or look-up) function, has the responsibility for accessing the initialized data set. It then returns results that are appropriate for the associated V/L method. How these two functions are incorporated into the server is explained in this section.

Both the data reader and the data loader functions must be defined and compiled into the server before "run time". The following steps outline the procedure for doing this:

1. Add an internal identifier to the list of enumerations that identify new data sources.
2. Write prototypes for the two functions and add them to a static table.
3. Add the source code for your prototyped functions.
4. Recompile the server.

These four steps are now covered in detail in the next four sections.

7.2.1 *Adding Internal Identifiers for New Types of Vulnerability Data (or formats)*

Unique internal identifiers are required for each new type of vulnerability data. A vulnerability data type may be considered "new" not only if it is new, but also if it is a previously defined vulnerability data type that is packaged in a different manner. For instance, a look-up table of IUA[3] data could be for a high explosive antitank (HEAT) munition threat, or it could be for a kinetic energy (KE) munition. Both data sets represent the same type of data (MFK), but both have tabular formats that differ (IUA HEAT format versus IUA KE). Hence, each needs its own data reader function (and internal identifier). The file $VLS_HOME/src/db/tbl_fmts.h contains

[3]IUA (individual unit action). This type of data represents loss of combat capability for a given threat. The tabulated data are formatted in columns and rows according to target range, aspect angle of attack, and kill level (MFK).

references to all the known data formats. Figure 6 displays this file. On lines 14 and 15 of Figure 6, we can see where internal enumerations for IUA data types are provided (for both HEAT and KE formats).

By way of example, we shall add a new format for the "collision" damage we defined on line 136 of vl_meth.h in Figure 3. On line 18 of Figure 6, we define `PKS_BINARY_COLLISION` to represent our new data type. This enumeration internally represents a data file format (or function, network protocol, etc.). Data look-up (also called reader) functions will be written, which will return data that are commensurate with whatever we expect to be returned by this newly defined data type. That data type is associated with our "BINARY" V/L methodology. This may sound rather enigmatic, but the fact is that the data source may be anything (not just a static data file of a particular format). It could be a database, a network connection, a "spawned" program, or just about anything. Our data reader (and initializer) functions (which we will write later) are the sole entities that need to know these particulars. What is important is that the V/L API, which indirectly calls the data reader function, receives what it expects to receive from the data reader. The important point is that the data reader function and the V/L API that calls it agree about what shall be returned and how it is properly used. In this particular example, it will expect to receive an array of probabilities from the "BINARY" vulnerability set. (This is what we assumed when we wrote the API function "`vl_binary_ArlDIS_ProbAll_NoNet()`" shown in Figure 5.) On line **206** of this function (Figure 5), the data reader returns its set of data, the results of the vulnerability analysis. This returned value (which is really of type "void *") is cast to a pointer to a floating point array ("float *"). Hence, the data reader is expected to return a pointer to a floating point array that contains the probability of a kill and the probability of not being killed (in accordance with the BINARY vulnerability methodology we established). The point is that the V/L API function had better understand quite clearly what type of data set is going to be returned for a given vulnerability methodology (BINARY method for collision damage in this case). As we continue to examine the file $VLS_HOME/src/db/tbl_fmts.h in Figure 6, we shall see how this association is established.

7.2.2. *Adding New "Table Lookup" Function Prototypes*

On lines 130 through 134, we added a new element to the `LookUp_Tbls[]` array. This is the point where the association is made between the data reader function and the data type identifier. On line 130, the internal enumeration we created on line 18 is used to identify the new type of vulnerability we are referencing. Following this, the ASCII string "**PKS_BINARY_COLLISION**" is used for the "name" field. This is an important field because it becomes the external representation of our new data type (and data source format). This name must appear in the

```
#ifndef _TBL_FMTS
#define _TBL_FMTS

#include <stdio.h>           /* ANSI C header files */
/* local header files */
#include <tbl_rdrs.h>        /*  prototypes of your table reader(s) go here */
#include "vl_meth.h"

enum _TblFmt_Enum {
     _TBLFMT_ERROR = 0       /* err==0, see tbl_fmt_is_valid_type() */
     ,_START_OF_TblFmt_Enum

     ,IUA_HE
     ,IUA_HEAT
     ,IUA_KE
     ,IUA_STAFF

     ,PKS_BINARY_COLLISION

     ,_END_OF_TblFmt_Enum
};
typedef enum _TblFmt_Enum  TblFmt_Enum ;

/*
 *  TblFmt_Enum    identifies the table (or data) format.
 *
 */



typedef struct TblFmt2ResultType_struct  {
   TblFmt_Enum        fmt;          /* format of data source */
   VL_Result    returned_type;  /* the type of result returned
                        * by the reader function
                        */
} TblFmt2Result_t;
/*
 *  TblFmt2Result[]    will have one entry for every
 *            known table format (that is one for every
 *            TblFmt_Enum).
 */


typedef struct _TblFmt_t {
     TblFmt_Enum type;       /* enumeration for your table format
                        *  - This identifies the format of the
                        *    data in the table.
                        */
     char *name;             /* a single word name for your table format */
     char *description;      /* a short description  of it */
     void *(*reader_func)(FILE *); /* reader function takes on FILE* arg */
     void *(*result_func)(void *);/* VL reporting function */
                       /* result_func() takes argument pointer
                        * to the look-up table structure loaded
                        * into memory.
                        */
                       /* returns vl data which  describes
                        *  the result of the lethality event.
                        * The format of the data is up to the
                        * returning function.
                        * However, it must agree with the
                        * output format which is implied by
                        * the  "name" field.
                        *
```

```
                        *
                        * For instance if
                        *  name = "IUA_HEAT"
                        * then result_func will return
                        * a floating-point array of 5 numbers.
                        * (Its returned value should therefore
                        *  be cast as (float *) ).
                        */
} TblFmt_t;



# ifndef _DB_C
   extern TblFmt2Result_t TblFmt2Result[];
   extern TblFmt_t LookUp_Tbls[];
# else
static TblFmt2Result_t TblFmt2Result[] =
{
     { IUA_HE,    __PS_MFK_LOWER_BOUND } /*  returns MFK data */
    ,{ IUA_HEAT, __PS_MFK_LOWER_BOUND } /*  returns MFK data */
    ,{ IUA_KE,    __PS_MFK_LOWER_BOUND } /*  returns MFK data */
    ,{IUA_STAFF, __PS_MFK_LOWER_BOUND } /*  returns MFK data */

      ,{PKS_BINARY_COLLISION,__PS_BINARY_LOWER_BOUND}/*returns  BINARY  pk  data  */

};
static  TblFmt_t LookUp_Tbls[_END_OF_TblFmt_Enum+1] = {
   { _TBLFMT_ERROR , "_error"   , "Not a known table format" , NULL , NULL }
  ,{_START_OF_TblFmt_Enum, NULL, NULL , NULL, NULL }
 /*-------------------------------------------------------------------------*/
 /* DO NOT ADD ABOVE THIS LINE                                              */
 /*+struct format is:-----------------------------------------------------+*/
 /*| enum        name         descript       reader_func    result_func|*/
 /*+-----------------------------------------------------------------------+*/

 /*
  * If you get a "tblfmt_WHATEVER_NAME" undeclared here.
  *  then you may have not added its prototype to the header file:
  *   #include <tbl_rdrs.h>
  */

  ,{IUA_HE,"IUA_HE","(IUA) High Explosive (HE)"
     ,tblfmt_iua_heat_rd
     ,tblfmt_iua_heat_result
   }

  ,{IUA_HEAT,"IUA_HEAT","(IUA) High Explosive Anti-Tank (HEAT)"
     ,tblfmt_iua_heat_rd
     ,tblfmt_iua_heat_result
   }

  ,{IUA_KE,"IUA_KE","(IUA) Kinetic Energy (KE)"
     ,tblfmt_iua_ke_rd
     ,tblfmt_iua_ke_result
   }
  /*
   * top attack ("Staff munition"):
   */
  ,{IUA_STAFF,"IUA_STAFF"
     ,"(IUA) STAFF Explosively Formed  Penatrator (EFP)"
     ,tblfmt_iua_staff_rd
     ,tblfmt_iua_staff_result
   }
```

```
129
130         ,{  PKS_BINARY_COLLISION,  "PKS_BINARY_COLLISION"
131         ,"(P-K)  For  Collisions  returns  BINARY  V/L  Methodology"
132         ,tblfmt_binary_collision_rd
133         ,tblfmt_binary_collision_result
134       }
135
136
137     /*-------------------------------------------------------------------*/
138     /* DO NOT ADD BELOW THIS LINE                                        */
139     /*-------------------------------------------------------------------*/
140      ,{_END_OF_TblFmt_Enum, NULL, NULL, NULL, NULL }
141
142   };
143   # endif /* ifndef TBL_FMTS_C */
144
145
146   VL_Result db_tbl_fmt_result_type(char *fmtname); /* data type returned by lookup*/
147
148   #endif /* ifndef _TBL_FMTS */
```

Figure 6. "tbl_fmts.h" Used for Data V/L Data Reading and Initialization.

external meta records that reference collision damage data returned in a format consistent with the BINARY vulnerability methodology. Line 131 describes the vulnerability data type and format in human terms and has no logical programming value (but is used in print statements). Lines 132 and 133 are the names of the vulnerability data initialization and reader functions, respectively.[4]

When a meta data record is read, the data format's external representation appears in the "format" field of the record ("**PKS_BINARY_COLLISION**" in the case of the collision damage described in terms of our BINARY methodology). For example, the meta data records shown in the **vls_db_init(5)** manual are repeated in Figure 7. On the last line of Figure 7, we see that "IUA_HEAT" is in the "format" field. This tells the server to use the record shown on lines 112 through 115 (of Figure 6) to determine which data initialization and data reader function to use. When the data initialization function is called, the last field of the meta data record is passed to it as an argument. The last line of Figure 7 shows that "`file:/Data/Tables/IUA/smplHEAT.iua`" is the argument that is passed to the initializing function under the conditions set forth by the record's target, threat, and vulnerability method as dictated on that line.[5] The initializer function must return a pointer. Later, when a lethality query is made, that same pointer will be available for use by the look-up (or reader) function. The lethality server maintains this pointer and provides it when

[4](This naming convention might be confusing since "**tblfmt_binary_collision_rd**" is not the reader function but the *initializer* function. (The "_rd" convention originated because, thus far, the server has only been used for static look-up tables; hence, the duty of the initialization function was to read [ergo, "_rd"] the static data into memory.) The duty of the second function (our current "reader" function) was to parse the static table (now in memory) and return the correct vulnerability results (ergo, the "_result" convention).

[5]Namely, the conditions are when a "T-80" tank is attacked by an "AT-5 Spandrel missile" and evaluated using the "DIS HitToKill" vulnerability methodology.

needed by the reader (or table look-up) function (i.e., when a vulnerability analysis query is made for the very same target, threat, and vulnerability method). This pointer could point to anything as long as the data reader (or table look-up) function is able to use the data set (referenced by the pointer) in such a way as to allow the function to return the correct lethality results (for the given vulnerability initial conditions[6]).

```
#
#
# DIS enumerations are IEEE 1278.1-1995 Standard.
# Note that the file URL location is taken relative
# to the $VLS_HOME directory.
#--next line's tgt and threat are: Soviet 125mm KE Threat VS. a T-80 target.
1 1 222 1 1 1, 2 2 222 2 11,"DIS HitToKill","IUA_KE","file:/Data/Tables/IUA/smplKE.iua"
#--next line's tgt and threat are: Soviet 120mm HEAT-FS VS. a T-80 tgt.
1 1 222 1 1 1, 2 2 222 2 18,"DIS HitToKill","IUA_HEAT","file:/Data/Tables/IUA/smplHEAT.iua"
#--next line's tgt and threat are: AT-5 Spandrel missile VS. a T-80 tgt.
1 1 222 1 1 1, 2 2 222 1 7,  "DIS HitToKill","IUA_HEAT","file:/Data/Tables/IUA/smplHEAT.iua"
```

Figure 7. Example of Records for a Meta Data File.

As mentioned, the lethality server architecture is designed to allow these functions to return any type of data. Thus far, however, they have only been used to initialize (and look up the results) of static look-up tables. This has been implemented by having the initialization file look for (and read) the data file referenced in the meta data record (the URL address in the last field of a meta data record). The last line of Figure 7 shows "`file:/Data/Tables/IUA/smplHEAT.iua`" as this data file for that meta data record. The initialization function loads this static data table into a data structure and returns a pointer to the memory location of that data structure. The result (or table look-up) function knows how to parse this data structure. When the result function is called, it receives a pointer to this data structure and proceeds to parse it and returns the correct results. Figure 8 displays our two new initializer and reader functions added (in bold text).

[6]Recall that those initial conditions are provided by the **VLParam sub-layer.**

```
1   /* $Id: tbl_rdrs.h,v 0.4 1998/03/23 04:20:48 geoffs Exp geoffs $ */
2
3   extern void  *tblfmt_iua_ke_rd(FILE *in_fp);
4   extern void  *tblfmt_iua_ke_result( void *);
5
6   extern void  *tblfmt_iua_he_rd(FILE * fp);
7   extern void  *tblfmt_iua_he_result(void *);
8
9   extern void  *tblfmt_iua_heat_rd(FILE * fp);
10  extern void  *tblfmt_iua_heat_result(void *);
11
12  extern void  *tblfmt_iua_staff_rd(FILE *fp);
13  extern void  *tblfmt_iua_staff_result(void *);
14
15  extern void    *tblfmt_binary_collision_rd(FILE  *fp);
16  extern void    *tblfmt_binary_collision_result(void  *);
```

Figure 8. Prototypes of Data Source Initialization and Reader Functions (in "tbl_rdrs.h").

7.2.3. *Adding Source Code for New "Table Look-up" Function*

The source code for these two functions is not included in this text since details of how they are implemented are not important (as long as the initialization function initializes the data set in some manner and the result function is able to use that initialized data set to return the correct lethality result). However, it is recommended that the source code for data initialization and reader functions be placed in the directory $VLS_HOME/src/TblReaders and incorporated into the directory's "makefile". It is required that the prototype for the reader and initialization file be placed in the header file $VLS_HOME/src/TblReaders/tbl_rdrs.h (shown in Figure 8). This is mandatory since "tbl_fmts.h" (Figure 6) requires the prototyped function names before they may be included into the `LookUp_Tbls[]` array (Figure 6, lines 94 through 142). The "tbl_rdrs.h" file is shown in Figure 8 with our two new initializer and reader functions added (in bold text).

7.2.4. *The Final Step in Adding New Table Look-up Functions (recompiling the server)*

In order for these changes to take place, the server must be recompiled. This may be accomplished by executing the "compile.sh" shell script outlined in Section 3.2. (This assumes that the "makefile" in the $VLS_HOME/src/TblReaders directory has been modified to incorporate the two new functions.) Following a successful recompilation, the server will be equipped to handle V/L API queries for the newly added vulnerability methodology and data source format.

Note, however, that in order to make such queries, an application program must be linked directly with the server V/L library (i.e., a direct function call must be made to the server API

functions). We still have not provided a method for a remote client application to make queries (for the newly created vulnerability methodology). To see how this is done, we examine our final code modification section, Section 7.3.

7.3 Adding Remote Access for a New Vulnerability Methodology

In Figure 1, it is seen that the **DIS Monitor** is an application that directly calls functions in the **VL API** layer (`vl_binary_ArlDIS_ProbAll_NoNet()` that we defined in Figure 5 would be one such function). This section shows how a remote client (the client of Figure 1) is able to have indirect access to the results from the same API. The steps are as follow:

1. Have the **DIS Monitor** monitor the virtual environment for important parameters.
2. Select a protocol syntax between the **client** and **DIS Server.**
3. Enable the DIS monitor to call the newly created **VL APIs** (to respond to client queries).

We cover these three steps in the next three sections.

7.3.1 *Expanding the Environmental Monitoring Capability of the **DIS Monitor***

Because the server currently implements just the "MFK" vulnerability methodology for munition type damage, the DIS monitor only monitors parameters required by that set of APIs. (This means that the DIS monitor monitors entity state, fire, and detonation PDUs because these are the only PDUs required by the V/L APIs to complete the "MFK" analysis for munition damage.) However, in Sections 7.1 and 7.2, we have provided for a new vulnerability methodology (BINARY) as a result of collision damage. APIs for this new methodology will require an additional argument (the collision PDU). (Note that on line 71 of Figure 5 the collision PDU is required to set the initial condition parameters for collision damage.)

The DIS monitor (see **dis_mon(1)** in Appendix B) watches DIS PDU traffic and maintains records of PDUs that are of interest to it. The PDUs it finds interesting are those that are needed for providing initial conditions for a vulnerability assessment. For instance, the API "vl_mfkDIS_ProbAll()" needs four PDUs: the entity state PDU for the target, entity state for the firer, the fire PDU, and detonation PDU (see **vl(3)** in Appendix B). The DIS monitor then listens to PDU traffic, and whenever someone fires (via a fire PDU) or a munition detonates (via the detonation PDU), it keeps a copy of that PDU, along with an entity state PDU of whoever did the firing and whoever was targeted at the time (if known). The DIS monitor is then able to call the API "vl_mfkDIS_ProbAll()" and provide all the required parameters. It would then

generate results for the DIS server module via shared memory (see Figure 1 and **vlserver(1)** in Appendix B). This same procedure should be followed for newly added APIs (such as the APIs for our collision damage/BINARY methodology). We must modify the DIS monitor to monitor the DIS network traffic (PDUs). It will have to keep records of parameters that could be used in a query. It will then be ready to call V/L APIs when required.

Source code for the DIS monitor is given in the $VLS_HOME/src/DisMon/ directory. The appropriate place to start modifying **dis_mon(1)** to listen for new parameters is in the file "process_pdu.c", specifically in the function `process_pdu_do()` which is reproduced in Figure 9.

The portion of the code in bold has been added in order to keep records of collision PDUs. The source code for the function `process_pdu_coll()` that is called on line 60 is left as an exercise for the student. All that `process_pdu_coll()` has to do is store the collision PDU in a data structure so that it can be retrieved for later use.

Unfortunately, this section of code will never be activated! This is because the DIS manager (see **dis_mgr(1)** in Appendix B) is excluding all PDUs except those that we have stated an interest in receiving (and we have not yet told the dis_mgr that we are interested in receiving collision PDUs). To start receiving collision PDUs, we modify the DIS monitor function "connect_to_dis_mgr()" (shown in Figure 10).

This section of the DIS monitor references the DIS manager library calls "dis_open()" and "dis_register_pdu()." The latter call is where we need to add a provision for the collision PDUs that we want to start monitoring. Lines 20 and 24 of Figure 10 show where we have provided for collision PDUs (shown in bold text). Now when `dis_register_pdu()` is called on line 27, all the PDU types seen (on lines 21 through 24) will be added to the list of PDU types that we are registering with the DIS manager (i.e., the list of PDUs we want "see"; all other PDUs will be excluded).

The DIS monitor is now able to monitor the virtual environment for parameters that are important to initializing a collision damage analysis. It is also maintaining a record of those parameters for later use in vulnerability calculations. We now turn our attention to how remote clients may access a new damage type by querying the server.

```
/*
 ~ process_pdu_do()
 *
 * int process_pdu_do( int indx, PDU_Type type, char * pdu)
 *
 *  handle a pdu ( based on which type of pdu it is that we are
 *  handling)
 *
 *  return TRUE if the pdu is to be freed (discarded).
 *       else return FALSE (if the pdu is being held somewhere).
 *
 */
int process_pdu_do( int indx, PDU_Type type, char * pdu)
{  char tmp[256];
   int free_this_pdu;
   static char *whoami="process_pdu_do()";
   extern char *Dis_Pdu_Names[];

   free_this_pdu = TRUE;

   switch (type) {
         case DL_NO_DATA:
            /* buffer empty */
            break;
         case EntityStatePDU_Type:
        /*
         * since we have already handle ES in process_pdu()
         *  there is nothing to do now.
         */
          free_this_pdu = process_pdu_es( (EntityStatePDU *) pdu );.
          break;
         case FirePDU_Type:
        /*
         * allocate a location in the fire/detonation event list.
         *  (this list will contain the most recent
         *   EntityState PDUs for the firer,
         *   Target (if one) and for the detonation.
         *   These PDUs will remain stored and will not be
         *     freed.
         */
        free_this_pdu = process_pdu_fir( (FirePDU *) pdu );
          break;
          case DetonationPDU_Type:
        /*
         * add the detonation pdu into the fire/detonation event
         * list.
         */
          free_this_pdu = process_pdu_det( (DetonationPDU *) pdu );
          break;

            case CollisionPDU_Type:
               /*
               * We saw a Collision PDU.   Keep it some where
               * for later use as an argument to a VL API call.
               *
               * process_pdu_coll(); is a function that would store all
               *                        collision pdu's in some data structure
               *                      for later retrieval.
               */
               process_pdu_coll( (CollisionPDU *) pdu );

                 free_this_pdu = FALSE; /* false since we need to keep a
                                         * copy of this pdu
```

```
                                                    */
            break;

          default:
           free_this_pdu = TRUE;
             sprintf(tmp
         ,"%s:saw a pdu_type %d (%s)...do not know how to handle it.\n"
            ,whoami
                 ,type
            ,Dis_Pdu_Names[type]
                      );
         fputs(tmp,stderr);
              break;
    }

    return  free_this_pdu;
}
```

Figure 9. Modifications of DIS Monitor to Listen for New PDU Types.

```
/*
 ~ connect_to_dis_mgr()
 *
 * Establish a connection with the server on the specified machine.
 */
int connect_to_dis_mgr(char *host)
{char  pdu_list[256];

      printf("Connecting to DIS manager on %s...\n",host);fflush(stdout);
        /*
         * Open a connection to the dis_mgr server on machine host.
         */
        if (dis_open(host) == FALSE) {
                return(-1);   /* connection failure! exit */
      } else {
                printf("Successful!\n");
                /*
                 * Register interest in PDUs.
                 */
                sprintf(pdu_list," %d %d %d %d "
                           ,EntityStatePDU_Type
                           ,FirePDU_Type
                           ,DetonationPDU_Type
                             ,CollisionPDU_Type
                 );
                printf("Sending : %s\n",pdu_list);
                dis_register_pdu(pdu_list);
                return(0);
       }
}
```

Figure 10. Removing Collision From DIS Manager's PDU "Filtering".

7.3.2 *Establishing a Protocol Between the **Client** and **DIS Server***

Figure 1 displays remote client applications communicating with the DIS server. The **vlserver(1)** manual page in Appendix B explains the syntax for "MFK" queries. We will now add the capability to make "BINARY" queries for collision damage. In the syntax of client-

server protocol established in the **vlserver(1)** manual page, we shall add new "QUERY" types. Each query type will correspond to one of the new V/L APIs we listed in Table 4 (presumably, we have already written all these APIs and added them to the V/L library). Figure 11 displays "vls_toke.h" where we add new key words for the client-server simple query language. This file and other **vlserver** source files are located in the $VLS_HOME/src/Server directory.

Changes made in vls_toke.h are shown in bold print. We added tokens to the `_VLS_Token` enumeration type for internal use, and corresponding ASCII strings to the `VLS_TokenString` array for parsing an external query. Clients send query tokens in the form of an ASCII string; these strings are then "tokenized" (the ASCII key words are converted to an internal numerical "token" representation) by a simple parser in the vls_toke.h file. Here, we added new query types (on lines 32 through 35 and 84 through 87 of Figure 11) to tell the server we wish to query for collision damage and receive the answer in a BINARY vulnerability format. On lines 49 and 102, we are accommodating arguments needed to complete this query. Specifically, when a query for collision damage is made, the querying client shall reference a number that identifies which collision event is to be evaluated. (DIS provides a unique identifier for each collision event on the virtual battlefield.) We next modify the behavior of the DIS server (vlserver) to respond to the tokens we just defined. To do this, we add to the function service_query_to_db() in the source code file $VLS_HOME/src/Server/vlserver.c as shown in bold text in Figure 12.

The bold print text in Figure 12 was added to `service_query_to_db()` to allow the server to understand and service the collision damage query. The query would be formatted by the client in a manner similar to the ASCII string that is shown in Figure 13. (See the **vlserver(1)** in Appendix B for how to format queries.)

The code segment shown in Figure 12 begins processing this query just after reading the key word "QUERY". On lines 50 and 79, `vls_tokenize()` transforms the type of query (`"TYPE_binaryCOLLISION_ProbAll"`) and the arguments identifier (`"ARGS_DIS_COLLISION_IDS"`) into their equivalent tokens. On line 97, the tokenized argument identifier is used to drive a *switch* statement. Since the tokenized value is equal to the enumeration `T_ARGS_DIS_COLLISION_IDS`, the section of code from lines 148 through 192 is executed. There we see that six integers are scanned. The first three represent the IDs of the target (or subject of our vulnerability analysis); the second three integers ("4 5 6" in Figure 13) are the unique collision event ID. On lines 179 through 181, these arguments and the token that identifies the type of query being made are placed in the shared memory link between the DIS server (vlserver) and the DIS monitor (dis_mon). (The manual page

```
/* $Id: vls_toke.h,v 0.20 1998/08/09 21:11:10 geoffs Exp geoffs $ */
#ifndef _TOKENS_H_
#define _TOKENS_H_
/*----------------------------start tiny vls_tokenizer----------------*/
enum _VLS_Token {
        T_ERROR
        ,_T_START_OF_TOKENS

      ,T_VLS_ECHO
      ,T_HELP                 /* ask for help */
      ,T_HELP1
      ,T_INFO_SERVER          /* get admin info */
         ,T_VLS_QUERY_SHMEM_ID      /* ask for shared memory ID */
         ,T_VLS_QUERY_PARSER_VER
         ,T_VLS_QUERY_PARSER_VERSION
         ,T_VLS_QUERY_DIS_VERSION

      ,T_VLS_QUERY_TYPE               /* expect the type of query
                                * to follow this vls_token
                                */

      ,__T_START_OF_T_QTYPE_TOKENS

      ,T_QTYPE_mfkDIS_Result         /* Requested Format of Answer  */
      ,T_QTYPE_mfkDIS_ProbAll
      ,T_QTYPE_mfkDIS_ProbK
      ,T_QTYPE_mfkDIS_ProbMF
      ,T_QTYPE_mfkDIS_ProbF
      ,T_QTYPE_mfkDIS_ProbM
      ,T_QTYPE_mfkDIS_ProbNoDamage

      ,T_QTYPE_binaryCOLLISION_Result              /*BINARY vulnerability method*/
      ,T_QTYPE_binaryCOLLISION_ProbAll            /*for damage from collision*/
      ,T_QTYPE_binaryCOLLISION_ProbALIVE
      ,T_QTYPE_binaryCOLLISION_ProbDEAD

      ,__T_END_OF_T_QTYPE_TOKENS

      ,__T_START_OF_T_QARGS_TOKENS

      ,T_VLS_QUERY_TYPE_MFK_BINARY_PDUS /* expect binary pdu args */
      ,T_VLS_QUERY_TYPE_MFK_DIS_IDS    /* expect ID args - implies
                                *   we have to get the
                                *  applicable pdus elsewhere
                                * (such as from shared
                                * memory)
                                */

      ,T_ARGS_DIS_COLLISION_IDS                    /* expect some ID args */

      ,__T_END_OF_T_QARGS_TOKENS


        ,_T_END_OF_TOKENS
};

typedef enum _VLS_Token VLS_Token;

static char *VLS_TokenString[] = {
        "<*ERROR NOT A vls_token*>"
        ,NULL
      ,"ECHO"
      ,"HELP"
```

```
64          ,"?"
65          ,"INFO_SERVER"
66              ,"SHMID"
67              ,"VER"
68              ,"VERSION"
69              ,"DIS_VERSION"
70
71          ,"QUERY"
72
73          , "__T START OF T QTYPE TOKENS"/* start of query types-Not a vls_token
74      */
75
76          ,"TYPE_mfkDIS_Result"
77          ,"TYPE_mfkDIS_ProbAll"
78          ,"TYPE_mfkDIS_ProbK"
79          ,"TYPE_mfkDIS_ProbMF"
80          ,"TYPE_mfkDIS_ProbF"
81          ,"TYPE_mfkDIS_ProbM"
82          ,"TYPE_mfkDIS_ProbNoDamage"
83
84          ,"TYPE_binaryCOLLISION_Result"          /*BINARY vulnerability method */
85          ,"TYPE_binaryCOLLISION_ProbAll"         /*for damage from collision*/
86          ,"TYPE_binaryCOLLISION_ProbALIVE"
87          ,"TYPE_binaryCOLLISION_ProbDEAD"
88
89          , "__T END OF T QTYPE TOKENS"/* end of query types-Not a vls_token */
90
91          ,"__T START OF T QARGS TOKENS" /*start of argument types */
92
93          ,"ARGS_mfkDIS_PDUS"             /* expect binary pdu args */
94          ,"ARGS_mfkDIS_IDS"              /* expect ID args - implies
95                                      *   we have to get the
96                                      *  applicable pdus elsewhere
97                                      * (such as from shared
98                                      * memory) - not implemented.
99                                      * Tue Oct 14 15:02:14 EDT 1997
100                                     */
101
102         ,"ARGS_DIS_COLLISION_IDS"                       /* expect some ID args */
103
104         ,"__T END OF T QARGS TOKENS" /*end of arg types - not a vls_token */
105
106           ,NULL
107     }; /*  the rest of vls_toke.h not shown...*/
```

Figure 11. Defining Client-Server Protocol (adding tokens to vls_toke.h).

mk_shm(3) in Appendix B describes the shared memory link between these two applications.) On line 182, a flag is set to inform the DIS monitor (**dis_mon(1)**) that the vlserver has placed a query in the shared memory link (and the vlserver is waiting for the answer to be returned). Vlserver then enters a loop (between lines 216 and 227) waiting for dis_mon to return the result of the vulnerability analysis. If an answer was successfully returned by dis_mon, then on line 278 vlserver passes that answer to the client who requested it in the first place. There is only one problem with all of this. The DIS monitor does not yet know how to respond to this query type from the DIS server. In the next section, we explain how dis_mon is modified to accomplish this task.

```
/*
 ~ service_query_to_db()
 *
 * static void service_query_to_db(pc,query_id,rest_of_query)
 *
 *   Service a QUERY type command from the client:
 *
 *    1. Grab rest of query command arguments.
 *    2. Place query and arguments into shared memory and set
 *        shared memory flag to let DisMonitor know
 *        that there is a query pending to be answered by the DisMonitor.
 *    3. Wait for DisMonitor to answer the query or be timed out.
 *    4. Return results to client.
 *
 */
void service_query_to_db(pc,query_id,rest_of_query)
struct pkg_conn       *pc;
int query_id;
char *rest_of_query;
{  int error, ch;
   register char *ptr, *eot;
   VLS_Token t, toke_query_type, toke_args_type;

   char *str_query_type, *str_args_type, *str_args_type_eot;
   /*    str_query_type   str_args_type   str_args_type_eot
    *    are used for making user-friendly error messages.
    */
   int query_placed, int_args_matched;
   char error_msg_buff[1024];
   char buf[1024];

   error = 1;
   query_placed = FALSE;
   int_args_matched = FALSE;

   if (pc!=NULL && rest_of_query != NULL) {

      ptr = rest_of_query;

     /*
      * get next vls_token of command. - query type.
      */
      ptr = sscan_skip_white(ptr);/* skip white space characters */
      eot = sscan_next_white(ptr);
      if (eot != NULL) {
          ch = *eot;
          *eot='\0';
      }

      toke_query_type = vls_tokenize( ptr );

      if (FALSE == vls_token_is_query_type( toke_query_type ) ) {
         ++error;
         str_query_type=ptr;
         sprintf(error_msg_buff,"sytax error. unknown query type seen :%30s "
                ,str_query_type);
         goto out; /* sytax error */
      }

      /*
       * restore rest_of_query for scaning
       */
      if (ch != 0 ) {
```

```
64              *eot = ch;
65          }
66          ptr = eot;
67
68          /*
69           * get next vls_token of command.  - args type.
70           */
71          ptr = sscan_skip_white(ptr);
72             str_args_type = ptr;
73          eot = sscan_next_white(ptr);
74             str_args_type_eot = eot;
75          if (eot != NULL) {
76              ch = *eot;
77              *eot='\0';
78          }
79          toke_args_type = vls_tokenize( ptr );
80
81          if (FALSE == vls_token_is_arg_type( toke_args_type ) ) {
82             ++error;
83           sprintf(error_msg_buff
84              ,"snytax error. unknown argument(s) identifier seen: %30s"
85              , str_args_type );
86             goto out; /* sytax error */
87          }
88
89          /*
90           * restore rest_of_query for scaning
91           */
92          if (ch != 0 ) {
93              *eot = ch;
94          }
95          ptr = eot;
96
97          switch ( toke_args_type ) {
98            int tgt_id[3],event_id[3], collision_id[3];
99
100         case T_VLS_QUERY_TYPE_MFK_DIS_IDS:
101          /* scan Tgt and Event ID (2 sets of (3 ints) )*/
102          if (Verbose)
103            printf("***scanner sees Tgt and Event: %s\n",ptr);
104
105          if ( 6 != sscanf(ptr," %d %d %d  %d %d %d ",
106                        &tgt_id[0],  &tgt_id[1],  &tgt_id[2],
107                        &event_id[0], &event_id[1], &event_id[2]
108                )
109            ) {
110           int_args_matched = FALSE;
111           ++error;
112
113           ch = *str_args_type_eot;
114           *str_args_type_eot='\0';
115
116                sprintf(error_msg_buff
117                 ,"sytax error. expected 6 integers to follow \"%s\""
118                 ,str_args_type
119                );
120           *str_args_type_eot=ch;
121
122           break; /* syntax error expected 6 ints */
123          } else {
124           /* set shared memory  */
125           int_args_matched = TRUE;
126           if (Verbose) {
127             printf("***puting to shm Tgt:%d %d %d ",
```

```
                tgt_id[0],  tgt_id[1],  tgt_id[2]);
      printf("***puting to shm Event:%d %d %d ",
                event_id[0], event_id[1],event_id[2]);
    }

    (void) shmSet_TargetID( tgt_id );
    (void) shmSet_EventID( event_id );
    (void) shmSet_QueryArgsType( toke_args_type );
    (void) shmSet_QueryType( toke_query_type );
    (void) shmClear_QueryAnswered();
    if ( 1 == shmSet_QueryPlaced() ) {
        query_placed = TRUE;
    } else {
        ++error;
        sprintf(error_msg_buff,"could not set share memory!");
        ; /* error could not set share memory */
    }
   }
   break;

  case T_ARGS_DIS_COLLISION_IDS:              /* expect some ID args */
   /*
    * We just saw "ARGS_DIS_COLLISION_IDS" in the query statement.
    * Following this we expect to see three integers
    * that together are the collision event ID
    */
  if ( 6 != sscanf(ptr," %d %d %d %d %d %d ",
                &tgt_id[0],          &tgt_id[1],          &tgt_id[2],
                &collision_id[0],    &collision_id[1],    &collision_id[2],
              );
      ) {
          int_args_matched  =  FALSE;
          ++error;

          ch  =  *str_args_type_eot;
          *str_args_type_eot='\0';

            sprintf(error_msg_buff
                ,"sytax error. expected 6 integers to follow \"%s\""
                ,str_args_type
                 );
          *str_args_type_eot=ch;

          break;       /* syntax error expected 6 ints */
   } else {
          /* set shared memory */
          int_args_matched  =  TRUE;
          if  (Verbose)  {
             printf("****puting to shm collision ID:%d %d %d ",
                collision_id[0],  collision_id[1],  collision_id[2]);
          }
          (void)  shmSet_TargetID(  tgt_id  );
          (void)  shmSet_EventID(  collision_id  );
          (void)  shmSet_QueryType(  toke_query_type  );
          (void)   shmClear_QueryAnswered();
          if  (  1  ==  shmSet_QueryPlaced()  )  {
               query_placed  =  TRUE;
          }  else  {
               ++error;
                sprintf(error_msg_buff,"could not set share memory!");
               ; /* error could not set share memory */
          }
   }
```

```
192          break;
193
194        case T_VLS_QUERY_TYPE_MFK_BINARY_PDUS: /* not implemented yet */
195          default:
196           ++error;
197           sprintf(error_msg_buff,"unsupported query type");
198           break;
199        }
200
201        if (query_placed == TRUE) {
202        static struct timeval timeout;
203        int polls;
204          int answered;
205        /*
206         * query is placed,
207         *  Now wait for DisMonitor to answer the query.
208         */
209        polls = 0;
210
211          Db_TimedOut_Clear(); /* set Db_Timedout==FALSE and start timer */
212                       /*
213                       * when timer goes off, then
214                       *   Db_Timedout is set to TRUE;
215                       */
216        while(  (1!=shmGet_QueryAnswered()) &&  (FALSE==Is_Db_TimedOut())  ) {
217              /*
218               * sleep a short time
219               */
220              timeout.tv_sec=0;
221              timeout.tv_usec= SERVER_DB_POLLING ;  /* sleep for
222                                                    * SERVER_DB_POLLING
223                                                    * micro seconds
224                                                    */
225              select(NullFile_fd , (fd_set *)NULL, (fd_set *)NULL,
226               (fd_set *)NULL,  &timeout);
227        }
228        answered=shmGet_QueryAnswered();
229
230          if (Verbose) {
231            if (answered==1) {
232              printf("server query answered after about %7.3f seconds\n"
233          ,    ((double) (polls*SERVER_DB_POLLING) )/ 1000.);
234            } else {
235              printf(
236               "server query NOT answered after %7.3f seconds (and %d polls).\n"
237          ,    ((double) (DB_TIMEOUT))/1.e+06
238          ,    polls);
239            }
240        }
241
242        if (answered) {
243          /*
244           *  Read answer from shared memory.
245           */
246
247           error = FALSE;            /* Success - finally */
248
249        } else {
250           ++error; /* ERROR! query timeout! */
251           sprintf(error_msg_buff,
252              "Timed-out waiting for VL DataManager response.");
253         }
254         } else if (FALSE == shmIsAttached()) {
255          ++error;
```

```
256          sprintf(error_msg_buff,
257          "VL server internal error: could not place query into shared memory!");
258        } else if (FALSE == int_args_matched)   {
259          ; /* leave error message as is - it describes #of ints expected */
260        } else {        /* UNKNOW ERROR - hopefully never will get here */
261          ++error;
262          sprintf(error_msg_buff,
263               "VL server internal error: could not propperly process query!");
264        }
265     }
266
267  out:
268
269     if (error) {
270          int len;
271        char msg[128];
272        if (Verbose)
273               printf("Query not understood from client %d\n",pc->pkc_fd);
274        sprintf(msg,"%d: VLS ERROR. %s",query_id,error_msg_buff);
275        len=strlen(msg)+1; /* add 1 to also send the NULL terminator*/
276          (void) pkg_send(VL_MSG_TO_CLIENT,msg,len,pc);
277     } else {
278        if (  TRUE != send_query_answer(pc,query_id,toke_query_type)) {
279          /* error could not read shared memory
280           * or else could not send client the answer...
281           * But is almost certainly is the later,
282           * since we already tested for shm writing
283           * when query_placed was set to true.
284           */
285              printf("*** server could not send to client %d (query %d)!!!\n"
286               ,pc->pkc_fd,query_id);
287        }
288     }
289  }
```

Figure 12. Enabling vlserver to Parse a New Query Type (service_query_to_db()).

```
"123 QUERY TYPE_binaryCOLLISION_ProbAll ARGS_DIS_COLLISION_IDS 1 2 3 4 5 6 "
```

Figure 13. Sample ASCII Query String (sent to the vlserver).

7.3.3. *Remote Access to New VL APIs (responding to client queries)*

In addition to monitoring the DIS environment (and storing certain PDUs for later use), the DIS monitor also periodically monitors the shared memory link (**mk_shm(3)** in Appendix B) for new queries placed by the vlserver. When it discovers that a query is pending, dis_mon uses the function `vls_link_service_query()` to service the query. Changes in `vls_link_service_query()` that address collision damage queries (using the BINARY vulnerability methodology) are shown in bold text in Figure 14. This function and other DIS monitor source code is located in the $VLS_HOME/src/DisMon directory.

```
1  /*
2   ~ vls_link_service_query()
3   *
4   * int vls_link_service_query(void);
5   *
6   *  Extract the VL server query and attempt to service it.
7   *  If serviced the query result is place in shared memory.
8   *  (A successful service includes placing indicators into
9   *   shared memory which tell the server to return an error
10  *   message to the client.)
11  *
12  * returns 1 (TRUE) if success (in placing the query into shared memory).
13  *         0 (FALSE) if unsuccess.
14  *
15  */
16 int vls_link_service_query(void)
17 {  VLS_Token ans_t, args_t;
18    int ret;
19    static char *whoami="vls_link_service_query()";
20
21    ret = FALSE;
22
23    _rpt_error(_RE_CLEAR_ERRORS,NULL);   /* clear error flags in rpt_perror*/
24    args_t =  shmGet_QueryArgsType();
25    ans_t  =  shmGet_QueryType();              /* called by DisMonitor */
26
27    switch ( args_t ) {
28        case T_VLS_QUERY_TYPE_MFK_DIS_IDS:
29           ret = vls_link_serve_mfkDIS_IDS(ans_t);
30           break;
31           case T_ARGS_DIS_COLLISION_IDS:
32                ret = vls_link_serve_binaryCollision_DIS_IDS(ans_t);
33             break;
34        default:
35           cprint(CH_ERR,"%s: cannot handle \"%s\" (%d) arguments in query\n"
36                      ,whoami
37                      ,vls_token_name( args_t )
38                      ,args_t
39                );
40           break;
41    }
42
43     shmClear_QueryPlaced();       /* clear query pending flag */
44     shmSet_QueryAnswered();       /* set   query answered flag */
45
46    return(ret);
47 }
```

Figure 14. Change (in **dis_mon**) to Accept New Queries.

Figure 14 shows on line 32 that the type of query is passed (via the variable `ans_t`) to a function that calls the appropriate VL API routines and returns the results. That function "`vls_link_serve_binaryCollision_DIS_IDS()`" is shown in Figure 15. This whole function would have to be written since it does not yet exist.

```
/*
 ~ vls_link_serve_binaryCollision_DIS_IDS()
 *
 *  static int vls_link_serve_binaryCollision_DIS_IDS( VLS_Token ans_t )
 *
 *  Service a query for a collision damage using BINARY methodology.
 * return 1 (TRUE) if successful in placing the query into shared  memory.
 *        0 (FALSE) if not.
 */
static int vls_link_serve_binaryCollision_DIS_IDS( VLS_Token ans_t )
{  DisID entityID[3], eventID[3];
   int i, flg, ret, set_err_msg, int3[3];
   VL_Result result;
   float *probspace;
   double prob;
   static const char *whoami="vls_link_serve_binaryCollision_DIS_IDS()";
   static char answer_buff[128];
   VLSetParam_t Dmg_Type;

   set_err_msg = FALSE;
   ret = FALSE;

   /* get entity and event ID's */
   (void) shmGet_TargetID( int3 );
   for (i=0; i<3; i++)
       entityID[i] = (DisID) int3[i]; /* int16 = int */
   (void) shmGet_EventID( int3 );
   for (i=0; i<3; i++)
       eventID[i] = (DisID) int3[i];
   printf("%s:IDs are: %d %d %d %d %d %d\n", whoami
       ,entityID[0]
       ,entityID[1]
       ,entityID[2]
       ,eventID[0]
       ,eventID[1]
       ,eventID[2]
         );
    Dmg_Type = VL_PARAM_SET_COLLISION;
    switch(ans_t) {
       case T_QTYPE_binaryCOLLISION_Result:  /* Requested Format of Answer  */
            result = vl_binary_DIS_Result(
                                       &flg, DmgType, entityID ,eventID);
            if (flg!=0)       /* result not from found table */
              set_err_msg = TRUE;
            ret = shmSet_VLResult(result,flg);  /* called by DisMonitor */
            break;
       case T_QTYPE_binaryCOLLISION_ProbAll:
            probspace = vl_binary_DIS_ProbAll(Dmg_Type, entityID, eventID );
            if (probspace==NULL)       /* a type of error */
              set_err_msg = TRUE;
            ret = shmSet_binaryPS( probspace );    /* called by DisMonitor */
            break;
       case T_QTYPE_binaryCOLLISION_ProbDEAD:
            prob = _vl_binary_DIS_ProbDEAD( Dmg_Type, entityID, eventID );
            goto set_prob;
            break;
       case T_QTYPE_binaryCOLLISION_ProbALIVE:
            prob = _vl_binary_DIS_ProbALIVE( Dmg_Type, entityID, eventID );
            goto set_prob;
            break;
set_prob:
            if (prob<0.)       /* a type of error */
              set_err_msg = TRUE;
```

```
64              ret = shmSet_prob(prob);     /* called by DisMonitor */
65              break;
66          default:
67              ret = FALSE;
68              cprint(CH_ERR,"%s: cannot handle %s vl result in query\n"
69                        ,whoami
70                        ,vls_token_name( ans_t ));
71             sprintf(answer_buff,"**DB Handler ERROR");
72             break;
73      }
74
75      if (set_err_msg == TRUE) {
76         char *b, buff[256];
77
78         /* Place error msg in shm (to be sent by server to client) */
79         /*-----------------------------------------------
80          *   this is enough info:
81          *      shmSet_ErrorMsg( rpt_error_getMsg() );
82          *-----------------------------------------------
83          *    but the following is even more info:
84          */
85         memset(buff,0,sizeof(buff));
86         strncpy(buff,rpt_error_getMsg(), sizeof(buff) );
87         buff[sizeof(buff)-1]='\0';
88         /*
89          * See if there is more to report that sheds light on the error
90          *   (that is more than the standard error msg).
91          */
92         if (NULL != (b= _rpt_error_getLastAddedMsg() ) ) {
93                int len;
94                len = strlen(buff);
95
96                if (len+3 < sizeof(buff) ) { /* add ". " */
97                   strcpy(buff+len,". ");
98                   len+=2;
99                }
100               strncpy(buff+len,b,sizeof(buff)-len); /* add extra msg*/
101               buff[sizeof(buff)-len-1 ]='\0';
102        }
103        shmSet_ErrorMsg( buff );
104     }
105     return (ret);
106 }
```

Figure 15. A Function to Process Collision Damage Queries (using BINARY method).

The first thing to notice about "`vls_link_serve_binaryCollision_DIS_IDS()`" is that it derives the arguments from the shared memory link between dis_mon and the vlserver. This is seen on lines 24 and 27 where the ID of the entity whose damage we are assessing and the ID of the collision event that causes the damage are retrieved. The *switch* statement on line 39 is used direct the program to call whichever V/L API will return the answer in a format that appropriately reflects the client's query. If you have a keen eye, then you may have noticed that the function "`shmSet_binaryPS()`" called on line 50 does not yet exist (and was not listed under the shared memory library calls shown in the **mk_shm(3)** "man" page in Appendix B). This function would have to be written, along with the corresponding "`shmGet_binaryPS()`", and a space in the shared

memory area would have to be allocated to store the data. This is because room is allocated in shared memory for the existing "MFK" methodology probability space, but there is no room as yet for our newly defined "BINARY" vulnerability method. Fortunately, there is only enough room for two floating point numbers (one for each of the outcomes we defined as possible for or BINARY probability space) (namely, `PS_BINARY_DEAD` and `PS_BINARY_ALIVE` as we defined in Figure 3, lines 42 and 43). Another thing missing from Figure 15 is the VL APIs that we call. We do have some VL APIs defined for the BINARY methodology (as shown in Table 4), but those APIs require PDUs as their arguments. However, the V/L APIs shown in Figure 15 (`vl_binary_DIS_Result()`, `vl_binary_DIS_ProbAll()`, `_vl_binary_DIS_ProbDEAD()`, and `_vl_binary_DIS_ProbALIVE()`) all require DIS "IDs" as arguments. Fortunately, writing these functions is fairly straightforward. Remember that the DIS monitor is keeping track of all PDUs of interest. It is simply a matter of finding the PDUs that are associated with the given IDs and then using those PDUs as arguments to the V/L APIs that are not defined. Those defined V/L APIs (Table 4 and Figure 5) use PDUs as arguments to initialize the vulnerability analysis, and therefore, we simply call them and return their results. If further assistance is required, the source code for any of the "MFK" "DIS-ID" functions may be examined. (These are the functions: `vl_mfkDIS_Result()`, `vl_mfkDIS_ProbAll()`, `_vl_mfkDIS_ProbK()`, `_vl_mfkDIS_ProbMF()`, `_vl_mfkDIS_ProbF()`, `_vl_mfkDIS_ProbM()`, and `_vl_mfkDIS_ProbNoDamage()`.) The source code is given in the file `$VLS_HOME/src/Vlapi/vl_dis.c`.

Notice in Figure 15 that the results of these VL APIs (assuming they are eventually written) are placed directly into shared memory (on lines 45, 51, and 64). The DIS server will retrieve them from there and pass them to the client who originally asked for the analysis, thus completing the remote query.

8. SUMMARY

To review, in Section 3 we showed how to unpack, install, and compile the DIS lethality server, as well as make some initial test runs.

We then explained the overall architecture of the server and the disposition of its major modules in Section 4. Section 5 described the data files needed to initialize the server and their location.

Section 6 explained how client applications may connect to the server and pose remote queries.

In Section 7, we showed in detail how the server may be expanded to service just about any vulnerability methodology (beyond just "MFK") and how it could be used to describe damage derived by other mechanisms (beyond just "munitions").

INTENTIONALLY LEFT BLANK

REFERENCES

1. IEEE Computer Society. "Standard for Distributed Interactive Simulation-Application Protocols." IEEE Standard 1278.1-1995, Institute of Electrical and Electronics Engineers, Inc., NY, 1995.

2. Institute for Simulation and Training. "Enumeration and Bit Encoded Values for Use with Protocols for Distributed Interactive Simulation Applications." IST-CF-97-23 (Section 4.3.1.1, Platforms of the Land Domain), Institute for Simulation and Training, Orlando, FL, 3 June 1997.

3. Sauerborn G.C. Proceedings of the Second International Workshop on Distributed Interactive Simulation and Real Time Applications. "The DIS Lethality Communications Server." IEEE Computer Society, pp. 82-87, 1998 (ISBN 0-8186-8594-8).

4. Sauerborn, G.C. "Communicating Platform Vulnerability in a Distributed Environment." Paper: 98F-SIW-130, Simulation Interoperability Workshop Papers, The Simulation Interoperability Standards Organization (SISO), pp. 809-815, September 1998.

5. Deitz, P.H., et al. "The Generation, Use, and Misuse of 'PKS' in Vulnerability/Lethality Analysis." ARL-TR-1640, U.S. Army Research Laboratory, Aberdeen Proving Ground, MD, 1997.

6. Roach, Lisa K. "A Methodology for Battle Damage Repair (BDR) Analysis." ALR-TR-330, U.S. Army Research Laboratory, Aberdeen Proving Ground, MD, 1994.

7. Walbert, J.N., et al. "Current Directions in the Vulnerability/Lethality Process Structure." ARL-TR-296, U.S. Army Research Laboratory, Aberdeen Proving Ground, MD, 1993.

8. Roach, L.K. "Fault Tree Analysis and Extensions of the V/L Process Structure." ARL-TR-149, U.S. Army Research Laboratory, Aberdeen Proving Ground, MD, 1993.

INTENTIONALLY LEFT BLANK

APPENDIX A

INITIAL COMPILATION'S SAMPLE OUTPUT

INTENTIONALLY LEFT BLANK

INITIAL COMPILATION'S SAMPLE OUTPUT

Sample output from compilation script $VLS_HOME/compile.sh is given here. Warnings and other messages will vary, depending on which compiler and operating system are used. The output from this example was generated using a Silicon Graphics®, Incorporated (SGI) IRIX™ 5.3 OS with SGI's ANSI C compiler (v3.19).

```
> ./compile.sh
starting in /usr/people/geoffs/Lserver
------------------------------libs--------------------------
cd src/Libs/CommaDelim
make CC=cc RANLIB=echo install
        cc -g  -c  -I. -c cdf.c
        ar urv libcdf.a cdf.o
a - cdf.o
s - creating archive symbol table. Wait...
s - done
/usr/lib/ar: Warning: creating libcdf.a

        echo libcdf.a
libcdf.a
        cp libcdf.a ../../../lib/
        cp cdf.h ../../../include
cd /usr/people/geoffs/Lserver
cd src/Libs/cprintf
make CC=cc RANLIB=echo install
        cc -DANSIIC -c cprint.c
        ar urv libcprint.a cprint.o
a - cprint.o
s - creating archive symbol table. Wait...
s - done
/usr/lib/ar: Warning: creating libcprint.a

        echo libcprint.a
libcprint.a
        cp libcprint.a ../../../lib/
        cp cprint.h ../../../include
cd /usr/people/geoffs/Lserver
cd src/Libs/Scanner
make CC=cc RANLIB=echo install
        cc -c -g -c  scanner.c
        ar urv libscanner.a scanner.o
a - scanner.o
s - creating archive symbol table. Wait...
s - done
/usr/lib/ar: Warning: creating libscanner.a

        echo libscanner.a
libscanner.a
        rm  scanner.o
        rm -f  ../../../lib//libscanner.a
        cp libscanner.a ../../../lib/
        chmod -w  ../../../lib//libscanner.a
        chmod ugo+r  ../../../lib//libscanner.a
        cp -p scanner.h ../../../include/scanner.h
cd /usr/people/geoffs/Lserver
cd src/Libs/Matrx
make CC=cc RANLIB=echo install
        cc -g -c -c matrx.c
        cc -g -c -c disXforms.c
        ar urv libmatrx.a matrx.o disXforms.o
a - matrx.o
a - disXforms.o
s - creating archive symbol table. Wait...
s - done
/usr/lib/ar: Warning: creating libmatrx.a

        rm  matrx.o disXforms.o
        rm -f ../../../lib/libmatrx.a
        mv libmatrx.a ../../../lib/libmatrx.a
        chmod -w  ../../../lib/libmatrx.a
        chmod ugo+r  ../../../lib/libmatrx.a
        cp -p matrx.h ../../../include
cd /usr/people/geoffs/Lserver
cd src/Db/TblReaders
make CC=cc RANLIB=echo install
        cc -DANSIIC -g -I../../../include -I../../../include/H -c
iua_ke.c
        cc -DANSIIC -g -I../../../include -I../../../include/H -c
iua_heat.c
        cc -DANSIIC -g -I../../../include -I../../../include/H -c
others.c
        ar urv libtbl_rdrs.a iua_ke.o iua_heat.o others.o
a - iua_ke.o
a - iua_heat.o
a - others.o
s - creating archive symbol table. Wait...
s - done
/usr/lib/ar: Warning: creating libtbl_rdrs.a

        echo libtbl_rdrs.a
libtbl_rdrs.a
        cp libtbl_rdrs.a ../../../lib/
        cp tbl_rdrs.h ../../../include
        rm libtbl_rdrs.a
cd /usr/people/geoffs/Lserver
cd src/VLapi
make CC=cc RANLIB=echo install
        cc -I../../include -g -c -I../Libs/DIS/include -I../Db/ -c vl.c
        cc -I../../include -g -c -I../Libs/DIS/include -I../Db/ -c
vl_bnry.c
        cc -I../../include -g -c -I../Libs/DIS/include -I../Db/ -c
vl_dis.c
        ar urv libvl.a vl.o vl_bnry.o vl_dis.o
a - vl.o
a - vl_bnry.o
a - vl_dis.o
s - creating archive symbol table. Wait...
s - done
/usr/lib/ar: Warning: creating libvl.a

        echo libvl.a
libvl.a
        cp libvl.a ../../lib
        cp vl.h ../../include
        rm libvl.a
cd /usr/people/geoffs/Lserver
DIS manager is made a little differently----------:
/usr/people/geoffs/Lserver
cd src_sgi5.3/LIB
make CC=cc
        cc -g -DBSD -c log.c
        cc -g -DBSD -c pdu.c
cfe: Warning 665: pdu.c, line 607: Modified an rvalue.
```

```
        ((ArticulatParamsNode *)*n) = (ArticulatParamsNode
*) 0 ;
        -^
cfe: Warning 665: pdu.c, line 610: Modified an rvalue.
        ((SupplyQtyNode *)*n) = (SupplyQtyNode *) 0  ;
        -^
cfe: Warning 665: pdu.c, line 613: Modified an rvalue.
        ((VarDatumNode *)*n) = (VarDatumNode *) 0  ;
        -^
cfe: Warning 665: pdu.c, line 616: Modified an rvalue.
        ((FixedDatumNode *)*n) = (FixedDatumNode *) 0  ;
        -^
cfe: Warning 665: pdu.c, line 619: Modified an rvalue.
        ((VarDatumValue *)*n) = (VarDatumValue *) 0  ;
        -^
cfe: Warning 665: pdu.c, line 622: Modified an rvalue.
        ((QueryDatumValueNode *)*n) =
(QueryDatumValueNode *) 0  ;
        -^
cfe: Warning 665: pdu.c, line 629: Modified an rvalue.
        ((TrackJam *)*n) = (TrackJam *) 0  ;
 ---    -^
cfe: Warning 665: pdu.c, line 632: Modified an rvalue.
        ((BeamDesc *)*n) = (BeamDesc *) 0  ;
 ---    -^
        cc -g -DBSD -c pkg.c
        cc -g -DBSD -c print.c
        cc -g -DBSD -c byte_bnd.c
cfe: Warning 709: byte_bnd.c, line 2027: Incompatible pointer type
assignment
      u32_ptr = ptr;
 ------------------^
        cc -g -DBSD -c dis_client.c
        cc -g -DBSD -c client_pdu_utils.c
        cc -g -DBSD -c coords.c
        cc -g -DBSD -c utm.c
        ar urv libdis.a log.o pdu.o pkg.o print.o byte_bnd.o
dis_client.o client_pdu_utils.o coords.o  utm.o
a - log.o
a - pdu.o
a - pkg.o
a - print.o
a - byte_bnd.o
a - dis_client.o
a - client_pdu_utils.o
a - coords.o
a - utm.o
s - creating archive symbol table. Wait...
s - done
/usr/lib/ar: Warning: creating libdis.a

        cc -g -DBSD -c dis_server.c
        cc -g -DBSD -c mgr_pdu_utils.c
        ar urv libdis_mgr.a log.o pdu.o pkg.o print.o byte_bnd.o
dis_server.o mgr_pdu_utils.o
a - log.o
a - pdu.o
a - pkg.o
a - print.o
a - byte_bnd.o
a - dis_server.o
a - mgr_pdu_utils.o
s - creating archive symbol table. Wait...
s - done
/usr/lib/ar: Warning: creating libdis_mgr.a

cd ../..
cd src_sgi5.3/MGR
make CC=cc
        cc -g -DIGN_SIGIO -DSGI  -c dis_mgr.c
        cc -g -DIGN_SIGIO -DSGI  -c client_rout.c
        cc -g -DIGN_SIGIO -DSGI  -c net.c
        cc -g -DIGN_SIGIO -DSGI  -c server.c
        cc -g -DIGN_SIGIO -DSGI  dis_mgr.o client_rout.o net.o
server.o ../LIB/libdis_mgr.a -lm -o dis_mgr
cd ../..
cd src_sgi5.3/CLIENT
make CC=cc
        cc -g -cckr -c client.c
        cc -g -cckr client.o  ../LIB/libdis.a -lm  -o client
cd ../..
cd src_sgi5.3/PLAYBACK
make CC=cc
        cc -g -DDEBUG -DSYS5  -c playback.c
        cc -g -DDEBUG -DSYS5  playback.o  ../LIB/libdis.a -lm
-o playback
cd ../..
cd src_sgi5.3/UTIL
make CC=cc
        cc -g -DDEBUG -DSYS5  -c btoa.c
        cc -g -DDEBUG -DSYS5  btoa.o ../LIB/libdis.a -o btoa
cd ../..
cd src_sgi5.3/UTIL/ByteBound
make CC=cc
        cc -I../../H -g -DMAIN -c byte_bnd.c
cfe: Warning 709: byte_bnd.c, line 2027: Incompatible pointer type
assignment
      u32_ptr = ptr;
 ------------------^
        cc byte_bnd.o -L../../LIB  -ldis
        a.out
DISLIB: byte_bounds_pr() unknown "OtherPDU" 0

byte boundaries for pdu type 1: EntityStatePDU
   1   1   1   1   4   2   1   1   2   2
   2   1   1   1   1   2   1   1   1   1
   1   1   2   1   1   1   1   4   4   4
   8   8   8   4   4   4   4   1   1   1
   1   1   1   1   1   1   1   1   1   1
   1   1   1   4   4   4   4   4   4   1
   1   1   1   1   1   1   1   1   1   1
   1   4   -1<var params start>


byte boundaries for pdu type 2: FirePDU
   1   1   1   1   4   2   1   1   2   2
   2   2   2   2   2   2   2   2   2   2
   1   1   1   1   8   8   8   1   1   2
   1   1   1   1   2   2   2   2   4   4
   4   4

byte boundaries for pdu type 3: DetonationPDU
   1   1   1   1   4   2   1   1   2   2
   2   2   2   2   2   2   2   2   2   2
   4   4   4   8   8   8   1   1   2   1
   1   1   1   2   2   2   2   4   4   4
   1   1   1   1   -1<var params start>


byte boundaries for pdu type 4: CollisionPDU
   1   1   1   1   4   2   1   1   2   2
   2   2   2   2   2   2   2   1   1   4
   4   4   4   4   4   4

byte boundaries for pdu type 5: ServiceRequestPDU
   1   1   1   1   4   2   1   1   2   2
   2   2   2   2   1   1   1   1   -1<var params start>


byte boundaries for pdu type 6: ResupplyOfferPDU
   1   1   1   1   4   2   1   1   2   2
   2   2   2   2   1   1   1   1   -1<var params start>
```

```
byte boundaries for pdu type 7: ResupplyReceivedPDU
   1   1   1   1   4   2   1   1   2   2
   2   2   2   2   1   1   1   1  -1<var params start>


byte boundaries for pdu type 8: ResupplyCancelPDU
   1   1   1   1   4   2   1   1   2   2
   2   2   2   2

byte boundaries for pdu type 9: RepairCompletePDU
   1   1   1   1   4   2   1   1   2   2
   2   2   2   2   2   1   1

byte boundaries for pdu type 10: RepairResponsePDU
   1   1   1   1   4   2   1   1   2   2
   2   2   2   2   1   1   1   1

byte boundaries for pdu type 11: CreateEntityPDU
   1   1   1   1   4   2   1   1   2   2
   2   2   2   2   4

byte boundaries for pdu type 12: RemoveEntityPDU
   1   1   1   1   4   2   1   1   2   2
   2   2   2   2   4

byte boundaries for pdu type 13: StartResumePDU
   1   1   1   1   4   2   1   1   2   2
   2   2   2   2   4   4   4   4   4

byte boundaries for pdu type 14: StopFreezePDU
   1   1   1   1   4   2   1   1   2   2
   2   2   2   2   4   4   1   1   1   1
   4

byte boundaries for pdu type 15: AcknowledgePDU
   1   1   1   1   4   2   1   1   2   2
   2   2   2   2   2   2   4

byte boundaries for pdu type 16: ActionRequestPDU
   1   1   1   1   4   2   1   1   2   2
   2   2   2   2   4   4   4   4  -1<var params start>


byte boundaries for pdu type 17: ActionResponsePDU
   1   1   1   1   4   2   1   1   2   2
   2   2   2   2   4   4   4   4  -1<var params start>


byte boundaries for pdu type 18: DataQueryPDU
   1   1   1   1   4   2   1   1   2   2
   2   2   2   2   4   4   4   4  -1<var params start>


byte boundaries for pdu type 19: SetDataPDU
   1   1   1   1   4   2   1   1   2   2
   2   2   2   2   4   1   1   1   1   4
   4  -1<var params start>


byte boundaries for pdu type 20: DataPDU
   1   1   1   1   4   2   1   1   2   2
   2   2   2   2   4   1   1   1   1   4
   4  -1<var params start>


byte boundaries for pdu type 21: EventReportPDU
   1   1   1   1   4   2   1   1   2   2
   2   2   2   2   4   1   1   1   1   4
   4  -1<var params start>


byte boundaries for pdu type 22: MessagePDU
   1   1   1   1   4   2   1   1   2   2
   2   2   2   2   1   1   1   1   4  -1<var params start>

DISLIB: byte_bounds_pr() unknown
"ElectromagneticEmissionsPDU" 23
DISLIB: byte_bounds_pr() unknown "       " 24

byte boundaries for pdu type 25: TransmitterPDU
   1   1   1   1   4   2   1   1   2   2
   2   2   8   1   1   1   1   8   8   8
   4   4   4   2   2   8   4   4   8   2
   2   1   1   1   1  -1<var params start>


byte boundaries for pdu type 26: SignalPDU
   1   1   1   1   4   2   1   1   2   2
   2   2   2   2   4   2   2  -1<var params start>


byte boundaries for pdu type 27: ReceiverPDU
   1   1   1   1   4   2   1   1   2   2
   2   2   2   1   1   4   2   2   2   2

byte boundaries for pdu type 28: DesignatorPDU
   1   1   1   1   4   2   1   1   2   2
   2   2   2   2   2   1   1   4   4   4
   4   4   8   8   8
DISLIB: byte_bounds_pr() unknown "CommentPDU" 29
DISLIB: byte_bounds_pr() unknown "NO SUCH DEFINED PDU"
30
DISLIB: byte_bounds_pr() unknown "NO SUCH DEFINED PDU"
31
cd ../..
UX:mkdir: ERROR: Cannot create directory "lib": File exists
UX:mkdir: ERROR: Cannot create directory "bin": File exists
UX:ls: ERROR: Cannot access lib/libdis.a: No such file or directory
src/Libs/DIS/lib/libdis.a
---------- data manager ----------
        ar urv libtbl_rdrs.a iua_ke.o iua_heat.o others.o
a - iua_ke.o
a - iua_heat.o
a - others.o
s - creating archive symbol table. Wait...
s - done
/usr/lib/ar: Warning: creating libtbl_rdrs.a

        echo libtbl_rdrs.a
libtbl_rdrs.a
        cp libtbl_rdrs.a ../../../lib/
        cp tbl_rdrs.h ../../../include
        rm libtbl_rdrs.a
        cc -I../../include -g -c  -I../Libs/DIS/include -c db_entity.c
        cc -I../../include -g -c  -I../Libs/DIS/include -c db_entmem.c
        cc -I../../include -g -c  -I../Libs/DIS/include -c strlink.c
        cc -I../../include -g -c  -I../Libs/DIS/include -c db_init.c
        cc -I../../include -g -c  -I../Libs/DIS/include -c util.c
        cc -I../../include -g -c  -I../Libs/DIS/include -c misc.c
        cc -I../../include -g -c  -I../Libs/DIS/include -c metatbls.c
        cc -I../../include -g -c  -I../Libs/DIS/include -c db.c
        cc -I../../include -g -c  -I../Libs/DIS/include -c meta_mem.c
        cc -I../../include -g -c  -I../Libs/DIS/include -c tiny_url.c
        cc -I../../include -g -c  -I../Libs/DIS/include -c tbl_fmts.c
        cc -I../../include -g -c  -I../Libs/DIS/include -c vlparam.c
        cc -I../../include -g -c  -I../Libs/DIS/include -c vl_meth.c
        ar urv libvldb.a db_entity.o db_entmem.o strlink.o db_init.o
util.o        misc.o metatbls.o db.o              meta_mem.o tiny_url.o
tbl_fmts.o vlparam.o vl_meth.o
a - db_entity.o
a - db_entmem.o
a - strlink.o
a - db_init.o
a - util.o
```

```
a - misc.o
a - metatbls.o
a - db.o
a - meta_mem.o
a - tiny_url.o
a - tbl_fmts.o
a - vlparam.o
a - vl_meth.o
s - creating archive symbol table. Wait...
s - done
/usr/lib/ar: Warning: creating libvldb.a

        echo libvldb.a
libvldb.a
        cp libvldb.a ../../lib
        cp vl_meth.h ../../include
        rm libvldb.a
------------------------- TCP Lethality Server ------------
        cc -g -DBSD  -I../../include  -I../../include/H/LIB
-I../../include -c client_lib.c
        cc -g -DBSD  -I../../include  -I../../include/H/LIB
-I../../include -c pkg.c
        ar urv libvlsclient.a client_lib.o pkg.o
a - client_lib.o
a - pkg.o
s - creating archive symbol table. Wait...
s - done
/usr/lib/ar: Warning: creating libvlsclient.a

        echo libvlsclient.a
libvlsclient.a
        cp libvlsclient.a       ../../lib
        cp vls_toke.h vlserver.h mk_shm.h ../../include
        cc -g -DBSD  -I../../include  -I../../include/H/LIB
-I../../include -c server.c
        cc -g -DBSD  -I../../include  -I../../include/H/LIB
-I../../include -c mk_shm.c
        cc -g -DBSD  -I../../include  -I../../include/H/LIB
-I../../include -c server_lib.c
        cc -o server.exe -g -DBSD  -I../../include
-I../../include/H/LIB -I../../include server.o pkg.o mk_shm.o
server_lib.o  -L../../lib -lscanner -lcprint -lvlsclient
        cc -g -DBSD  -I../../include  -I../../include/H/LIB
-I../../include -c client.c
        cc -o client.exe -g -DBSD  -I../../include
-I../../include/H/LIB -I../../include client.o -lvlsclient  -L./
------------------------- DIS monitor --------------------
        cc -c -I../../include/H -I../../include/H/LIB -I../..//include -g
-DNOCURSES -DCONNECT_TO_VL_SERVER  Main.c
        cc -c -I../../include/H -I../../include/H/LIB -I../..//include -g
-DNOCURSES -DCONNECT_TO_VL_SERVER  dis_misc.c
        cc -c -I../../include/H -I../../include/H/LIB -I../..//include -g
-DNOCURSES -DCONNECT_TO_VL_SERVER  process_pdu.c
        cc -c -I../../include/H -I../../include/H/LIB -I../..//include -g
-DNOCURSES -DCONNECT_TO_VL_SERVER  usrcmd.c
        cc -c -I../../include/H -I../../include/H/LIB -I../..//include -g
-DNOCURSES -DCONNECT_TO_VL_SERVER  munitions.c
        cc -c -I../../include/H -I../../include/H/LIB -I../..//include -g
-DNOCURSES -DCONNECT_TO_VL_SERVER  ent_list.c
        cc -c -I../../include/H -I../../include/H/LIB -I../..//include -g
-DNOCURSES -DCONNECT_TO_VL_SERVER  fire_det.c
        cc -c -I../../include/H -I../../include/H/LIB -I../..//include -g
-DNOCURSES -DCONNECT_TO_VL_SERVER  mk_shm.c
        cc -c -I../../include/H -I../../include/H/LIB -I../..//include -g
-DNOCURSES -DCONNECT_TO_VL_SERVER  vls_link.c
        cc -o dis_mon.exe -g  -DNOCURSES
-DCONNECT_TO_VL_SERVER Main.o dis_misc.o  process_pdu.o
usrcmd.o munitions.o ent_list.o  fire_det.o mk_shm.o vls_link.o
-L../..//lib -ldis  -lcprint -lcurses -lm -lvlsclient -lvldb -lvl  -ltbl_rdrs
-lscanner  -lmatrx -lcdf
UX:rm: ERROR: Cannot access
/usr/people/geoffs/Lserver/bin/*.exe: No such file or directory

DIS lethality server finished
installing libs and header files
in $VLS_HOME/lib and $VLS_HOME/include.
set $VLS_HOME to: /usr/people/geoffs/Lserver
```

APPENDIX B

MANUAL "MAN" PAGES

INTENTIONALLY LEFT BLANK

MANUAL "MAN" PAGES

These manuals are presented in the UNIX™ "man page" format. The directory $VLS_HOME/doc/ contains these manuals in "man" page (.man) format. In addition, there are versions in Hypertext Markup Language (HTML) (.html), PostScript (.ps), plain text (.txt), and rich text format (.rtf).

The man pages are presented in alphabetical order:

cdf(3)
cprint(3)
db(3)
dis_mon(1)
dismgr(1)
matrx(3)
mk_shm(3)
scanner(3)
vl(3)
vlparam(3)
vls_db_init(5)
vlsclient(3)
vlserver(1)

The "(1)", "(3)", and "(5)" delineations are conventions used for "man" pages. These indicate the general category of application that the manual addresses.

Some variants of UNIX™ stray from this numbering scheme (e.g., IRIX™ and Sun Solaris® use "(4)" to describe file formats instead of "(5)"). In general, these are the applied manual section numbers:

Section "(1)" man pages are for applications and user commands.
Section "(2)" indicates an operating system level library call.
Section "(3)" manuals are for user library calls (such as math library functions, etc.)
Section "(4)" manuals document devices (such as memory, tape, etc).
Section "(5)" manuals describe file formats.
Section "(8)" are for "miscellaneous" other things (such as a man page describing the ASCII codes, etc.).

The directory structure and how it relates to the various modules in the DIS lethality server are shown in the Figures B-1 and B-2. Figure B-1 is a replaction of Figure 2 showing where module source code is located. Figure B-2 shows the current directory organization.

The directory structure and how it relates to the various modules in the DIS lethality server are shown in the Figures B-1 and B-2. Figure B-1 is a replaction of Figure 2 showing where module source code is located. Figure B-2 shows the current directory organization.

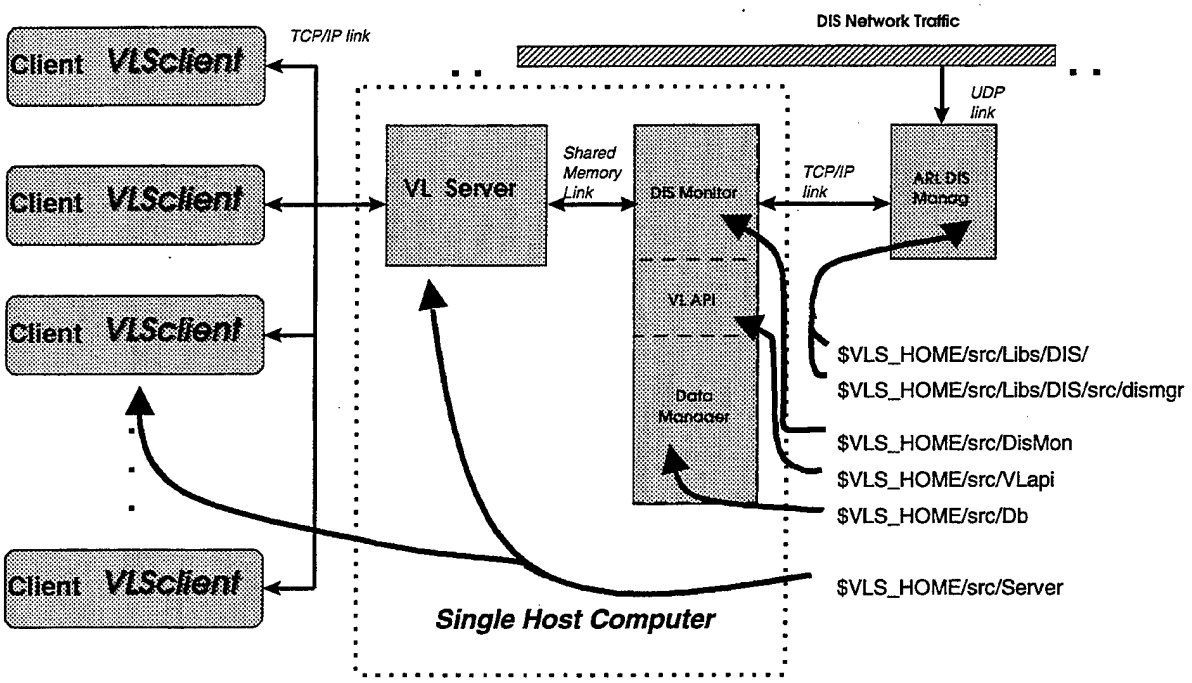


Figure B-1 . Major Module Source Code Locations.

VL Server Directory Structure

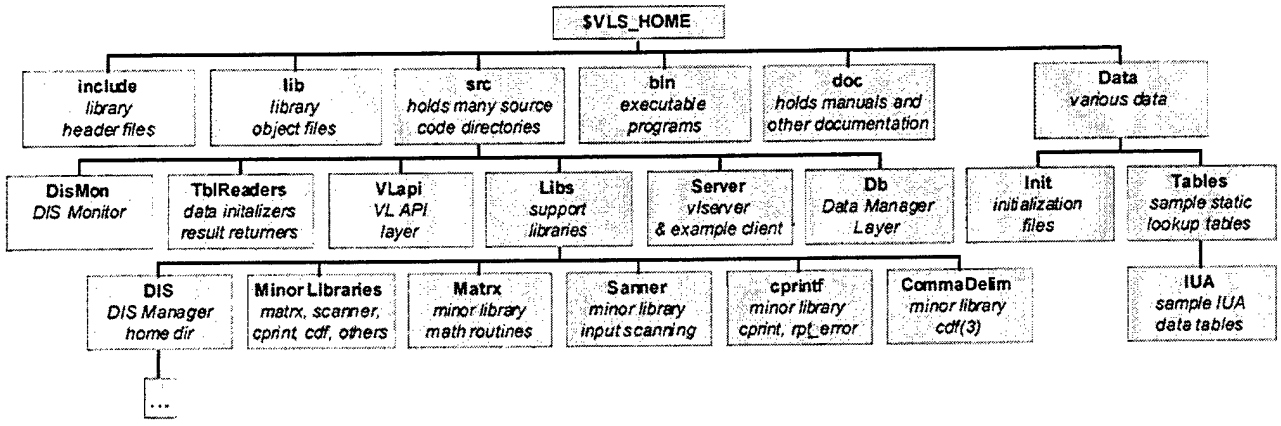


Figure B-2 . Directory Structure.

NAME

cdf_strtok() cdf_scan_fields()

Read ASCII data in comma-delimited fields.

SYNOPSIS

#include "cdf.h"

char *cdf_strtok(char *str , int *flag, int literal);
int cdf_scan_fields(char *fields[], int num_fields, char*buffer,int *cdfflag);

DESCRIPTION

cdf_strtok() is similar to strtok() but specialized for comma-delimited fields. **cdf_strtok()** reads and returns pointers to the comma separated field(s) within the argument string *str.* The returned string will be the characters found between field separators (the comma). [Note: If the field between commas is empty, (e.g. ",," then an empty string will be returned). On subsequent calls if *str* is set to NULL, **cdf_strtok()** will return subsequent comma-delimited fields from within the original string until no more fields are found (at which point a NULL is then returned). The integer pointed to by *flag* is set to -1 if the field ended before a comma or newline was found (this can only occur in the case of a quoted string field being read. That is when (") is the first character in a field, (at which point **cdf_strtok()** expects a quoted string field). When a quoted string is read the whole field content is return (including the enclosing quotes (")). Within a string field the quote character (") itself may be quoted by placing the literal character directly in front of the quote character. The literal character is defined by passing it to **cdf_strtok()** via the *literal* argument. Some databases may use the quote '"' character as the literal (e.g. "This is a string with ""embedded quotes""""). Also the literal character is often the backslash '\' (e.g. "This is a string with \"embedded quotes\"").

cdf_scan_fields() scans the comma separated fields found in the string buffer pointed to by *buffer.* The number of fields expected is specified by *num_fields.* **cdf_scan_fields()** will attempt to read this number of fields and will return the memory allocated (via **malloc()**) string duplicates of these fields in the string pointer array pointed to by *fields.* **cdf_scan_fields()** returns 0 on an error. If a quoted comma separated field ends prematurely, then -1 is returned in *cdfflag.* If there was no error, then the number of fields read is returned and the (malloc'd) string contents of those fields are returned in *fields[].* It is the caller's responsibility to free this allocated memory (by calling **free(3)**).

BUGS

The quote character is fixed (as ") (this might be made user selectable).

SEE ALSO

strtok(3) free(3) malloc(3)

AUTHOR

Geoff Sauerborn <*geoffs@arl.mil*> 1996, 1997, 1998.

NAME

cprint_control, cprint, cprint_fflush, cflush

rpt_perror, rpt_error_getErrno, rpt_error_getMsg, _rpt_error_getLastAddedMsg,

SYNOPSIS

```
#include "cprint.h"

int cprint_control( int channel, int onoff, FILE*dest );
int cprint( int channel, char *fmt, ...);
int cprint_fflush(int channel);
int cflush( int channel );

void _rpt_error( int re_msg_num, char *addedmsg );
void rpt_perror( char *moreinfo );
int rpt_error_getErrno(void);
char* rpt_error_getMsg(void);
char *_rpt_error_getLastAddedMsg();
```

DESCRIPTION

This library provides some error handling routines and a means for printing to and redirecting channeled text messages.

The **cprint** routines are used to print to a channel. Three channels are predefined: **CH_ERR, CH_WARN,** and **CH_STAT.** These are intended for printing error, warning, and status messages respectively. By default these channels start as "stderr", however, applications can turn on/off or redirect any of these channels by calling **cprint_control().** Channels are written to via **cprint().**

The error message functions are meant to assist library writers in tracking errors in lower level functions. When an error occurs in a lower level library **_rpt_error()** is called. The higher level library has the option of ignoring the message or using it (via **rpt_perror()**). This is similar to the way **perror(3)** works. The advantage to rpt_error() is that you may add your own system error codes and messages. Additionally there is provision for calling error handling functions. These changes are made by editing the static structure of error signals and messages. This structure is an array called *re_msg_messages[]* found in cprint.c

Library function details:

```
/*
~ cprint_control();
*
*
*   int cprint_control( int msg_type, int on_off, FILE* destination );
*
*       an interface to allow applications to control lower LIBRARY
*               msg level & destination.
*
*               msg_type is one of:     CH_ERR
*                                       CH_WARN
*                                       CH_STAT
*               on_off is one of:  0 (turns off reporting all msgs of msg_type)
*                                  1 (turns on reporting message of msg_type)
*
*               destination if not NULL, will redirect
*                       and write all messages of type msg_type to
```

```
*                       the file pointed to by destination.
*
*       returns 1 on success 0 on failure.
*
*/


/*
~ cprint_fflush()  and  cflush()
*
* int cflush( msg_channel )
* int cprint_fflush( msg_channel )
*
*       channel is one of
*               CH_ERR
*               CH_WARN
*               CH_STAT
*
*    mimics return value of fflush( FILE *)
*
* On successful completion these functions return a value of
* zero. Otherwise EOF is returned. For fflush(NULL), an
* error is returned if any files encounter an error.
*
* cprint_fflush()  and  are synonyms for themselves cflush()
*
*/


/*
~ cprint()
*
* int cprint( msg_channel, fmt, args.... )
*
*    prints a message on a PRG Error channel.
*       defined default channels ("msg_channel") are:
*               CH_ERR
*               CH_WARN
*               CH_STAT
*
*       fmt is standard formatted print format (see printf())
*
*       args are optional variable arguments for formatted print.
*
* Return value:
* mimics vprintf (returns #of characters transmitted or -1 on err).
*
* On successful completion these functions return a value of
* zero. Otherwise EOF is returned. For fflush(NULL), an
* error is returned if any files encounter an error.
*
* see also:
*       cflush(),  cprint_control()
*
```

```
 *
 */

/*=========================================================*/
/*    error handler (for known errors)                     */
/*=========================================================*/

/*
~ _rpt_error()
 *
 *  void _rpt_error( int re_msg_num, char *addedmsg )
 *
 *      report a known message number to error system for processing.
 *
 *        re_msg_num       is the numeric id of the error.
 *
 *        addedmsg is an additional string of text to be printed
 *           along with the system default message.
 *           The system default message, (and addedmsg), are only
 *           printed when rpt_perror() is called.
 *
 *           if addedmsg == NULL, then just the system default
 *           message is printed (only after rpt_perror() is called).
 *
 *          An internal PRG system error handling function is called
 *           for each known error (re_msg_num).
 *
 */


/*
~ rpt_perror()
 *
 *  void rpt_perror( char *s )
 *
 *   print to stderr the last reported error
 *      (which was generated by a _rpt_error() call.
 *
 */


/*
~ rpt_error_getErrno()
 *
 * int rpt_error_getErrno(void);
 *
 *    returns the integer value of the last error generated.
 *
 *      The integer returned is not the same as the values defined
 *      in the unix intro(2),
 *      This value should never be compare to a number (e.g. "3")
 *      Rather, compare the value with the enumerations defined
 *      in rpt_err.h
 *
```

```
*       See cprint.c for a list of errors numbers and messages.
*
*       e.g. if ( rpt_error_getErrno() == RE_EISCONN )
*                    printf("socket in use");
*
*/


/*
~ rpt_error_getMsg()
*
*  char* rpt_error_getMsg()
*
* Return a text message associated with the last error generated.
* The last error was generated via the last call to _rpt_error().
*
*/


/*
~ _rpt_error_getLastAddedMsg()
*
* char *_rpt_error_getLastAddedMsg(void);
*
* Get the last "added message" that was
* added to the standard error message
* via the _rpt_error( int error_no, static char *an_added_msg )
*
* RETURNS
*       a pointer to the "an_added_msg" string.
*       NULL if no message was ever added on the last call to _rpt_error().
*
*
*/
```

Defined error messages (for use in _rpt_error()):

RE_EPERM	No permission match
RE_ENOENT	No such file or directory
RE_ESRCH	No such process
RE_EINTR	Interrupted system call
RE_EIO	I/O error
RE_ENXIO	No such device or address
RE_E2BIG	Arg list too long
RE_ENOEXEC	Exec format error
RE_EBADF	Bad file number
RE_ECHILD	No child processes
RE_EAGAIN	Resource temporarily unavailable
RE_EWOULDBLOCK	Operation would block
RE_ENOMEM	Not enough space
RE_EACCES	Permission denied

RE_EFAULT	Bad address
RE_ENOTBLK	Block device required
RE_EBUSY	Device or resource busy
RE_EEXIST	File exists
RE_EXDEV	Cross-device link
RE_ENODEV	No such device
RE_ENOTDIR	Not a directory
RE_EISDIR	Is a directory
RE_EINVAL	Invalid argument
RE_ENFILE	Too many open files in system
RE_EMFILE	Too many open files in a process
RE_ENOTTY	Inappropriate IOCTL operation
RE_ETXTBSY	Text file busy
RE_EFBIG	File too large
RE_ENOSPC	No space left on device
RE_ESPIPE	Illegal seek
RE_EROFS	Read-only file system
RE_EMLINK	Too many links
RE_EPIPE	Broken pipe
RE_EDOM	Argument out of range
RE_ERANGE	Result too large
RE_ENOMSG	No message of desired type
RE_EIDRM	Identifier removed
RE_EDEADLK	Deadlock situation detected/avoided
RE_ENOLCK	No record locks available
RE_ENOSTR	Not a stream device
RE_ENODATA	No data available
RE_ETIME	Timer expired
RE_ENOSR	Out of stream resources
RE_ENOPKG	Package not installed
RE_EPROTO	Protocol error
RE_EBADMSG	Not a data message
RE_ENAMETOOLONG	File name too long
RE_EOVERFLOW	Value too large for defined data type
RE_ELIBACC	Can not access a needed shared library
RE_ELIBBAD	Accessing a corrupted shared library
RE_ELIBSCN	.lib section in a.out corrupted
RE_ELIBMAX	Attempting to link in more shared libraries than system
RE_ELIBEXEC	Cannot exec a shared library directly
RE_ENOSYS	Operation not applicable
RE_ELOOP	Too many symbolic links in path name traversal
RE_ERESTART	Restartable system call
RE_ESTRPIPE	If pipe/FIFO, don't sleep in stream head
RE_ENOTEMPTY	Directory not empty
RE_ENOTSOCK	Socket operation on non-socket

RE_EDESTADDRREQ	Destination address required
RE_EMSGSIZE	Message too long
RE_EPROTOTYPE	Protocol wrong type for socket
RE_ENOPROTOOPT	Option not supported by protocol
RE_EPROTONOSUPPORT	Protocol not supported
RE_ESOCKTNOSUPPORT	Socket type not supported
RE_EOPNOTSUPP	Operation not supported on socket
RE_EPFNOSUPPORT	Protocol family not supported
RE_EAFNOSUPPORT	Address family not supported by protocol family
RE_EADDRINUSE	Address already in use
RE_EADDRNOTAVAIL	Can't assign requested address
RE_ENETDOWN	Network is down
RE_ENETUNREACH	Network is unreachable
RE_ENETRESET	Network dropped connection on reset
RE_ECONNABORTED	Software caused connection abort
RE_ECONNRESET	Connection reset by peer
RE_ENOBUFS	No buffer space available
RE_EISCONN	Socket is already connected
RE_ENOTCONN	Socket is not connected
RE_ESHUTDOWN	Can't send after socket shutdown
RE_ETOOMANYREFS	Too many references: can't splice
RE_ETIMEDOUT	Connection timed out
RE_ECONNREFUSED	Connection refused
RE_EHOSTDOWN	Host is down
RE_EHOSTUNREACH	No route to host
RE_EALREADY	Operation already in progress
RE_EINPROGRESS	Operation now in progress
RE_ESTALE	Stale NFS file handle
RE_ECANCELLED	Cancelled
RE_EDQUOT	Disc quota exceeded
RE_ENFSREMOTE	Too many levels of remote in path

---

the following are examples of application system specific errors which have been added to the library. You may remove these and add your own.

RE_DBERR	General Master Data Base Error
RE_DBFLDERR	Invalid field in database element
RE_DBBADKEY	Invalid or dangling key pointer
RE_TGT_UKNOWN	Invalid or undefined target entity
RE_THREAT_UKNOWN	Invalid or undefined threat entity type
RE_DET_EVENT_UKNOWN	Invalid or undefined detonation event
RE_NO_META_REC	V/L Data meta record not found.
RE_VLSOURCE_INTERP	Error interpreting V/L source data.
RE_NO_ENVIRON_DATA	Could not find or set V/L environment (initial) parameters for this case.
RE_NOSHM	Shared memory not attached. An attempt was made to access or set shared memory which has not been attached to the current process.

BUGS

Really this should be separated into two libraries (cprint and rpt_error).

SEE ALSO

printf(3), perror(3), errno(3), strsignal(3)

Author

Geoff Sauerborn <*geoffs@arl.mil*> , US Army Research Lab. 1996, 1997, 1998.

NAME

A Data Management API layer.

SYNOPSIS

db_init, db_MetaTable_query, db_MetaTable_retrieve, db_tbl_retrieve, db_tbl_reader_func, db_tbl_result_func, db_tbl_fmt_type, db_tbl_fmt_result_type

DESCRIPTION

DB (which should really have a name change to *DM*) is an Application Programming Interface (API) to the data management layer of the Distributed Interactive Simulation (DIS) Lethality server.

The Data Manager (db) keeps track of vulnerability tables, and DIS entity IDs. It is mostly data driven in that you specify which entities are associated with which tables. This is done in the in the **DAMAGE_SOURCE_META_DATA_FILE** record of the initialization file (see **vls_db_init(5)**). The file names for the DAMAGE_SOURCE_META_DATA_FILE, DIS_ENTITIES_FILE, and DIS_AUXILIARY_ENTITIES_FILE are found in the data manager initialization file. The file name of the data manager initialization file is passed to **db_init()**. **db_init()** assumes that the filename passed is relative to the current working directory, or if the environmental variable **VLS_HOME** is set, relative to the $VLS_HOME/Data/Init/ directory. Once **db_init()** is called (and does not return an error), other Data Manager API calls may be made.

Internally the data manager maintains correlations between p/k tables and their associated entities via meta data records. These records are structures which contain DIS enumerations for the target and threat as well as identifiers for the Vulnerability/Lethality methodology to be used, the location of the lethality data, and an identifier specifying what the format is for that data. This meta data record contains the following fields:

TargetID	
ThreatID	
VL method used	(defines how to interpret data results)
Table format	(how to parse the data to find results)
Table location	(how/where to get data)

The *Table format* field serves a second purpose. It is also used to determine what *type* of data is returned by the data reader function for that particular table format. *(Currently only MFK probabilities are returned by any data reader function known to the data manager (see the* **VL(3)** *api vl_mfk_ArlDIS_ProbAll_NoNet() for further explanation of the MFK data). Any type of data may be returned by a reader function (not just MFK data). The api* **db_tbl_fmt_result_type()** *is used to determine the type of data that is returned).*

After calling db_init(), meta data for all targets/threats will have been read. However, none of the actual data sources (Tables) will have been loaded into memory. To read a source data (table) into memory, first identify the meta data record associated with the entity and target of interest. This is done via the **db_MetaTable_query()** API

```
unsigned int* db_MetaTable_query( MetaTable_t *qrec, int *nfound );
```

db_MetaTable_query() returns index key(s) which can be used to access the meta record(s) queried from the metadata database. The passed argument (qrec) points to a meta record in the form of a query. In this passed query record, the target, threat, analysis method, and table format to be sought are entered. NULL fields will match all records in a category. For example:

```
TargetID        = T72
ThreatID        = M829
```

```
VL method used   = Munition
Table format     = IUA_KE
```

Note that the above examples are illustrative only of the type (meaning of) data passed to db_MetaTable_query(). For example the `ThreatID` *is actually set to a record of seven (7) integers (the DIS enumeration) and* **not** *the litteral text* `M829`. *See* **db_MetaTable_query()** *for information on the actual data structure formats which are passed and returned.* **db_MetaTable_query()** returns a key(s). Using a key obtained in this manner, the table's meta record can be retrieved from the data manager's internal database via **db_MetaTable_retrieve().**

MetaTable_t *db_MetaTable_retrieve(unsigned int key);

Since the meta record returned by **db_MetaTable_retrieve()** contains the actual location of the lethality data (in URL format), **db_tbl_retrieve()** can then be used to retrieve a pointer to the actual source data of that table (and load its data structure in memory).

void *db_tbl_retrieve(MetaTable_t *mrec)

If this table has never been read into memory, the table will be loaded into memory (and remain there) at this time. Subsequent calls to db_tbl_retrieve() will not re-load the table into memory, rather they will just return a pointer to the data structure of the (already loaded) table. There is currently no facility (API) to unload tables and free memory or force a re-read.

After a successful call to **db_tbl_retrieve()** results from this table may be looked-up. However, first the appropriate data look-up function must be obtained by calling **db_tbl_result_func().** To actually look up the lethality results, the function pointer returned by **db_tbl_result_func()** is then used and passed a pointer to the internal data structure of that lethality data. (That is, it is passed the value returned by **db_tbl_retrieve()**).

An important point to note is that before calling the lookup function, all parameters which define the lethality event's initial conditions must first be set. These parameters are global variables defined in the $VLS_HOME/src/Db/vlparam.h and are set in the **_vlp** API (see **vlparam(3)**). For instance **vlp_zero_all_params()** may be called set all these parameters to zero and should be called prior to setting initial conditions. See **vlparam(3)**

Often this is referred to as the **VLPARAM** layer. The lethality result function (which was returned by **db_tbl_result_func()**) will use these parameters to calculate (or look-up) the vulnerability effects.

To review, these are the steps needed to retrieve lethality results:

1. Set the parameters which define lethality initial conditions (via the **VLPARAM** layer).
2. Call **db_MetaTable_query()** to find out if there is data available which applies to the target and threat of interest.
3. If so, then call **db_MetaTable_retrieve()** to retrieve a meta record for the data of interest.
4. Using the this meta record, call **db_tbl_result_func()** to get a pointer to a data retrieval function that will lookup and return the lethality result (when it is called).
5. Using the same meta record, call **db_tbl_retrieve()** to get a pointer to the lethality data (in memory).
6. Pass this lethality data pointer as an argument to the data retrieval function that was obtained in step 4.
7. The data retrieval function will then return the appropriate lethality results.

The following code segment example is based on the undocumented test application **$VLS_HOME/src/Db/dbtest.c:**

```
{  MetaTable_t  mquery, *mrec;
   VL_Meth *method_struct;
   float *f, *(*funcptr)(void *);
   static dbEntityType _Mk_82 = {2, 9, 225, 2, 73, 1 }; /* 500 lb bomb */
   static dbEntityType _T80 = {1, 1, 222, 1, 1, 1, 0}; /* Russian tank*/

   if ( 0 == db_init("vls_db_init.ini") ) {
      perror("could not open vls_db_init.ini");
      exit(1);
   }

   /*
    * Get meta record.
    */
   mquery.tgt    = &_T80;
   mquery.threat = &_Mk_82;
   mquery.vl_meth= "DIS HitToKill";
      /* "DIS HitToKill"
       *  indicates that VL_PARAM_SET_METH_DIS_HitToKill
       *  is the method  ID.
       */
      /*
       * VL_PARAM_SET_METH_DIS_HitToKill
       *
       * Identifies both the type of output and the type
       * of inputs (PDUs) required to do the look-up.
       * It also defines certain actions that may be taken.
       * For instance (in higher API layers (the VL layer)) the
       * DIS server will ignore the returned results if the
       * munition did NOT make a direct hit against the target.
       *
       */
   mrec = MetaTable_get_rec( mquery.tgt, mquery.threat, VL_PARAM_SET_METH_DIS_Hi

   if (mrec == NULL) {
      puts("Error not such data record");
      exit(0);
   }
   /*
    *  data retrieval function.
    */
   funcptr = db_tbl_result_func(mrec);
   if ( funcptr != NULL )
   /*
    *  use data retrieval function to lookup results.
    *    Pass a pointer to the look-up table.
    *    The look-up table is already loaded into memory.
    */
      f = funcptr( db_tbl_retrieve( mrec ) );
   else
      f = NULL;

   if (f!= NULL) {        /* The reader function returned some data...*/
       int j;
```

```
        VL_Result type_of_output;
       /*
        *   we now print what this data returns as its output.
        */
        type_of_output =  db_tbl_fmt_result_type( mrec->tblfmt) ;
        if (type_of_output ==  __PS_MFK_LOWER_BOUND ) {
       /*  - Note: by examining the meta record describing this data's
        *       format "mrec->tblfmt" with db_tbl_fmt_result_type()
        *       (which returned  " __PS_MFK_LOWER_BOUND"),
        *       we now know that the data returned are an array of
        *       kill probabilities ( M,F,MF,K,and No Damage).
        */
          for ( j=PS_MFK_M; j<=PS_MFK_NODAMAGE; j++) {
               printf("%d: %f0,j,f[j]);        /* show some data*/
          }
        }
     } else {
        puts("no results from table lookup! -ERROR");
     }
}
```

Synopsis of the API functions are now given in the following order:

db_init()
db_MetaTable_query()
db_MetaTable_retrieve()
db_tbl_retrieve()
db_tbl_reader_func()
db_tbl_result_func()
db_tbl_fmt_type()
db_tbl_fmt_result_type()

```
/*
 ~ db_init()
 *
 *  int db_init( char *db_init_filename )
 *
 *  General initialization for database level.
 *  Initialization file name (in string form) is passed as an argument.
 *  This file is opened and parsed.  In the file various filenames will
 *  be found identifying things like:
 *
 *        all DIS dbEntity ID's           (DIS_ENTITIES_FILE)
 *        extra DIS dbEntity ID's        (DIS_AUXILIARY_ENTITIES_FILE)
 *
 * and where to find damage mechanism and data for different
 * target/threat interactions:  (DAMAGE_SOURCE_META_DATA_FILE)
 *
 * These files are opened and read by there appropriate initialization
 * function.
 *
 * Returns:
 *
```

```
 *        1 if successful.
 *
 *        0 if an error occurs somewhere (either in reading the
 *                db_init_filename filename itself or one of the
 *                other files to be read). e.g.:
 *                                DIS_ENTITIES_FILE,
 *                                DIS_AUXILIARY_ENTITIES_FILE, etc...
 *
 */
int db_init( char *db_init_filename );


/*
 ~ db_MetaTable_query()
 *
 * unsigned int *db_MetaTable_query( MetaTable_t *mrec, int *num_elements)
 *
 * Try to find record(s) matching portions of the data record passed.
 * Elements are logically ANDed together.
 * Wildcard is a NULL field.  (all zeros in the case of the DIS entity
 * fields.
 *
 *  If the query is successful (data items found), their database keys
 *  are returned in an array.  The length of the array is returned
 *  by setting the int pointed to by num_elements to the number of
 *  elements (keys) in the array.   These keys can be used with
 *  mtbl_retrieve_data() to retrieve the record(s).
 *
 *
 * currently only the fields:
 *
 *        tgt          - DIS ID of tgt
 *        threat;      - DIS ID of threat
 *        vl_meth;     - type of analysis identifier
 *        tblfmt;      - format of lookup table
 *
 * are queried on. (all other fields are ignored).
 *
 * RETURNS NULL is returned by the function if there were no matches
 *         otherwise (if there were matches) the the function returns
 *        a pointer to an array of keys.  (A key is used to retrieve
 *        the record - see db_MetaTable_retrieve()).
 *        The number of elements in recored is passed by setting num_elements.
 *
 *        NOTE: this array must be free by calling functions.
 *        NOTE: MTBL_QUERY_MAX is an internally defined constant of the Maximum
 *                number of elements that will be returned in the array...
 *
 */
#define  MTBL_QUERY_MAX 100
unsigned int *db_MetaTable_query( MetaTable_t *mrec, int *num_elements);


/*
```

```
 ~ db_MetaTable_retrieve()
 *
 *  MetaTable_t *db_MetaTable_retrieve( unsigned int key );
 *
 *  db_MetaTable_retrieve() returns the meta record associated with
 *        a database index value  "key".
 *
 *  RETURNS
 *        pointer to the table meta record (MetaTable_t *)
 *        NULL if no record found or an error occurred.
 */
MetaTable_t *db_MetaTable_retrieve( unsigned int key );


/*
 ~ db_tbl_result_func()
 *
 * void * db_tbl_result_func( MetaTable_t *mrec )
 *
 *   returns a pointer to the result function
 *   which knows how to interpret the set
 *   VL_Parameters, lookup the approprate results
 *   in the lookup table.
 *
 *   The Table data is in the format specified
 *   found in the tblfmt field of the passed mata record
 *   parameter argument (mrec->tblfmt).
 *
 *   Returns:  pointer to the table look-up (result) function.
 *             NULL on error.
 *
 *   See also: db_tbl_reader_func()
 *             db_tbl_fmt_type()
 *
 */
void * db_tbl_result_func( MetaTable_t *mrec  );


/*
 ~ db_tbl_retrieve()
 *
 * void *db_tbl_retrieve( MetaTable_t * );
 *
 *   Retrieve a pointer to the data structure which holds
 *   the source data of a table loaded into memmory.
 *   This pointer is passed to the table look-up function which
 *   the function that knows how to parse this data structure.
 *
 *   RETURNS: pointer to a table loaded into memmory.
 *            NULL if table is not loaded into memmory or other error.
 *
 *   See also:
 *            _db_tbl_load_source() - called only once. (not needed)
 *
```

```
 */
void *db_tbl_retrieve( MetaTable_t *metr_ptr )


/*
 ~ db_tbl_fmt_type()
 *
 *  TblFmt_Enum db_tbl_fmt_type( char *fmtname )
 *
 *    returns a TblFmt_Enum that corresponds to the string "fmtname"
 *                (positive integer) if the string represents
 *                a recognized format.
 *              _TBLFMT_ERROR (0) otherwise (a failure).
 *
 *  See also:  db_tbl_fmt_returned_data_type()
 *               the TblFmt_t structure (struct _TblFmt_t)
 *               the LookUp_Tbls[] and TblFmt2Result[] arrays.
 *
 */
int db_tbl_fmt_type( char *fmtname )


/*
 ~ db_tbl_fmt_result_type();
 *
 *  VL_Result db_tbl_fmt_result_type(char *fmtname);
 *
 *   RETURNS the data type returned by the table's reader function
 *           which reads the data source described by "fmtname".
 *
 *           a returned value of __PS_MFK_LOWER_BOUND means "MFK" type.
 *
 *           PS_ERROR is returned if "fmtname" is unrecognized
 *                    or another error occurs.
 *
 *  SEE ALSO:  db_tbl_fmt_type()
 *             db_tbl_result_func()
 *
 */
VL_Result db_tbl_fmt_result_type(char *fmtname)
```

FILES

$VLS_HOME/Data/Init/vls_db_init.ini

vls_db_init.ini is the initialization file for the data manager. The environmental variable **VLS_HOME** must be set to the root directory of the DIS Lethality server or if not, the initialization file is looked for relative to the current working directory of the parent process.

SEE ALSO

Other DIS Lethality server components:

vls_db_init(5) vl(3) vlparam(3)

AUTHOR

Geoff Sauerborn <*geoffs@arl.mil*> , US Army Research Lab. 1997, 1998.

NAME

dis_mon

SYNOPSIS

dis_mon -d DIS_mgr_host -e exercise_ID -F [big | little] -D Dport -v Vport -s

DESCRIPTION

Dis_mon (DIS Damage Monitor) is a part of the DIS Lethality server, but may also be used to monitor DIS exercises for lethality effects in a stand alone mode.

When **dis_mon** runs in conjunction with the DIS Lethality server **vlserver** , it serves as a "back end" for **vlserver** - doing the grunt work for the server. (e.g. looking-up lethality results).

Dis_mon uses the ARL DIS Manager to connect to the DIS network, therefore the ARL DIS Manager must be installed and running first. (See **dis_mgr(1)**).

Dis_mon listens to DIS Protocol Data Units (PDUs) paying attention to only certain PDUs that may have an influence on the damage state of simulation entities. Currently the DIS Monitor only listens to Fire, Denotation, and Entity State PDUs. The reason only these PDUs are monitored is because the DIS lethality server only knows how to respond to queries relating to "munition" type damage. Other vulnerability/lethality methodologies which might require other types of "trigger" events might require the monitoring of other types of PDUs. For example, system damage caused by some means of electronic warfare may require monitoring the Electromagnetic Emissions PDU.

When started the DIS Monitor will attempt to connect to the DIS lethality server via a shared memory link. This implies that the DIS lethality server (**vlserver**) must already be running on the same computer. Once connected to **vlserver** the DIS Monitor may respond to queries made by the **vlserver** on behalf of vlserver's clients.

Note, if for any reseason the vlserver is stopped or restarted, the DIS monitor must also be restarted. This is because the **vlserver** creates a shared memory location for joint use during start up. When **dis_mon** is started, **vlserver** communicates to **dis_mon** the location of the shared memory resources. If **dis_mon** is already running when **vlserver** is started, they will both be using *different* shared memory locations. Thus if **vlserver** is restarted, **dis_mon** must also be restarted. Of course, restarting **dis_mon** means that all knowledge of lethality events already monitored will be lost up to the point of the restart.

Stand Alone Mode:

Dis_mon does not need to operate in conjunction with **vlserver.** If run by itself, **dis_mon** may be used to monitor DIS entities and their lethality states. The following commands are accepted from the keyboard:

r - provides a "rollup report" to the console.

q - may be used to quit and exit **dis_mon.**

Data in the rollup report merely reflect what is being broadcast by the DIS simulations controlling the entities. A rollup report output will produce a list (one entity per line) showing certain entity states as reported from the "Entity Appearance Field" of the Entity State PDU. Its output will look similar to the following example:

```
tracking 2 Entities in Exercise 37

Mon Mar  9 16:13:40 EST 1998
```

```
--Entity------------------  ---KILL--- ------Damage------ ----Smoke----  Times
Frc  ID  Type               Mobil FireP Slght Modrt Dstryd Plm Eng PlmEng  Hit

  1 1005 " FMC M113 Armor     0     0     0     0     0     0   0     0     0
  2 1006 " BRDM-2 Reconna     0     0     0     0     0     0   0     0     0

          FRIENDLY              FOES
        (Force ID 1)        (Force ID 2)
           (blue)               (red)
        ------------        -----------
KKilled         0                   0
MKilled         0                   0
FKilled         0                   0
MFKilled        0                   0
```

The columns seen represent the following information. Values denoted by "Bool:" are boolean values (where a value of 1 means "TRUE" and 0 means "FALSE"). Unless otherwise stated, these data are all extracted from the Entity State PDU:

Column		Meaning
Entity		
	Frc	Entities Force ID. This information comes from the Force ID Field of the Entity State PDU. Valid Force IDs are: 0 = Other 1 = Friendly 2 = Foe 3 = Neutral
	ID	This is the Entity ID Field portion of the PDU's Entity Identifier Record.
	Type	This column reports the name of the entity type. The entity type a numeric value defined in the Entity Type Record (of the Entity State PDU). The name seen in this column is the text name associated with that numeric entity type ID. The text name comes from the V/L Data Manager initialization file's "DIS_ENTITIES_FILE" record. (See vls_db_init(5)).
KILL		
	Mobil	Bool: Reports if entity is mobility killed.
	FireP	Bool: Reports if entity is fire power killed.
Damage		
	Slight	Bool: Reports if entity is slightly damaged.
	Modrt	Bool: Reports if entity is moderately damaged.
	Dstryd	Bool: Reports if entity is destroyed.
Smoke		
	Plm	Bool: Smoke plume is rising from the entity
	Eng	Bool: Entity is emitting engine smoke
	PlmEng	Bool: Entity is emitting engine smoke and smoke plume is rising from the entity
Times Hit		This field displays the number of times that dis_mon saw the entity "hit" by a munition. This is a derived number an does not appear in the Entity State PDU.

Following the entity appearance rollup report, a summary is provided for friendly and foe kills (Mobility **MKilled,** Fire Power **FKilled,** Mobility and Fire Power **MFKilled,** and completely destroyed (or Catastrophic) (**KKilled**).

Naturally, when run If run by itself, **dis_mon** will not be able to provide damage state (look-up table) results since it will not be receiving queries from the vlserver.

OPTIONS

-d *DIS_mgr_host* The ARL DIS Manager is running on the computer whose IP address/name is *DIS_mgr_host.* By default **dis_mon** looks for the ARL DIS Manager to on to the same host on which it is running.

-e exercise_ID *This tells* **dis_mon** to monitor the DIS exercise whose exercise identification number is *exercise.* By default **dis_mon** monitors exercise number 1.

-F [big | little] The **-F** options forces big (or little) endian conversion of the incoming binary DIS traffic. Normally this option is unnecessary since **dis_mon** will auto-detect whether it is running on a big endian (MIPS (SGI), RISC etc..) or little (Intel, DEC Alpha, etc) computer architecture. The byte order of DIS traffic is supposed to be network byte ordered (big endian). However, if another DIS host is publishing PDUs in the incorrect (little endian) format, then this option may be used to interpret the incorrectly broadcast data in a meaningful way.

-D Dport *Dport* is the port number used by the ARL DIS manager (which is running on *DIS_mgr_host* host computer. By default the ARL DIS manager uses port 4978.

-v Vport This option tells **dis_mon** to connect to the **vlserver** on TCP/IP port number *Vport*. **dis_mon** makes this connection in order to find out what shared memory identification is being used for **vlserver** to **dis_mon** communication. This is the same port the **vlserver** uses to communicate with all its clients. By default **Vport** is 4976.

-s The **-s** option runs the DIS Monitor in stand alone mode (not communicating with the **vlserver**).

FILES

./roleup.out provides Damage state (as reported by the DIS entities).
./roleup.det Lists of Detonations events and the entities involved.
$VLS_HOME/Data/Init/vls_db_init.ini
vls_db_init.ini is the initialization file for the DIS Monitor. (This is the same initialization file needed by the **data manager (db)** API layer of the DIS Lethality server). The environmental variable **VLS_HOME** must be set to the root directory of the DIS Lethality server.

SEE ALSO

Other DIS Lethality server components:

dis_mgr(1), vlserver(1), vls_db_init(5)

AUTHOR

Geoff Sauerborn <*geoffs@arl.mil*> , US Army Research Lab. 1997, 1998.

NAME

dis_mgr - run the ARL DIS Manager (server)

SYNOPSIS

dis_mgr [-v] [-B bridge_host] [-c num_clients] [-n network_interface] [-r recv -s send] [-MP port] [-BP port] [-m] [-g groupname] [-t time] [-SiteIDMask mask_file] [-H host_id] [-S site_id] [-V version] [-X exercise] [-x ON|OFF] [-F pdu_type] [-P] [-l filename] [-ap] [-overl-overwrite] [-il-inl-incoming] [-ol-outloutgoing] [-h] [-hn] [-a] [-b] [-ab] [-P]

DESCRIPTION

This application is part of a suite of utilities and libraries collectively called the DIS Manger. This particular program is also called the DIS manager (but it is not to be confused with the collective suite).

This program monitors DIS (Distributed Interactive Simulation) Protocol Data Units (PDU) on a UDP (User Datagram Protocol) network and transmits the data to clients applications listening on a TCP/IP network connection. Client applications use local library calls (supplied as part of the ARL DIS Manager suite) to receive these PDUs as an internal (C language) data structure representation of these PDUs. Providing PDUs this way in a native language data structure makes it easier for application programs to manipulated and use the PDU data.

The DIS Manager server (dis_mgr or Dis_Mgr) has the following command line options:

-v verbose mode. Incoming UDP and outgoing TCP/IP traffic and other information is reported.
-P print a list of all PDU types.
-B bridge_host where **bridge_host** is the name of the machine running another instance of Dis_Mgr to be bridged. Bridge mode uses the internet tunneling wherein one may simulate a local (DIS UDP) network connection across a long distance network (like the internet). The dis_mgr is run on both computers (See options **-MP and**
-BP).
-c num_clients where **num_clients** is the maximum number of clients the dis_mgr will support (the default is 8).
-n network_interface - where network_interface is the name of the ethernet interface (e.g., le0, ec0, lo0 {for loopback}). As a default, the Dis_Mgr will attempt to determine this (via the internal function call get_ethernet_interface()). Use this option to bypass the default.

Port information:

-r recv Where **recv** is a port number for receiving PDUs (DIS traffic) on the UDP network (default is port 3000).
-s send Where **send** is a port number for sending PDUs on the UDP network (default is port 2099).
-MP port Where **port** is the port number for internal DIS_Mgr and Client TCP/IP communication. By default this is port number 4978.
-BP port Where **port** is the port number for DIS_Mgr to remote site DIS_Mgr bridge (internet tunneling) connections.

Multicast information: If you wish to avoid the internet tunneling method to facilitate long distance networking you may use multicast if your network supports multicast. A network administrator will have to do additional configuring to prepare the multicast group.

-m Use multicast communication mode.
-g groupname Set the multicast group to **groupname.**
-t time time to live for multicast.

Specific to DIS:

DIS site ID masking:

-SiteIDMask mask_file Where **mask_file** is the name of a file with list of valid site IDs. The file is expected to be a list of integers. the number of integers in this file is between 0 and MAX_SITE_ID. Where MAX_SITE_ID is an internally defined integer (whose default value is FD_SETSIZE, which is normally equal to the maximum number of open files supported by the operating system). Use this option to list valid DIS application "sites". The "site" is a number placed in the DIS PDU header as part of the DIS standard.

DIS PDU header information:

As a service to client applications the DIS Manager pre-fills certain header fields in PDU the header. This is done at the time that client applications request a new PDU. These fields are the site, host, and exercise field. In a DIS exercise these field collectively identify the host computer from which PDUs originated. The Dis_Mgr determines the values for these fields via environmental variables or the following command line switches:

-H host_id Specifies that **host_id** will be the DIS host ID number (the host field in the PDU header). This overides the environmental variable DIS_HOST_ID and the default internal value used by the DIS manager.

-S site_id Specifies the DIS site ID number (in the site field of the PDU header record). This overides the environmental variable DIS_SITE_ID and the default internal value used by the DIS manager (a defined constant SITE_ID_ABERDEEN).

-X exercise Specifies the DIS exercise ID (which will appear in the exercise field of the PDU header record). This overides the environmental variable DIS_EXERCISE_ID and the default internal value of 1 used by the DIS manager.

-V version Where version is one of 2, 3, or 4 (for DIS versions 2.0.2, 2.0.3, and 2.0.4 respectively). This changes the default DIS protocol version number to be associated with out going PDUs and overides the default which was set in the DIS Manager at compile time. Note: that the internal structure of PDUs will not be changed to match a particular DIS protocol version. This option merely changes the value placed in the "version" field of the PDU header. To change the PDU format to conform to a different DIS version you must edit H/protocol_ver.h and define either DIS2_0_2, DIS2_0_3, or DIS2_0_4. Following this, recompile the DIS Manager and all its utilities. (See the recompilation script ${MGR}/compile.sh).

PDU filtering:

-x ON|OFF If -x if followed by **ON,** then all PDUs which do not match the DIS_EXERCISE_ID will be filtered out of the stream and not passed on to client applications. If **-x OFF** is used, then any PDUs seen will be passed to clients (so long as the clients have requested that *type* of PDU). *Client applications have the option to request that all or only certain types of PDUs get sent to them by the dis_mgr. This is done via the* `dis_register_pdu()` *API call.*

LOGGING

The DIS Manager can create binary files containing a log of the PDU traffic. The following options apply to DIS traffic logging:

-l filename - turn logging on. Use **filename** to hold the record of PDUs.
-ap - appends logged PDUs to the log file.
-over|-overwrite - overwrites the log file.

-i|-in|-incoming Logs incoming PDUs only. (PDUs coming into the Dis_Mgr via the DIS UDP network).
-o|-out|-outgoing Logs outgoing PDUs only. (These are PDUs sent by clients to the dis_mgr).
-h Prepends header to each logged PDU. (This is the default). The header information is needed for the DIS Manager utility **playback** to work properly. (Playback is used to play back PDUs from a recorded exercise).
-hn Logs PDUs with no header information.
-a Logs in ascii format.
-b Logs in binary format. (This is the default).
-ab Logs in ascii-binary format,

SEE ALSO

"Distributed Interactive Simulation (DIS) Network Manager",Dec 1994, ARL-TR-780, Ken Smith.
Other DIS Manager utilities. **playback(1), client(1), btoa(1)/btoab(1).** Look in ${MGR}/doc for man pages to these applications.

FILES

The collective DIS Manager is in a file system starting from its own "home" directory. This directory may be located anywhere but, naturally, should be accessable by users who are building client applications. (In order for them to link the manager's object libraries with their application). In the list of files below we use "${MGR}" to represent the DIS Manager's "home" directory . From this root, the subdirectories hold the following files:

${MGR}/src/H - C header "include" files.
${MGR}/src/MGR - source code for the dis_mgr.
${MGR}/src/CLIENT - source code for an example client application.
${MGR}/src/CLIENTX - source code for a Motif X client (unsupported).
${MGR}/src/PLAYBACK - source code for logged exercise playback utility.
${MGR}/src/UTIL - source code for other utilities (btoa, btoab).
${MGR}/src/LIB - library source code (for clients and the dis_mgr).
${MGR}/lib - object code libraries.
${MGR}/bin - compiled executables.
${MGR}/doc - some documentation.
${MGR}/compile.sh - shell program to compile everything.
${MGR}/scrub.sh - shell program to remove compiled objects and executables, etc..

AUTHOR

Original Author: Ken Smith, US Army Research Lab. 1994, 1995, 1996. with additional help from others: Holly A. Ingham, *<hollyo@arl.mil>* and Geoff Sauerborn *<geoffs@arl.mil>* Mark Thomas *<markt@arl.mil>*. James Bowen made an early port to the PC and wrote the original PLAYBACK. Geoff Sauerborn is the most recent maintainer of this software.

NAME

(Matrix Single precision floating point routines)

mat_mult, mat_pm, mat_fpm, mat_fread, mat_fwrite, mat_build_psi, mat_build_phi, mat_build_theta, mat_build_ident, mat_build_rot3, mat_distance, mat_distance_xy, mat_build_DISEntity2World, mat_build_DISWorld2Entity, mat_build_DISEntity2World3x3, mat_build_DISWorld2Entity3x3

(Matrix Double precision floating point routines)

mat_dmult, mat_dpm, mat_fdpm, mat_dfread, mat_dfwrite, mat_dbuild_psi, mat_dbuild_phi, mat_dbuild_theta, mat_dbuild_ident, mat_dbuild_rot3, mat_ddistance, mat_ddistance_xy, mat_dbuild_DIS-Entity2World, mat_dbuild_DISWorld2Entity, mat_dbuild_DISEntity2World3x3, mat_dbuild_DIS-World2Entity3x3, mat_calcDISPsiWrtXAxis

SYNOPSIS

#include "matrx.h"

void mat_mult(float *dest, float *m1 , float *m2, int rows1,int cols1, int rows2);
void mat_pm(float *m,int rows,int cols); /* PRINT A MATRIX */
void mat_fpm(FILE *fp, float *m,int rows,int cols); /* PRINT A MATRIX to file */
int mat_fread(FILE *fp, float *m,int rows,int cols); /* READ A MATRIX from a file */
void mat_fwrite(FILE *fp,float *m,int rows,int cols); /* WRITE A MATRIX to file */
void mat_build_psi(float*psimat, float psi); /* rotate clockwise about the z axis by psi radians */
void mat_build_phi(float *phimat, float phi); /* rotate clockwise about the y axis by phi radians */
void mat_build_theta(float *thetamat, float theta);/* rotate clockwise about the x axis by theta radians */
void mat_build_ident(float *ident, int n);/* make ident an NxN identity matrix */
float *mat_build_rot3(float *mat, double psi, double theta, double phi);
double mat_distance(float *pnt1 , float *pnt2); /* distance in 3 space*/
double mat_distance_xy(float *pnt1 , float *pnt2); /* ignore Z */

float *mat_build_DISEntity2World(float *mat,double psi,double theta,double phi);
float *mat_build_DISWorld2Entity(float *mat,double psi,double theta,double phi);
float *mat_build_DISEntity2World3x3(float *mat,double psi,double theta,double phi);
float *mat_build_DISWorld2Entity3x3(float *mat,double psi,double theta,double phi);

void mat_dmult(double *dest, double *m1 , double *m2, int rows1,int cols1, int rows2);
void mat_dpm(double *m,int rows,int cols); /* PRINT A MATRIX */
void mat_fdpm(FILE *fp, double *m,int rows,int cols); /* PRINT A MATRIX to file */
int mat_dfread(FILE *fp, double *m,int rows,int cols); /* READ A MATRIX from a file */
void mat_dfwrite(FILE *fp,double *m,int rows,int cols); /* WRITE A MATRIX to a mat */
void mat_dbuild_psi(double*psimat, double psi); /* rotate clockwise about the z axis by psi radians */
void mat_dbuild_phi(double *phimat, double phi); /* rotate clockwise about the y axis by phi radians */
void mat_dbuild_theta(double *thetamat, double theta);/* rotate clockwise about the x axis by theta radians */
void mat_dbuild_ident(double *ident, int n);/* make ident an NxN identity matrix */
double *mat_dbuild_rot3(double *mat, double psi, double theta, double phi);
double mat_ddistance(double *pnt1 , double *pnt2); /* ignore Z */
double mat_ddistance_xy(double *pnt1 , double *pnt2); /* ignore Z */

double *mat_dbuild_DISEntity2World(double *mat,double psi,double theta,double phi);
double *mat_dbuild_DISWorld2Entity(double *mat,double psi,double theta,double phi);
double *mat_dbuild_DISEntity2World3x3(double *mat,double psi,double theta,double phi);
double *mat_dbuild_DISWorld2Entity3x3(double *mat,double psi,double theta,double phi);

double mat_calcDISPsiWrtXAxis(double x, double y); /* calc psi in Entity coord. sys */

DESCRIPTION

This library provides some general matrix manipulation functions. It is small and simple.

All library functions treat matrices as a continuous block of memory (that is in a single array). Matrix elements are stored ((R)*(NCOLS) + (C)) elements from the matrix base address. Where "NCOLS" are the total number of columns in the matrix, R and C are matrix number of rows and columns of interest respectively. (Note: with the exception of NCOLS, we start counting at 0. Therefore the very first row and first column are indexed as the 0'th row and 0'th column.) For example, let M be the 3 row by 4 column matrix:

1	2	3	4
5	6	7	8
9	10	11	12

Then the following C language statments are true:

M[((0)*(4) + (0))] ==1
M[((1)*(4) + (2))] ==7
M[((1)*(4) + (3))] ==8
M[((2)*(0) + (2))] ==9

The convenience macro **MAT_INDX** is provided for indexing matrix elements.

usage: MAT_INDX(row, col, NCOLS)

For example, using matrix M above the following C language statments are true:

M[MAT_INDX(0, 0, 4)]==1
M[MAT_INDX(1, 2, 4)]==7
M[MAT_INDX(1, 3, 4)]==8
M[MAT_INDX(2, 2, 4)]==11

A second convenience macro **MAT_MTRX** is also provided.

usage: MAT_MTRX(matrix, row, col, NCOLS)

Using matrix M above and the MAT_MTRX macro the following C language statments are true:

MAT_MTRX(M, 0, 0, 4)==1
MAT_MTRX(M, 1, 2, 4)==7
MAT_MTRX(M, 1, 3, 4)==8
MAT_MTRX(M, 2, 2, 4)==11

There are a few functions which are *not* general at all but specifically apply to the Distributed Interactive Simulation standard (DIS). These are conversion routines used to translate points to and from the DIS world coordinate system and the DIS Entity coordinate system. *(These are functions with the capitalized letters "DIS" found in their name).*

The DIS standard specifies that the following sequence of rotations occur to transform from the DIS World

to the DIS Entity coordinate systems. First rotated about the Z axis (by psi radians). This produces a transformed X and Y axis (called X' and Y'). The next rotation occurs about the transformed Y axis (Y') by theta radians (producing a new X axis again [X"]). The last rotation is by phi radians about the X" axis. These three rotation angles (psi, theta, phi) are call the Euler angles. Positive angles of rotation about an axis are clockwise about the axis ("clockwise" as viewed from axis origin out towards the positive path of the axis). The functions **mat_build_DISWorld2Entity()** and **mat_build_DISEntity2World()** and their double precision counterparts may be used to create matrices which may be used to accomplish these DIS transformations. The transformation matrix produced would then be used to multiply a point (or set of points) to translate that point(s) to the other DIS coordinate system.

For instance, the following code segment will use the Euler angles psi, theta, phi to build the transformation matrix (XMat). This matrix wil then be used to transform the matrix MEntity (which contains 2 points in the first two rows) to the DIS world coordinate system:

```
#include <math.h>
#include "matrx.h"
.
.
.
double XMat[16];     /* will hold transformation matrix a 4x4 */
static double psi=M_PI , theta=M_PI , phi=M_PI ; /* maded-up angles */
double MEntity = { 0., 0., 0., 1.  /* 1st pt.   (0,0,0) */
                   1., 2., 3., 1.  /* 2nd pt.   (1,2,3) */
                 };
double MWorld[8];/* will hold the transformed points */

if (NULL != mat_dbuild_DISEntity2World(Xmat, psi, theta, phi) ) {

   mat_dmult(MWorld, MEntity, Xmat, 2, 4, 4);
   /*
    * MWorld now holds the transformed points
    *
    */
};
```

The reason for the fourth matrix column is because these functions transform homogeneous coordinates whose usefulness is not covered here but is found elsewhere [ROGERS]. Therefore matrices used in these functions **must** have a 4th column (even if not used) as a place holder. Any number will suffice as a place holder, but the use of the number one (1) is preferred. There are alternate functions **mat_build_DISEntity2World3x3(), mat_build_DISWorld2Entity3x3()** and their double precision counter parts **mat_dbuild_DISEntity2World3x3()** and **mat_dbuild_DISWorld2Entity3x3().** These functions do not use the heterogeneous fourth dimension. Therefore only a 3 dimensional point (matrix with exactly **three** (3) columns) will be returned by these functions.

Note that rotation matrices produced by **mat_build_psi(), mat_build_theta(),** and **mat_build_phi()** and their double precision counterpart functions apply to a **single** rotation about the **ordinal** (untransformed) coordinate axis. This differs from the DIS standard method for translating between the DIS Entity *(sometimes called the "missile coordinate system")* and the DIS World coordinate systems *(sometimes called the "earth centered earth fixed, or the geocentric Cartesian coordinate system").*

One important final note before introducing the functions. The function **mat_build_rot()** and its double precision counter part **mat_dbuild_rot()** have *nothing* to do with the DIS coordinate rotations. These functions build a rotational matrix based on the assumption that the ordinal axis (the (original X, Y, and Z axis) *remain fixed* and are never transformed (into X', Y', Z', and X", Y", Z"). (See the IEEE standard 1278.1). *Therefore never use these functions in combination to build a DIS world to entity coordinate transformation*

matrix. Use the DIS specific functions instead (the ones with **DIS** *in their function name).*

Further details on library functions:

```
/*
~ mat_dmult()
*
* void mat_dmult(double *dest,double *m1,double *m2,int rows1,int cols1,int rows2)
*
*    Matrix multiply       [dest] = [m1][m2]
*
*    Multiply matrix "m1" by "m2" and store the results in "dest".
*
 *       m1 is a rows1, by cols1 matrix
 *       m2 is a rows2, by cols1 matrix
 *     dest is a rows1, by cols1 matrix
 *
 *   for example:      dest           m1          m2
 *                   |1 2 3 0|  =  |1 2 3 0|   |1 0 0 0|
 *                                             * |0 1 0 0|
 *                                               |0 0 1 0|
 *                                               |0 0 0 1|
 *        dest is 1 row  by 4 columns
 *          m1 is 1 row  by 4 columns
 *          m2 is 4 rows by 4 columns
*
* In this case the proper call to mat_dmult is:
*
 *        mat_dmult( dest, m1, m2, 1,4,4);
*
*/


/*
~ mat_fdpm()
*
* void mat_fdpm(FILE *fp, double *m,int rows,int cols)
*
* Print a double precision matrix to the file pointed by
*   the file pointer fp.  m is a matrix with "rows" rows
*   and "cols" columns.
* The purpose is to present the matrix in a fairly human
*  readable format.
*
* BUGS:
*   This is just for a 'popular range' of numbers.
*   (It assumes 6 decimal places an no more than 19 digits).
*
* SEE ALSO:
*   mat_fdwrite()
*
*/
```

```
/*
~ mat_dpm()
*
* void mat_dpm(double *m,int rows,int cols)
*
* Print a double precision matrix to the standard output
*    the file pointer fp.  m is a matrix with "rows" rows
*    and "cols" columns.
* The purpose is to present the matrix in a fairly human
*  readable format.
*
* BUGS:
*  This is just for a 'popular range' of numbers.
*   (It assumes 6 decimal places an no more than 19 digits).
*
* SEE ALSO:
*   mat_fdpm(), mat_fdwrite()
*
*/


/*
~ mat_dfwrite()
*
* void mat_dfwrite(FILE *fp,double *m,int rows,int cols)
*
* Write a matrix to a file showing more precision
* but in a less human readable format.  The purpose
* is for storing matrix contents (which can then be read
* back (by mat_dfread()).
*
* BUGS
* Limited to about 21 digits of precision, but you can
* always change the source code if you have say a 64 bit
* architecture (with 128 bit double precision floating point
* numbers).
*
* Numbers are converted from their native binary
*           format to scientific notation numbers
* Disadvantage: Therefore this may cause common digital to
*           real numbers conversion ambiguities.
* Advantage:   However this allows portability of your data
*           between systems.
*
* SEE ALSO:
*  mat_dfread(), mat_fdpm()
*/


/*
~ mat_dfread()
*
* int mat_dfread(FILE *fp, double *m,int rows,int cols)
*
```

```
* Read a matrix from file, storing it internally in the
* (matrix) double precision array "m".
*
* Stored matrix values are ASCII numbers.
*
* Once a matrix reading is started (from the
* file), only the matrix elements are expected (i.e. NO
* comments are allowed in the file (unless they proceed
* or come before the matrix).
*
* SEE ALSO:
*   mat_dfwrite()
*/


/*
~ mat_dbuild_psi()
*
* void mat_dbuild_psi(double*psimat,double psi)
*
*    psimat - the 4x4 matrix
*    psi    - the rotation angle psi (about the Z axis in radians)
*
* Build a Psi rotation matrix of doubles. (Psi, rotation about the Z
* axis in radians) and return it in psimat.
*
* RETURNS
*    The transformation matrx is returned in the passed array psimat.
*    Note that a 4x4 matrix is returned therefore
*    "psimat" must be an array of at least 16 doubles.
*    Furthermore if psimat is then used in a multiplication,
*    "4" columns must be specified as an argument to mat_dmult().
*
*/

/*
~ mat_dbuild_theta()
*
* void mat_dbuild_theta(double*thetamat,double theta)
*
*    thetamat - the 4x4 matrix
*    theta    - the rotation angle theta (about the Y axis in radians)
*
* Build a theta rotation matrix of doubles. (theta, rotation about the Y
* axis in radians) and return it in thetamat.
*
* RETURNS
*
*    The transformation matrx is returned in the passed array thetamat.
*    Note that a 4x4 matrix is returned therefore
*    "thetamat" must be an array of at least 16 doubles.
*    Furthermore if thetamat is then used in a multiplication,
*    "4" columns must be specified as an argument to mat_dmult().
*
```

```
*/


/*
~ mat_dbuild_phi()
*
* void mat_dbuild_phi(double*phimat,double phi)
*
*   phimat - the 4x4 matrix
*   phi    - the rotation angle phi (about the X axis in radians)
*
* Build a phi rotation matrix of doubles. (phi, rotation about the X
* axis in radians) and return it in phimat.
*
* RETURNS
*   The transformation matrx is returned in the passed array phimat.
*   Note that a 4x4 matrix is returned therefore
*   "phimat" must be an array of at least 16 doubles.
*   Furthermore if phimat is then used in a multiplication,
*   "4" columns must be specified as an argument to mat_dmult().
*
*/

/*
~ mat_dbuild_ident()
*
* void mat_dbuild_ident(double *ident, int N)
*
*   Make ident an NxN identity matrix
*   For example if N equals 3 then
*
*     1  0  0
*     0  1  0
*     0  0  1
*
*   will be returned in "ident". Naturally
*   ident must be an array with at least N*N doubles.
*
*   RETURNS
*
*    An NxN identity matrix in "ident"
*
*/


/*
~ mat_ddistance_xy()
*
* double mat_ddistance_xy( double *pnt1 , double *pnt2 )
*
*   pnt1 and pnt2 are arrays containing the X and Y coordinates
*   for each of the points in question.  mat_ddistance_xy() returns
*   the distance (root sum square) between these points.
*   (Z if present is ignored)
```

```
*
* RETURNS
*
*   The distance between two points in the XY plane.
*
*/

/*
~ mat_ddistance()
*
* double mat_ddistance( double *pnt1 , double *pnt2 )
*
*  pnt1 and pnt2 are arrays containing the X, Y, and Z coordinates
*  for each of the points in question.  mat_ddistance() returns
*  the distance (root sum square) between these points.
*
* RETURNS
*
*   The distance between two points in 3 space.
*
*/


/*
~ mat_dbuild_rot3()
*
* double *mat_dbuild_rot3(double *mat, double psi, double theta, double phi)
*
* Build the 3 angle rotation matrix (rotating
*          psi about the Z axis,
*          theta about the Y' axis
*          phi about the X" axis)
*     the rotation matrix is returned in the 4x4 matrix argument
*     "mat" (which is a double precision floating point
*          array of at least 16 elements).
*
*    NOTE:  The ordinal axis are not themselves transformed
*          between rotations.  Therefore this function may
*          NOT be used to create transformation matrices for
*          DIS Euler angles.
*
*    RETURNS
*
*          mat or NULL if an error.
*
*     SEE ALSO:
*
*          mat_dbuild_DISWorld2Entity(), mat_dbuild_DISEntity2World()
*
*/

/*-----------------------float functions----------------------*/
/*-----------------------float functions----------------------*/
/*-----------------------float functions----------------------*/
```

```
/*
~ mat_mult()
*
* void mat_mult(float *dest,float *m1,float *m2,int rows1,int cols1,int rows2)
*
*    Matrix multiply       [dest] = [m1][m2]
*
*     Multiply matrix "m1" by "m2" and store the results in "dest".
*
 *        m1 is a rows1, by cols1 matrix
 *        m2 is a rows2, by cols1 matrix
 *      dest is a rows1, by cols1 matrix
 *
 *   for example:      dest            m1           m2
 *                   |1 2 3 0|  =   |1 2 3 0|    |1 0 0 0|
 *                                               * |0 1 0 0|
 *                                                 |0 0 1 0|
 *                                                 |0 0 0 1|
 *
 *         dest is 1 row  by 4 columns
 *           m1 is 1 row  by 4 columns
 *           m2 is 4 rows by 4 columns
*
* In this case the proper call to mat_mult is:
*
 *          mat_mult( dest, m1, m2, 1,4,4);

*
*/


/*
~ mat_fpm()
*
* void mat_fpm(FILE *fp, float *m,int rows,int cols)
*
* Print a single precision matrix to the file pointed by
*    the file pointer fp.  m is a matrix with "rows" rows
*    and "cols" columns.
* The purpose is to present the matrix in a fairly human
*  readable format.
*
* BUGS:
*   This is just for a 'popular range' of numbers.
*   (It assumes 6 decimal places an no more than 19 digits).
*
* SEE ALSO:
*   mat_fwrite()
*
*/


/*
~ mat_pm()
*
```

```
 * void mat_pm(float *m,int rows,int cols)
 *
 * Print a single precision matrix to the standard output
 *    the file pointer fp.  m is a matrix with "rows" rows
 *    and "cols" columns.
 * The purpose is to present the matrix in a fairly human
 *  readable format.
 *
 * BUGS:
 *  This is just for a 'popular range' of numbers.
 *   (It assumes 6 decimal places an no more than 19 digits).
 *
 * SEE ALSO:
 *   mat_fpm(), mat_fwrite()
 *
 */


/*
 ~ mat_fwrite()
 *
 * void mat_fwrite(FILE *fp,float *m,int rows,int cols)
 *
 * Write a matrix to a file showing more precision
 * but in a less human readable format.  The purpose
 * is for storing matrix contents (which can then be read
 * back (by mat_fread()).
 *
 * BUGS
 * Limited to about 21 digits of precision, but you can
 * always change the source code if you have, say, a 64 bit
 * architecture (with 64 bit single precision floating point
 * numbers).
 *
 * Numbers are converted from their native binary
 *          format to scientific notation numbers
 * Disadvantage: Therefore this may cause common digital to
 *           real numbers conversion ambiguities.
 * Advantage:    However this allows portability of your data
 *           between systems.
 *
 * SEE ALSO:
 * mat_fread(), mat_fpm()
 */


/*
 ~ mat_fread()
 *
 * int mat_fread(FILE *fp, float *m,int rows,int cols)
 *
 * Read a matrix from file, storing it internally in the
 * (matrix) single precision array "m".
 *
```

```
* Stored matrix values are ASCII numbers.
*
* Once a matrix reading is started (from the
* file), only the matrix elements are expected (i.e. NO
* comments are allowed in the file (unless they proceed
* or come before the matrix).
*
* SEE ALSO:
*   mat_fwrite()
*/


/*
~ mat_build_psi()
*
* void mat_build_psi(float*psimat,float psi)
*
*    psimat - the 4x4 matrix
*    psi    - the rotation angle psi (about the Z axis in radians)
*
* Build a Psi rotation matrix of floats. (Psi, rotation about the Z
* axis in radians) and return it in psimat.
*
* RETURNS
*    The transformation matrx is returned in the passed array psimat.
*    Note that a 4x4 matrix is returned therefore
*    "psimat" must be an array of at least 16 floats.
*    Furthermore if psimat is then used in a multiplication,
*    "4" columns must be specified as an argument to mat_dmult().
*
*/

/*
~ mat_build_theta()
*
* void mat_build_theta(float*thetamat,float theta)
*
*    thetamat - the 4x4 matrix
*    theta    - the rotation angle theta (about the Y axis in radians)
*
* Build a theta rotation matrix of floats. (theta, rotation about the Y
* axis in radians) and return it in thetamat.
*
* RETURNS
*
*    The transformation matrx is returned in the passed array thetamat.
*    Note that a 4x4 matrix is returned therefore
*    "thetamat" must be an array of at least 16 floats.
*    Furthermore if thetamat is then used in a multiplication,
*    "4" columns must be specified as an argument to mat_dmult().
*
*/
```

```
/*
~ mat_build_phi()
*
* void mat_build_phi(float*phimat,float phi)
*
*   phimat - the 4x4 matrix
*   phi    - the rotation angle phi (about the X axis in radians)
*
* Build a phi rotation matrix of floats. (phi, rotation about the X
* axis in radians) and return it in phimat.
*
* RETURNS
*   The transformation matrx is returned in the passed array phimat.
*   Note that a 4x4 matrix is returned therefore
*   "phimat" must be an array of at least 16 floats.
*   Furthermore if phimat is then used in a multiplication,
*   "4" columns must be specified as an argument to mat_dmult().
*
*/

/*
~ mat_build_ident()
*
* void mat_build_ident(float *ident, int N)
*
*   Make ident an NxN identity matrix
*   For example if N equals 3 then
*
*     1  0  0
*     0  1  0
*     0  0  1
*
*   will be returned in "ident".  Naturally
*   ident must be an array with at least N*N floats.
*
*   RETURNS
*
*    An NxN identity matrix in "ident"
*
*/


/*
~ mat_distance_xy()
*
* double mat_distance_xy( float *pnt1 , float *pnt2 )
*
*   pnt1 and pnt2 are single precision floating point arrays
*   containing the X and Y coordinates for each of the points
*   in question.
*   mat_distance_xy() returns the distance (root sum square)
*   between these points. (Z if present is ignored)
*
* RETURNS
```

```
*
*    The distance between two points in the XY plane.
*
*/

/*
~ mat_distance()
*
* double mat_distance( float *pnt1 , float *pnt2 )
*
*   pnt1 and pnt2 are single precision floating point arrays
*   containing the X, Y, & Z coordinates for each of the points
*   in question.
*   mat_distance_xy() returns the distance (root sum square)
*   between these points.
*
* RETURNS
*
*    The distance between two points in 3 space.
*
*/

/*
~ mat_build_rot3()
*
* float *mat_build_rot3(float *mat, double psi, double theta, double phi)
*
* Build the 3 angle rotation matrix (rotating
*           psi about the Z axis,
*           theta about the Y' axis
*           phi about the X" axis)
*      the rotation matrix is returned in the 4x4 matrix argument
*      "mat" (which is a floating point array of 16 elements).
*
*     NOTE:  The ordinal axis are not themselves transformed
*           between rotations.  Therefore this function may
*           NOT be used to create transformation matrices for
*           DIS Euler angles.
*
*     RETURNS
*
*           mat or NULL if an error.
*
*      SEE ALSO:
*
*           mat_build_DISWorld2Entity(), mat_build_DISEntity2World()
*
*/


/*------------DIS  Transformation functions -------------------*/
/*------------DIS  Transformation functions -------------------*/
/*------------DIS  Transformation functions -------------------*/
```

```
/*
~ mat_dbuild_DISEntity2World()
*
* double *mat_dbuild_DISEntity2World(double *mat16,double psi,double theta, double phi)
*
* mat16 points to an array of 16 doubles (which represents a 4x4 homogeneous
*    matrix to the "mtrx" library procedures).
*
* psi, theta, and phi are the DIS (Distributed Interactive Simulation)
*      Euler angles as described in the DIS standard.
*    (they represent the successive rotation about the Z, Y', and X"
*     axis in order to transform from the DIS World Coordinate System
*     to the DIS Entity Coordinate system).
*
*      Entity Coordinate system:
*
 *            ^         -Z (Entity's coordinate system used in IEEE 1278.1)
 *            |               Note that "up" is -Z
 *            |
 *            |
 *            |'top'
 *            |
 *            |_
 *          ../_/|.. entity
 *         / |_| ======    'front'
 *        |       | |  --------------->
 *        |__/___|/                   X
 *          /   |
 *         /    |
 *        /     |
 *       /      |
 *      /       v
 *     /         +Z
 *  |/
 *     Y                                          Geoff Sauerborn
*
*          DIS World coordinate system
*
*            ^ +Z          (used in IEEE 1278.1)
*            |                The origin at the center of the earth
*            |                +Z goes through the north pole.
*            |                +X goes through prime meridian at the equator
*            |
*            |
*          #####  the earth
*        ########
*       ##########
*       ##########----------------->
*        ##/#####                  X
*         /#####
*        /    |
*       /     |
*      /      |
*     /       v
```

```
 *    /          -Z
 *  |/
 *     Y                                                Geoff Sauerborn
*
*
* NOTE: these are the Euler angles which are used to transform from
*       the World to Entity coordinate system (even though this
*        function uses them to translate from the
*       Entity to the World coordinate system.
*
* Returns mat16 on success
*       NULL if an error occurred somewhere.
*
* Written by Geoff Sauerborn. geoffs@arl.mil
*  However the transformation were derived elsewhere
*  by others.
*  Special thanks to Rich Pearson (pearson@arl.mil).
*
*/

/*
~ mat_build_DISEntity2World()
*
*float *mat_build_DISEntity2World(float *mat16,double psi,double theta, double phi)
*
*  mat16 points to an array of 16 floats (which represents a 4x4 homogeneous
*     matrix to the "mtrx" library procedures).
*
* psi, theta, and phi are the DIS (Distributed Interactive Simulation)
*       Euler angles as described in the DIS standard.
*    (they represent the successive rotation about the Z, Y', and X"
*     axis in order to transform from the DIS World Coordinate System
*     to the DIS Entity Coordinate system).
*
*  NOTE: these are the Euler angles which are used to transform from
*       the World to Entity coordinate system (even though this
*        function uses them to translate from the
*       Entity to the World coordinate system.
*
*  Returns mat16 on success
*       NULL if an error occurred somewhere.
*
* Written by Geoff Sauerborn. geoffs@arl.mil
*  However the transformation were not derived elsewhere
*  by others.
*  Special thanks to Rich Pearson (pearson@arl.mil).
*
*/


/*
~ mat_build_DISWorld2Entity()
*
*float *mat_build_DISWorld2Entity(float *mat16,double psi,double theta, double phi)
```

```
*
* mat16 points to an array of 16 floats (which represents a 4x4 homogeneous
*    matrix to the "mtrx" library procedures).
*
* psi, theta, and phi are the DIS (Distributed Interactive Simulation)
*      Euler angles as described in the DIS standard.
*    (they represent the successive rotation about the Z, Y', and X"
*    axis in order to transform from the DIS World Coordinate System
*    to the DIS Entity Coordinate system).
*
* Returns mat16 on success
*      NULL if an error occurred somewhere.
*
* Written by Geoff Sauerborn. geoffs@arl.mil
*  However the transformation were not derived elsewhere
*  by others.
*  Special thanks to Rich Pearson (pearson@arl.mil).
*
*/

/*
~ mat_dbuild_DISWorld2Entity()
*
*double *mat_dbuild_DISWorld2Entity(double *mat16,double psi,double theta, double phi)
*
* mat16 points to an array of 16 floats (which represents a 4x4 homogeneous
*    matrix to the "mtrx" library procedures).
*
* psi, theta, and phi are the DIS (Distributed Interactive Simulation)
*      Euler angles as described in the DIS standard.
*    (they represent the successive rotation about the Z, Y', and X"
*    axis in order to transform from the DIS World Coordinate System
*    to the DIS Entity Coordinate system).
*
* Returns mat16 on success
*      NULL if an error occurred somewhere.
*
* Written by Geoff Sauerborn. geoffs@arl.mil
*  However the transformation were not derived elsewhere
*  by others.
*  Special thanks to Rich Pearson (pearson@arl.mil).
*
*/

/*
~ mat_dbuild_DISEntity2World3x3()
*
*double *mat_dbuild_DISEntity2World3x3(double *mat9,double psi,double theta, double phi)
*
* mat9 points to an array of 9 doubles (which represents a 3x3
*    matrix to the "mtrx" library procedures).
*
* psi, theta, and phi are the DIS (Distributed Interactive Simulation)
*      Euler angles as described in the DIS standard.
```

```
*    (they represent the successive rotation about the Z, Y', and X"
*     axis in order to transform from the DIS World Coordinate System
*     to the DIS Entity Coordinate system).
*
* NOTE: these are the Euler angles which are used to transform from
*       the World to Entity coordinate system (even though this
*         function uses them to translate from the
*       Entity to the World coordinate system.
*
* Returns mat9 on success
*       NULL if an error occurred somewhere.
*
* Written by Geoff Sauerborn. geoffs@arl.mil
*  However the transformation were not derived elsewhere
*  by others.
*  Special thanks to Rich Pearson (pearson@arl.mil).
*
*/


/*
~ mat_build_DISEntity2World3x3()
*
*float *mat_build_DISEntity2World3x3(float *mat9,double psi,double theta, double phi)
*
* mat9 points to an array of 9 floats (which represents a 3x3
*     matrix to the "mtrx" library procedures).
*
* psi, theta, and phi are the DIS (Distributed Interactive Simulation)
*       Euler angles as described in the DIS standard.
*     (they represent the successive rotation about the Z, Y', and X"
*     axis in order to transform from the DIS World Coordinate System
*     to the DIS Entity Coordinate system).
*
* NOTE: these are the Euler angles which are used to transform from
*       the World to Entity coordinate system (even though this
*         function uses them to translate from the
*       Entity to the World coordinate system.
*
* Returns mat9 on success
*       NULL if an error occurred somewhere.
*
* Written by Geoff Sauerborn. geoffs@arl.mil
*  However the transformation were not derived elsewhere
*  by others.
*  Special thanks to Rich Pearson (pearson@arl.mil).
*
*/

/*
~ mat_build_DISWorld2Entity3x3()
*
*float *mat_build_DISWorld2Entity3x3(float *mat9,double psi,double theta, double phi)
*
```

```
 * mat9 points to an array of 9 floats (which represents a 3x3
 *    matrix to the "mtrx" library procedures).
 *
 * psi, theta, and phi are the DIS (Distributed Interactive Simulation)
 *      Euler angles as described in the DIS standard.
 *   (they represent the successive rotation about the Z, Y', and X"
 *    axis in order to transform from the DIS World Coordinate System
 *    to the DIS Entity Coordinate system).
 *
 * Returns mat9 on success
 *      NULL if an error occurred somewhere.
 *
 * Written by Geoff Sauerborn. geoffs@arl.mil
 *  However the transformation were not derived elsewhere
 *  by others.
 *  Special thanks to Rich Pearson (pearson@arl.mil).
 *
 */

/*
 ~ mat_dbuild_DISWorld2Entity3x3()
 *
 *double *mat_dbuild_DISWorld2Entity3x3(double *mat9,double psi,double theta, double phi)
 *
 * mat9 points to an array of 9 floats (which represents a 3x3
 *    matrix to the "mtrx" library procedures).
 *
 * psi, theta, and phi are the DIS (Distributed Interactive Simulation)
 *      Euler angles as described in the DIS standard.
 *   (they represent the successive rotation about the Z, Y', and X"
 *    axis in order to transform from the DIS World Coordinate System
 *    to the DIS Entity Coordinate system).
 *
 * Returns mat9 on success
 *      NULL if an error occurred somewhere.
 *
 * Written by Geoff Sauerborn. geoffs@arl.mil
 *  However the transformation were not derived elsewhere
 *  by others.
 *  Special thanks to Rich Pearson (pearson@arl.mil).
 *
 */

/*
 ~ mat_calcDISPsiWrtXAxis()
 *
 *  double  mat_calcDISPsiWrtXAxis( double x, double y )
 *
 *  Calculate Psi with respect to the X axis.
 *
 *  Using the Missile Coordinate system (DIS Entity Coordinate system).
 *  find Psi (the clock wise rotation about Z relative to the positive
 *  X axis, given x,y coordinate of a point in this system.
 *  x,y is taken to be the end point of a vector whose origin is 0,0.
```

```
* psi is the "rotation" that this vector makes relative to the X-axis.
* with its origin fixed (at 0,0).
*
* In the xy plane the DIS Entity Coordinate system looks like this:
*
 *                    ^ X
 *                    |
 *          II        |          I <----- (Quadrant I)
 *                    |
 *                    |
 *          ----------+------------->
 *          -Y        |                Y
 *                    |
 *          III       |        IV
 *                    |
 *                     -X
 *
*
* returns Psi in radians.
*
* Written by Geoff Sauerborn <geoffs@arl.mil>
*
*/
```

SEE ALSO

IEEE Standard 1278.1.

[ROGERS] "Mathematical Elements for Computer Graphics", by David F. Rogers, J. Alan Adams., 1990, ISBN: 0070535299

Author

Geoff Sauerborn *<geoffs@arl.mil>* , US Army Research Lab. 1995, 1996, 1997. The DIS transformation were derived elsewhere by others. Special thanks to Rich Pearson *<pearson@arl.mil>* who provided the DIS tranformations to and from world and entity coordinates.

NAME

shmCreateSharedMem(void), shmGetID(void), shmIsAttached(void), shmCreateSharedMem(void), shmGetID(void), shmDestroy(), shmClear_QueryPlaced(), shmClear_QueryAnswered(), shmSet_QueryPlaced(), shmSet_QueryAnswered(), shmGet_QueryPlaced(), shmGet_QueryAnswered(), shmClear_TargetES_PDU(), shmClear_ShooterES_PDU(), shmClear_Fire_PDU(), shmClear_Detonation_PDU(), shmSet_TargetES_PDU(), shmSet_ShooterES_PDU(), shmSet_Fire_PDU(), shmSet_Detonation_PDU(), shmGet_TargetES_PDU(), shmGet_ShooterES_PDU(), shmGet_Fire_PDU(), shmGet_Detonation_PDU(), shmSet_TargetID(), shmSet_EventID(), shmSet_QueryType(), shmSet_QueryArgsType(), shmGet_TargetID(), shmGet_EventID(), shmGet_QueryType(), shmGet_QueryArgsType(), shmSet_DisVersion(), shmGet_DisVersion(), shmSet_VLResult(), shmSet_mfkPS(), shmSet_prob(), shmGet_VLResult(), shmGet_mfkPS(), shmGet_prob(), shmClear_ErrorMsg(), shmGet_ErrorMsg()

SYNOPSIS

```
#include "mk_shm.h"

int shmCreateSharedMem(void); /* allocate shared memory block */
int shmGetID(void);           /* return share memory ID */
Bool shmIsAttached(void);     /* return TRUE iff shared memory is attched*/
/*---------- Shared memory manipulators-----------------*/
int shmCreateSharedMem(void); /* called by Server */
int shmGetID(void);           /* called by Server and passed
                          *  to DisMonitor via tcp socket.
                          */
int shmDestroy();           /* called by Server */
int shmClear_QueryPlaced();         /* called by DisMonitor */
int shmClear_QueryAnswered(); /* called by Server */
int shmSet_QueryPlaced();     /* called by Server */
int shmSet_QueryAnswered();         /* called by DisMonitor */
int shmGet_QueryPlaced();           /* called by DisMonitor */
int shmGet_QueryAnswered();   /* called by Server */
int shmClear_TargetES_PDU();        /* called by DisMonitor */
int shmClear_ShooterES_PDU();       /* called by DisMonitor */
int shmClear_Fire_PDU();            /* called by DisMonitor */
int shmClear_Detonation_PDU();            /* called by DisMonitor */
int shmSet_TargetES_PDU();    /* called by Server */
int shmSet_ShooterES_PDU();   /* called by Server */
int shmSet_Fire_PDU();        /* called by Server */
int shmSet_Detonation_PDU();  /* called by Server */
void * shmGet_TargetES_PDU();       /* called by DisMonitor */
void * shmGet_ShooterES_PDU();            /* called by DisMonitor */
void * shmGet_Fire_PDU();           /* called by DisMonitor */
void * shmGet_Detonation_PDU();           /* called by DisMonitor */
int shmSet_TargetID();        /* called by Server */
int shmSet_EventID();         /* called by Server */
int shmSet_QueryType();       /* called by Server */
int shmSet_QueryArgsType();   /* called by Server */
int shmGet_TargetID();              /* called by DisMonitor */
int shmGet_EventID();               /* called by DisMonitor */
VLS_Token shmGet_QueryType();       /* called by DisMonitor */
VLS_Token shmGet_QueryArgsType();         /* called by DisMonitor */

int shmSet_DisVersion();                  /* called by DisMonitor */
const char *shmGet_DisVersion();/* called by Server */
```

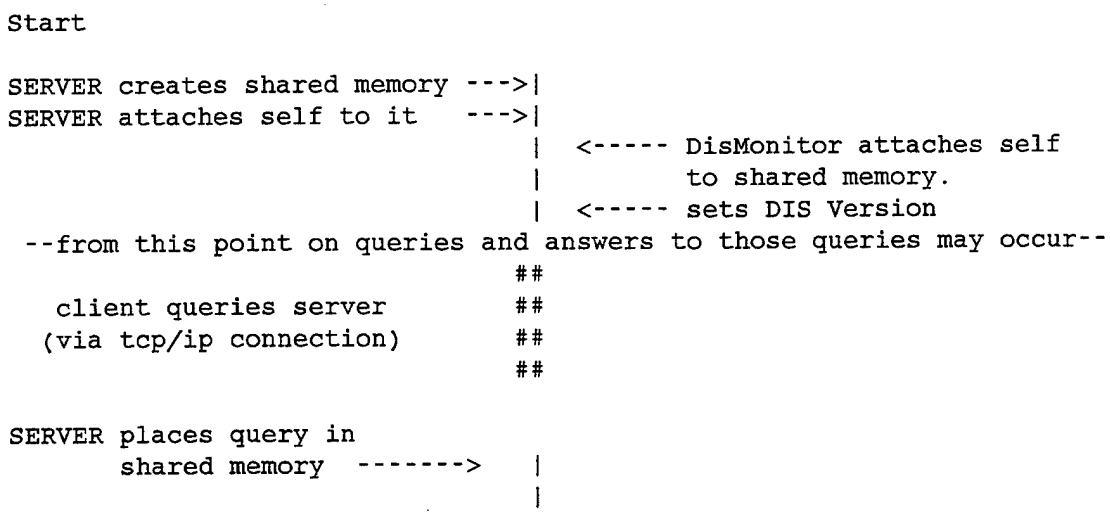
```
int shmSet_VLResult();                    /* called by DisMonitor */
int shmSet_mfkPS();                       /* called by DisMonitor */
int shmSet_prob();                        /* called by DisMonitor */
int shmGet_VLResult();          /* called by Server */
float* shmGet_mfkPS();          /* called by Server */
double shmGet_prob();           /* called by Server */

void shmClear_ErrorMsg();       /* called by Server */
const char *shmGet_ErrorMsg();       /* called by Server */
int  shmSet_ErrorMsg();                   /* called by DisMon */
```

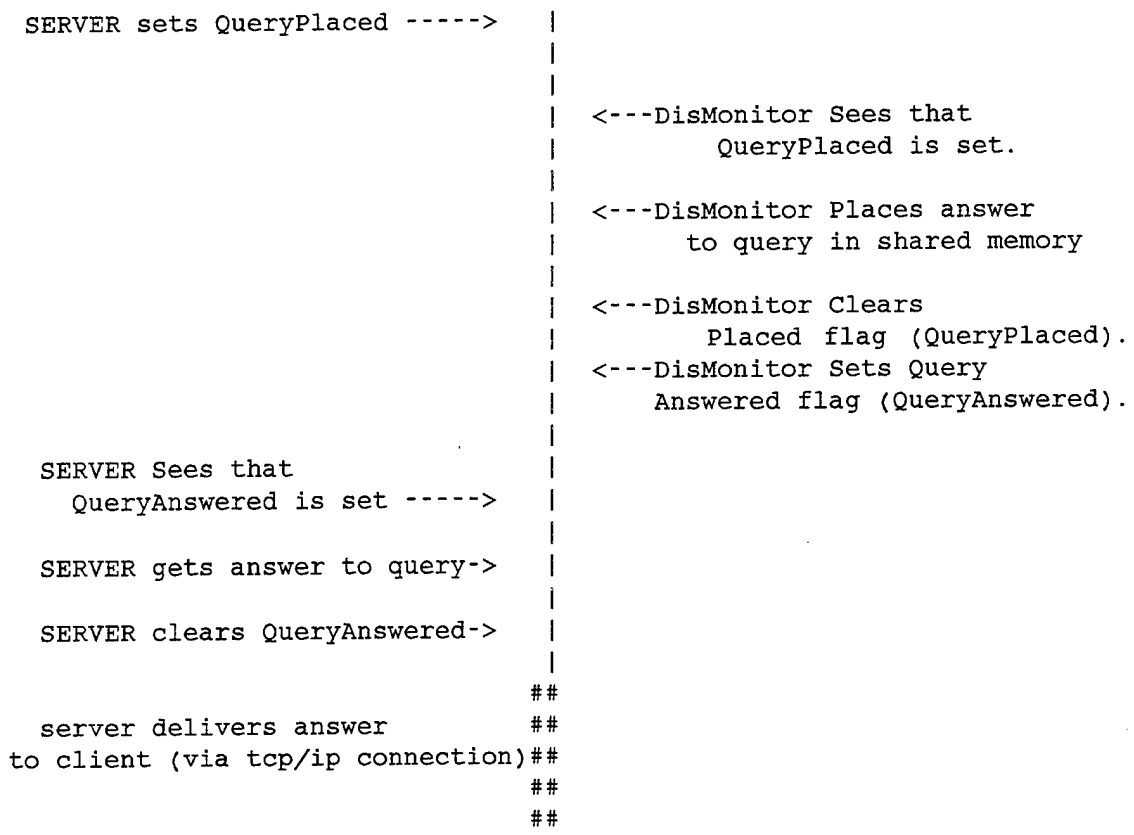
DESCRIPTION

Make (and manipulate) Shared Memory. This is a special purpose library. It links the DIS Server portion of the DIS Lethality Server (the "vlserver" application - see **vlserver(1)**) with the DIS monitor portion (see **dis_mon(1)**).

The functions in this library are "user friendly" in that the shared memory creation is automated (with no need to maintain track of the shared memory "ID"s. The **vlserver** establishes the share memory by calling **shmCreateSharedMem()**. Later, **dis_mon** connects (as a client) to **vlserver** and queries **vlserver** for the shared memory ID. Using this ID, **dis_mon** establishes a connection to the same shared memory location and closes it's client connection with the **vlserver.** (This takes place in the vls_link_connect() function within the **dis_mon** source code). After this, all further communication between the **vlserver** and the DIS Monitor occurs via share memory (through the functions defined in this library).

The purpose for the link between the DIS Server and the DIS Monitor is so that the server may pass vulnerability analysis queries on to the DIS Monitor. The DIS monitor then returns the results via the same shared memory link. The main loop for this process proceeds by having the DIS monitor periodically check to see if a lethality result query has been "queued" into the shared memory. If so, the DIS monitor reads the query from shared memory, calls the appropriate **VL API** function (which will perform the analysis) and then places the result in the shared memory. Once the DIS monitor has completed these steps it sets a flag (via *shmSet_QueryAnswered()*) to inform **vlserver** that query has been answered and is placed in shared memory. The **vlserver** is now free to retrieve the answer (and pass it on to the client who requested it). The following figure maps the sequence of events, and specifies when **vlserver** or **dis_mon** access the shared memory (via these library calls). Access could be either putting data in or coping data out of the shared memory. The sequence of events proceeds forward as one reads down the page. The line running down the middle of the page represents the shared memory. The left side of this line shows when the **vlserver** (SERVER) accesses shared memory. The right side displays access by the DIS Monitor.





Short explanations for the existing **mk_shm(3)** functions follow. Other functions may have to be added if vulnerabilities are described in a manner different from the "MFK" methodology or if additional initialization parameters are needed to complete the vulnerability analysis. Thus far most of the parameters found in the Entity State, Detonation, and Fire PDUs are provided by **mk_shm(3)** functions. Functions are described in the following order:

shmCreateSharedMem()
shmDestroy()
shmGetID()
shmClear_QueryPlaced()
shmClear_QueryAnswered()
shmSet_QueryPlaced()
shmSet_QueryAnswered()
shmGet_QueryPlaced()
shmGet_QueryAnswered()
shmSet_TargetES_PDU()
shmClear_TargetES_PDU()
shmGet_TargetES_PDU()
shmClear_TargetES_PDU()
shmClear_ShooterES_PDU()
shmClear_Fire_PDU()
shmClear_Detonation_PDU()
shmSet_TargetES_PDU()
shmSet_ShooterES_PDU()
shmSet_Fire_PDU()
shmSet_Detonation_PDU()
shmGet_TargetES_PDU()
shmGet_ShooterES_PDU()

shmGet_Fire_PDU()
shmGet_Detonation_PDU()
shmSet_TargetID()
shmSet_EventID()
shmGet_TargetID()
shmGet_EventID()
shmSet_DisVersion( )
shmGet_DisVersion( )
shmGet_QueryType()
shmGet_QueryArgsType()
shmSet_QueryType()
shmSet_QueryArgsType()
shmSet_VLResult()
shmGet_VLResult()
shmGet_mfkPS()
shmSet_mfkPS()
shmGet_prob()
shmSet_prob()
shm_zero_mem()
shmIsAttached()
shmClear_ErrorMsg()
shmGet_ErrorMsg()
shmSet_ErrorMsg()

```
/*
 ~ shmCreateSharedMem()
 *
 * int shmCreateSharedMem(void)
 *
 * Establish shared memory for inter process communitcation
 *  between the Lethality Server and the DIS Monitor
 *
 *  The memory must be large enought to hold
 *   1. arguments to the Lethality Data Query.
 *   2. answers of resulting from the Lethality Data Query.
 *   3. and a few overhead bytes to keep track of when data is read
 *    for reading.
 *
 *        Arguments (1.) are in the form of a set of PDUs (in
 *    binary string form - as seen on the UDP net).  Currently the
 *    only arguments needed are a FirePDU, DetonationPDU, and two
 *    EntityStatePDUs (one for the firer and one for the target).
 *    If you would like to add more arguments increase NARG_TYPES and
 *    add to the ArgEnum list.  Also, if the addtion includes
 *    a different type of answer from the server, then add
 *    that to the AnsTypes union.  Any VLS_Token(s) which represent a new
 *    type of answer(s) is then added as a valid VLS_Token (in vls_toke.h).
 *    (This vls_token must be inserted in (enum _VLS_Token) somewhere between the
 *     __T_END_OF_T_QTYPE_TOKENS and __T_START_OF_T_QTYPE_TOKENS and
 *     coorsponding entries(s) are added in (char *VLS_TokenString[]).
 *
 *  returns 1 on success, 0 on failure.
 *
 */
```

```
int shmCreateSharedMem()

/*
 ~ shmDestroy()
 *
 * int shmDestroy( int id );
 *
 * shared memory is marked for destruction after the last detached
 *  process detaches.  An attempt is made to detatch the current
 *  thread from the shared memory.
 *
 *  return 0 on faliure
 *         1 on success. (if memory is marked for destruction).
 *
 */
int shmDestroy( int id );

/*
 ~ shmGetID()
 *
 * int shmGetID(void)
 *
 *  Returns the shared memory id established by shmCreateSharedMem().
 *  this is the same id returned by the unix system call shmget().
 *  Returns -1 if no shared memory was established.
 *
 */
int shmGetID()

/* ------------------------------------------------------------ */
/*  Shared Memory data  ------------------------------------- */
/*    manipulator APIs ------------------------------------- */
/* ------------------------------------------------------------ */

/*
 ~ shmClear_QueryPlaced()
 *
 * int shmClear_QueryPlaced(void);
 *
 *  Clears (sets to FALSE) the QueryPlaced Bool in shared memory.
 *
 *  returns 1 on sucess.
 *          0 on failure (likely because shared memory not available)
 */
int shmClear_QueryPlaced(void);

/*
 ~ shmClear_QueryAnswered()
 *
 * int shmClear_QueryAnswered(void);
 *
 *  Clears (sets to FALSE) the QueryAnswered Bool in shared memory.
 *
 *  returns 1 on sucess.
```

```
 *          0 on failure (likely because shared memory not available)
 */
shmClear_QueryAnswered()

/*
 ~ shmSet_QueryPlaced()
 *
 * int shmSet_QueryPlaced(void);
 *
 *  Sets (assigns TRUE to) the QueryPlaced Bool in shared memory.
 *
 *  returns 1 on sucess.
 *          0 on failure (likely because shared memory not available)
 */
int shmSet_QueryPlaced(void);

/*
 ~ shmSet_QueryAnswered()
 *
 * int shmSet_QueryAnswered(void);
 *
 *  Sets (assigns TRUE to) the QueryAnswered Bool in shared memory.
 *
 *  returns 1 on sucess.
 *          0 on failure (likely because shared memory not available)
 */
int shmSet_QueryAnswered(void);

/*
 ~ shmGet_QueryPlaced()
 *
 * int shmGet_QueryPlaced(void);
 *
 *  returns 1 (TRUE)  if Query data as been Placed in the shared memory.
 *          0 (FALSE) if a complete query is not there yet.
 *       -1 on failure (likely because shared memory not available)
 */
int shmGet_QueryPlaced(void);

/*
 ~ shmGet_QueryAnswered()
 *
 * int shmGet_QueryAnswered(void);
 *
 *  This function returns the value of a boolean flag in shared memory.
 *   the flag is set (TRUE) by the function shmSet_QueryAnswered()
 *   the flag is set (FALSE) by the function shmClear_QueryAnswered().
 *
 * returns 1 (TRUE) if a placed Query has been answerd by the DIS Monitor
 *          (and this answer has been placed in shared memory).
 *          0 (FALSE) if an answer is not yet there.
 *         -1 on failure (likely because shared memory not available)
 */
int shmGet_QueryAnswered(void);
```

```
/*
 ~ shmSet_TargetES_PDU()
 *
 * int  shmSet_TargetES_PDU( void* bin_arry, int len );
 *
 * Copies the argument (bin_arry) into shared memory.
 * The argument "bin_arry" is a PDU (in binary continious string
 *   form).   "len" is its length.
 * This PDU is the Entity State PDU of the  TARGET (the threatend entity).
 *
 * RETURNS:
 *         1 is on sucess.
 *       0 on failure (likely because shared memory not available)
 */
int shmSet_TargetES_PDU( void* bin_arry, int len );

/*
 ~ shmClear_TargetES_PDU()
 *
 * int  shmClear_TargetES_PDU(void );
 *
 * Marks as clear the shared memroy array associated with TargetES_PDU
 *
 * RETURNS:
 *         1 is on sucess.
 *       0 on failure (likely because shared memory not available)
 */
int  shmClear_TargetES_PDU(void );

/*
 ~ shmGet_TargetES_PDU()
 *
 * void  * shmGet_TargetES_PDU( int *len );
 *
 *  Returns pointer to a PDU (in binary continious string
 *   form).  Its length is returned in "len".
 *  The PDU is the Entity State PDU of the  TARGET (the threatend entity).
 *
 * RETURNS:
 *         ptr to (binary string) PDU on sucess.
 *       NULL on failure. (likely because shared memory not available)
 *
 * Note
 *         a returned length of 0 is a good indicator that the
 *          pdu was not set.
 *
 */
void * shmGet_TargetES_PDU( int *len );

/*
 ~ shmClear_TargetES_PDU()
 *
 * int  shmClear_TargetES_PDU(void );
 *
```

```
 * Marks as clear the shared memroy array associated with TargetES_PDU
 *
 * RETURNS:
 *         1 is on sucess.
 *       0 on failure (likely because shared memory not available)
 */
int shmClear_TargetES_PDU( void )


/*
 ~ shmClear_ShooterES_PDU()
 *
 * int  shmClear_ShooterES_PDU(void );
 *
 * Marks as clear the shared memroy array associated with ShooterES_PDU
 *
 * RETURNS:
 *         1 is on sucess.
 *       0 on failure (likely because shared memory not available)
 */
int  shmClear_ShooterES_PDU(void );

/*
 ~ shmClear_Fire_PDU()
 *
 * int  shmClear_Fire_PDU(void );
 *
 * Marks as clear the shared memroy array associated with Fire_PDU
 *
 * RETURNS:
 *         1 is on sucess.
 *       0 on failure (likely because shared memory not available)
 */
int  shmClear_Fire_PDU(void );

/*
 ~ shmClear_Detonation_PDU()
 *
 * int  shmClear_Detonation_PDU(void );
 *
 * Marks as clear the shared memroy array associated with Detonation_PDU
 *
 * RETURNS:
 *         1 is on sucess.
 *       0 on failure (likely because shared memory not available)
 */
int  shmClear_Detonation_PDU(void );


/*
 ~ shmSet_TargetES_PDU()
 *
 * int  shmSet_TargetES_PDU( void* bin_arry, int len );
 *
```

```
 * Copies the argument (bin_arry) into shared memory.
 * The argument "bin_arry" is a PDU (in binary continious string
 *   form).   "len" is its length.
 * This PDU is the Entity State PDU of the  TARGET (the threatend entity).
 *
 * RETURNS:
 *         1 is on sucess.
 *       0 on failure (likely because shared memory not available)
 */
int  shmSet_TargetES_PDU( void* bin_arry, int len );


/*
 ~ shmSet_ShooterES_PDU()
 *
 * int  shmSet_ShooterES_PDU( void* bin_arry, int len );
 *
 * Copies the argument (bin_arry) into shared memory.
 * The argument "bin_arry" is a PDU (in binary continious string
 *   form).   "len" is its length.
 * This PDU is the Entity State PDU of the Shooting Entity (the entity
 *  who is shooting at the TARGET) - if the Shooting Entity is known).
 *
 * RETURNS:
 *         1 is on sucess.
 *       0 on failure (likely because shared memory not available)
 */
int  shmSet_ShooterES_PDU( void* bin_arry, int len );

/*
 ~ shmSet_Fire_PDU()
 *
 * int  shmSet_Fire_PDU( void* bin_arry, int len );
 *
 * Copies the argument (bin_arry) into shared memory.
 * The argument "bin_arry" is a PDU (in binary continious string
 *   form).   "len" is its length.
 * This "Fire PDU" which describes the weapon launch of the threat munition.
 *
 * RETURNS:
 *         1 is on sucess.
 *       0 on failure (likely because shared memory not available)
 */
int  shmSet_Fire_PDU( void* bin_arry, int len );

/*
 ~ shmSet_Detonation_PDU()
 *
 * int  shmSet_Detonation_PDU( void* bin_arry, int len );
 *
 * Copies the argument (bin_arry) into shared memory.
 * The argument "bin_arry" is a PDU (in binary continious string
 *   form).   "len" is its length.
 * This "Detonation PDU" of the munition threating the target.
```

```
 * (Which describes the munitions impact or detonation.)
 *
 * RETURNS:
 *         1 is on sucess.
 *       0 on failure (likely because shared memory not available)
 */
int  shmSet_Detonation_PDU( void* bin_arry, int len );

/*
 ~ shmGet_TargetES_PDU()
 *
 * void  * shmGet_TargetES_PDU( int *len );
 *
 *  Returns pointer to a PDU (in binary continious string
 *   form).  Its length is returned in "len".
 *  The PDU is the Entity State PDU of the  TARGET (the threatend entity).
 *
 * RETURNS:
 *         ptr to (binary string) PDU on sucess.
 *       NULL on failure. (likely because shared memory not available)
 */
void  * shmGet_TargetES_PDU( int *len );

/*
 ~ shmGet_ShooterES_PDU()
 *
 * void  * shmGet_ShooterES_PDU( int *len );
 *
 *  Returns pointer to a PDU (in binary continious string
 *   form).  Its length is returned in "len".
 * This PDU is the Entity State PDU of the Shooting Entity (the entity
 *  who is shooting at the TARGET) - if the Shooting Entity is known).
 *
 * RETURNS:
 *         ptr to (binary string) PDU on sucess.
 *       NULL on failure. (likely because shared memory not available)
 */
void  * shmGet_ShooterES_PDU( int *len );

/*
 ~ shmGet_Fire_PDU()
 *
 * void  * shmGet_Fire_PDU( int *len );
 *
 *  Returns pointer to a PDU (in binary continious string
 *   form).  Its length is returned in "len".
 * This "Fire PDU" which describes the weapon launch of the threat munition.
 *
 * RETURNS:
 *         ptr to (binary string) PDU on sucess.
 *       NULL on failure. (likely because shared memory not available)
 */
void  * shmGet_Fire_PDU( int *len );
```

```
/*
 ~ shmGet_Detonation_PDU()
 *
 * void  * shmGet_Detonation_PDU( int *len );
 *
 *  Returns pointer to a PDU (in binary continious string
 *   form).  Its length is returned in "len".
 * This "Detonation PDU" of the munition threating the target.
 * (Which describes the munitions impact or detonation.)
 *
 * RETURNS:
 *         ptr to (binary string) PDU on sucess.
 *       NULL on failure. (likely because shared memory not available)
 */
void  * shmGet_Detonation_PDU( int *len );

/*
 ~ shmSet_TargetID()
 *
 * int shmSet_TargetID( int arry3[] );
 *
 *  Set target entity ID in shared memory.
 *  The function's argument (int arry3[]) is an array of
 *  3 integers, (the site, application, and ID) which
 *  together serve to identify an entity in the DIS protocol.
 *
 *  return 1 on success
 *       0 on failure.
 */
int shmSet_TargetID( int arry3[] );

/*
 ~ shmSet_EventID()
 *
 * int shmSet_EventID( int arry3[] );
 *
 *  Set the DIS event ID in shared memory.
 *  The function's argument (int arry3[]) is an array of
 *  3 integers, (the site, application, and ID) which
 *  together serve to identify an event the DIS protocol.
 *
 *  return 1 on success
 *       0 on failure.
 */
int shmSet_EventID( int arry3[] );

/*
 ~ shmGet_TargetID()
 *
 * int shmGet_TargetID( int arry3[] );
 *
 *  Get the DIS event ID from shared memory.
 *  The function's argument (int arry3[]) is an array of
 *  3 integers.  These 3 integers shall be set within the function
```

```
 *  (to the site, application, and ID) of the DIS Target EntityID
 *  in shared memory.
 *
 *  return 1 on success
 *       0 on failure.
 */
int shmGet_TargetID( int arry3[] );

/*
 ~ shmGet_EventID()
 *
 * int shmGet_EventID( int arry3[] );
 *
 *  Called by DisMonitor.
 *
 *  Get the DIS event ID from shared memory.
 *  The function's argument (int arry3[]) is an array of
 *  3 integers.  These 3 integers shall be set within the function
 *  (to the site, application, and ID) of the DIS EventID in shared
 *  memory.
 *
 *  return 1 on success
 *       0 on failure.
 */
int shmGet_EventID( int arry3[] );

/*
 ~ shmSet_DisVersion( );
 *
 * int shmSet_DisVersion ( char *str );
 *
 *  Copies string into the "DIS Version" location of shared memory.
 *
 *  returns 1 if some or all of the string was copied.
 *  returns 0 if none of the string could be copied.
 */
int shmSet_DisVersion( char *str );

/*
 ~ shmGet_DisVersion( );
 *
 * const char *shmGet_DisVersion();      -       called by Server
 *
 *  Copies string into the "DIS Version" location of shared memory.
 *
 *  returns ptr to static shared memory holding "DIS Version"
 *  returns NULL   if none is the memory could not be accessed.
 */
const char *shmGet_DisVersion();

/*
 ~ shmGet_QueryType()
 *
 * VLS_Token shmGet_QueryType(void);
```

```
 *
 *  Get the QueryType from shared memory.
 *
 *  Called by DisMonitor.
 *
 *  The QueryType specifies the form in which the query answer
 *  is to appear.  As of this writing, some of the
 *  valid query types are:
 *
 *          T_QTYPE_mfkDIS_Result
 *          T_QTYPE_mfkDIS_ProbAll
 *          T_QTYPE_mfkDIS_ProbK
 *          T_QTYPE_mfkDIS_ProbMF
 *          T_QTYPE_mfkDIS_ProbF
 *          T_QTYPE_mfkDIS_ProbM
 *          T_QTYPE_mfkDIS_ProbNoDamage
 *
 *  return a valid QueryType VLS_Token on success
 *       T_ERROR on failure.
 */
VLS_Token shmGet_QueryType(void);

/*
 ~ shmGet_QueryArgsType()
 *
 * VLS_Token shmGet_QueryArgsType(void);
 *
 *  Get the QueryArgsType from shared memory.
 *
 *  Called by DisMonitor.
 *
 *  The QueryArgsType specifies the arguments which are used
 *  in setting up the initial conditions for a vulnerability
 *  assessment.  As of this writing, some of the
 *  valid QueryArgsType's are:
 *
 *          T_VLS_QUERY_TYPE_MFK_BINARY_PDUS - expect binary pdu args
 *          T_VLS_QUERY_TYPE_MFK_DIS_IDS    - expect ID args
 *
 *  return a valid QueryArgsType VLS_Token on success
 *       T_ERROR on failure.
 */
VLS_Token shmGet_QueryArgsType(void);

/*
 ~ shmSet_QueryType()
 *
 * int shmSet_QueryType( VLS_Token type );
 *
 *  Set the QueryType in shared memory.
 *
 *  Called by Server.
 *
 *  The QueryType specifies the form in which the query answer
```

```
 *  is to appear.  As of this writing, some of the
 *  valid query types are:
 *
 *          T_QTYPE_mfkDIS_Result
 *          T_QTYPE_mfkDIS_ProbAll
 *          T_QTYPE_mfkDIS_ProbK
 *          T_QTYPE_mfkDIS_ProbMF
 *          T_QTYPE_mfkDIS_ProbF
 *          T_QTYPE_mfkDIS_ProbM
 *          T_QTYPE_mfkDIS_ProbNoDamage
 *
 *  return 1 on success.
 *       0 on failure.
 */
int shmSet_QueryType( VLS_Token type );

/*
 ~ shmSet_QueryArgsType()
 *
 * int shmSet_QueryArgsType( VLS_Token type );
 *
 *  Set the QueryArgsType in shared memory.
 *
 *  Called by Server.
 *
 *  The QueryArgsType specifies the arguments which are used
 *  in setting up the initial conditions for a vulnerability
 *  assessment.  As of this writing, some of the
 *  valid QueryArgsType's are:
 *
 *          T_VLS_QUERY_TYPE_MFK_BINARY_PDUS - expect binary pdu args
 *          T_VLS_QUERY_TYPE_MFK_DIS_IDS   - expect ID args
 *
 *  return 1 on sucess.
 *       0 on failure.
 */
int shmSet_QueryArgsType( VLS_Token type );

/*
 ~ shmSet_VLResult()
 *
 * int shmSet_VLResult( VL_Result result, int flag );
 *
 *  return 1 on sucess.
 *       0 on failure.
 */
int shmSet_VLResult( VL_Result result, int flag );

/*
 ~ shmGet_VLResult()
 *
 * int shmGet_VLResult( VL_Result *result, int *flag );
 *
 * Sets *result and *flag to the VLResult in shared memory and source
```

```
 *  flag respectively.  (See the vl API layer of the VL Data manager.
 *
 *  returns 1 on success;
 *        0 on failure
 */
int shmGet_VLResult( VL_Result *result, int *flag );

/*
 ~ shmGet_mfkPS()
 *
 * float *shmGet_mfkPS( float probs5[] );
 *
 *  return probs5 on sucess.
 *       NULL on failure.
 */
float *shmGet_mfkPS( float probs5[] );

/*
 ~ shmSet_mfkPS()
 *
 * int shmSet_mfkPS( float probs5[] );
 *
 *  return 1 on sucess.
 *       0 on failure.
 */
int shmSet_mfkPS( float probs5[] );

/*
 ~ shmGet_prob()
 *
 * double shmGet_prob(void);
 *
 *  returns WHATEVER is in that share memory location.
 *
 */
double shmGet_prob(void);

/*
 ~ shmSet_prob()
 *
 * int shmSet_prob( double prob );
 *
 *  return 1 on sucess.
 *       0 on failure.
 */
int shmSet_prob( double prob );


/*
 ~ shm_zero_mem()
 *
 *  void shm_zero_mem(int unused)
 *
 *  This function is called when a HUP signal is recieved
```

```
 *  by the vlserver.  It sets all of the shared memory
 *  area to zero (0).
 *
 */
void shm_zero_mem(int unused);

/*
 ~ shmIsAttached()
 *
 *  Bool shmIsAttached(void)
 *
 *  Return 1 (TRUE) if shared memory is currently attached.
 *  Return 0 (FALSE) if not.
 */
Bool shmIsAttached(void);

/*
 ~ shmClear_ErrorMsg()
 *
 * void shmClear_ErrorMsg(void);
 *
 *  Called by Server to effectively clear the error message buffer.
 *  (so that the next call to shmGet_ErrorMsg() returns NULL;
 *
 */
void shmClear_ErrorMsg(void);

/*
 ~ shmGet_ErrorMsg()
 *
 * const char *shmGet_ErrorMsg(void);
 *
 *  Called by Server to fetch the null terminated string
 *  message placed in the error message buffer.
 *  (presumably to pass on to the client).
 *
 *  See Also: DIS Server utility APIs cprint(3), rpt_error(3)
 *
 *  returns pointer to the error message string.
 *        NULL if the error message string is not set.
 *           (i.e. there is no error message).
 *
 */
const char *shmGet_ErrorMsg(void);

/*
 ~ shmSet_ErrorMsg()
 *
 *  int shmSet_ErrorMsg(char *str_error_msg);
 *
 *  Called by DIS Monitor to set the null terminated string
 *  message placed in the error message buffer.
 *  (presumably to pass on to the client by the server).
 *
```

```
 *  See Also: DIS Server utility APIs cprint(3), rpt_error(3)
 *
 *  returns 1 (TRUE) if string was copied.
 *        0 (FALSE) not (shared memmory was not accessable).
 *
 *  NOTES:
 *        A maximum of the system defined BUFSIZ bytes
 *      can be placed into the error buffer.
 *      The server may pass even fewer bytes onto the client.
 *      Generally, the server will strive to send less than
 *      1024 bytes to a client at in any one message.
 */
int shmSet_ErrorMsg(char *str_error_msg);
```

SEE ALSO

Other DIS Lethality server components:

vl(3), vlserver(1), dis_mon(1), vlexample_client.c - an undocumented example client program provided with the DIS Lethality server (look in $VLS_HOME/src/Server).

Author

Geoff Sauerborn <*geoffs@arl.mil*> , US Army Research Lab. 1997, 1998.

NAME

scan_int, scan_double, scan_long, scan_next_white, scan_skip_cmnt, scan_is_eof, scan_linenum, fscan_int, fscan_double, fscan_long, fscan_next_white, fscan_skip_cmnt, fscan_is_eof, fscan_linenum - general scanning routines (for scanning ascii input).

SYNOPSIS

```
#include "scanner.h"

/* Initialization / Closing routines: */

int fscan_reg(FILE *fp, char *filename);
FILE *fscan_fopen(char *filename,char *typeopen);
FILE *fscan_fopen_wMsgOnErr(char *filename,char *typeopen,char *callingfunc);
int fscan_unreg(FILE *fp);
int fscan_fclose(FILE *fp);

/* File status functions: */

const char *fscan_filename( FILE *fp);
int fscan_linenum(FILE *fp);
int scan_linenum(void);
int scan_is_eof();
int fscan_is_eof(FILE *fp);

/* Read head movement functions: */

int scan_next_white( void );
int fscan_next_white( FILE *fp);
int fscan_skip_cmnt(FILE *fp);
int scan_skip_cmnt(void);
/* *Experts only* - fscan_getc(), fscan_ungetc(), scan_getc(), scan_ungetc(), *BY-PASSES* COMMENT &
white space filtering. */

int fscan_getc( FILE *fp);
int scan_getc( void );
int fscan_ungetc(int c, FILE *fp);
int scan_ungetc( int c);

/* Read head movement (and data interpretation) functions: */

int fscan_int(FILE *fp );
int scan_int(void );
double fscan_double(FILE *fp);
double scan_double(void);
long fscan_long(FILE *fp );
long scan_long(void );

/* Read head movement (and strings/line scanning) functions: */

int fscan_quoted_string(FILE *fp, char *buff, int buffer_size);
int scan_quoted_string(char *buff, int buffer_size);
int fgetline(FILE *fp,char string[] ,int  limit  );
char *scan_string(char *buff,  int buffsize);
char *fscan_string(FILE *fp, char *buff,  int buffsize);
```

int fscan_gets(FILE *fp, char *s, int n);
int scan_gets(char *s, int n);

/ String operating functions: */*

char *sscan_skip_white(char* str);
char *sscan_next_white(char* str);
int sscan_int(char *str);
double sscan_double(char *str);
void sscan_strip_quotes(char *str);
void sscan_add_quoted_quotes(char *str);

DESCRIPTION

This library provides general scanning functions with internal input line number tracking.

Line number tracking: It is important to use **fscan_unreg()** (or **fscan_fclose()**) when finished with a file. This is because if a file is closed outside of the scanner library without notifying the library (via **fscan_unreg()**) then on many operating systems it it likely the next file opened will have the same FILE pointer; hence, the scanner library will think it is continuing to operate the on the original file *(and the line numbers reported will be incorrect).*

fscan_fopen() , or **fscan_reg()** must be used to open and register a file in order to track line numbers and file names. All other library function may still be used to scan through a file, however if fscan_reg() and fscan_fopen() are *not* used, then **fscan_filename()** *will not be able to return the correct filename.* All routines keep track of each new file pointer **fp** sent to the library. In this way information can be retrieved about the current line number and EOF status of any file (via the **fscan_linenum()** and **fscan_is_eof()** functions.

WARNING **fscan_getc(FILE *fp),** and **scan_getc()** bypass comment and white-space filtering. These should only by used by *expert* experts.

BUGS

Really "just special features". *fscan_linenum(fp)* returns the current line number for the opened file pointed to by file pointer *fp*. In this way the user can report line numbers associated with read errors. *fscan_linenum()* returns an int, so files with more than INT_MAX lines, will have a unpredictable value returned.

A maximum of SCANNERLIB_OPEN_FILES_TRACKED files are monitored. (If a user wants to track more than SCANNERLIB_OPEN_FILES_TRACKED files, the library will need recompiling).

Unless **fscan_unreg()** is called, scanner library functions cannot tell when a file is closed. If the scanner library does not know when the file is closed memory used by the library is not recycled. If a file is closed without informing the library (*via* **fscan_unreg()** or **fscan_fclose()**) and then a new file is opened and that new file has the same file pointer *(FILE*)* value as the (now) closed first file, then the newly opened file will incorrectly be tracked as if it were the old file (and its first reported linenumber will equal the last linenumber read from the original file).

Further details on library functions:

Initialization and closing routines:
/*
~ fscan_reg()

```
*
* int  fscan_reg(FILE *fp, char *filename);
*
*  Register a file pointer (fp) (and optional file name (filename)
*      with the scanner library.
*      Once registerd, fscan_linenum() and fscan_filename() can be
*        used to report the current line number and the file name
*        being read.
*      If filename is passed as NULL, then the "UNKNOWN_FILE_NAME"
*        is used to report the filename (via: fscan_filename()).
*
*  returns an integer >= 0 on success.
*          a negative integer on failure.
*
*/


/*
~ fscan_fopen()
*
* FILE *fscan_fopen( char *filename, char *typeopen)
*
*   attempt to open file "filename", for the purpose of "typeopen"
*           (* these two args are passed directly to fopen() *)
*      if successful, the file is then registered into the scanner library.
*           (* via fscan_reg() *)
*
*      returns FILE pointer to the opened file on success.
*           NULL on failure.
*/

/*
~ fscan_fopen_wMsgOnErr()
*
* FILE *fscan_fopen_wMsgOnErr( char *filename, char *typeopen, char *callingfunc)
*
*  the function behaves like fscan_fopen() with the addition of
*  printing  error message to stderr if file could not be opened.
*  the messaged are either:
*     "fscan_fopen_wMsgOnErr(): called with NULL arg(s)"
* or   "CallingFunctionName(): could not open "file" for "r".
*
*      (Where "CallingFunctionName()" represents the string pointed to by
*        char *callingfunc, the argument passed to this function).
*
*      returns a FILE pointer to the opened file on success.
*           NULL on failure.
*/

/*
~ fscan_unreg()
*
* int fscan_unreg( FILE *fp)
*
```

```
 *      unregister a file from the scanner library.
 *      (but do not attempt a close of the file stream).
 *
 * return 0 if the if successful close.
 *       EOF is there is an error.
 *
 *            (passing a NULL file pointer "fp" or a
 *             file pointer that was never registered
 *              [via fscan_fopen_wMsgOnErr() or fscan_reg()]
 *             are errors.)
 *
 */

/*
 ~ fscan_fclose()
 *
 * int fscan_fclose( FILE *fp)
 *
 *      close a file and unregister it with the scanner library.
 *
 * return 0 if the if successful close.
 *       EOF is there is an error.
 *
 *            (passing a NULL file pointer "fp" or a
 *             file pointer that was never registered
 *              [via fscan_fopen_wMsgOnErr() or fscan_reg()]
 *             are errors.)
 *
 */
```

Functions to maintain and track file status information:

```
/*
 ~ fscan_linenum()
 *
 * int fscan_linenum(FILE *fp)
 *
 *   Returns the current line number for the
 *   opened file pointed to by file pointer "fp".
 *   In this way the user can report line numbers associated with
 *   read errors.
 *
 *    fscan_linenum() returns an int, so files with more
 *    than INT_MAX lines, will have a unpredictable value returned.
 *
 */
  scan_linenum() is equivalent to fscan_linenum(stdin)

/*
 ~ fscan_is_eof()
 *
 * int fscan_is_eof(FILE *fp)
 *
 * Determine if the file associated with the file pointer (fp)
```

```
 * is at the end-of-file.  (Has it's read head at the end-of-file).
 *
 *  Return 1 if at EOF.
 *  otherwise return 0.
 */
  scan_is_eof() is equivalent to fscan_is_eof(stdin)

/*
~ fscan_filename()
 *
 * const char *fscan_filename( FILE *fp)
 *
 *        returns pointer to the string file name registered via  fscan_reg().
 *          or  NULL upon an error.
 *                (such as the name or file pointer (fp) was never registered).
 */
```

Read-head movement functions.

```
/*
~ fscan_next_white()
 *
 * int fscan_next_white( FILE *fp)
 *
 *  For the file associated with (fp). move it's read head
 *  to the next "whitespace" character.
 *
 *  See Also:
 *      isspace(3)
 */
scan_next_white() is equivalent to fscan_next_white(stdin)

/*
~ fscan_skip_cmnt()
 *
 * fscan_skip_cmnt(FILE *fp)
 *
 * Move the read head past comments (lines beginning with a pound #)
 * and any white space.  That is, bring the read head to the first
 * non-comment, non-whitespace line.  (This might be an EOF).
 *
 * Returns the next character to be read or
 *       EOF if at end of file.
 *
 * NOTES:
 *
 *    All comments are denoted by a '#' as the first character of a line.
 *    There is currently library function to change the comment character.
 */
```

scan_skip_cmnt() is equivalent to fscan_next_white(stdin)

```
/*
 ~ fscan_getc()
 *
 * int fscan_getc( FILE *fp);
 *
 *  Read the next character in the file (move the read-head forward).
 *
 *    return the next character.
 *    return EOF if at file end.
 *
 *       *WARNING* this routine by-passes comment and white-space
 *            filtering and should only by used if you know
 *            what you are doing ... (you probably do).
 *
 */
```

scan_getc() is equivalant to fscan_getc(stdin)

```
/*
 ~ fscan_ungetc();
 *
 * int fscan_ungetc(int c, FILE *fp);
 *
 *  Place the character "c" , back into the input stream
 *  of the file associated with the file pointer (fp).
 *
 *
 */
```

scan_ungetc(c) is equivalent to fscan_ungetc(c,stdin)

Other read-head movement functions (with data interpretation).

```
/*
 ~ fscan_int()
 *
 * int fscan_int( FILE *fp)
 *
 *  Scan the input stream (fp) and attempt to interpret the next character
 *  string as an integer.  Any white space and comment lines
 *  (see fscan_skip_cmnt() are skipped prior to the attempted interpretation.
 *
 *  Returns
 *       the integer if successful.
 *
 *       if unsuccessful, the returned value is undefined and
 *         an error message is printed.
 *
 */
```

scan_int() is equivalent to fscan_int(stdin)
fscan_double() is similar except the next floating point number is returned.
scan_double() is equivalent to fscan_double(stdin).
fscan_long() is similar except the next long is returned.

scan_long() is equivalent to fscan_long(stdin).

```
/*
~ fscan_quoted_string(FILE *fp, char *buffer, int buffer_size)
*
* Read a quoted string from the input stream
* "this is an example" and place it in buffer.
*  fscan_quoted_string() will treat all lines ending in the
*  single backslash character (\)
* as a continuation line.
*
*  buffer is filled with the string (to include quotes) up to
*     buffer_size characters.
*
*  White space an commented lines are skipped prior
*  to reading the quoted string.
*
*  returns 1 on success,
*        0 on failure. (such as no quoted string found).
*/
```

scan_quoted_string() is is equivalant to scan_quoted_string(stdin)

```
/*
~ fgetline()
*
* int fgetline(FILE *fp, char *string , int limit )
*
*  Read from the file stream pointed to by "fp".
*  place the contents up to (and including) the first
*  newline character '\n'.
*
*  Characters read (and placed
*  into the buffer "string") and null terminates the
*  buffer (with \0 after the last character read).
*  No more than "limit"-1 characters will be placed
*  into the buffer "string".
*
*  RETURNS the number of characters placed into "string"
*        when "limit" is not reached.  When limit
*        is reached, then "limit" is always returned
*        even though "limit"-1 characters will have
*        been placed in the buffer.
*
*        EOF is returned if EOF is the first character read.
*
*  adopted from the K&R getline()  by  Geoff Sauerborn.
*
*  See also  fgets(3)
*
*/
```

```
/*
~ fscan_string()
*
* char *fscan_string(FILE *fp, char *buff,  int buffsize)
*
*  Reads a white space delimited string, returns it in
*       the buffer "buff".  no more than "buffsize"-1
*       characters will be placed in "buff".
*
*  RETURNS pointer to string that was read.
*         if no string found before EOF, (or just EOF seen) then
*             NULL is returned.
*/
   scan_string( b, bsz) is equivalent to fscan_string(stdin,b,bsz)


/*
~ fscan_gets()
*
*  int fscan_gets( FILE *fp, char *buff, int n )
*
*  fscan_gets() is used to read the rest of a line
*  [from the fp to the next occurrence of
*                   END-OF-LINE (0
*                               or comment character
*                               or END-OF-FILE]
*
*   all read characters are placed into buff.
*   no more than n chars are read.  The buffer is null terminated.
*
*   RETURNS
*    the number of characters read or EOF if END-OF-FILE is read.
*
*    If the limit "n" is reached, then n is always returned
*         even though n-1 characters will have been placed
*         in the buffer.
*
*/
   scan_gets(str, n)  is equivalent to fscan_gets(stdin,str,n)
```

Some string equivalent functions (similar to other library functions, however they operate on strings not file streams):

```
/*
~ sscan_skip_white()
*
* char *sscan_skip_white(char* str)
*
* Scan the passed null terminated character string array (str).
```

```
 *
 * Return the address of the first non-white character.
 *            if there is only white space in the string,
 *            then the address of the terminating NULL is returned.
 * NULL is returned if the str is NULL.
 */

/*
 ~ sscan_next_white()
 *
 * char *sscan_next_white(char* str)
 *
 *  return the address of the next white space character in the string str.
 *       NULL is returned on failure or end-of-string is reached
 *         without finding a white space character.
 *
 *  see also:  isspace(3)
 */


/*
 ~ sscan_int()
 *
 * int  sscan_int(char *str);
 *
 *  read and return int from the string str.
 *
 *  RETURNs
 *    the scanned integer value.
 *    If an integer is not scanned,
 *    an error message is printed and the
 *     is returned.
 *
 */


/*
 ~ sscan_double()
 *
 * double  sscan_double(char *str);
 *
 *  read and return double from the string str.
 *
 *  RETURNs
 *    the scanned floating point number value.
 *    If number cannot be scanned,
 *    an error message is printed and the
 *    0 is returned.
 *
 */

/*
 ~ sscan_strip_quotes()
 *
```

```
 * void sscan_strip_quotes( char *str )
 *
 *   remove quotes from a string.  (but handle quoted quotes i.e.
 *
 */

/*
~void sscan_add_quoted_quotes( char *str )
 *
 *         add '' before any quotes .  " ->becomes->
 *    returns a pointer to a static character buffer
 *                    (which will be over-written on next call).
 *           The internal buffer is BUFFSIZE in length.
 *                    The enquoted string is in this buffer.
 */
```

SEE ALSO

gets(3), getc(3), ungetc(3), sscanf(3), scanf(3), strtok(3)

AUTHOR

Geoff Sauerborn <*geoffs@arl.mil*> , US Army Research Lab. 1995, 1996, 1997, 1998.

NAME

vl_mfk_ArlDIS_Result_NoNet, vl_mfk_ArlDIS_ProbAll_NoNet, _vl_mfk_ArlDIS_ProbM_NoNet, _vl_mfk_ArlDIS_ProbMF_NoNet, _vl_mfk_ArlDIS_ProbF_NoNet, _vl_mfk_ArlDIS_ProbK_NoNet, _vl_mfk_ArlDIS_ProbNoDamage_NoNet

vl_mfk_binaryDIS_Result_NoNet, vl_mfk_binaryDIS_ProbAll_NoNet, _vl_mfk_binaryDIS_ProbK_NoNet, _vl_mfk_binaryDIS_ProbMF_NoNet, _vl_mfk_binaryDIS_ProbF_NoNet, _vl_mfk_binaryDIS_ProbM_NoNet, _vl_mfk_binaryDIS_ProbNoDamage_NoNet

vl_mfkDIS_Result, vl_mfkDIS_ProbAll, _vl_mfkDIS_ProbK, _vl_mfkDIS_ProbMF, _vl_mfkDIS_ProbF, _vl_mfkDIS_ProbM, _vl_mfkDIS_ProbNoDamage

_vl_drandom, vl_GetResultErrorValue, vl_mfk_directFireIsAHit, VL_Result VL_mfkDIS_ResultGenericRandomDraw

SYNOPSIS

```
#include <vl.h>
/*-----------------vl_mfk_ArlDIS...() functions-----------------*/
VL_Result vl_mfk_ArlDIS_Result_NoNet(int*flg, VLSetParam_t itype, ...);
float* vl_mfk_ArlDIS_ProbAll_NoNet( VLSetParam_t itype, ... );
double _vl_mfk_ArlDIS_ProbM_NoNet(VLSetParam_t itype, ...);
double _vl_mfk_ArlDIS_ProbMF_NoNet(VLSetParam_t itype, ...);
double _vl_mfk_ArlDIS_ProbF_NoNet(VLSetParam_t itype, ...);
double _vl_mfk_ArlDIS_ProbK_NoNet(VLSetParam_t itype, ...);
double _vl_mfk_ArlDIS_ProbNoDamage_NoNet(VLSetParam_t itype, ...);

/*-----------------vl_mfk_binaryDIS...() functions---------------*/
VL_Result vl_mfk_binaryDIS_Result_NoNet(int *flg, VLSetParam_t itype, ...);
float* vl_mfk_binaryDIS_ProbAll_NoNet( VLSetParam_t itype, ...);
double _vl_mfk_binaryDIS_ProbK_NoNet(VLSetParam_t itype, ...);
double _vl_mfk_binaryDIS_ProbMF_NoNet(VLSetParam_t itype, ...);
double _vl_mfk_binaryDIS_ProbF_NoNet(VLSetParam_t itype, ...);
double _vl_mfk_binaryDIS_ProbM_NoNet(VLSetParam_t itype, ...);
double _vl_mfk_binaryDIS_ProbNoDamage_NoNet(VLSetParam_t itype, ...);

/*-----------------vl_mfkDIS...() functions---------------------*/
/* DisID  * an array of 3 16-bit unsigned integers (Uint16[3])  */
VL_Result vl_mfkDIS_Result(int*flag, DisID *entityID, DisID *eventID);
float*  vl_mfkDIS_ProbAll( DisID *entityID, DisID *eventID );
double  _vl_mfkDIS_ProbK( DisID *entityID, DisID *eventID );
double  _vl_mfkDIS_ProbMF(  DisID *entityID, DisID *eventID );
double  _vl_mfkDIS_ProbF(  DisID *entityID, DisID *eventID );
double  _vl_mfkDIS_ProbM(  DisID *entityID, DisID *eventID );
double  _vl_mfkDIS_ProbNoDamage(  DisID *entityID, DisID *eventID );

/*-----------------vl_...() Utility functions-----------------*/
void _vl_drandom_seed( int seed );
double _vl_drandom(void);
int vl_GetResultErrorValue(void);
int vl_mfk_directFireIsAHit( DetonationResult DIS_det_result );
VL_Result VL_mfkDIS_ResultGenericRandomDraw( void );
```

DESCRIPTION

The **vl** API layer is used for a particular class of vulnerability / lethality methodology (or taxonomy). A vulnerability / lethality methodology is a means by which one divides the set of all possible outcomes for a vulnerability result. The current API only includes the Mobility, Firepower, Catastrophic (MFK) Kill result set. In this set the only possible outcomes of a lethal result are:

Outcome	Meaning
MKILL	Mobility Kill only.
FKILL	Fire Power Kill only.
MFKILL	Both Mobility and Fire Power Kills.
KKILL	Catastrophic Power Kill
NODAMAGE	Probability that no additional damage occurs.

Application programs call an API. They pass enough information to describe the initial conditions of the vulnerability calculation. Internally the VL API will set the appropriate parameters in the VLparam layer (see **vlparam(3)**). Then the VL API will call the appropriate vulnerability "lookup" function (see **db_tbl_reader_func**() from the DIS lethality server's **db(3)** layer) and return the results.

There are three sets of MFK APIs. Each set may be used to retrieve the equivalent V/L results. Which API is selected for use depends on which inputs are expected by a particular API. The inputs required by the three sets are:

API layer name	Type of input expected
vl_mfk_binaryDIS_...	DIS PDUs are input. The PDU format is a continuous binary array containing the DIS PDU data as specified in the standard IEEE 1278.1
vl_mfk_ArlDIS_...	DIS PDUs are input. The PDU format is a data structure particular to the ARL DIS Manager.
vl_mfkDIS_...	Input comprises of the DIS Entity ID of the entity whose vulnerability is being assessed, and the DIS ID of the munition detonation event which is of interest.

API layer names in the table above indicate the first sequence of characters in the name of the indicated functions. Most of the functions are actually proceeded by an underscore (_). If the application program is monitoring the DIS environment then one of the "vl_mfkDIS..." APIs might be the best choice of APIs. If the calling application is a client to the *ARL DIS Manager* (see **dis_mgr(1)**), then it might be most convenient to use the "vl_mfk_ArlDIS_..." set of functions.

Synopsis of the API functions are now given in the following order:

```
/*----------------vl_mfkDIS...() functions---------------------*/
VL_Result vl_mfkDIS_Result(int*flag, DisID *entityID, DisID *eventID);
float*  vl_mfkDIS_ProbAll( DisID *entityID, DisID *eventID );
double  _vl_mfkDIS_ProbK( DisID *entityID, DisID *eventID );
double  _vl_mfkDIS_ProbMF(  DisID *entityID, DisID *eventID );
double  _vl_mfkDIS_ProbF(  DisID *entityID, DisID *eventID );
double  _vl_mfkDIS_ProbM(  DisID *entityID, DisID *eventID );
double  _vl_mfkDIS_ProbNoDamage(  DisID *entityID, DisID *eventID );
/*----------------vl_mfk_ArlDIS...() functions-----------------*/
VL_Result vl_mfk_ArlDIS_Result_NoNet(int*flg, VLSetParam_t itype,  ...);
float* vl_mfk_ArlDIS_ProbAll_NoNet( VLSetParam_t itype,  ... );
double _vl_mfk_ArlDIS_ProbM_NoNet(VLSetParam_t itype,  ...);
```

```
double _vl_mfk_ArlDIS_ProbMF_NoNet(VLSetParam_t itype, ...);
double _vl_mfk_ArlDIS_ProbF_NoNet(VLSetParam_t itype, ...);
double _vl_mfk_ArlDIS_ProbK_NoNet(VLSetParam_t itype, ...);
double _vl_mfk_ArlDIS_ProbNoDamage_NoNet(VLSetParam_t itype, ...);
/*----------------vl_...() Utility functions----------------*/
VL_Result vl_mfk_binaryDIS_Result_NoNet(int *flg, VLSetParam_t itype, ...);
float* vl_mfk_binaryDIS_ProbAll_NoNet( VLSetParam_t itype, ...);
double _vl_mfk_binaryDIS_ProbK_NoNet(VLSetParam_t itype, ...);
double _vl_mfk_binaryDIS_ProbMF_NoNet(VLSetParam_t itype, ...);
double _vl_mfk_binaryDIS_ProbF_NoNet(VLSetParam_t itype, ...);
double _vl_mfk_binaryDIS_ProbM_NoNet(VLSetParam_t itype, ...);
double _vl_mfk_binaryDIS_ProbNoDamage_NoNet(VLSetParam_t itype, ...);
/*----------------vl_mfk_binaryDIS...() functions----------------*/
void _vl_drandom_seed( int seed );
double _vl_drandom(void);
int vl_GetResultErrorValue(void);
int vl_mfk_directFireIsAHit( DetonationResult DIS_det_result );

/*-----------------vl_mfkDIS...()  functions----------------------*/
/*
 ~ vl_mfkDIS_Result()
 *
 * int vl_mfkDIS_Result(int *source, DisID *entityID, DisID *eventID  )
 *
 * This function returns the result of the detonation with
 * identification "eventID" verses the target whose DIS entity ID
 * is "entityID".
 *
 *     The type DisID is a pointer to an array of 3 16-bit integers.
 *     "eventID" refers to the identification used to denote a
 *     fire/detonation event sequence in DIS (i.e. the "Event ID" field
 *     of a DIS Detonation PDU), while "entityID refers to the
 *     identification used to denote an entity for a particular DIS
 *     exercise (i.e. the "entity identification" field of the Entity
 *     State PDU.
 *
 * RETURNS:
 *
 * The FLAG (*flg) parameter is always set.  (See below).
 * The function will always return one of the following results:
 *
 *           PS_MFK_M
 *           PS_MFK_F
 *           PS_MFK_MF
 *           PS_MFK_K
 *           PS_MFK_NODAMAGE
 *
 *           PS_ERROR
 *
 *
 * The naming convention for these results is as follows:
 *
 *  #1_#2_#3
 *
```

```
*  #1  is "PS_" (meant to imply "Probability Space")
*  #2  is used to indicate the analysis method being applied.
*       (in this case the analysis method is to divide the probability
*       space between the M,F,K kills (and combinations) as well as the
*       implied No Damage possibility).
*  #3  is used to indicate a particular event in that space.
*
* Hypothetically, there may be other results returned depending on what is
* defined as the result set for the targetted entity.  (For example for a
* helicopter, a PS_MISSION_ABORT might be a logical addition to a class of
* the result sets).
*
* The integer referenced by the parameter "source" is a flag which is set
* to inform the caller whether the source of the function's result was from
* a valid PKH table or from a less authoritative source.
*
* The "FLAG" (*flg) parameter
* ---------------------------
* The "FLAG" (*flg) parameter is always set with
* one of the following values:
*
*     Value
*    of FLAG            MEANING
*    --------  ------------------------------------------------
*
*       -1      Unknown error.
*
*               A generic pkh result is returned but is not authoritative.
*
*               In this case calling the function rpt_perror() might shed
*               some light on the source of the error.  (This is an
*               internal vlserver library procedure whose purpose is
*               similar to perror()).
*
*       0       Success.
*
*               The pkh source for the referenced entity and threat
*               munition (as defined in the DAMAGE_SOURCE_META_DATA_FILE)
*               was successfully found, interpreted, and used in
*               the calculation of the returned (VL_Result) value.
*
*       1       No Table.
*
*               A generic pkh result is returned but is not authoritative.
*
*               A reference to a vulnerability source could not be found
*               in the DAMAGE_SOURCE_META_DATA_FILE for this
*               combination of entity and threat.
*
*
*       2       Corrupt Table.
*
*               A generic pkh result is returned but is not authoritative.
*
```

```
 *              The the referenced vulnerability source data was found,
 *              however there was an error when attempting to interpret
 *              the data.
 *
 *      3       No Environment Data.
 *
 *              A generic pkh result is returned but is not authoritative.
 *
 *              Data describing the fire and detonation events were
 *              never observed while monitoring the run time
 *              environment.
 *
 *      4       Unknown Target.
 *
 *              A generic pkh result is returned but is not authoritative.
 *
 *              A reference to the threatened (targeted) entity could
 *              not be found in the DIS_ENTITIES_FILE nor in the
 *              DIS_AUXILIARY_ENTITIES_FILE.
 *
 *      5       Unknown Threat.
 *
 *              A generic pkh result is returned but is not authoritative.
 *
 *              A reference to the threat munition could
 *              not be found in the DIS_ENTITIES_FILE nor in the
 *              DIS_AUXILIARY_ENTITIES_FILE.
 *
 *
 * Note: This function depends on the DIS Network having been monitored by
 * the Lethality server long enough to have detected the fire,
 * detonation, and at least one Entity State PDU from both the target and
 * firer.  (Otherwise FLAG will not be set to "Success.").
 *
 * See also:
 *
 *   VL_Result vl_mfk_ArlDIS_Result_NoNet(int*Flag, VLSetParam_t itype, ...)
 *   VL_Result vl_mfk_binaryDIS_Result_NoNet(int*Flag,VLSetParam_t itype, ...)
 *
 */
VL_Result vl_mfkDIS_Result(int*flg, DisID *entityID, DisID *eventID)

/*
 ~ vl_mfkDIS_ProbAll()
 *
 *   float*  vl_mfkDIS_ProbAll( DisID *entityID, DisID *eventID  )
 *
 * This function returns the a static array containing probabilities of
 * certain kill levels.  The indices of the array are as follows:
 *
 * Array    Element (index)
 * Element  Value                      Value  Meaning
 * -------  ---------------     ---------------------------------
 *  0       PS_MFK_M            Mobility Kill only.
```

```
*  1        PS_MFK_F            Fire Power Kill only.
*  2        PS_MFK_MF           Both Mobility and Fire Power Kills.
*  3        PS_MFK_K            Catastrophic Power Kill
*  4        PS_MFK_NODAMAGE     Probability that no additional
*                               damage occurs.
*
*     The type DisID is a pointer to an array of 3 16-bit integers.
*     "eventID" refers to the identification used to denote a
*     fire/detonation event sequence in DIS (i.e. the "Event ID" field
*     of a DIS Detonation PDU), while "entityID refers to the
*     identification used to denote an entity for a particular DIS
*     exercise (i.e. the "entity identification" field of the Entity
*     State PDU.
*
*
* DIAGNOSTICS:
*
*         returns a static array containing additive probabilities of
*                 K,MF,F,M, and No Damage.  The values in the array must
*                 be used be for subsequent calls to this function.
*
*         returns NULL on an error.
*
*
* These values are "additive" (sometimes referred to as
* a thermometer redistribution).  They are added together in the
* following manner.
*
*         p[PS_MFK_M] = Probability of Mobility Kill Only.
*         p[PS_MFK_F] = Probability of Mobility Kill Only
*                         + Probability of Fire Power Kill only.
*         p[PS_MFK_MF] = Probability of Mobility Kill Only
*                         + Probability of Fire Power Kill only
*                         + Both Mobility and Fire Power Kills.
*         p[PS_MFK_K] = Probability of Mobility Kill Only
*                         + Probability of Fire Power Kill only
*                         + Probability of Both Mobility and Fire Power Kills
*                         + Probability of Catastrophic Power Kill.
*         p[PS_MFK_NODAMAGE] = 1.0
*
* That is these values are arranged so that they appear on [0,1] in order
* that one random number may select the event which occurs.
*
*         e.g. if       p(m only) = .10
*                       p(f only) = .10
*                       p(m & f)  = .25
*                       p(k)      = .25
*       (and therefore) p(no damg)= 1.0 - (.1+.1+.25.+.25) = .30
*
*      then a probability event space would be:
*
*
*        | M | F |  M & F  |    K    |  no dmg   |
*        |   |   |         |         |           |
```

```
 *       +---+---+---+---+---+---+---+---+---+---+
 *       0. .1  .2  .3  .4  .5  .6  .7  .8  .9  1.0
 *
 *   resulting in the vector:
 *
 *        p[PS_MFK_M]         = p[0] = .10
 *        p[PS_MFK_F]         = p[1] = .20
 *        p[PS_MFK_MF]        = p[2] = .45
 *        p[PS_MFK_K]         = p[3] = .70
 *        p[PS_MFK_NODAMAGE] = p[4] =1.00
 *
 *  being returned by vl_mfk_ArlDIS_ProbAll_NoNet()
 *
 */
float*  vl_mfkDIS_ProbAll( DisID *entityID, DisID *eventID )
```

The following functions expect the same input as **vl_mfkDIS_ProbAll()** however, unlike **vl_mfkDIS_ProbAll()** they each return only one probability and not all the probabilities of every event occurring.

```
/*
 *
 * double  _vl_mfkDIS_ProbK( DisID *entityID, DisID *eventID );
 *
 *   Returns the probability of a K-KILL (catostrophic kill)
 *   and only a K-KILL.
 *
 *   The parameters passed to this function are the same as passed to
 *   vl_mfkDIS_ProbAll().
 *
 *  RETURNS:
 *      a number LESS THAN 0 on an error,
 *      otherwise the probability of an K-KILL and only a K-KILL.
 *
 */

/*
 * double  _vl_mfkDIS_ProbMF(  DisID *entityID, DisID *eventID );
 *
 * Returns the probability of an MF-KILL (both mobility and firepower kill)
 *   and only an MF-KILL.
 *
 *   The parameters passed to this function are the same as passed to
 *   vl_mfkDIS_ProbAll().
 *
 *  RETURNS:
 *      a number LESS THAN 0 on an error,
 *      otherwise the probability of an MF-KILL and only an MF-KILL.
 *
 */

/*
 * double  _vl_mfkDIS_ProbF(  DisID *entityID, DisID *eventID );
 *
```

```
 *   Returns the probability of an F-KILL (fire power kill)
 *   and only an F-KILL.
 *
 *   The parameters passed to this function are the same as passed to
 *   vl_mfkDIS_ProbAll().
 *
 *  RETURNS:
 *      a number LESS THAN 0 on an error,
 *      otherwise the probability of an F-KILL and only an F-KILL.
 *
 */

/*
 * double  _vl_mfkDIS_ProbM(  DisID *entityID, DisID *eventID );
 *
 *   The parameters passed to this function are the same as passed to
 *   vl_mfkDIS_ProbAll().
 *
 *  RETURNS:
 *      a number LESS THAN 0 on an error,
 *      otherwise the probability of an M-KILL and only an M-KILL.
 */

/*
 * double  _vl_mfkDIS_ProbNoDamage(  DisID *entityID, DisID *eventID );
 *
 *   Returns the probability of an M-KILL (mobility kill)
 *   and only an M-KILL.
 *
 *   The parameters passed to this function are the same as passed to
 *   vl_mfkDIS_ProbAll().
 *
 *  RETURNS:
 *      a number LESS THAN 0 on an error,
 *      otherwise the probability of NoDamage.
 */


/*-----------------vl_mfk_ArlDIS...()  functions------------------*/
/*
 ~ vl_mfk_ArlDIS_Result_NoNet()
 *
 *
 * VL_Result vl_mfk_ArlDIS_Result_NoNet(int*flg, VLSetParam_t itype,  ...)
 *
 * This function returns an MFK kill type (i.e. one of Mobility, Fire
 * Power, Mobility & Fire, Catastrophic Kills, or No Damage) as the result
 * of the interaction of the target and threat.
 *
 * The answer may possibly be "made up" in the event
 *  that there is not enough information to find the correct
 *  answer.  If this is the case, the parameter (flg) is set to
 *  a non-zero number.  (See "The FLAG (*flg) parameter" below).
 *
```

```
* The target, threat, and their environment are determined by the passed
* arguments.  The VLSetParam_t data object (itype) identifies the
* required inputs needed to prepare a vulnerability assessment. (Namely
* it identifies the variables that are being passed, which are required
* to set the initial conditions of the vulnerability assessment.)
*
* In the event that itype == VL_PARAM_SET_METH_DIS_HitToKill
* It indicates that passing the DIS PDUS
*       Entity State (target)
*       Entity State (firer)
*       FirePDU
*       DetonationPDU
*    shall be sufficient to set the VL parameters to return the
*    correct result from the lookup table (or other data source).
*
* (pointers to these data structures will be passed to this routine (after
* itype):
*
*
* This function returns the result of the target/threat/detonation
* interaction
*
* RETURNS:
*
* The FLAG (*flg) parameter is always set.  (See below).
* The function will always return one of the following results:
*
*        PS_MFK_M
*        PS_MFK_F
*        PS_MFK_MF
*        PS_MFK_K
*        PS_MFK_NODAMAGE
*
*        PS_ERROR
*
*
* The naming convention for these results is as follows:
*
*  #1_#2_#3
*
*  #1  is "PS_" (meant to imply "Probability Space")
*  #2  is used to indicate the analysis method being applied.
*       (in this case the analysis method is to divide the probability
*       space between the M,F,K kills (and combinations) as well as the
*       implied No Damage possibility).
*  #3  is used to indicate a particular event in that space.
*
* Hypothetically, there may be other results returned depending on what is
* defined as the result set for the targetted entity.  (For example for a
* helicopter, a PS_MISSION_ABORT might be a logical addition to a class of
* the result sets).
*
* The integer referenced by the parameter "source" is a flag which is set
* to inform the caller whether the source of the function's result was from
```

```
 * a valid PKH table or from a less authoritative source.
 *
 * The "FLAG" (*flg) parameter
 * ---------------------------
 * The "FLAG" (*flg) parameter is always set with
 * one same values as it is set in the function  vl_mfkDIS_Result().
 *
 * Note: This function depends on the DIS Network having been monitored by
 * the Lethality server long enough to have detected the fire,
 * detonation, and at least one Entity State PDU from both the target and
 * firer.  (Otherwise FLAG will not be set to "Success.").
 *
 * See also:
 *
 *   VL_Result vl_mfkDIS_Result(int EntityID, int DetID )
 *   VL_Result vl_mfk_binaryDIS_Result_NoNet(int*Flag,VLSetParam_t itype, ...)
 *
 */
VL_Result vl_mfk_ArlDIS_Result_NoNet(int*flg, VLSetParam_t itype,  ... )

/*
 ~ vl_mfk_ArlDIS_ProbAll_NoNet()
 *
 *  float * vl_mfk_ArlDIS_ProbAll_NoNet( VLSetParam_t itype,  ... )
 *
 * This function returns the a static array containing probabilities of
 * certain kill levels.   The function's behaviour and return values
 * are the same as the function vl_mfkDIS_ProbAll().
 * The only difference is the parameter set passed to the function.
 * This function expects the following parameters:
 *
 * The first parameter argument is of type VLSetParam_t.
 *
 * This type is used to indicate which data sources (inputs)
 * are sufficient to set the VL parameters in order to be able
 * to return the correct result from the lookup table
 * (or other data source).   These indicated data sources
 * shall then be the 2nd, 3rd, 4th, ... etc. parameter arguments
 * to the function.
 *
 * To date there are only two VLSetParam_t types defined:
 *
 *      VL_PARAM_SET_METH_DIS_HitToKill
 *      VL_PARAM_SET_METH_DIS_ProxKill
 *
 * They both require the same arguments.  Namely, pointers to
 * the DIS PDUS:
 *
 *       Entity State (for the target),
 *       Entity State (for the firer),
 *       Fire PDU,
 *       and Detonation PDU.
 *
 *  These PDUs will be the 2nd, 3rd, 4th, and 5th arguments to the
```

```
 *  function.  These PDUs must be in the
 *  ARL DIS Manager PDU data structure format.
 *  (See  vl_mfk_binaryDIS_ProbAll_NoNet() for passing other
 *        PDU data structures).
 *
 *  RETURNS:
 *
 *     Same as vl_mfkDIS_ProbAll().
 *
 *  SEE ALSO:
 *      float * vl_mfk_ArlDIS_ProbAll_NoNet( VLSetParam_t itype,  ...)
 *      float*  vl_mfkDIS_ProbAll( EntityID, DetID )
 *
 *        ************************************************
 *        *                                              *
 *        * NOTE:                                        *
 *        *                                              *
 *        * PDU arguments are in the form of a ARL DIS   *
 *        * Manager PDU data structure format.           *
 *        *                                              *
 *        ************************************************
 *
 */
float * vl_mfk_ArlDIS_ProbAll_NoNet( VLSetParam_t itype,  ... )
```

The following functions expect the same input as **vl_mfk_ArlDIS_ProbAll()** however, unlike **vl_mfk_ArlDIS_ProbAll()** they each return only one probability and not all the probabilities of every event occurring.

```
/*
 ~ _vl_mfk_ArlDIS_ProbK_NoNet(VLSetParam_t itype,  ...)
 *
 * double _vl_mfk_ArlDIS_ProbK_NoNet(VLSetParam_t itype,  ...)
 *
 *   Returns the probability of a K-KILL (catostrophic kill)
 *   and only a K-KILL.
 *
 *   The parameters passed to this function are the same as passed to
 *       vl_mfk_ArlDIS_Result_NoNet()
 *   and vl_mfk_ArlDIS_ProbAll_NoNet()
 *
 *  Namely an indentifier (itype) telling the server which
 *  parameters are needed to set the initail state of the vulnerabilty
 *  analysis (and hence which variables will follow as arguments).
 *
 *  RETURNS:
 *      a number LESS THAN 0 on an error,
 *      otherwise the probability of an K-KILL and only a K-KILL.
 *
 */

/*
 ~ _vl_mfk_ArlDIS_ProbMF_NoNet(VLSetParam_t itype,  ...)
 *
```

```
 * double _vl_mfk_ArlDIS_ProbMF_NoNet(VLSetParam_t itype,  ...)
 *
 * Returns the probability of an MF-KILL (both mobility and firepower kill)
 *   and only an MF-KILL.
 *
 *   The parameters passed to this function are the same as passed to
 *       vl_mfk_ArlDIS_Result_NoNet()
 *   and vl_mfk_ArlDIS_ProbAll_NoNet()
 *
 *  Namely an indentifier (itype) telling the server which
 *  parameters are needed to set the initail state of the vulnerabilty
 *  analysis (and hence which variables will follow as arguments).
 *
 *  RETURNS:
 *      a number LESS THAN 0 on an error,
 *      otherwise the probability of an MF-KILL and only an MF-KILL.
 *
 */


/*
 ~ _vl_mfk_ArlDIS_ProbF_NoNet(VLSetParam_t itype,  ...)
 *
 * double _vl_mfk_ArlDIS_ProbF_NoNet(VLSetParam_t itype,  ...)
 *
 *   Returns the probability of an F-KILL (fire power kill)
 *   and only an F-KILL.
 *
 *   The parameters passed to this function are the same as passed to
 *       vl_mfk_ArlDIS_Result_NoNet()
 *   and vl_mfk_ArlDIS_ProbAll_NoNet()
 *
 *  Namely an indentifier (itype) telling the server which
 *  parameters are needed to set the initail state of the vulnerabilty
 *  analysis (and hence which variables will follow as arguments).
 *
 *  RETURNS:
 *      a number LESS THAN 0 on an error,
 *      otherwise the probability of an F-KILL and only an F-KILL.
 *
 */


/*
 ~ _vl_mfk_ArlDIS_ProbM_NoNet(VLSetParam_t itype,  ...)
 *
 * double _vl_mfk_ArlDIS_ProbM_NoNet(VLSetParam_t itype,  ...)
 *
 *   Returns the probability of an M-KILL (mobility kill)
 *   and only an M-KILL.
 *
 *   The parameters passed to this function are the same as passed to
 *       vl_mfk_ArlDIS_Result_NoNet()
 *   and vl_mfk_ArlDIS_ProbAll_NoNet()
 *
 *  Namely an indentifier (itype) telling the server which
```

```
 *  parameters are needed to set the initail state of the vulnerabilty
 *  analysis (and hence which variables will follow as arguments).
 *
 *  RETURNS:
 *      a number LESS THAN 0 on an error,
 *      otherwise the probability of an M-KILL and only an M-KILL.
 */

/*
 ~ _vl_mfk_ArlDIS_ProbNoDamage_NoNet(VLSetParam_t itype,  ...)
 *
 * double _vl_mfk_ArlDIS_ProbNoDamage_NoNet(VLSetParam_t itype,  ...)
 *
 *   Returns the probability of No further Damage
 *   occuring and and only No further Damage occuring.
 *
 *   The parameters passed to this function are the same as passed to
 *       vl_mfk_ArlDIS_Result_NoNet()
 *   and vl_mfk_ArlDIS_ProbAll_NoNet()
 *
 *  Namely an indentifier (itype) telling the server which
 *  parameters are needed to set the initail state of the vulnerabilty
 *  analysis (and hence which variables will follow as arguments).
 *
 *  RETURNS:
 *      a number LESS THAN 0 on an error,
 *      otherwise the probability of NoDamage.
 */


/*-----------------vl_mfk_binaryDIS...()  functions---------------*/
/*
 ~ vl_mfk_binaryDIS_Result_NoNet()
 *
 *
 * VL_Result vl_mfk_binaryDIS_Result_NoNet( VLSetParam_t itype,  ...)
 *
 * This function returns an MFK kill type (i.e. one of Mobility, Fire
 * Power, Mobility & Fire, Catastrophic Kills, or No Damage) as the result
 * of the interaction of the target and threat.
 *
 * The target, threat, and their environment are determined by the passed
 * arguments.  The VLSetParam_t data object (itype) identifies the
 * required inputs needed to prepare a vulnerability assessment. (Namely
 * it identifies the variables that are being passed, which are required
 * to set the initial conditions of the vulnerability assessment.)
 *
 * In the event that itype == VL_PARAM_SET_METH_DIS_HitToKill
 * It indicates that passing the DIS PDUS
 *       Entity State (target)
 *       Entity State (firer)
 *       FirePDU
 *       DetonationPDU
 *
```

```
 *    shall be sufficient to set the VL parameters to return the
 *    correct result from the lookup table (or other data source).
 *
 * The passed arguments to the function are pointers to DIS PDU
 * structures whose bits are packed into a continuous binary string
 * of bytes.  (That is, the arguments are pointers
 * to the binary PDU string argument as it would "appear" as a DIS
 * broadcast PDU on the DIS network).
 *
 * This function returns the result of the target/threat/detonation
 * interaction
 *
 * one of the following results are returned:
 *
 *        PS_MFK_M
 *        PS_MFK_F
 *        PS_MFK_MF
 *        PS_MFK_K
 *        PS_MFK_NODAMAGE
 *
 *        PS_ERROR
 *
 * The vl_library random number generator (vl_drandom()) is used to
 * randomly select a point in the probability space (and hence return the
 * "result".
 *
 * See also:
 *
 *       VL_Result vl_mfkDIS_Result(int*flg, int EntityID, int DetID )
 *       VL_Result vl_mfk_ArlDIS_Result_NoNet()
 *
 *        ************************************************
 *        *                                              *
 *        * NOTE:                                        *
 *        *                                              *
 *        * PDU arguments are in the form of a continuous *
 *        * BINARY STRING.                               *
 *        *                                              *
 *        ************************************************
 *
 */
VL_Result vl_mfk_binaryDIS_Result_NoNet(int*flg, VLSetParam_t itype,  ... )

/*
 ~ vl_mfk_binaryDIS_ProbAll_NoNet()
 *
 *  float * vl_mfk_binaryDIS_ProbAll_NoNet( VLSetParam_t itype,  ... )
 *
 * This function returns the a static array containing probabilities of
 * certain kill levels.   The function's behaviour and return values
 * are the same as the function vl_mfkDIS_ProbAll().
 * The only difference is the parameter set passed to the function.
 * This function expects the following parameters:
 *
```

```
* The first parameter argument is of type VLSetParam_t.
*
* This type is used to indicate which data sources (inputs)
* are sufficient to set the VL parameters in order to be able
* to return the correct result from the lookup table
* (or other data source).   These indicated data sources
* shall then be the 2nd, 3rd, 4th, ... etc. parameter arguments
* to the function.
*
* To date there are only two VLSetParam_t types defined:
*
*      VL_PARAM_SET_METH_DIS_HitToKill
*      VL_PARAM_SET_METH_DIS_ProxKill
*
* They both require the same arguments.  Namely, pointers to
* the DIS PDUS:
*
*       Entity State (for the target),
*       Entity State (for the firer),
*       Fire PDU,
*       and Detonation PDU.
*
*  These PDUs will be the 2nd, 3rd, 4th, and 5th arguments to the
*  function.  These PDUs must be in "binary format" that is in
*  the same bit format that DIS PDUs are trasported in when
*  communicated on the DIS network.  This means that each
*  PDU argument points an address contains a continuous
*  array of bits representing the content of the PDU.
*
*  (See  vl_mfk_ArlDIS_ProbAll_NoNet() for passing other
*        PDU data structures).
*
* Example:
*
*    assumes    tgt_entity_state_pdu, shooter_entity_state_pdu,
*                fire_pdu, detonation_pdu
*               are pointers (void *)
*
*   {  float *probability_space;
*
*      probability_space =
*         vl_mfk_binaryDIS_ProbAll_NoNet(
*                VL_PARAM_SET_METH_DIS_HitToKill
*            , tgt_entity_state_pdu
*            , shooter_entity_state_pdu
*            , fire_pdu
*            , detonation_pdu
*         );
*   }
*
*  RETURNS:
*
*     Same as vl_mfkDIS_ProbAll().
*
```

```
 *  SEE ALSO:
 *      float * vl_mfk_ArlDIS_ProbAll_NoNet( VLSetParam_t itype,  ...)
 *      float*  vl_mfkDIS_ProbAll( EntityID, DetID )
 *
 *        ************************************************
 *        *                                              *
 *        * NOTE:                                        *
 *        *                                              *
 *        * PDU arguments are in the form of a continuous *
 *        * BINARY STRING.                               *
 *        *                                              *
 *        ************************************************
 *
 */
float * vl_mfk_binaryDIS_ProbAll_NoNet( VLSetParam_t itype,  ... )
```

The following functions expect the same input as **vl_mfk_binaryDIS_ProbAll()** however, unlike **vl_mfk_binaryDIS_ProbAll()** they each return only one probability and not all the probabilities of every event occurring.

```
/*
 ~ _vl_mfk_ArlDIS_ProbK_NoNet(VLSetParam_t itype,  ...)
 *
 * double _vl_mfk_ArlDIS_ProbK_NoNet(VLSetParam_t itype,  ...)
 *
 *   Returns the probability of a K-KILL (catostrophic kill)
 *   and only a K-KILL.
 *
 *   The parameters passed to this function are the same as passed to
 *       vl_mfk_ArlDIS_Result_NoNet()
 *   and vl_mfk_ArlDIS_ProbAll_NoNet()
 *
 *  Namely an indentifier (itype) telling the server which
 *  parameters are needed to set the initail state of the vulnerabilty
 *  analysis (and hence which variables will follow as arguments).
 *
 *  RETURNS:
 *      a number LESS THAN 0 on an error,
 *      otherwise the probability of an K-KILL and only a K-KILL.
 *        ************************************************
 *        *                                              *
 *        * NOTE:                                        *
 *        *                                              *
 *        * PDU argurments are in the form of a continuous*
 *        * BINARY STRING.  the argurment list consists of*
 *        * a pointer (to the PDU in binary form) followed*
 *        * by the length of the PDU string (in bytes)    *
 *        *                                              *
 *        *  see: vl_mfk_binaryDIS_...() functions       *
 *        *                                              *
 *        ************************************************
 *
 */
```

```
/*
 ~ _vl_mfk_ArlDIS_ProbMF_NoNet(VLSetParam_t itype,  ...)
 *
 * double _vl_mfk_ArlDIS_ProbMF_NoNet(VLSetParam_t itype,  ...)
 *
 * Returns the probability of an MF-KILL (both mobility and firepower kill)
 *   and only an MF-KILL.
 *
 *   The parameters passed to this function are the same as passed to
 *       vl_mfk_ArlDIS_Result_NoNet()
 *   and vl_mfk_ArlDIS_ProbAll_NoNet()
 *
 *  Namely an indentifier (itype) telling the server which
 *  parameters are needed to set the initail state of the vulnerabilty
 *  analysis (and hence which variables will follow as arguments).
 *
 *  RETURNS:
 *      a number LESS THAN 0 on an error,
 *      otherwise the probability of an MF-KILL and only an MF-KILL.
 *
 */

/*
 ~ _vl_mfk_ArlDIS_ProbF_NoNet(VLSetParam_t itype,  ...)
 *
 * double _vl_mfk_ArlDIS_ProbF_NoNet(VLSetParam_t itype,  ...)
 *
 *   Returns the probability of an F-KILL (fire power kill)
 *   and only an F-KILL.
 *
 *   The parameters passed to this function are the same as passed to
 *       vl_mfk_ArlDIS_Result_NoNet()
 *   and vl_mfk_ArlDIS_ProbAll_NoNet()
 *
 *  Namely an indentifier (itype) telling the server which
 *  parameters are needed to set the initail state of the vulnerabilty
 *  analysis (and hence which variables will follow as arguments).
 *
 *  RETURNS:
 *      a number LESS THAN 0 on an error,
 *      otherwise the probability of an F-KILL and only an F-KILL.
 *
 */

/*
 ~ _vl_mfk_ArlDIS_ProbM_NoNet(VLSetParam_t itype,  ...)
 *
 * double _vl_mfk_ArlDIS_ProbM_NoNet(VLSetParam_t itype,  ...)
 *
 *   Returns the probability of an M-KILL (mobility kill)
 *   and only an M-KILL.
 *
 *   The parameters passed to this function are the same as passed to
 *       vl_mfk_ArlDIS_Result_NoNet()
```

```
 *   and vl_mfk_ArlDIS_ProbAll_NoNet()
 *
 *  Namely an indentifier (itype) telling the server which
 *  parameters are needed to set the initail state of the vulnerabilty
 *  analysis (and hence which variables will follow as arguments).
 *
 *  RETURNS:
 *      a number LESS THAN 0 on an error,
 *      otherwise the probability of an M-KILL and only an M-KILL.
 */

/*
 ~ _vl_mfk_ArlDIS_ProbNoDamage_NoNet(VLSetParam_t itype,  ...)
 *
 * double _vl_mfk_ArlDIS_ProbNoDamage_NoNet(VLSetParam_t itype,  ...)
 *
 *   Returns the probability of No further Damage
 *   occuring and and only No further Damage occuring.
 *
 *   The parameters passed to this function are the same as passed to
 *       vl_mfk_ArlDIS_Result_NoNet()
 *   and vl_mfk_ArlDIS_ProbAll_NoNet()
 *
 *  Namely an indentifier (itype) telling the server which
 *  parameters are needed to set the initail state of the vulnerabilty
 *  analysis (and hence which variables will follow as arguments).
 *
 *  See Also:
 *     double _vl_mfk_binaryDIS_ProbNoDamage_NoNet(VLSetParam_t itype,  ...)
 *     double _vl_mfk_binaryDIS_ProbNoDamage_NoNet(VLSetParam_t itype, ...)
 *     double _vl_mfkDIS_ProbNoDamage( EntityID, DetID )
 *
 *  RETURNS:
 *      a number LESS THAN 0 on an error,
 *      otherwise the probability of NoDamage.
 */
```

The remaining API calls are utility functions.

_vl_drandom() and **_vl_drandom_seed()** are used to generate pseudo random numbers. The random number generater is used to select returned results for the *"Result"* functions (**vl_mfkDIS_Result()**, **vl_mfk_ArlDIS_Result_NoNet()**, and **vl_mfk_binaryDIS_Result_NoNet()**). The function **vl_GetResultErrorValue()** may be used to debug at an application level when unexected results are returned. **vl_mfk_directFireIsAHit()** applies specifically to "direct fire" munitions to see if they hit the target in question. The server uses this function to possibly overide the look-up table results. For example if a bullet lands within a lethal radious of a target, however, the target is oriented in such a manner that the bullet actually missed. (And hence, **vl_mfk_directFireIsAHit()** returns a *FALSE* result. Then the server will overide the look-up table's damage result and return a "No Damage" result instead. Assuming, of course, that a hit is *required* to cause damage. **vl_mfk_directFireIsAHit()** does no geometric calculations but rather it is reliant on the "detonation result" field of the deonation PDU to determine if a bullet misses of hits a target. **VL_mfkDIS_ResultGenericRandomDraw()** is used to return (an invalid) result in the event that look-up tables or valid data cannot be accessed for some reason.

```
/*-----------------vl_...()  Utility  functions------------------*/
/*
 ~ _vl_drandom()
 *
 *   double _vl_drandom(void)
 *
 *  returns a random number on [0,1]
 *
 * See also:  _vl_drandom_seed()
 *   (to re-seed pseudo random sequence).
 *
 */
double _vl_drandom(void)

/*
 ~ _vl_drandom_seed()
 *
 *  void _vl_drandom_seed( int seed )
 *
 *  re-seeds the pseudo random number sequence using the integer "seed"
 *  (Don't use a seed of 0).
 *  See also:
 *      _vl_drandom()
 */
void _vl_drandom_seed( int seed )

/*
 ~ vl_GetResultErrorValue();
 *
 *  int vl_GetResultErrorValue(void);
 *
 *  Called only (and immediately) after an unsuccessful attempt
 *  has been made to one of the
 *        "float *vl_mfk_SOMETHING_ProbAll(...)" functions
 *
 *   (e.g. calling any one of:
 *      float * vl_mfkDIS_ProbAll( DisID *entityID, DisID *eventID );
 *      float * vl_mfk_ArlDIS_ProbAll_NoNet( VLSetParam_t itype,  ...)
 *      float * vl_mfk_binaryDIS_ProbAll_NoNet( VLSetParam_t itype,  ...)
 *    have returned a NULL
 *
 *  RETURNS an error value (as follows:
 *
 *      VL_RSLT_ERR_GENERAL             /* Unknown error. */
 *      VL_RSLT_SUCCESS                 /* success - NO error.  */
 *      VL_RSLT_ERR_NO_TABLE            /* No Table. */
 *      VL_RSLT_ERR_CURRUPT_TABLE       /* Corrupt Table. */
 *      VL_RSLT_ERR_NO_ENVIRON_DATA     /* No Environment Data. */
 *      VL_RSLT_ERR_UNKNOWN_TARGET      /* Unknown Target. */
 *      VL_RSLT_ERR_UNKNOWN_THREAT      /* Unknown Threat. */
 *
 */
int vl_GetResultErrorValue(void)
```

```
/*
 ~ vl_mfk_directFireIsAHit()
 *
 *  int vl_mfk_directFireIsAHit( DetonationResult DIS_det_result )
 *
 *  This function returns TRUE if the result field indicates a
 *    "hit" with a direct fire (or prox-fuze) weapon.
 *
 *  The argument DIS_det_result is the IEEE 1278.1 (DIS) Detonation
 *   Result field (An 8 bit unsigned integer)
 *   [the type DetonationResult is defined in the ARL DIS Manager
 *    "pdu_basic.h"].
 *
 *   Note: for a kinetic energy munitions
 *   (That is type VL_Meth = "DIS HitToKill"
 *   in the DAMAGE_SOURCE_META_DATA_FILE).
 *   only:
 *         1    Entity Impact
 *         2    Entity Proximate Detonation
 *         5    Detonation
 *
 *   will currently have an effect on
 *   the targetted entity.
 *
 */
int vl_mfk_directFireIsAHit( DetonationResult  DIS_det_result )

/*
 ~ VL_mfkDIS_ResultGenericRandomDraw()
 *
 *  VL_Result VL_mfkDIS_ResultGenericRandomDraw(void)
 *
 * A generic probabilility of a kill given a hit
 *   result is returned (but is not authoratative).
 *  Results are drawn from a fixed distribution.
 *
 * This is used as the default answer when a target or threat
 * is unknown, or else if the V/L data set cannot be found or
 * can be found but cannot be interpreted correctly.
 *
 *  RETURNS
 *
 *    An "MFK" probability result (as described
 *    in vl_mfkDIS_Result() ).  I.E. one of the
 *    following values are returned
 *
 *           PS_MFK_M
 *           PS_MFK_F
 *           PS_MFK_MF
 *           PS_MFK_K
 *           PS_MFK_NODAMAGE
 */
VL_Result VL_mfkDIS_ResultGenericRandomDraw(void)
```

SEE ALSO

Other DIS Lethality server components:

vlparam(3) db(3)

AUTHOR

Geoff Sauerborn <*geoffs@arl.mil*> , US Army Research Lab. 1997, 1998.

NAME

vlp_setp_all_Munition_Frm_DIS, vlp_zero_all_params, vlp_print_all_params

VLP_ang_aspect, VLP_ang_attack ,VLP_tvel[3], VLP_impact[3], VLP_range, VLP_result, VLP_target_id, VLP_target_type, VLP_firer_id, VLP_firer_type, VLP_threat_id, VLP_threat_type

SYNOPSIS

```
#include <vlparam.h>

extern Float32 VLP_ang_aspect, VLP_ang_attack ,VLP_tvel[3], VLP_impact[3];
extern Float64   VLP_range;
extern int       VLP_result;
extern EntityID  VLP_target_id;
extern EntityType VLP_target_type;
extern EntityID  VLP_firer_id;
extern EntityType VLP_firer_type;
extern EntityID  VLP_threat_id;
extern EntityType VLP_threat_type;

int vlp_setp_all_Munition_Frm_DIS(PDU*, PDU*, PDU*, PDU*);/* Arl DIS Manager */
void vlp_zero_all_params(void);  /* initialize parameters */
void vlp_print_all_params(void); /* Display settings of all VL parameters */
```

DESCRIPTION

The **Vulnerabilty Parameter sub-layer** of the DIS lethality server. This layer is the means by which vulnerability data look-up functions determine the initial conditions for the vulnerability calculation.

The **VLParam** layer serves to insolate the V/L result data from the virtual environment (initial conditions to the vulnerability analysis). (This separation allows different DIS networking packages to be swapped in an out of the server; and even allowing for a non-DIS networking paradigm, such as HLA, to be used).

When a new look-up function is written for data of a certain format, the author of the look-up function will have to reference this layer. The variables defined here are the only means by which the initial conditions relevant to the V/L calculation may be determined.

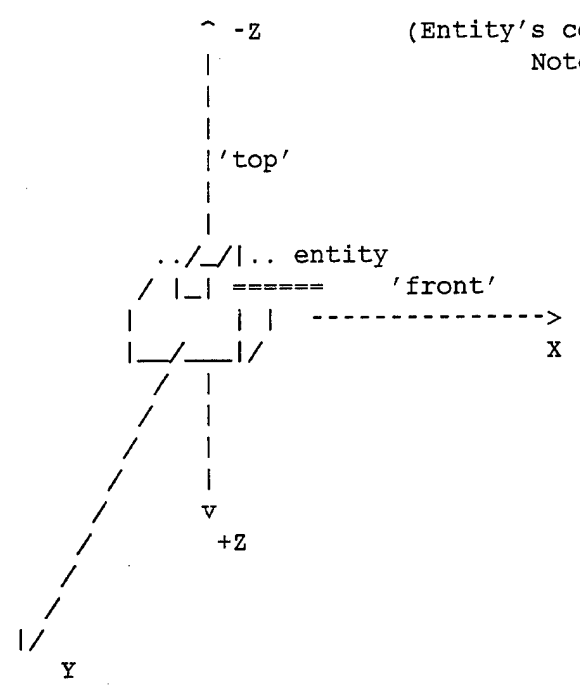
If there are crucial parameters missing from this layer, then those parameters will have to be added. Following this, API functions which set these parameters (prior to calling the look-up function) need to be modified or added to the **VLParam(3)** API layer of the lethality server. *As an example the source code for the API function* **vlp_setp_all_Munition_Frm_DIS()** *may be examined. The function* **vlp_setp_all_Munition_Frm_DIS()** *sets the VLparam layer parameters based on inputs provided by Entity State, Fire, and Detonation PDUs.* Finally, the reader (result look-up function) will have to be modified (or written) that can access the newly added parameters in order to look-up the proper results from the lethality data set.

Data Dictionary:

Note: "target" refers to the queried entity who's resulting vulnerability is being asked for of the server.

```
EntityID - is the DIS Entity ID record (site, application, entity).
     IEEE 1278.1

float32 - 32bit floating point number.
float64 - 64bit floating point number.
enum    - a unitless enumerated value.
```

Entity Coordinate system:



PARAMETER	TYPE	UNITS	MEANING
VLP_ang_aspect	float32	radians	'horizontal' orientation of munition's directional attack (aspect angle) (relative to the target entity). (Rotation is about the target entity's positive "z" axis in a clockwise direction). The positive direction of rotation about an axis is defined as clockwise when viewed towards the positive direction along the axis of rotation. Clockwise direction: For example, a 90 degree (PI/2 radian) clockwise rotation about the z axis will make the positive x-axis co-linear to where the positive y-axis was before the rotation. Targetted entity's coordinate system is that of IEEE 1278.1 with the positive X-Axis axis extending from the "front" of the entity. Positive Z-Axis extending "down". Positive Y-Axis extending out of the entity's "right". See Also: VLP_ang_attck.

VLP_ang_attck	float32	radians	"Angle of attack". 'vertical' orientation of munition's directional attack (aspect angle) (relative to the target). (Rotation is about the target entity's new "X" axis after having been rotated by the angle VLP_ang_aspect. See Also: VLP_ang_aspect.
VLP_impact[3]	float32	meters	Location of munition impact point relative to the target. Location is in target entity's coordinate system. (IEEE 1278.1)
VLP_tvel[3]	float32	m/s	Terminal velocity of the munition immediately before impact. This is in the DIS world coordinate system linear velocity vector record 1278.1 Units are in meters per second. (Same as the "velocity" field of the DIS Detonation PDU).
VLP_range	float64	meters	(line of sight) range from the target to the origin of the munition. (i.e. distance from where the munition was fired to where it detonated). The DIS Standard states that the "range" field in the Fire PDU is set to 0 if the range is unknown. If this is the case, then the VL server shall attempt to guess at the approximate range by setting the VLP_range to the distance between the target and firing entity (if known). If this approximation fails for some reason, then VLP_Range shall remain set to 0.
VLP_result	int	enum	result of the detonation (if known) the enumeration are according to the DIS standard (IEEE 1278.1) Note: for a kinetic energy munitions (That is type VL_Meth = "DIS HitToKill" in the DAMAGE_SOURCE_META_DATA_FILE). only: 1 Entity Impact 2 Entity Proximate Detonation 5 Detonation will currently have an effect on the targetted entity.

Value	Description
0	Other
1	Entity Impact
2	Entity Proximate Detonation
3	Ground Impact
4	Ground Proximate Detonation
5	Detonation
6	None
7	HE hit, small
8	HE hit, medium
9	HE hit, large
10	Armor-piercing hit
11	Dirt blast, small
12	Dirt blast, medium
13	Dirt blast, large
14	Water blast, small
15	Water blast, medium
16	Water blast, large
17	Air hit
18	Building hit, small
19	Building hit, medium
20	Building hit, large
21	Mine-clearing line charge
22	Environment object impact
23	Environment object proximate detonation
24	Water Impact
25	Air Burst

VLP_target_id EntityID enum — Targeted Entity's ID. (site, host, id)
If there was an entity impact
indicated by the= VLP_result field,
then this is the entity which was
impacted.

VLP_target_type EntityType enum — Type of entity Targeted. (Entity Enumeration)
If there was an entity impact
indicated by the= VLP_result field,
then this is the type of entity which
was impacted (e.g. "T72M1", "M48").

VLP_threat_id EntityID enum — Threat Entity's ID. (site, host, id)
If the treating object (impacting or
detonating object) is an entity,
then this is its Entity's ID. (site, host, id)
(Normally the threat is not an entity,
but an inanimate munition, in which case
the VLP_threat_id

VLP_threat_type EntityType enum — Type of threating object. (Entity Enumeration)
(Normally the threat is a munition
in which case this field will be derived
from the DIS Burst descriptor field of
the detonation and fire PDUs).

VLP_firer_id	EntityID enum	If the originating entity (the shooter) can be determined, then this is its entity ID.
VLP_firer_type	EntityType enum	If the originating entity (the shooter) can be determined, then this field identifies the DIS entity type.

The vulnerability table reader function must derive all of it's required environmental information from these data structures. If it requires additional environmental data, then the lethality server code will have to be modified to provide that data.

Synopsis of the API functions are now given in the following order:

vlp_setp_all_Munition_Frm_DIS()
vlp_zero_all_params()
vlp_print_all_params()

```
/*
 ~ vlp_setp_all_Munition_Frm_DIS()
 *
 *  int vlp_setp_all_Munition_Frm_DIS( PDU *es_tgt,
 *                                PDU *es_firer, PDU *pfire, PDU *pdet)
 *
 *  Map all DIS data (from the pdu's) to their appropriate parameter.
 *  *NOTE* Assumes PDUs are pointers to ARL DIS Manager PDU structures.
 *
 * This function uses the data found in the Target and Firer's
 * Entity State, the Fire, and the Detonation Protocol Data Units (PDUs)
 * to set the appropriate VLparam layer parameters.
 *
 *  Returns 0 on an error.
 *
 */

/*
 ~ vlp_zero_all_params()
 *
 *  void vlp_zero_all_params(void)
 *
 *  initialize all parameters.
 *  sets to zero all parameters in the VLP layer.
 *
 */

/*
 ~ vlp_print_all_params()
 *
 *  void vlp_print_all_params(void)
 *
 *  Display settings of all VL parameters values.
 *   These values are used by table lookup functions to parse
 *   their individual vulnerability tables.  See definitions for all
 *   the named variable in the DIS Lethality server's data dictionary.
```

```
*
*   See Also:
*           vlp_zero_all_params();
*           _vlp_setp_....functions();
*
*/
```

BUGS

The global variables should be hidden. APIs should be written instead to set and get parameter values. (e.g. _vlp_setp_target_id(), _vlp_getp_target_id()). Most of these have already been written, but since they the set was incomplete, the naked global variables are left exposed for now. (Another reason for leaving them exposed for now is that access time is slightly faster when not having to go through the overhead of calling an extra function layer).

SEE ALSO

Other DIS Lethality server components:

vl(3) db(3)

AUTHOR

Geoff Sauerborn <*geoffs@arl.mil*> , US Army Research Lab. 1997, 1998.

NAME

dis_mon

DESCRIPTION

vls_db_init.ini (This is the same initialization file needed by the **data manager (db)** API layer of the DIS Lethality server. This is an ISO 646, 7 bit (ASCII) file. Lines whose first non-white spaces character is the "#" symbol are ignored. The file consists of name-value pairs which define the file names to be read in order to initialize certain data structures used by the DIS Lethality server's **data manager (db)** API layer. File names specified in **vls_db_init.ini** are assumed to be found relative to the $VLS_HOME/Data/Init/ directory. **VLS_HOME** is an environmental variable which must be set to the root (home) directory of the DIS Lethality server. If not set, then the value of $VLS_HOME is taken as the current working directory ("./").

An example **vls_db_init.ini** initialization file follows:

```
# $Id: vls_db_init.5,v 0.5 1999/01/21 21:27:50 geoffs Exp geoffs $
#
# This file is input to db_init() routine.
#  e.g. if this file was called "./vls_db_init.ini",
#  then calling the API db_init():
#
#        db_init( "./vls_db_init.ini" );
#
#  would initialize the database.
#
#  syntax:
#
#   DIS_ENTITIES_FILE - the file containing all DIS Entity ID's
#                       file's format in is comma separated fields.
#                       (there are 15 fields.  The last field is the
#                        record number.  the first 14 fields comprise
#                       the 7 DIS "Entity Type" enumerations. These are
#                       ordered in pairs of enumeration integer,
#                       followed by ascii text enumeration description.
#
#  DIS_AUXILIARY_ENTITIES_FILE - user added (or non-standard dis entity ID's.
#
#  DAMAGE_SOURCE_META_DATA_FILE - contains pointers to where lookup
#                       tables can be found.
#
DIS_ENTITIES_FILE               ./dis2_0_4_ids.csf
DIS_AUXILIARY_ENTITIES_FILE     ./dis_entities_aux.csf
DAMAGE_SOURCE_META_DATA_FILE    ./tst_tbls_HEAT.csf
```

Identifiers for the name-value pairs are as follows:

DIS_ENTITIES_FILE The file name which follows this keyword identifies the file containing all DIS Entity Enumerations. Currently the entity enumeration file format in is comma separated fields. One record appears per line. There are 15 fields per record. The last field is the record number. the first 14 fields comprise the 7 DIS "Entity Type" enumeration fields. These are ordered in pairs of enumeration integer, followed by ASCII text enumeration description. The enumeration integers correspond to the value for the 7 DIS "Entity Type" enumeration fields (Kind, Domain, etc.) as see in the table below.

DIS "Entity Type" Enumeration	
Field Name	Bit Length
Kind	8
Domain	8
Country	16
Category	8
Subcategory	8
Specific	8
Extra	8

The field that follows each enumeration containts a brief quoted text string to name to that enumeration value. Enumeration values and their names placed in the **DIS_ENTITIES_FILE** file should only be enumerations found in the DIS enumeration standard (see: http://siso.sc.ist.ucf.edu/dis/dis-dd/). The **DIS_AUXILIARY_ENTITIES_FILE** file can be used to add non-standard and experimental enumerations.

A (very) short excerpt of an example **DIS_ENTITIES_FILE** file follows:

```
#
# DIS 2.0.4 Enumerations   (1995)
#   contained in the IEEE 1278.1-1995 Standard for DIS
#
# Derived from the DIS Data Dictionary.
1,"Platform",1.00,"Land",225,"United States",1.00,"Tank",1.00," M1 Abrams",1.00,"M1 Abrams",1.00,"VERSION 5",1
1,"Platform",1.00,"Land",225,"United States",1.00,"Tank",2.00," M60 Main Battle Tank (MBT)",1.00," M60A3",1.00,"",2
1,"Platform",1.00,"Land",225,"United States",1.00,"Tank",4.00," M48 medium tank",1.00," M48C",1.00,,3
1,"Platform",1.00,"Land",225,"United States",1.00,"Tank",4.00," M48 medium tank",2.00," M48A1",1.00,"",4
```

DIS_AUXILIARY_ENTITIES_FILE Identifies the file to be used for adding any additional (auxiliary) DIS enumerations. Sometimes it is handy for adding exercise specific entities or entities not in the latest release of the DIS enumeration standard. The **DIS_AUXILIARY_ENTITIES_FILE** file follows the same format as the **DIS_ENTITIES_FILE** file.

DAMAGE_SOURCE_META_DATA_FILE This denotes the file which contains references to lethality data sources. This tells the the data manager where to find look-up tables associated with different target/threat combinations. The data format found in this file is currently a comma-separated field list. There are five fields which identify the following items.

The threatened entity (in DIS standard enumeration).
The threat (in DIS standard enumeration).
The type of V/L analysis method to be used (i.e. MFK direct or indirect fire).
The table format identifier.
The table's location (in URL format).

The V/L analysis method must be a "quoted" string as identified in the array VL_Meth_List[] (source code $VLS_HOME/src/Db/vl_meth.h). A short excerpt from a **DAMAGE_SOURCE_META_DATA_FILE** follows:

```
#
#  DIS enumerations are IEEE 1278.1-1995 Standard.
#      Note that the file URL location is taken relative
#      to the $VLS_HOME directory.
#--next line's tgt and threat are:  Soviet 125mm KE Threat VS. a T-80 target.
1 1 222 1 1 1,  2 2 222 2 11,"DIS HitToKill","IUA_KE","file:/Data/Tables/IUA/smplKE.iua"
```

```
#--next line's tgt and threat are:  Soviet 120mm HEAT-FS VS. a T-80 tgt.
1 1 222 1 1 1,          2 2 222 2 18,"DIS HitToKill","IUA_HEAT","file:/Data/Tables/IUA/smpl
#--next line's tgt and threat are:  AT-5 Spandrel missile VS. a T-80 tgt.
1 1 222 1 1 1,          2 2 222 1  7,"DIS HitToKill","IUA_HEAT","file:/Data/Tables/IUA/smpl
```

Note that High Explosive Anti-Tank (HEAT) munitions are designated as using the "DIS HitToKill" vulnerability methodology. This means they have to actually hit the target in order for the server to look-up lethality effects in the look-up tables. This is due to the type of vulnerability data that is being used (in the look-up table). The vulnerability data that is being used are treated as "probability of a kill" (at some level) *given* a hit on the target. If therefore the munition actually missed the target, it would not make sense to used the data in this look-up table to describe the results of the event. (Even if it was only a near miss that detonated right next to the target). The designation "DIS ProxKill" could be used if the data set was one which did not require a "Hit" directly on the target.

FILES

$VLS_HOME/Data/Init/vls_db_init.ini

SEE ALSO

Other DIS Lethality server components:

dis_mon(1), db(3), vl(3) and $VLS_HOME/src/Db/vl_meth.c within the **vl(3)** API in particular.

Author

Geoff Sauerborn <*geoffs@arl.mil*> , US Army Research Lab. 1997, 1998.

NAME

vls_open, vls_close, vls_send, vls_read

SYNOPSIS

```
#include "vlserver.h"

int vls_open( char *host);
int vls_close(void);
int vls_send(char *data);
int vls_read(char **ptr);
```

DESCRIPTION

vlserver client library functions. **V/L server** client applications must call these functions to communicate with a running DIS V/L Communications Server.

```
/*
 ~ vls_open()
 *
 * int vls_open( char *host)
 *
 * This API is used by client programs to open a connection
 * to the DIS V/L server.  The argument "host" is an
 * ASCII null terminated string that is the name (or IP address)
 * of the computer which is running the V/L server.
 *
 * REQUIRES:
 *   The DIS vlserver program to already be running on the "host"
 *   computer.
 *
 * RETURNS
 *
 *   TRUE (1) if connection is successful.
 *  FALSE (0) if connection was unsuccessful.
 *
 * SEE ALSO:  /etc/hosts
 */
int vls_open( char *host);

/*
 ~ vls_close()
 *
 * int vls_close(void)
 *
 * This API used by client programs to
 * terminate connection with V/L server.
 *
 * RETURNS
 *
 *  TRUE (1)       -  (* on a success close. *)
 *  VL_BAD_NETCONN -  (* if there never was a connection to begin with *)
 *
 */
int vls_close(void);

/*
```

```
 ~ vls_send()
 *
 * int vls_send(char *data)
 *
 * This routine takes a data msg received from a client
 * and sends it the the VL server.  It is up to the client
 * to make sure that the message is a legal message to the
 * server.  The message a null terminated ASCII string.
 *
 * For the proper syntax of see the manual page: vlserver(1)
 *
 * RETURNS 1               - on success
 *         VL_BAD_NETCONN  - if there is not a connection established.
 *       less than zero    - on failure.
 *
 */
int vls_send(char *data);

/*
 ~ vls_read()
 *
 * int vls_read(char **ptr)
 *
 * This function checks to see if the VL Server has sent data
 * back to the client.  (Usually because the client has posed
 * a query to the server).  If a message has been sent, then
 * the message type that was sent from the VL Server will be
 * the returned value of the function (this identifies the
 * message type).
 *
 * In addition to the message type, the message itself is passed
 * to the client by setting the the argument "ptr" to point to
 * some data message.  (However, not all message types will pass
 * message and set the "ptr" argument.)  If a message is
 * passed, then it is an ASCII (NULL terminated) string.  This
 * string is allocated memory and it is the responsibility
 * of the calling application to free it (via free()) when
 * no longer need. The interpretation of the string message
 * depends on the message type value returned.
 *
 * Client vulnerability queries are usually answered by a
 * VL_MSG_TO_CLIENT message type.  (Unless an exception
 * occurred).  An improperly formatted query returns
 * VL_BAD_QUERY.  If the VL server stopped running for some
 * reason VL_SERVER_SHUTDOWN is returned.  If the client sends
 * too many quarries before looking for an answer, (or if too
 * many clients are queuing faster than the server can respond,
 * it is possible for the server to end up dropping queries and
 * returning a VL_MSG_OVERFLOW message type.  If the client
 * failed to call vls_open() prior to forming a query to the
 * server, then a VL_BAD_NETCONN message type is returned.  If
 * the client application mis-formatted a query, then the a
 * VL_BAD_QUERY message type is returned.  How a proper query
 * is formated is explain in the vlserver(1) manual page.
```

```
 *
 * RETURNS
 *
 * The function returns a message type and may set the
 * argument "ptr" to point to a text message.  Values
 * (message types) returned are one of the following values:
 *
 *    Message Type                    Meaning
 *    ------------          ---------------------------------
 *    VL_NO_DATA            (* no message (yet?) ...keep trying... *)
 *    VL_MSG_TO_CLIENT      (* incoming MSG from server *)
 *    VL_SERVER_SHUTDOWN    (* server shutting down *)
 *    VL_MSG_OVERFLOW       (* too many queries in server's que *)
 *    VL_BAD_QUERY          (* couldn't format or understand query *)
 *    VL_BAD_NETCONN        (* Never connected - call vls_open() 1st. *)
 *
 * NOTE:  The returned message point to by "ptr"
 *        is allocated memory and it is the responsibility
 *        of the calling application to free it (via free()) when
 *        no longer need.
 *
 * SEE ALSO:
 *
 *    vlserver(1)
 *
 */
int vls_read(char **ptr);
```

EXAMPLE CODE EXCERPT

The code except would be linked with the libvlsclient.a library (via a compile option such as "-lvlsclient". (i.e. cc example.c -lvlsclient").

```
#include "vlserver.h"

int status, query_id;
char *srv_msg;                  /* will point to message return by server */

/* code excerpt... */
vls_send("1 echo hello world");
while ( VL_NO_DATA == (status=vls_read(&srv_msg)))
 ; /* null loop waiting for server to respond */
/* got a message back! */
puts(srv_msg);
free(srv_msg);

/* send another message: */
vls_send("2 QUERY TYPE_mfkDIS_Result ARGS_mfkDIS_IDS 1185 33086 1110 1185 33086 12");
while ( VL_NO_DATA == (status=vls_read(&srv_msg)))
 ; /* null loop waiting for server to respond */
/* got a message back */
if (status == VL_MSG_TO_CLIENT) {
 query_id = atoi( srv_msg );
 if ( query_id == 2 ) { /* 2 is the ID we sent with our 2nd call to vls_send() */
   puts("Got back an answer to our query!");
   printf("Answer is: %s0,srv_msg);
```

```
 }
}
free(srv_msg);
```

SEE ALSO

Other DIS Lethality server components:

vlserver(1), vlexample_client.c - an undocumented example client program provided with the DIS Lethality server (look in $VLS_HOME/src/Server).

Author

Geoff Sauerborn <*geoffs@arl.mil*> , US Army Research Lab. 1997, 1998.

NAME

vlserver

SYNOPSIS

vlserver [-P port#] [-V]

DESCRIPTION

vlserver (The DIS Vulnerability/Lethality Server front end) is a part of the a collective set of programs and APIs known as the DIS Lethality server. This application is a TCP/IP server for clients to the DIS Lethality server. It works in conjunction with **dis_mon** (the DIS Damage Monitor). In fact **vlserver** does very little processing itself, it merely passes client queries on to the DIS Damage Monitor and returns the results. **vlserver** must be started before **dis_mon** since **vlserver** both creates the shared memory link between the two processes and communicates the location of that link to **dis_mon.**

OPTIONS

-P port# Where **port#** Is the port socket number where V/L server clients shall connect to the server. (Default is port 4976).

-V Turn on verbose mode. Gives extra information printed to the local console.

PROTOCOL

client/server protocol is a very simple set of ASCII commands, queries, and responses.

All commands sent to the server by a client are preceded by an integer serial number (e.g. "3 ECHO hello world"). The serial number has no significance to the server (other than it is required for proper syntax). This same serial number is returned to client along with the server's reply for that particular command. The following are recognized commands a client may send to the server:

ECHO <string> Where <string> is any text. The server returns <string> to client.

SHMID The server returns the shared memory ID to client (see **shmget(3), shmctl(3)**).

VERSION The server returns server/client protocol version ID. This an identification associated with a set of recognized commands a client may send to the server. (The client-server synatx language version). **VER** Is a synonym for VERSION.

QUERY <QTYPE> <ARG_TYPES> [arguments . . .]

This is the main query mechanism. The server returns an answer to a vulnerability query. The format of the answer is specified by QTYPE where QTYPE is one the valid type specifiers (see QTYPE). ARG_TYPES tells the server what kind of arguments will follow.

<QTYPE> specifies the form in which the query answer is to appear. Valid query types are:

TYPE_mfkDIS_Result
TYPE_mfkDIS_ProbAll
TYPE_mfkDIS_ProbK
TYPE_mfkDIS_ProbMF
TYPE_mfkDIS_ProbF
TYPE_mfkDIS_ProbM
TYPE_mfkDIS_ProbNoDamage

<ARG_TYPES> tells the server what kind of arguments will follow. Currently the only valid argument type is:

ARGS_mfkDIS_IDS

ARGS_mfkDIS_IDS tells the server to expect ID arguments. (These IDs will specify the DIS Entity ID and DIS Detonation Event ID relevant to the query. That is the arguments shall be the DIS Identity of the threatened Entity followed by the DIS Identity of the Detonation event which poses a threat to the Entity.

Since DIS expresses these identities in the form of a set of triple integers (for: site, application, id), then the arguments shall appear as six integers. (Two sets of triplets, one set for the threatened (target ID) followed by another set for the Detonation event ID:

tgt_site tgt_app tgt_id event_site event_app event_id

An example query syntax:

123 QUERY TYPE_mfkDIS_Result ARGS_mfkDIS_IDS 1185 33086 1110 1185 33086 12

This query asks the server to supply an "MFK" type result for the entity 1185 33086 12 as a consequence of detonation event 1185 33086 1110. The server might respond with something like:

123: 4 0

"123:" matches the query ID that was passed to the server. "4 and 0" are the RESULT and FLAG codes respectively. The following tables describe RESULT and FLAG codes for TYPE_mfkDIS_Result type results:

RESULT		
Numeric code	Enumerated equivalent	Meaning
0	PS_MFK_M	Mobility Kill
1	PS_MFK_F	Fire Power Kill
2	PS_MFK_MF	Mobility and Fire Power Kill
3	PS_MFK_K	Catastrophic Kill
4	PS_MFK_NODAMAGE	No Additional Damage
5	PS_ERROR	unknown error

The FLAG returned may have the following values and meanings associated with them:

FLAG return codes	
Value	Meaning
-1	Unknown error. A generic pkh result is returned but is not authoritative. In this case calling the function rpt_perror() might shed some light on the source of the error. (This is an internal vlserver library procedure whose purpose is similar to perror()).
0	Success. The pkh source for the referenced entity and threat munition (as defined in the DAMAGE_SOURCE_META_DATA_FILE) was successfully found, interpreted, and used in the calculation of the returned (VL_Result) value.
1	No Table. A generic pkh result is returned but is not authoritative. A reference to a vulnerability source could not be found in the DAMAGE_SOURCE_META_DATA_FILE for this combination of entity and threat.
2	Corrupt Table. A generic pkh result is returned but is not authoritative. The the referenced vulnerability source data was found, however there was an error when attempting to interpret the data.
3	No Environment Data. A generic pkh result is returned but is not authoritative. Data describing the fire and detonation events were never observed while monitoring the run time environment.

FLAG return codes	
Value	Meaning
4	Unknown Target. A generic pkh result is returned but is not authoritative. A reference to the threatened (targeted) entity could not be found in the DIS_ENTITIES_FILE nor in the DIS_AUXILIARY_ENTITIES_FILE.
5	Unknown Threat. A generic pkh result is returned but is not authoritative. A reference to the threat munition could not be found in the DIS_ENTITIES_FILE nor in the DIS_AUXILIARY_ENTITIES_FILE. (See vls_db_init(5)).

More example client commands and server responses:

client's command	server's response
1 ECHO Hello World	1: Hello World
2 VER	2: 19970930
3 FOO BAR	3: VLS_QUERY_SYNTAX_ERROR

BUGS

"Surely you aren't serious." "Yes I am....and don't call me Shirley."

SEE ALSO

Other DIS Lethality server components:

dis_mgr(1), vlsclient(3), vls_db_init(5) vlexample_client.exe vlexample_client.exe is an undocumented example client program provided with the DIS Lethality server (look in $VLS_HOME/bin).

Author

Geoff Sauerborn <*geoffs@arl.mil*> , US Army Research Lab. 1997, 1998.

NO. OF COPIES	ORGANIZATION
2	ADMINISTRATOR DEFENSE TECHNICAL INFO CENTER ATTN DTIC OCP 8725 JOHN J KINGMAN RD STE 0944 FT BELVOIR VA 22060-6218
1	DIRECTOR US ARMY RESEARCH LABORATORY ATTN AMSRL CS AL TA REC MGMT 2800 POWDER MILL RD ADELPHI MD 20783-1197
1	DIRECTOR US ARMY RESEARCH LABORATORY ATTN AMSRL CI LL TECH LIB 2800 POWDER MILL RD ADELPHI MD 207830-1197
1	DIRECTOR US ARMY RESEARCH LABORATORY ATTN AMSRL D R WHALIN 2800 POWDER MILL RD ADELPHI MD 20783-1197
1	DIRECTOR US ARMY RESEARCH LABORATORY ATTN AMSRL DD J J ROCCHIO 2800 POWDER MILL RD ADELPHI MD 20783-1197
1	DOD JOINT CHIEFS OF STAFF ATTN J39 CAPABILITIES DIV CAPT J M BROWNELL THE PENTAGON RM 2C865 WASHINGTON DC 20301
1	OFC OF THE DIR RSRCH AND ENGRG ATTN R MENZ PENTAGON RM 3E1089 WASHINGTON DC 20301-3080
2	OFC OF THE SECY OF DEFNS ATTN ODDRE (R&AT) G SINGLEY ODDRE (R&AT) S GONTAREK THE PENTAGON WASHINGTON DC 20301-3080
1	OSD ATTN OUSD(A&T)/ODDDR&E(R) ATTN R J TREW WASHINGTON DC 20301-7100

NO. OF COPIES	ORGANIZATION
1	AMCOM MRDEC ATTN AMSMI RD W C MCCORKLE REDSTONE ARSENAL AL 35898-5240
1	CECOM ATTN PM GPS COL S YOUNG FT MONMOUTH NJ 07703
1	CECOM SP & TERRESTRIAL COMMCTN DIV ATTN AMSEL RD ST MC M H SOICHER FT MONMOUTH NJ 07703-5203
1	US ARMY INFO SYS ENGRG CMND ATTN ASQB OTD F JENIA FT HUACHUCA AZ 85613-5300
1	US ARMY NATICK RDEC ACTING TECHNICAL DIR ATTN SSCNC T P BRANDLER NATICK MA 01760-5002
1	US ARMY RESEARCH OFC 4300 S MIAMI BLVD RESEARCH TRIANGLE PARK NC 27709
1	US ARMY SIMULATION TRAIN & INSTRMNTN CMD ATTN J STAHL 12350 RESEARCH PARKWAY ORLANDO FL 32826-3726
1	US ARMY TANK-AUTOMOTIVE & ARMAMENTS CMD ATTN AMSTA AR TD M FISETTE BLDG 1 PICATINNY ARSENAL NJ 07806-5000
1	US ARMY TANK-AUTOMOTIVE CMD RD&E CTR ATTN AMSTA TA J CHAPIN WARREN MI 48397-5000
1	US ARMY TRAINING & DOCTRINE CMD BATTLE LAB INTEGRATION & TECH DIR ATTN ATCD B J A KLEVECZ FT MONROE VA 23651-5850
1	NAV SURFACE WARFARE CTR ATTN CODE B07 J PENNELLA 17320 DAHLGREN RD BLDG 1470 RM 1101 DAHLGREN VA 22448-5100

NO. OF COPIES	ORGANIZATION
1	DARPA ATTN B KASPAR 3701 N FAIRFAX DR ARLINGTON VA 22203-1714
1	UNIV OF TEXAS ARL ELECTROMAG GROUP CAMPUS MAIL CODE F0250 ATTN A TUCKER AUSTIN TX 78713-8029
1	HICKS & ASSOCIATES, INC. ATTN G SINGLEY III 1710 GOODRICH DR STE 1300 MCLEAN VA 22102
1	ARL ELECTROMAG GROUP CAMPUS MAIL CODE F0250 A TUCKER UNIVERSITY OF TEXAS AUSTIN TX 78712
1	SPECIAL ASST TO THE WING CDR 50SW/CCX CAPT P H BERNSTEIN 300 O'MALLEY AVE STE 20 FALCON AFB CO 80912-3020
1	HQ AFWA/DNX 106 PEACEKEEPER DR STE 2N3 OFFUTT AFB NE 68113-4039
1	APPLIED RESEARCH ASSOCIATES INC ATTN MR. ROBERT SHANKLE 219 W BEL AIR AVENUE SUITE 5 ABERDEEN MD 21001
1	CDR US ARMY AVIATION RDEC CHIEF CREW ST R7D (DR N BUCHER) MS 243-4 AMES RESEARCH CENTER MOFFETT FIELD CA 94035
1	ITT INDUSTRIES ATTN CHARLES WOODHOUSE 2560 HUNTINGTON AVE ALEXANDRIA VA 22303
1	ITT INDUSTRIES ATTN MICHAEL O'CONNOR 600 BLVD SOUTH SUITE 208 HUNTSVILLE AL 35802
1	RAYTHEON SYSTEMS COMPANY ATTN JOHN D POWERS 6620 CHASE OAKS BLVD MS 8518 PLANO TX 75023

NO. OF COPIES	ORGANIZATION
1	OPTOMETRICS INCORPORATED ATTN FREDERICK G SMITH 3115 PROFESSIONAL DRIVE ANN ARBOR MI 48104-5131
1	DIR US ARL ATTN AMSRL SL EP (G MAREZ) WHITE SANDS MISSILE RANGE NM 88002
1	DIR US ARMY TRAC ATTN ATRC WE (LOUNELL SOUTHARD) WHITE SANDS MISSILE RANGE NM 88002
4	DIR US ARMY TRAC ATTN ATRC WEC JOE AGUILAR CARROL DENNY DAVID DURDA PETER SHUGART WHITE SANDS MISSILE RANGE NM 88002
3	CDR TARDEC ATTN AMSTA TR D M/S 207 FSCS ROGER HALLE GEORGE SIMON WARREN MI 48397-5000
3	CDR ARDEC ATTN AMSTA AR FSS JULIE CHU DON MILLER BILL DAVIS PICATINNY ARSENAL NJ 07806-5000
1	DEFENSE THREAT REDUCTION AGENCY ATTN SWE (WALTER ZIMMERS) 6801 TELEGRAPH ROAD ALEXANDRIA VA 22310
	ABERDEEN PROVING GROUND
2	DIRECTOR US ARMY RESEARCH LABORATORY ATTN AMSRL CI LP (TECH LIB) BLDG 305 APG AA
1	US ARMY EDGEWOOD RDEC ATTN SCBRD TD J VERVIER APG MD 21010-5423

NO. OF COPIES	ORGANIZATION
8	DIR AMSAA ATTN P DEITZ M BORROUGHS B BRADLEY J BREWER D HODGE D JOHNSON R NORMAN A WONG
26	US ARMY RESEARCH LABORATORY ATTN AMSRL WM BF J LACETERA AMSRL WM BF G SAUERBORN (25 CYS) BLDG 120
4	US ARMY RESEARCH LABORATORY ATTN AMSRL SL BV R MEYER AMSRL SL BV J ANDERSON AMSRL SL BV C KENNEDY AMSRL SL BV M MUUSS BLDG 238
4	US ARMY RESEARCH LABORATORY ATTN AMSRL M SMITH AMSRL G MOSS BLDG 321
2	DIR USARL AMSRL WM W DR INGO MAY LARRY JOHNSON BLDG 4600
1	DIR USARL AMSRL WM B A. HORST BLDG 4600
1	DIR USARL AMSRL-SL-B J SMITH BLDG 328

ABSTRACT ONLY

NO. OF COPIES	ORGANIZATION
1	DIRECTOR US ARMY RESEARCH LABORATORY ATTN AMSRL CS AL TP TECH PUB BR 2800 POWDER MILL RD ADELPHI MD 20783-1197
1	COMMANDER US ARMY MATERIEL COMMAND ATTN AMCRDA TF 5001 EISENHOWER AVENUE ALEXANDRIA VA 22333-0001
1	PRIN DPTY FOR TECH GY HDQ US ARMY MATL CMND ATTN AMCDCG T 5001 EISENHOWER AVE ALEXANDRIA VA 22333-0001
1	PRIN DPTY FOR ACQTN HDQ US ARMY MATL CMND ATTN AMCDCG A D ADAMS 5001 EISENHOWER AVE ALEXANDRIA VA 22333-0001
1	DPTY CG FOR RDE HDQ US ARMY MATL CMND ATTN AMCRD BG BEAUCHAMP 5001 EISENHOWER AVE ALEXANDRIA VA 22333-0001
1	COMMANDER US ARMY MATERIEL COMMAND ATTN AMCDE AQ 5001 EISENHOWER AVENUE ALEXANDRIA VA 22333-0001

INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	February 1999	Final

4. TITLE AND SUBTITLE
The Distributed Interactive Simulation (DIS) Lethality Communication Server
Volume II: User and Programmer's Manual

5. FUNDING NUMBERS
PR: 1L162618AH80

6. AUTHOR(S)
Sauerborn, G.C. (ARL)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Research Laboratory
Weapons & Materials Research Directorate
Aberdeen Proving Ground, MD 21010-5066

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U.S. Army Research Laboratory
Weapons & Materials Research Directorate
Aberdeen Proving Ground, MD 21010-5066

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
ARL-TR-1775

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Volume 1 presented the distributed interactive simulation lethality communication server, a client-server approach to handling battle simulation lethality. Although Volume 1 explained the approach and its benefits and limitations, it presented no information about how to set up, run, or modify the server. In this volume, these vital (yet sometimes tedious) details are provided.

14. SUBJECT TERMS
client server, DIS, lethality, degraded states, distributed simulation, vulnerability

15. NUMBER OF PAGES
183

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	