ADVANCED DISTRIBUTED SIMULATION TECHNOLOGY

1

ModSAF

PROGRAMMER'S REFERENCE MANUAL

VOL. 1

(LibArtyEdit - LibHulls)

Ver 1.0 - 20 December 1993

CONTRACT NO. N61339-91-D-0001

D.O.: 0021

CDRL SEQUENCE NO. A001

Prepared for:

U.S. Army Simulation, Training, and Instrumentation Command (STRICOM)
12350 Research Parkway
Orlando, FL 32826-3276

Prepared by:

LORAL
Systems Company

ADST Program Office
12151-A Research Parkway
Orlando, FL 32826

466

ADST-TR-W003268

ADVANCED DISTRIBUTED SIMULATION TECHNOLOGY

ModSAF

PROGRAMMER'S REFERENCE

MANUAL

VOL. 1

(LibArtyEdit - LibHulls)

Ver 1.0 - 20 December 1993

CONTRACT NO. N61339-91-D-0001

D.O.: 0021

CDRL SEQUENCE NO. A001

Prepared for:

U.S. Army Simulation, Training, and Instrumentation Command (STRICOM)
12350 Research Parkway
Orlando, FL 32826-3276

Prepared by:

LORAL
Systems Company

ADST Program Office
12151-A Research Parkway
Orlando, FL 32826

REPORT DOCUMENTATION PAGE

Form approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)

2. REPORT DATE
12/20/93

3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE
ModSAF Programmer's Reference Manual

5. FUNDING NUMBERS
C N61339-91-D-0001, Delivery Order (0021), ModSAF (CDRL A001)

6. AUTHOR(S)
Dr. Andy Ceranowicz, Joshua Smith, Anthony Courtenache, et. al

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Loral Systems Company
ADST Program Office
12151-A Research Parkway
Orlando, FL 32826

8. PERFORMING ORGANIZATION REPORT NUMBER
ADST-TR-W003268

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Simulation Training and Instrumentation Command (STRICOM)
12350 Research Parkway
Orlando, FL 32826-3275

10. SPONSORING ORGANIZATION REPORT
ADST-TR-W003268

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

12b. DISTRIBUTION CODE
A

13. ABSTRACT (Maximum 200 words)

This document provides in-depth information on all libraries within the ModSAF application. Each section is devided into an overview of the library, and a functional description.

14. SUBJECT TERMS
Modular Semi-Automated Forces, DIS, ADST, BDS-D

15. NUMBER OF PAGES
Approx 1500

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
UNCLASSIFIED

17. SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

17. SECURITY CLASSIFICATION OF ABSTRACT
UNCLASSIFIED

20. LIMITATION OF ABSTRACT
UL

ADST-TR-W003268

LibArtyEdit

1 Overview

The artillery editor allows the user to drop arbitrary artillery rounds on the DIS battlefield. The editor supports two modes of operation:

Set up then Fire

In this mode, the user selects the Mode **`Set Up Mission`**, then sets the location, round types, quantities, etc. When everything is set, the user selects the Mode **`Fire Mission`** which sends the selected detonations. The Mode then automatically returns to **`Set Up Mission`**.

Point to shoot

In this mode, the user selects the Mode **`Fire When Location Is Set`**, then sets everything *except* the location. When the round types, quantities, etc. are as desired, the user selects `Location` then clicks on the map. The mission will be fired when the user releases the mouse button. A separate mission will fire for each click.

The artillery editor is not reinitialized when it is raised, so that the default values for each mission will be whatever values were used in the previous mission.

2 Functions

The following sections describe each function provided by libartyedit, including the format and meaning of its arguments, and the meaning of its return values (if any).

2.1 artyedit_init

```
void artyedit_init()
```

artyedit_init initializes libartyedit. Call the before any other libartyedit function.

2.2 artyedit_init_gui

```
int32 artyedit_init_gui(data_path, reader_flags, gui,
                        tactmap, tcc, map_erase_gc, sensitive,
                        refresh_event, db, ctdb, valve, sim_addr,
                        event_id, exercise, protocol)
    char               *data_path;
    uint32              reader_flags;
    SGUI_PTR            gui;
    TACTMAP_PTR         tactmap;
    COORD_TCC_PTR       tcc;
    GC                  map_erase_gc;
    SNSTVE_WINDOW_PTR   sensitive;
    CALLBACK_EVENT_PTR refresh_event;
    PO_DATABASE        *db;
    CTDB               *ctdb;
    PV_VALVE_PTR        valve;
    SimulationAddress  *sim_addr;
    int32              *event_id;
    int32               exercise;
    int32               protocol;
```

'data_path'
: Specifies source directory for data files

'reader_flags'
: Specifies data file reading flags

'gui' Specifies the GUI

'tactmap' Specifies the tactical map

'tcc' Specifies the coordinate system

'map_erase_gc'
Specifies the GC used to erase on the map

'sensitive'
Specifies the sensitive window

'refresh_event'
Specifies the event invoked when the map is refreshed

'db' Specifies the PO database

'ctdb' Specifies the terrain database

'valve' Specifies the packet valve for transmission of detonations

'sim_addr'
Specifies the simulation address

'event_id'
Specifies, by referenence, the gloabl application event ID which is used, then incremented for each outgoing detonation

'exercise'
Specifies the exercise on which PDUs should be sent

'protocol'
Specifies the protocol to use (SIMNET, DIS 1.0, DIS 2.0, etc.)

artyedit_init_gui create the artillery tool. The **data_path** and **reader_flags** are used to read the editor definition file, and are as in **reader_read**. A non-zero return value indicates a libreader error occured.

ADST-TR-W003268

LibAssign

Table of Contents

1 Overview

LibAssign implements the mission assignment editor. Currently this editor can only be used to assign a single unit a single mission, however, in the future it will manage execution matrices, or some other complex assignment paradigm.

2 Usage

The software library ‘**libassign.a**’ should be built and installed in the directory ‘**/common/lib/**’. You will also need the header file ‘**libassign.h**’ which should be installed in the directory ‘**/common/include/libinc/**’. If these files are not installed, you need to do a ‘**make**’ in the libassign source directory. If these files are already built, you can skip the section on building libassign.

2.1 Building Libassign

The libassign source files are found in the directory ‘**/common/libsrc/libassign**’. ‘**RCS**’ format versions of the files can be found in ‘**/nfs/common_src/libsrc/libassign**’.

If the directory ‘**common/libsrc/libassign**’ does not exist on your machine, you should use the ‘**genbuild**’ command to update the common directory hierarchy.

To build and install the library, do the following:

```
# cd common/libsrc/libassign
# co RCS/*,v
# make install
```

This should compile the library ‘**libassign.a**’ and install it and the header file ‘**libassign.h**’ in the standard directories. If any errors occur during compilation, you may need to adjust the source code or ‘**Makefile**’ for the platform on which you are compiling. libassign should compile without errors on the following platforms:

- Mips
- SGI Indigo
- Sun Sparc

2.2 Linking with Libassign

Libassign can be linked into an application program with the following link time flags: ‘**ld [source .o files] -L/common/lib -lassign [other libraries]**’. If your compiler does not support '-L' syntax, you can use the archive explicitly: 'ld [source .o files] /common/lib/libassign.a'.

Libassign depends directly on the following libraries: libsafgui, libtactmap, libcoordinates, libsensitive, libcallback, libpo, libeditor, and libreader.

3 Functions

The following sections describe each function provided by libassign, including the format and meaning of its arguments, and the meaning of its return values (if any).

3.1 assign_init

```
void assign_init()
```

assign_init initializes libassign. Call the before any other libassign function.

3.2 assign_init_gui

```
EDT_EDITOR_PTR assign_init_gui(data_path, reader_flags, dialog_parent,
                               gui, tactmap, tcc, map_erase_gc,
                               sensitive, refresh_event, db,
                               exit_fcn, exit_arg, reinit_fcn)
    char                   *data_path;
    uint32                  reader_flags;
    Widget                  dialog_parent;
    SGUI_PTR                gui;
    TACTMAP_PTR             tactmap;
    COORD_TCC_PTR           tcc;
    GC                      map_erase_gc;
    SNSTVE_WINDOW_PTR       sensitive;
    CALLBACK_EVENT_PTR      refresh_event;
    PO_DATABASE            *db;
    ASSIGN_EXIT_FUNCTION    exit_fcn;
    ADDRESS                 exit_arg;
    ASSIGN_REINIT_FUNCTION reinit_fcn;
```

'data_path'
: Specifies the directory where data files are expected

'reader_flags'
: Specifies flags to be passed to **reader_read** when reading data files

'dialog_parent'
: Specifies top-level widget that should parent popup dialogs

'gui' Specifies the SAF GUI

'tactmap' Specifies the tactical map

'tcc' Specifies the map coordinate system

'map_erase_gc'
Specifies the GC which can erase things from the tactical map

'sensitive'
Specifies the sensitive window for the tactical map

'refresh_event'
Specifies the event which fires when the map is refreshed

'db' Specifies the persistent object database

'exit_fcn, exit_arg'
Specify a function to call when the assignment is completed

'reinit_fcn'
Specify a function to call to reinitialize a task frame prior to assignment (pass **taskedit_reinit**

assign_init_gui create the mission assignment editor. The data file (**'delobj.rdr'**) is read either from '.' or the specified data path, depending upon the **reader_flags**. The **reader_flags** are as in **reader_read**. A **NULL** return value indicates an error occured.

The **exit_fcn** should be declared as follows:

```
void exit_fcn(exit_arg, status)
    ADDRESS         exit_arg;
    EDT_EXIT_STATUS status;
```

3.3 assign_set_unit

```
void assign_set_unit(editor, unit_id)
    EDT_EDITOR_PTR  editor;
    ObjectID       *unit_id;
```

'editor' Specifies the assignment editor

'unit_id' Specifies the unit for default assignment

assign_set_unit initializes the **unit** field of the running assignment editor to the passed value.

3.4 assign_set_taskframe

```
    void assign_set_taskframe(editor, taskframe_id)
        EDT_EDITOR_PTR  editor;
        ObjectID       *taskframe_id;
```

`editor' Specifies the assignment editor

`taskframe_id'
Specifies the taskframe for default assignment

assign_set_taskframe initializes the **taskframe** field of the running assignment editor to the passed value.

3.5 assign_make_assignment

```
    void assign_make_assignment(gui, unit, frame, assigner, instruction)
        SGUI_PTR                     gui;
        ObjectID                    *unit;
        ObjectID                    *frame;
        ObjectID                    *assigner;
        TaskInstallationInstruction instruction;
```

`gui' Specifies the GUI

`unit' Specifies the unit to execute the mission

`frame' Specifies a frame of the mission to execute (automatically finds first frame)

`assigner'
Specifies the unit responsible for making the assignment (pass **NULL** to indicate the user)

`instruction'
Specifies how to start the mission

assign_make_assignment does an assignment, as though the user had selected a unit and frame from the assignment editor.

Libbalgun

Table of Contents

1 Overview

Libbalgun implements an instance of the gun class of components. It provides a low-fidelity model of generic ballistic gun behavior which is suitable for ModSAF tank main guns and machine guns. libbalgun guns support burst shooting, multiple types of munitions, and table driven hit probabilities. Also, the capability to hit un-intended targets is supported.

The parameters of a generic ballistic gun are specified in the configuration file for the vehicle containing such a gun as follows:

```
(SM_BallisticGun (physdb_name <name of gun>)
                 (sensor_name <name of tracking sensor>)
                 (hit_obscuring_vehicles <true | false>)
                 (rates <min-elevation-rate> <max-elevation-rate>)
                 (magazine_size <n>)
                 (loading_block <n>)
                 (load_time <integer milliseconds>)
                 (track_time <integer milliseconds>)
                 (munition <munition-table>
                           <munition-table>
                            ...))
```

The **<name of gun>** must match the name of a gun as specified in the libphysdb database. The **<name of tracking sensor>** must match the name of some sensor component of the vehicle. **hit_obscuring_vehicles** indicates whether this gun can accidentally hit vehicles that obscure an intended target. The **rates** for elevation are in degrees per second. **magazine_size** indicates how many rounds of a munition may be simultaneously loaded in the gun. The time to load up to **loading_block** rounds is specified by **load_time**.

A **munition-table** is of the form:

```
(<munition object type>  (round_velocity <real meters/sec>)
                         (rate <integer> burst rate shells/sec>)
                         (mass <real kg>)
                         (min_range <real meters>)
                         (max_range <real meters>)
                         (hit_table <hit-table-filename>)
                         (tracktime_table <tracktime-table>))
```

<munition object type> indicates the '**librdrconst**' object type that will be sent in the Fire and Impact PDUs. **rate** indicates the maximum sustainable burst rate in rounds per minute. A value of -1 means the gun can be fired as fast as it has rounds available. A rate of -1 should not be used if the gun has a magazine size larger than 1 (this would imply that the entire magazine could be shot at once. **min_range** and **max_range** should be considered as minimum and maximum effective ranges.

A **tracktime_table** is of the form:

```
(<real moving_firer_factor> <real moving_target_factor>
 (<real range meters> <first_med msecs> <sub_med msecs> <sub_fixed msecs>)
 (<real range meters> <first_med msecs> <sub_med msecs> <sub_fixed msecs>)
 ...
```

The tracktime table defines how long the target will be tracked before firing. The moving factors are used to modify the tracktime for these conditions. The first_med describes the lognormal distribution for the first shot by a firer at a specific target. The sub_med and sub_fixed are for subsequent shots at a target. The median and fixed values are not interpolated for range. The values are also not range-limited. Finally, if either moving factor is -1, then the tracktime will be 0. The actual calculation of the tracktime is described below.

So a simple table could be: (1.0 1.0 (0.0 3000 0 0))

This table says that a) moving firer and target have no effect on tracktime; b) the first shot's tracktime will have a median of 3000 msecs, regardless of range; c) the subsequent shots' tracktimes will be 0.

The **hit-table-filename** has a table, derived directly from AMSAA approved weapon delivery accuracy data, with the following form:

```
(
 (<range> <time of flight>
  <horizontal fixed bias>      <vertical fixed bias>
  <horizontal variable bias>   <vertical variable bias>
  <horizontal random error>    <vertical random error>
  <horizontal stationary/moving add-on dispersion>
  <vertical stationary/moving add-on dispersion>
  <horizontal stationary/moving subtractive dispersion>
  <vertical stationary/moving subtractive dispersion>
  <horizontal moving/stationary add-on dispersion>
  <vertical moving/stationary add-on dispersion>
  <horizontal moving/stationary subtractive dispersion>
  <vertical moving/stationary subtractive dispersion>)
 (<range> <time of flight> ...)

  ...)
```

)

range is the distance between the shooter and the target. (real in meters)

time of flight is the time of flight for the specific range. (real in seconds)

horizontal fixed bias and **vertical fixed bias** are fixed discrepancies between the desired aim-point and the actual hit point. (real in mils)

horizontal variable bias and **vertical variable bias** are occasional discrepancies between the desired aim-point and the actual hit point. (std deviation, real in mils)

horizontal random error and **vertical random error** are the random error factors. (std deviation, real in mils)

horizontal stationary/moving add-on dispersion and **vertical stationary/moving add-on dispersion** are add-on dispersion when the shooter is stationary and the target is moving. (std deviation, real in mils)

horizontal stationary/moving subtractive dispersion and **vertical stationary/moving subtractive dispersion** are subtractive dispersion when the shooter is stationary and the target is moving. (std deviation, real in mils)

horizontal moving/stationary add-on dispersion and **vertical moving/stationary add-on dispersion** are add-on dispersion when the shooter is moving and the target is stationary. (std deviation, real in mils)

horizontal moving/stationary subtractive dispersion and **vertical moving/stationary subtractive dispersion** are subtractive dispersion when the shooter is moving and the target is stationary. (std deviation, real in mils)

The algorithm to calculate missed distance is described below:

1. Look up the hit table entry for a given range.
2. Calculate the overall bias (B).
   a. Extract the fixed bias (F) and the variable bias standard deviation (V_STD_DEV) from the entry.
   b. The variable bias (V) is selected by drawing a normal random number from the variable

bias distribution with zero mean, and standard deviation deviation is V_STD_DEV. V = rnd_normal_distribution(0.0, V_STD_DEV)

c. The overall bias is calculated by adding F and V. B = F + V

3. Calculate the overall error (E).

   a. Extract the random error standard deviation (R_STD_DEV), the add-on dispersion standard deviation (A_STD_DEV) and the subtractive dispersion standard deviation (S_STD_DEV).

   b. Calculate overall random error standard deviation (E_STD_DEV) by adding the variances of the random error and add-on dispersion. Then subtract the variance of the subtractive dispersion and take the square root of the result. E_STD_DEV = sqrt(R_STD_DEV^2 + A_STD_DEV^2 - S_STD_DEV^2)

   c. The overall error is calculated by drawing a normal random number from the overall random error distribution with zero mean, and standard deviation is E_STD_DEV. E = rnd_normal_distribution(0.0, E_STD_DEV)

4. Calculate the missed distance.

   a. Given the overall bias (B) and overall error (E), the miss distance in mils (M_MILS) can be calculated as M_MILS = B + E

   b. Given the range (R), the missed distance in meters (M_METERS) can be calculated as M_METERS = R*tan(MIL_TO_RAD(M_MILS))

>>>>>>> ../acu/libbalgun.texinfo Libbalgun supports up to 4 instantiations per vehicle (i.e., a vehicle can have up to 4 generic ballistic guns). This number can be easily changed by recompilation.

The libbalgun library defines a common set of functions (and the semantics of those functions) which are invoked on instances of the guns class (such as those instantiated by libbalgun or libmlauncher). It is possible to modify the ballistic gun model by changing an exisiting guns interface function or by adding a completely new function.

To modify an existing libbalgun interface function would require the following actions:

1. If the change occurs only in the function body, a change to the function code in the libbalgun library if all that is needed. If the change occurs to the function's argument list, change the function code in the libbalgun library and the guns interface structure definition found in libguns.h. Also to maintain the common guns interface, change the code for the modified function in any other gun specific component library (such as libmlauncher).
2. Recompile ModSAF.

To add an additional libbalgun function to the current model would require the following actions:

1. Write the function as part of the libbalgun library. The function is written in the code that manages the libbalgun class information attached to each vehicle (bgun_class.c).
2. Add the function and its declaration to any of the other gun specific component libraries. This maintains the common guns interface.
3. In the libguns source code that handles libguns initialization processing, include a **function_number, function** entry identifying the new function for the **cmpnt_define_instance** function and every other gun instance library (such as libmlauncher).
4. In libguns.h, add an entry to identify the new macro and associate it with a function code number. This new addition means that the number of guns interface functions must be incremented by one. The guns interface structure definition that appears in libguns.h must include a structure to define the new function's argument list.
5. Recompile ModSAF.

To replace this ballistic gun model with a completely different one would require the following actions:

1. Decide on the get functions and set functions that would be required in the new model. Try to map these needed functions to the existing guns interface. A function can map if its argument list can remain the same. Functions that can not map must be added to the guns interface.
2. For those functions that can map to the existing guns interface but whose code body you want to change, edit the code for the function in the libbalgun source file that contains the code to manage the libbalgun class information (bgun_class.c).
3. For those functions that can't map to the existing guns interface, add an additional function to the guns interface. The addition procedure was described above.
4. Recompile ModSAF.

If an interface function is no longer needed, it is possible but not required, to remove it. Deletion of an interface function is only allowed when that function is not needed in any of the specific component libraries,

The deletion process requires these steps:

1. Delete the function code from each specific component library.
2. In the generic component library, remove the "function_number, function" entry identifying the excess function in the "cmpnt_define_instance" function call. This function call is found in the library's initialization code segment. In the library's public header file, remove the entry for the excess macro and its associatiated function code number. Decrease the number of interface functions by one. Delete the structure that defines the excess function's argument

list in the interface structure definition.

3. Recompile ModSAF.

2 Algorithms

The following sections describe the tick and firing event processing algorithms used by libbalgun.

2.1 Gun Tick Processing

When a gun is ticked via `bgun_tick(vehicle_id, ctdb)` the following processing occurs:

1. Retrieve the name of the sensor used by this gun.
2. Calculate dT, the time period between the last tick and this tick.
3. If the gun's state is `BALGUN_STATE_DESTROYED` then exit, otherwise do the ballistic gun simulation by invoking three independent state machines: Loader, Tracker, and Firer. Loader, implemented via the `bgun_loader_tick` function, does loading or unloading operations on quantities of requested munition. Tracker, implemented via the `bgun_tracker_tick` function, does tracking operations on requested targets or locations. Firer, invoked via the `bgun_firer_tick` function, does shooting operations on requested targets or locations. These functions which are each passed the argument list, `(vehicle_id, user_data_handle, gun, dT)`, are described below.

2.2 bgun_loader_tick

The processing for `bgun_loader_tick` (which is determined by the gun's loading state) is as follows:

- If the loading state is `BALGUN_LOADING_STATE_IDLE`, then exit if the requested transfer quantity is 0. Otherwise call libsupplies via the function `supp_get_amount` to see how many rounds are currently in storage. A requested transfer quantity greater than 0 implies an intended load and a requested transfer quantity less than 0 implies an intended unload. For an intended load, clip the request if necessary, down to the amount available in storage or the amount that there is enough room for in the breach. If the requested transfer quantity gets clipped down to zero then exit, otherwise change the loading state to `BALGUN_LOADING_STATE_LOADING`. For an intended unload, if there aren't enough rounds in the breach for the requested transfer quantity to get unloaded, change the request to reflect the amount currently loaded. If the change sets the requested transfer quantity to zero then exit, otherwise set the loading state to `BALGUN_LOADING_STATE_UNLOADING`. Set the loader time to the load time specified in the configuration file.

- If the loading state is **BALGUN_LOADING_STATE_LOADING**, then reduce the loading time by the input dT, the time between ticks. If the loader time is still greater than zero, not enough time has elapsed between ticks to load a round, so exit. Otherwise simulate the loading of one round by increasing the loaded quantity by one and reducing the requested transfer quantity by one. Invoke libsupplies to decrement the munition amount in the storage area by one round. If the requested transfer quantity has dropped down to zero, set the loading state to **BALGUN_LOADING_STATE_IDLE**, otherwise reset the loader time back to the load time of the configuration file.
- If the loading state is **BALGUN_LOADING_STATE_UNLOADING**, then reduce the loader time by the input dT, the time between ticks. If the loader time is still greater than zero, not enough time has elapsed between ticks to unload a round, so exit. Otherwise simulate the unloading of one round by decreasing the loaded quantity by one and increasing the requested transfer quantity by one. Invoke libsupplies to increment the munition amount in the storage area by one round. If the requested transfer quantity has increased to zero, set the loading state to **BALGUN_LOADING_STATE_IDLE**, otherwise reset the loader time back to the load time of the configuration file.
- If the loading state is not one of those named above, print an error message and set the loading state to **BALGUN_LOADING_STATE_IDLE**.

2.3 bgun_tracker_tick

The processing for **bgun_tracker_tick** is as follows:

1. Get the physical limits (the elevation up limit and down limit) and the turret name for this gun.
2. The gun's physical elevation needs to be updated based on the projected elevation and rate of the previous tick plus dT (the time between ticks). Since the projected elevation and rate of the previous tick are recorded in the last vehicle appearance packet, retrieve them via a libentity call. The new physical elevation is the sum of the retrieved elevation plus the change in elevation this tick. That change is calculated by multiplying the retrieved elevation rate by dT. Test whether the new elevation needs to be clipped to keep the gun elevation within limits. If clipping is needed, determine the elevation direction (up or down) and then clip to the appropriate limit. This handles the case when a long time between times could permit a gun to go completely past a stopping point. Clipping causes a discontinuity in elevation and requires a resetting of the retrieved elevation rate to zero. Call libentity to pass it the new elevation.
3. Set the new elevation rate based on the retrieved elevation rate. Clip the rate, if necessary, to keep the rate within the limits specified in the configuration file. Call libentity to pass it the

new elevation rate.

4. If the tracking mode is **BALGUN_TRACKING_MODE_MANUAL**, the gun will use the elevation that the user asks for rather than a requested tracking elevation. Therefore, set the gun's turret_request field to FALSE and set **turret_at_desired** to FALSE.
5. If the tracking mode is **BALGUN_TRACKING_MODE_TARGET**, retrieve the location of the requested tracking target via a call to libentity. Use the same code as that used for **BALGUN_TRACKING_MODE_LOCATION** to calculate a tracking azimuth and tracking elevation.
6. If the tracking mode is **BALGUN_TRACKING_MODE_LOCATION** or **BALGUN_TRACKING_MODE_LOCATION**, a requested azimuth and a requested elevation must be calculated. First, retrieve the location and azimuth of the gun's vehicle via calls to libentity. Using the position data, calculate a vector between the simulated vehicle and the requested tracking location. Using the vehicle's azimuth, calculate the position to shoot at in hull coordinates. The desired elevation and desired azimuth can then be determined using an asin, an atan2, and a sqrt. The azimuth calculation is an atan2. The elevation calculation is done by first calculating the 3D range (which can be used later as input to the Fire PDU), and using an asin on the Z linear elevation in vehicle hull coordinates with respect to this 3D range. Note that a less useful way of calculating this elevation is with an atan2 on Z with respect to the distance in the XY plane, but this doesn't give the more useful 3D range as a side-effect. Since the gun is in a tracking mode, it is necessary to note whether the turret maintains the requested tracking azimuth. When it doesn't, **turret_at_desired** is set to FALSE. This setting will be used in the tests that determine the gun's tracking status. When the turret is not at the gun's requested azimuth, set the gun's turret_request field to TRUE and invoke a turret component macro to call a libgenturret function that simulates the moving of the turret to the requested azimuth. If the turret is at the gun's requested azimuth, then set the gun's turret_request field to FALSE to indicate that this tracked gun is pointing at the proper location and does not need to request that the turret rotate.
7. If the gun's tracking mode is not one of the valid modes (**BALGUN_TRACKING_MODE_MANUAL**, **BALGUN_TRACKING_MODE_MANUAL**, or **BALGUN_TRACKING_MODE_MANUAL**) print an error message, and set the mode to **BALGUN_TRACKING_MODE_MANUAL**.
8. Adjust the requested elevation to make sure it does not exceed the gun's up or down limits.
9. Calculate dE (the change in distance to reach the requested elevation). If the gun is already close enough to the requested elevation (within 1.5 degrees) and the elevation rate is very slow (less than or equal to 1.5 degrees/sec) consider the gun to be at the requested elevation and set the elevation rate to 0 degrees/second.
10. When the gun arrives at the requested elevation, set the gun's at_desired field to TRUE and call libentity to update the gun elevation and elevation rate in the vehicle appearance packet.
11. If the gun has not yet arrived at the requested elevation, set a new elevation rate and set the gun's at_desired field to FALSE. This FALSE setting will be used in tests that determine the gun's tracking status. The new elevation rate is an adjustment that takes into account the size

of the dE (elevation change) and the how often this tracker code ticks.

12. The tracking phase begins when a target is defined, and the gun and turret are first pointed at the desired location. This event is synonomous with the vehicle commander giving the command "Gunner". The sense, detect, recognize, and identify steps have already taken place. The coarse lay of the gun has not yet taken place. The length of the tracking phase is defined by bgun_get_tracktime, which considers the firer's and target's velocities, the range to the target, and whether this is the firer's first shot at this target.
13. The tracktime table for the loaded munition contains factors for moving firer and moving target. The factors for the stationary condition are 1. The overall factor F is the product of the firer and target factors. The tracktime table also contains median and fixed times versus range. A median M defines a lognormal distribution, where the logs of the values are normally distributed with a standard deviation of 0.5, and the median of the actual values is M. The tracktime for the firer's first shot at a target and the firer's subsequent shots at a target are given by: first tracktime = (F * lognormal(m=ffire_median)) sub. tracktimes = (F * lognormal(m=sfire_median)) + sfire_fixed
14. After the tracking period has elapsed, the gun considered "tracked" (indicated by its tracking field set at TRUE). If the gun is loaded, the gun can be fired.
15. Any successive ticks while the tracking field is set at TRUE, will test whether a loss of tracking occurs because the gun and turret are not still where they are supposed to be.

2.3.1 bgun_firer_tick

The processing for `bgun_firer_tick` is as follows:

1. If this gun does not have a fire request (either an at-target or an at-location request), exit.
2. If the gun has ammunition loaded and its loading state is idle (rather than loading or unloading), then continue. Otherwise, set the fire requests to FALSE and exit.
3. If the gun can not fire because of pending firing going on from a previous burst, exit.
4. If the loaded quantity is less then the firing request quantity, clip the firing request quantity down to the loaded quantity.
5. Get munition specifics, such as the loaded munition type, from the gun's munition table.
6. If the request is a fire at-target request, reset the requests to FALSE. If this gun is tracked and its tracking mode is `BALGUN_TRACKING_MODE_TARGET`, compare the requested firing target to the requested tracking target. If the two targets are different invoke the function `shoot_at_pointing` to calculate the location the gun is pointing at (this function is described later). If the two targets are the same, then test for visibility to the target location via a call to `bgun_target_intersection`. This function will invoke the `SENSORS_GET_TARGET_VISIBILITY`

macro to determine if it would be possible for the gun to hit the target. When there isn't visibility, this function will return an intermediate vehicle or location to which there is visibility. If there is visibility, shoot the target via the function **bgun_shoot_to_hit_vehicle**; otherwise shoot the intermediate vehicle or location via the function **bgun_shoot_intermediate**.

7. If the request is a fire at-location request, reset the requests to FALSE. If this gun is tracked and its tracking mode is **BALGUN_TRACKING_MODE_LOCATION**, compare the requested firing location to the requested tracking location. If the two locations are different invoke the function **shoot_at_pointing** to calculate the location the gun is pointing at (this function is described later). If the two targets are the same, then test for visibility to the target location via a call to **bgun_location_intersect**. This function will invoke the **SENSORS_GET_LOCATION_VISIBILITY** macro to determine if it would be possible for the gun to hit a 3.0 X 3.0 meter cube at the target location. When there isn't visibility, this function will set an intermediate vehicle or location to which there is visibility. If there is visibility, shoot the location via the function **bgun_shoot_to_hit_location**; otherwise shoot the intermediate vehicle or location via the function **bgun_shoot_intermediate**.
8. Subtract the fired quantity from the loaded quantity.
9. Increment the next shot time by the time necessary to handle this request. This is determined by the rate in the munition table.

2.4 Firing Event Processing

The following functions handle the firing event processing of libbalgun.

2.4.1 bgun_shoot_to_hit_location

The function:

```
bgun_shoot_to_hit_location(vehicle_id,user_data_handle,
                           location, gun, mun_table)
```

does the following:

1. Gets the location of the shooting vehicle plus the burst specifics (location).
2. Calculates the range and time of burst.
3. Calls the function **send_ballistic_fire** to build and send a Fire PDU.

4. Schedules an Impact PDU via a deferred function call of **send_ballistic_impact**. The Impact packet will be sent in accordance with the calculated burst time.

2.4.2 bgun_shoot_to_hit_vehicle

The function:

```
bgun_shoot_to_hit_vehicle(vehicle_id, user_data_handle,
                          target_id, gun, mun_table)
```

does the following:

1. Gets the location of the shooting vehicle plus the burst specifics (location).
2. Performs the range and time of burst calculations.
3. Calls the function **send_ballistic_fire** to build and send a Fire PDU.
4. Computes either a hit or miss based on the gun's hit probability tables.
5. If the shot is determined to be a hit, schedules an Impact PDU via a deferred function call of **send_ballistic_impact**. The Impact packet will be sent in accordance with the calculated burst time.
6. If the shot is determined to be a miss, computes a miss position as being between 3 and 8 meters directly short of the target and then schedules an Impact PDU via a deferred function call of **send_ballistic_impact**. The Impact packet will be sent in accordance with the calculated burst time.

2.4.3 bgun_shoot_intermediate

The function:

```
bgun_shoot_intermediate(vehicle_id, user_data_handle, gun,
                        int_location, int_vehicle, mun_table)
```

does the following:

1. Gets the location of the shooting vehicle.
2. Updates the gun's derived quantities (vector and range) to accommodate the different location.

3. Calls **bgun_shoot_to_hit_location** if this is a shot to the ground (or building) or if the configuration file has the **hit_obscuring_vehicles** parameter set to FALSE.
4. Calls **bgun_shoot_to_hit_vehicle** if this is a shot to an intermediate vehicle.

2.4.4 shoot_at_pointing

Libbalgun makes use of the function **shoot_at_pointing** when the gun wants to shoot but is not pointing where it should be. To handle this fall through case, an alternate location is calculated and shot at with the following steps:

1. Calculate a location that the gun is pointing at that is no further away from the gun than the loaded munition's maximum range will permit.
2. Test visibility to that point.
3. If the gun has visibility to that maximum range point, place the round on the ground at that location via a call to **bgun_shoot_to_hit_location**, otherwise place the round at somewhere or something in between the gun and the max range location via a call to **bgun_shoot_intermediate**.

3 Examples

To get the component number of a gun with a particular name (such as "main-gun"):

```
int32 gun;

if ((gun = cmpnt_locate(vehicle_id, name)) ==
    CMPNT_NOT_FOUND)
  printf("Vehicle %d does not seem to have a gun called \"%s\".\n",
         vehicle_id,
         name);
```

To then give a command to that gun (the macro is defined by libguns; it assembles a `GUNS_INTERFACE` structure, and calls `cmpnt_invoke`):

```
if (gun != CMPNT_NOT_FOUND)
  GUNS_SET_ELEVATION(vehicle_id, gun, elevation);
```

4 Functions

The following sections describe each function provided by libbalgun, including the format and meaning of its arguments, and the meaning of its return values (if any).

4.1 bgun_init

```
void bgun_init(packet_valve, event_id, sim_addr, protocol,
               data_path, flags)
    PV_VALVE_PTR        packet_valve;
    int32              *event_id;
    SimulationAddress  *sim_addr;
    int32               protocol;
    char               *data_path;
    uint32              flags;
```

'packet_valve'
Specifies the packet valve used to send fire and impact pdus

'event_id'
Specifies a pointer to a static host event counter

'sim_addr'
Specifies simulation address for outgoing event DIS IDs

'protocol'
Specifies protocol in use (0 for SIMNET, **DIS_PROTOCOL_VERSION_*** for DIS)

'flags' **libreader** flags to pass to **reader_read()** (see section 'reader_read' in LibReader Programmer's Manual) when hit table files are read

bgun_init initializes libbalgun. Call this before any other libbalgun function.

4.2 bgun_class_init

```
void bgun_class_init(parent_class)
    CLASS_PTR parent_class;
```

'parent_class'
Class of the parent (declared with **class_declare_class**)

bgun_class_init creates a handle for attaching balgun class information to vehicles. The parent_class will likely be safobj_class.

4.3 bgun_tick

```
void bgun_tick(vehicle_id, ctdb)
    int   vehicle_id;
    CTDB *ctdb;
```

'vehicle_id'
Specifies the vehicle ID

'ctdb' Specifies the terrain database

bgun_tick ticks the guns for a vehicle. ctdb is required in order to access terrain information for shooting misses.

4.4 bgun_component_availability

```
void bgun_component_availability(vehicle_id, component, is_available)
        int32 vehicle_id;
        int32 component;
        int32 is_available;
```

'vehicle_id'
Specifies the vehicle ID

'component'
Specifies a component which has changed availability

'is_available'
Specifies whether the component is available or not

bgun_component_availability informs a vehicle's balguns about a change in status for components. If a component that a balgun is using, (such as the turret that the gun is attached to) becomes unavailable, the gun becomes unavailable.

4.5 bgun_damage

```
    void bgun_damage(vehicle_id, damage)
      int32 vehicle_id;
      int32 damage;
```

'**vehicle_id**'
Specifies the vehicle ID

'**damage**' Specifies whether the gun is damaged. When the function is invoked with damage set to TRUE, the gun state will be set to **BALGUN_STATE_DESTROYED** and libentity will be invoked to set the gun elevation value to -0.3. When the function is invoked with damage set to FALSE, a destroyed gun will have its the gun state reset to **BALGUN_STATE_HEALTHY** and libentity will be invoked to reset the elevation value to 0.0.

bgun_damage informs a vehicle's balguns about whether they should be damaged or not. This is used to model external damage assessment.

ADST-TR-W003268

LibC2obj

Table of Contents

1 Overview

Libc2obj provides the class superstructure in which all command-and-control sub-classes reside. Examples of these sub-classes include: unit, graphics, task, task frame, and unitorg. They can be thought of as the pieces that make up the c2obj class.

Various libraries define classes of objects which are instantiated. Most notably: LibPO creates line graphics, units, task frames, etc. Often the same object is represented in two ways, once at a low software layer, and again in a higher layer. For example:

Object Type	Low-Layer Representation	High-Layer Representation
route (line graphic)	libpo persistent object	libc2obj object

For each class of object, there is a senior layer (libc2obj) responsible for management of that class. This will generally not be the layer which instantiates the objects (libpo). Hence for persistent object classes:

High layer	libc2obj
Mid layers	libunits, libgraphics, libtaskframe,...
Low layer	libpo

Instances of the c2obj class are attached as user data to Persistent Objects of various classes such as the line class, the point class, and the unit class (e.g. see section 'Unit Class' in LibPO Programmer's Guide).

Libc2obj registers a
new_object_event_handler (see section 'new_object_event_handler' in LibPO Programmer's Guide) and an **object_gone_event_handler** (see section 'object_gone_event_handler' in LibPO Programmer's Guide) in the PO database to cause the creation and destruction of c2obj instances when new PO objects are received or old PO objects are destroyed. For example, when a user at a SAFstation creates a new unit, the resulting new_object_event_handler instructs LibPO to build an instance of the unit class in the PO database. When a user deletes an existing point graphic from the SAFSstation, the resulting object_gone_event_handler instructs libpo to remove the instance of the point class in the PO database.

Adding a sub-class requires making the following changes to this library:

1. Modify **c2obj_init** to initialize the sub-class.
2. Modify **c2obj_create** to create the sub-class at the appropriate point (after those sub-classes in lower layers, and before those in higher layers).
3. Modify **c2obj_destroy** to also destroy the new sub-class.

2 Functions

The following sections describe each function provided by libc2obj, including the format and meaning of its arguments, and the meaning of its return values (if any).

2.1 c2obj_init

```
void c2obj_init(db)
    PO_DATABASE *db;
```

c2obj_init initializes libc2obj. Call this before any other libc2obj function. Note that this function will call the *_class_init routines for all c2obj subclasses. Hence, you should call their primary init routines before c2obj_init.

ADST-TR-W003268

LibCallback

Table of Contents

1 Overview

Information can be classified into two categories: state and events. State information is present at all times, and describes things like the location of a vehicle, or the name of the mission being executed. State information can be easily controlled and made available to all software modules using the ModSAF data hiding/sharing techniques (see the ModSAF methodology documentation in the ModSAF Programmer's Guide for more on this subject). Events are one time occurrences in the simulation, such as the explosion of an artillery shell. Often, it is convenient to treat a change of state as an event, such as when a vehicles runs out of fuel (the value of the fuel state variable changed to zero).

Sharing event information is problematic. The most common method used in real time simulation, is strict function chaining: when event 'X' happens, function 'Y' will be called. To add another consequence to event 'X' requires that function 'Y' be changed. This causes interoperability problems, and may make layering of software impossible.

An alternative method of event handling is through the use of "callback" functions. A software module which is interested in an event, requests that the software module generating that event call a function when the event happens. If the software can only register one callback function, however, this solution is little better than hard coding responses to events. What is needed is a way for an arbitrary number of software modules to register event handlers, such that all of them may be called.

X windows provides callback functionality using an extremely flexible (and consequently, very computationally expensive) method. Events are identified by name (a character string), and are uniquely tied to an object (e.g., pressing a button on the screen generates an event, pressing a different button generates a different event); software modules can register callback routines by specifying the object involved and the name of the event. No distinction is made between private and public events. No facility exists for registering object-independent callback functions (a software module must re-register the callback function with every object to make it apply to all of them).

Libcallback implements a similar functionality with a simple, real time technique. A software module registers an event with libcallback, giving the types and order of arguments (libcallback will need this information later); note that no name is given. Libcallback returns a handle to the software module, which the module may either make public, or keep private. Other software modules may register callback routines which are associated with this event handle, along with a constant which it requests be passed as the last argument (this allows the receiver to define one callback function to be used with a variety of similar events). When the event occurs, the software generating the event passes libcallback the event handle and arguments; libcallback then calls all registered callback routines (the order of calling is not guaranteed, and cannot be depended upon).

A software module may create a unique event for each object created (as in X windows), or it may create one event for an entire class of objects (such as the PO database object_changed event). Unlike X, the number and type of arguments are defined on a per-event basis (X has a standard set of three arguments which are passed to every callback routine).

The use of libcallback for distribution of events is not always appropriate. In general, libcallback should be only be used to distribute event information from low layer service providers (such as libpo or libvtab) to high layer independent object classes (such as libdfdam and libtactmap). The use of libcallback for distribution of events within an object is strongly discouraged; direct function invocation should be used instead. The problem here is that a renegade programmer can use libcallback to violate normal layering constraints. Consider the following examples:

Example 1: Proper use of libcallback

Indirect fire appears on the network in the form of a DIS PDU. A low layer service provider (such as libpktvalve) receives this PDU and must distribute the indirect fire event to all interested high layer object classes. One of these classes is the local vehicle simulation (which evaluates damage from indirect fire). Another is the map display system (which draws indirect fire on the map). One or both of these classes may appear in the simulation, depending upon how the program is linked (as a SAFSIM, SAFSTATION, or POCKET SAF).

Libcallback provides a useful service in this case. The generator of the event is in a low layer, and is not part of either receiver class (vehicle simulation, or map display). The receiving classes are in high layers, are completely independent, and may or may not appear in the same executable system.

Example 2: Improper user of libcallback

The physical simulation of a vehicle (e.g., libtracked) determines that the vehicle has run out of fuel. Running out of gas may trigger several responses within the vehicle. There are three options for sharing the out-of-gas event:

1. The class which inherits libtracked (e.g., safobj class) may poll a public state variable within the libtracked class to determine if the event has occurred.
2. Libtracked may provide a service to its superior class, in the form of a callback function passed in at initialization, and called when the vehicle runs out of fuel. The superior class would then call appropriate functions in all the affected subsystems.

3. Libtracked can define an event via libcallback, and all other subsystems of the vehicle which respond to running out of fuel can register handlers for this event.

The first option is undesirable because it is both inefficient and requires making public state which could otherwise be kept private. Hence, the only decision is between the second and third options. Consider the following attributes:

Layering In option 2, the libtracked layer provides a service to the safobj class (telling it the vehicle just ran out of fuel). Each other subsystem also provides services to the safobj (responding appropriately to running out of fuel). Hence, layering is ensured.

In option 3, libtracked provides a service to all subsystems. Since some subsystems can be in higher layers than libtracked, and others in lower layers, layering cannot be guaranteed. Put another way, option 3 guarantees, that given enough time, someone WILL violate layering.

Specification

In option 2, each service (the out-of-gas notification, and each out-of-gas reaction) is specified. That a callback is provided by libtracked is specified in its initialization routine; the other subsystem reaction functions are each specified as available public function calls.

In option 3, only the existence of the event is specified. The subsystems which respond to this event can only be discovered by an examination of the software.

Clarity In option 2, safobj will contain a function which calls all the subsystems which respond to running out of fuel. The order will be explicit, and interactions between subsystems will be predictable.

In option 3, the order in which subsystems react to running out of fuel cannot be predicted, and the interactions between subsystems can only be found through experimentation or detailed analysis.

Given these considerations, it is clear that libcallback is less desirable than using an explicit callback within libtracked. The problem can be summarized succinctly: libtracked is in too high a layer for libcallback to be useful.

2 Examples

The following program demonstrates the use of libcallback functions. This program appears as 'test.c' in the libcallback source directory, and can be compiled with the command 'make test'.

```
/* Include these header files in any file using libcallback functions */
#include <libcallback.h>
#include <stdext.h> /*common/include/global*/

/* This test program demonstrates a callback relationship between
 * three libraries: (1) A network interface library which is detecting
 * direct- and indirect-fire events; (2) A simulation library which
 * queues these events for vehicle processing; (3) A display library
 * which draws effects on a map.
 *
 * The network library defines the event, and puts the handle in a
 * public variable.  The other libraries know where to find the handle
 * and the format of the event from the documentation of the network
 * library; they attach their handlers.
 *
 * Note that while the simulation uses two different handlers, the
 * display routine uses only one handler and discriminates via the user
 * data.  The latter is OK only if the events have the same number/types
 * of arguments (in this case DOUBLE, PTR).
 */

/******************************************************************/
/*                     Network Library                            */

CALLBACK_EVENT_PTR df_event_handle;
CALLBACK_EVENT_PTR if_event_handle;

void network_init()
{
    df_event_handle = callback_define_event(A_DOUBLE, A_PTR, A_END);
    if_event_handle = callback_define_event(A_DOUBLE, A_PTR, A_END);
}

void network_detect_df(when, data)
    float64 when;
    char *data;
{
    callback_fire_event(df_event_handle, when, data);
}

void network_detect_if(when, data)
    float64 when;
    char *data;
{
    callback_fire_event(if_event_handle, when, data);
}

/*****************************************************************/
/*                    Simulation Library                         */

/* #include network public header */

void distribute_df(when, data, ignored)
    float64 when;
    char *data;
    ADDRESS ignored;
{
    /* allocate buffers and queue for vehicle processing... */
    printf("Simulation: df at %f <%s>\n", when, data);
}

void distribute_if(when, data, ignored)
    float64 when;
    char *data;
    ADDRESS ignored;
{
    /* allocate buffers and queue for vehicle processing... */
    printf("Simulation: if at %f <%s>\n", when, data);
}

void simulation_init()
{
    callback_register_handler(df_event_handle, distribute_df,
                              0/*Ignored*/);
    callback_register_handler(if_event_handle, distribute_if,
                              0/*Ignored*/);
}


/*****************************************************************/
/*                      Display Library                          */

/* #include network public header */

#define DF_TYPE 1
#define IF_TYPE 2

void display_df_or_if(when, data, what)
    float64 when;
    char *data;
    ADDRESS what;
{
    switch((int32)what)
    {
      case DF_TYPE:
        printf("Display:   df at %f <%s>\n", when, data);
        break;
      case IF_TYPE:
        printf("Display:   if at %f <%s>\n", when, data);
        break;
    }
}

void display_init()
{
    callback_register_handler(df_event_handle, display_df_or_if,
                              DF_TYPE);
    callback_register_handler(if_event_handle, display_df_or_if,
                              IF_TYPE);
}

void display_no_if()
{
    callback_unregister_handler(if_event_handle, display_df_or_if,
                                IF_TYPE);
}

void display_if_too_much()
{
    int32 i;

    for(i=0;i<10;i++)
      callback_register_handler(if_event_handle, display_df_or_if,
                                IF_TYPE);
}

/***********************************************************************/

main()
{
    /* Initialize the libraries.  Note that the library which defines
     * the event must be initialized first.
     */
    network_init();
    simulation_init();
    display_init();

    /* Trigger some events */
    network_detect_df(1.0, "Bang");
    network_detect_if(2.0, "Boom");
    network_detect_df(3.0, "Pow");
    /* Disable the display of indirect fire */
    display_no_if();

    /* Trigger some events */
    network_detect_df(4.0, "Bang");
    network_detect_if(5.0, "Boom");
    network_detect_df(6.0, "Pow");

    /* Add a whole bunch of callbacks */
    display_if_too_much();

    /* Trigger some events */
    network_detect_df(4.0, "Bang");
    network_detect_if(5.0, "Boom");
    network_detect_df(6.0, "Pow");

    /* Disable the if display again */
    display_no_if();

    /* Trigger some events */
    network_detect_df(7.0, "Bang");
    network_detect_if(8.0, "Boom");
    network_detect_df(9.0, "Pow");

    /* Clean up and exit */
    callback_destroy_event(df_event_handle);
    callback_destroy_event(if_event_handle);
}
```

3 Functions

The following sections describe each function provided by libcallback, including the format and meaning of its arguments, and the meaning of its return values (if any).

3.1 callback_define_event

```
CALLBACK_EVENT_PTR callback_define_event(arg_type0, arg_type1, ...,
                                         A_END)
    int32 arg_type0, arg_type1, ...,
```

'arg_types'
Specifies the types of the arguments passed when the event fires. Chosen from the set: A_INT, A_DOUBLE, A_PTR, A_SHORT, A_CHAR, A_FLOAT (defined in common/include/global/stdext.h).

'A_END' Constant which should always be passed as the last argument.

callback_define_event creates an event and returns a handle which can be used to register handlers, fire the event, or destroy the event. The arguments specify the types of up to four arguments which will be passed when the event fires. (e.g., callback_define_event(A_INT, A_INT, A_DOUBLE, A_END)).

3.2 callback_register_handler

```
void callback_register_handler(event_handle, handler_function, user_data)
    CALLBACK_EVENT_PTR event_handle;
    CALLBACK_HANDLER   handler_function;
    ADDRESS            user_data;
```

'event_handle'
Identifies the event.

'handler_function'
Specifies the function to be called when the event occurs.

'user_data'
Specifies a constant which will be passed to the handler_function as the last argument.

callback_register_handler notes that a function is to be called when an event occurs. The event_handle identifies the event (created with callback_define_event), the handler_function specifies the address of the function which is to be called. The handler function should expect the argument types specified when the event was created (via callback_define_event), as well as a trailing argument which is the constant value passed here as user data.

3.3 callback_fire_event

```
void callback_fire_event(event_handle, arg, arg...)
    CALLBACK_EVENT_PTR event handle;
```

'**event_handle**'

Identifies the event.

'**args**' Specifies up to four arguments to pass to the handlers.

callback_fire_event invokes all the handlers defined for the event in no particular order (if order is important, libcallback is probably not the right solution, see Chapter 1 [Overview], page 1). The types of the arguments are assumed to be those passed to callback_define_event (see Section 3.1 [callback'define'event], page 9).

3.4 callback_unregister_handler

```
void callback_unregister_handler(event_handle, handler_function, user_data)
    CALLBACK_EVENT_PTR event_handle;
    CALLBACK_HANDLER   handler_function;
    ADDRESS            user_data;
```

'**event_handle**'

Identifies the event.

'**handler_function, user_data**'

Specifies the function/user_data pair which should no longer be called when the event occurs.

If a handler for the event_handle passed can be found which matches the handler_function/user_data passed, callback_unregister_handler will removed it the list of functions called when the event is fired.

3.5 callback_destroy_event

```
void callback_destroy_event(event_handle)
    CALLBACK_EVENT_PTR event_handle;
```

'event_handle'

Identifies the event.

callback_destroy_event frees the memory associated with the passed event_handle. Referencing this handle after after a call to this function will probably make the program crash (i.e., don't do it).

ADST-TR-W003268

LibClass

Table of Contents

1 Overview

An *object* is defined: "An area in computer memory that serves as a basic structural unit of analysis" (Baron). A *class* defines the organization of data in that memory, and the functions which operate on a group of objects which use the same organization. Typically a library will define a single class, and will provide functions to create, destroy, or operate on objects in that class.

Often the same object is represented in two ways, once at a low software layer, and again in a higher layer. For example:

Object Type	Low-Layer Representation	High-Layer Representation
Simulated Entity	LibVTab vehicle	LibSAFObj object
Route	LibPO persistent object	LibC2Obj object

Various libraries define classes of objects which are instantiated. Most notably:

- libpo creates graphics, units, task frames,etc.
- libvtab creates vehicles
- Xt creates widgets

Different high-layer classes need to attach different user data to these objects. For example, when a route is created in libpo, the user interface software makes a bunch of widgets and stuff which it wants to attach as user data to the object; simultaneously, the simulation software wants to compile the route into its internal format, and attach that as user data. This leads to an incompatibility which will prevent linking the workstation and simulation software together. What is needed is a way to declare at run time the number of pieces of user data which will be attached to each class of object.

Low layer classes generally allow the attachment of one piece of user data to each object (this is true of libpo, libvtab, and Xt). Libclass provides a sort of user data multiplexer service to allow each class (each library) within an application to attach its own kind of user data to each object.

As shown in the figure below, a single slot is also provided for a 'global' piece of user data to be attached to each block. This slot is accessed through functions rather than the sub-class slots.

```
Low Layer
Object
-------
|.....|
|.....|
|.....|     LibClass User Data Block
|User |    ----------------------
|Data----->| Global Slot        |
|.....|    | Debug Info         |
|.....|    ----------------------
-------    | SubClass User Data |
           ----------------------
           | SubClass User Data |
           ----------------------
           | SubClass User Data |
           ----------------------
```

Libclass also provides functions to aid in debugging. It manages run time modifiable flags which enable or disable debugging for each class, on both a global and a per-instance basis. Furthermore, it manages 'show' routines for each class, to aid in debugging.

2 Examples

The program '`test.c`' in the libclass source directory demonstrates the mechanics of using libclass. It can be compiled with the command '`make test`'. The following example provides a *bigger picture* of how libclass is to be used in the ModSAF application.

For each class of object, there is a senior layer responsible for management of that class. This will generally not be the layer which instantiates the objects. Hence for persistent object classes:

High layer	libc2obj
Mid layers	libunits, libgraphics, libtaskframe, ...
Low layer	libpo

High layer	libsafobj
Mid layers	libcollision, libcomponents, libentity, ...
Low layer	libvtab

In each case, the low layer creates the objects (provides an object creation service), while the high layer is responsible for managing all the other libraries which need to attach user data (and hence expand upon the object). The low-layer representation of the object class does not define many functions. In the case of persistent objects, the kernel does have a lot of state; in the case of safobj's, there is next to none. In both cases, there is more than one layer between the high and low layers which contributes state. Furthermore, these libraries differ depending upon how the program is linked (STATION, SIM, POCKET). STATION means that the application has no simulation capabilities, SIM means that the application has no user interface capabilities, and POCKET means that the application has both user interface and simulation capabilities.

At initialization, libclass will provides unique handles by which classes may be referenced. For example, libsafobj will do the following:

```
static CLASS_PTR safobj_class;

safobj_init(...)
    ...
{
    safobj_class = class_declare_class();
}
```

Each client library that needs to attach user data to a class thus declared will do so in its initialization routine (the senior layer will pass it the handle just allocated). In the hypothetical examples that follow, it is assumed that a vehicle of the safobj class has a "driver" sub-layer (subclass).

```
static int32 driver_user_data_handle;

driver_init(safobj_class, ...)
    CLASS_PTR safobj_class;
    ...
{
    driver_user_data_handle =
      class_reserve_user_data(safobj_class, "driver", driver_printer);
    ...
}
```

When an object of that class is instantiated, the most senior layer asks libclass to allocate space for all the sub-layer's user data. Each sub-layer allocates its own user data in its instantiation routine, and asks libclass to store a pointer to it.

```
safobj_create(...)
    ...
{
    VTAB_USER_DATA_TYPE user_data;

    /* libvtab provides a kernel object (referenced via the
     * vehicle id), with a user_data slot which is filled in here.
     */

    /* Ask libclass to allocate space for all the pointers */
    user_data =
      (VTAB_USER_DATA_TYPE)class_alloc_user_data(safobj_class);

    /* Store a pointer to this space with libvtab for later
     * retrieval.
     */
    vtab_set_vehicle(vehicle_id, user_data);

    ...

    driver_create(vehicle_id, ...);
}

-----------

driver_create(vehicle_id, ...)
    int32 vehicle_id;
    ...
{
    /* Allocate space to store our unique state variables.
     * Note that DRIVER_VARS is a private structure to
     * libdriver.
     */
    DRIVER_VARS *driver = ALLOCATE_DRIVER_VARS();

    class_set_user_data(vtab_get_vehicle(vehicle_id),
                        driver_user_data_handle,
                        driver);

    ...
}
```

Thereafter, routines which need to access their class variables will get them from libclass:

```
void driver_do_something(vehicle_id, ...)
    int32 vehicle_id;
{
    DRIVER_VARS *driver = (DRIVER_VARS *)
      class_get_user_data(vtab_get_vehicle(vehicle_id),
                          driver_user_data_handle);

    if (!driver) /* Passive error detection */
      return;

    ... operate on driver ...
}
```

Note that both `get` routines are macros, so the code here is really just doing array references. Furthermore, the mess of code making up the first three lines of the function will be the same for every functions, and hence can be encoded in a macro. An actual function should look more like this:

```
void driver_do_something(vehicle_id, ...)
    int32 vehicle_id;
{
    GET_VARS(vehicle_id, driver);

    if (!driver) /* Passive error detection */
      return;

    ... operate on driver ...
}
```

Since all the pointers to all the state structures are now stored in a generic array (allocated by libclass, and pointed to by the user data elements of the various kernel classes), debugging is a problem. dbx is not able to cast variables for printing, so examining the state of a vehicle would not be possible (except by looking at the raw numbers in memory, and matching them to structure members by hand). The solution to this problem is the debugging information which was passed to libclass at initialization time. Specifically, this includes:

- A string which identifies the name of the module ("driver", "gunner", ...)
- A pointer to a function which takes a pointer to the local data structure, and will print its contents:

```
static void driver_print(driver)
    DRIVER_VARS *driver;
{
    printf("State: %d (%s)\n", driver->state,
           state_string(driver->state));
    printf("Leader: %d\n", driver->leadveh);
    ...
}
```

With this information, libclass provides the **class_show** service. This function can be called from an application (such as from a tty command parser), or from '**dbx**':

```
(dbx) print class_show(vtab_get_vehicle(1054), "driver")
------
driver
------
State: 1 (Idle)
Leader: 52
...
0
(dbx) print class_show(vtab_get_vehicle(1054), "driver entity")
------
driver
------
State: 1 (Idle)
Leader: 52
...
------
entity
------
Vehicle ID: 54
Location: < 0.0 0.0 0.0 >
...
0
(dbx) print class_show(vtab_get_vehicle(1054), "all")
...prints all safobj information...
```

Finally, libclass also provides run time debugging support. Rather than putting conditionally compiled `printf`'s into a library, an author should use the macro CLASS_DEBUG, as follows:

```
#define DRIVER_DEBUG(vehicle_id, format) \
  CLASS_DEBUG(vtab_get_vehicle(vehicle_id), driver_user_data_handle, \
              format);

void driver_do_something(vehicle_id, ...)
    int32 vehicle_id;
{
    ...

    DRIVER_DEBUG(vehicle_id,
                 ("Vehicle %d: Driver doing something\n", vehicle_id));
    ...
}
```

3 Functions

The following sections describe each function provided by libclass, including the format and meaning of its arguments, and the meaning of its return values (if any).

3.1 class_declare_class

```
CLASS_PTR class_declare_class()
```

class_declare_class declares an object class. Each object class can have an arbitrary number of slots for pointers to sub-class data. The class handle returned by this function is referred to by each sub-class when reserving a spot for its data.

3.2 class_reserve_user_data

```
int32 class_reserve_user_data(parent_class, name, printer)
    CLASS_PTR     parent_class;
    char          *name;
    CLASS_PRINTER printer;
```

'parent_class'
Identifies the class.

'name' An ASCII string naming the sub-class.

'printer' Specifies a function which can print the user data. This function should take one argument, namely the user data.

class_reserve_user_data notes a potential sub-class of the **parent_class** (some instances of the parent class may not have this as a sub-class). The **parent_class** was declared with **class_declare_class** (see Section 3.1 [class'declare'class], page 9). Note that name and printer are used only for debugging purposes.

It is safe, although short-sighted, to pass 0 for the printer.

3.3 class_alloc_user_data

```
CLASS_USER_DATA_TYPE class_alloc_user_data(class)
    CLASS_PTR class;
```

'class' Identifies the class.

class_alloc_user_data allocates a block of memory which will hold pointers to the various sub-class data structures of the passed class. Store the return value of this as the **user_data** of the libpo entry, libvtab vehicle, Xt widget, or whatever. The passed class is the id of the class declared with **class_declare_class** (see Section 3.1 [class`declare`class], page 9).

3.4 class_free_user_data

```
void class_free_user_data(user_data)
    CLASS_USER_DATA_TYPE user_data;
```

'user_data'
Identifies the user data to be freed.

class_free_user_data frees the block of memory created with **class_alloc_user_data**, as well as private data structures (like the debugging flags).

Never reference passed the **user_data** again after calling this function.

3.5 class_set_user_data

```
void class_set_user_data(parent_user_data, handle, user_data)
    CLASS_USER_DATA_TYPE parent_user_data;
    int32                handle;
    CLASS_USER_DATA_TYPE user_data;
```

'parent_user_data'
Pointer to a block of libclass user data.

'handle' Specifies which sub-class user data slot to use.

'user_data'
Specifies the user data to be stored in that slot.

class_set_user_data sets the slot for a particular subclass of the passed user data memory block. The **parent_user_data** is a block created with **class_alloc_user_data** (see Section 3.3 [class'alloc'user'data], page 10), and the handle is the one created for this subclass with **class_reserve_user_data** (see Section 3.2 [class'reserve'user'data], page 9).

3.6 class_get_user_data

```
CLASS_USER_DATA_TYPE class_get_user_data(parent_user_data, handle)
    CLASS_USER_DATA_TYPE parent_user_data;
    int32                handle;
```

'parent_user_data'
Pointer to a block of libclass user data.

'handle' Specifies which sub-class user data slot to get.

class_get_user_data gets the data stored in a particular slot of the passed block of **user_data** memory. The **parent_user_data** is a block created with **class_alloc_user_data** (see Section 3.3 [class'alloc'user'data], page 10), and the handle is the one created for this subclass with **class_reserve_user_** (see Section 3.2 [class'reserve'user'data], page 9). If no user data was stored in this slot with **class_set_user_data** (see Section 3.5 [class'set'user'data], page 10), a **NULL** pointer will be returned.

3.7 class_set_global_data

```
void class_set_global_data(parent_user_data, user_data)
    CLASS_USER_DATA_TYPE parent_user_data;
    CLASS_USER_DATA_TYPE user_data;
```

'parent_user_data'
Pointer to a block of libclass user data.

'user_data'
Specifies the user data to be stored in the global slot.

class_set_global_data sets the global slot of the passed user data memory block. This is analogous to **class_set_user_data**, except that only one global slot is provided.

3.8 class_get_global_data

```
CLASS_USER_DATA_TYPE class_get_global_data(parent_user_data)
    CLASS_USER_DATA_TYPE parent_user_data;
```

'**parent_user_data**'
Pointer to a block of libclass user data.

class_get_global_data gets the global slot of the passed user data memory block. This is analogous to **class_get_user_data**, except that only one global slot is provided.

3.9 class_show

```
void class_show(user_data, fields)
    CLASS_USER_DATA_TYPE user_data;
    char                 *fields;
```

'**user_data**'
Pointer to a block of libclass user data.

'**fields**' Selects which fields to show.

class_show is a debugging print routine. It prints the named fields within the body of data pointed to by **user_data**. Multiple fields can be given, separated by a space, comma, or other delimiter. Also, one of the following may be given as the only field:

what Shows names of slots which have non-NULL values.

all Shows all fields.

This function can be called from '**dbx**' to examine variables, as shown in the example (see Chapter 2 [Examples], page 3).

3.10 CLASS_DEBUG

```
void CLASS_DEBUG(user_data, handle, (format_string, arg, args...))
    CLASS_USER_DATA_TYPE user_data;
    int32                handle;
```

'user_data'
Pointer to a block of libclass user data.

'handle' Specifies which sub-class user data slot to test.

CLASS_DEBUG is a macro which calls printf with the passed format/args on if the debugging is on for that slot on that class in this block. Will also print if debugging is on for that slot/class globally. See the example (see Chapter 2 [Examples], page 3) for example usage.

3.11 class_get_debug

```
int32 class_get_debug(user_data, handle)
    CLASS_USER_DATA_TYPE user_data;
    int32                handle;
```

'user_data'
Pointer to a block of libclass user data.

'handle' Specifies which sub-class user data slot to test.

class_get_debug determines if debugging output is appropriate for the particular handle of the particular instance (or if it is set globally). Returns 1 if debugging is on, 0 if off. CLASS_DEBUG is implemented using this function. An application may call this when enabling debugging triggers more than just a printf (such as turning on extra error checks).

3.12 class_debug_byname

```
void class_debug_byname(user_data, fields, bit)
    CLASS_USER_DATA_TYPE user_data;
    char                *fields;
    int32                bit;
```

'**user_data**'
Pointer to a block of libclass user data.

'**fields**' Selects which fields to enable/disable debugging.

'**bit**' 1 enables debugging; 0 disables.

class_debug_byname sets debugging flags associated with classes listed in fields either on (bit == 1), or off (bit == 0). If a **parent_user_data** is passed, debugging is set for that object only. If a NULL pointer is passed as the **parent_user_data**, debugging is set globally. The special value **all** can be passed as the field, to set all debugging field for the object.

Note that the sequence **class_debug_byname(NULL, "all", 1)** enables all debugging for all objects.

3.13 class_show_names

```
void class_show_names()
```

class_show_names prints the named fields within the slots of all valid classes.

ADST-TR-W003268

LibCmdLine

Table of Contents

1 Overview

Libcmdline provides a flexible command line processor with the following features:

- Command line arguments are specified succinctly in the code, in a way which makes it easy to add new ones.
- Each argument has accompanying information so libcmdline can print help, summarize selected values, and do error checking.
- Each user can modify the default options used with each application program via an environment variable. The name of this variable is derived from the name of the executable by capitalizing it and appending ARGS. Hence, default arguments to an executable named phantom could be stored in the environment variable PHANTOMARGS. This allows the creation of one executable which can use symbolic links to yield different names and default behavior (safstation, safsim, logger, etc.).
- Hence, arguments can be specified in three places: in the code, in an environment variable, and on the command line (in increasing order of precedence).
- Each command line argument can be a switch, the mere presence of which indicates a value, or it can be followed by one or more values (integers, floating point numbers, strings, or any combination).
- Arguments can be interdependent, so for example, the user may only be allowed to specify an exercise ID if running with the network; or specify a starting X if a starting Y is also specified (and vice-versa).
- Multiple dependencies can be either conjunctively or disjunctively combined, so for example, the interdependent X and Y can also be dependent upon a Z and upon there not being a starting grid specified, and the Z can be dependent upon there being an X or a starting grid specified.
- Unrecognized arguments are given back to the caller, so they can be passed to other command line processors (such as `XtInitialize`). These unrecognized arguments can be given either on the command line or in the user environment variable (so for example, STATIONARGS can be "-g 800x500", which might not be recognized by libcmdline, but instead would be passed on to Xt).
- Arguments can be abbreviated by ending them with a '.'. For example, if the application expects -exercise 1, the user can type -e. 1 to achieve the same result. The reason for the explicit '.' is that some arguments are not intended to be recognized (such as the -g option just described).

The interface to libcmdline is through an array of `CMD_OPTION` structures, although an application will generally instead use a structure of various parallel structures, and speci values with aggregate initialization (see Chapter 2 [Examples], page 3).

2 Examples

The following code segment gives examples of all libcmdline features. This code serves as a hypothetical example; it does not represent actual ModSAF code. See Chapter 4 [Options Structures], page 13, for more information.

```
#include "libcmdline.h"
#include <stdext.h> /*common/include/global*/

struct
{
    CMD_TOGGLE_OPTION safsim;
    CMD_TOGGLE_OPTION safstation;
    CMD_STRING_OPTION terrain;
    CMD_TOGGLE_OPTION network;
    CMD_INTEGER_OPTION exercise;
    CMD_BOOLEAN_OPTION synch;
    CMD_INTEGER_OPTION mcache;
    CMD_STRING_OPTION startmap;
    CMD_INTEGER_OPTION scalenum;
    CMD_INTEGER_OPTION scaledenom;
    CMD_FLOAT_OPTION rndseed;
    CMD_INTEGER_OPTION xloc;
    CMD_INTEGER_OPTION yloc;
} options = {
    {
        "SAFSIM" , "Selects whether to run SAF simulation processes" ,
        NULL ,
        CMD_TOGGLE ,   "sim" ,          "nosim" ,       TRUE ,
    } ,
    {
        "SAFSTATION" , "Selects whether to run SAF workstation processes" ,
        NULL ,
        CMD_TOGGLE ,   "station" ,      "nostation" ,   TRUE ,
    } ,
    {
        "Terrain Database Name" , "Specifies terrain database" ,
        "station|sim" ,
        CMD_STRING ,   "terrain" ,      "nowhere" ,     "knox-0311" ,
    } ,
    {
        "Network" , "Selects whether to use network" ,
        NULL ,
        CMD_TOGGLE ,   "network" ,      "nonet" ,       TRUE ,
    } ,
    {
        "Exercise ID" , "Specifies simulation exercise id" ,
        "network" ,
        CMD_INTEGER ,   "exercise" ,    NULL ,          1 ,
    } ,
    {
        "Synchronous" , "Run X windows in synchronous mode for debugging" ,
        "station" ,
        CMD_BOOLEAN ,   "synch" ,       "asynch" ,      FALSE ,
    } ,
    {
        "Map Cache" , "Specifies number of map screens to cache" ,
        "station" ,
        CMD_INTEGER ,   "mcache" ,      "nomcache" ,    2 ,
    } ,
    {
        "Map Starting Grid" , "Specifies initial location of map" ,
        "station" ,
        CMD_STRING ,    "startmap" ,    "center" ,      NULL ,
    } ,
    {
        "Map Starting Scale" , "Specifies initial map scale # : #" ,
        "station" ,
        CMD_INTEGER ,   "scale" ,       NULL ,          1 ,
    } ,
    {
        /* Map Starting Scale takes two arguments */
        NULL , NULL , NULL , CMD_INTEGER , NULL , NULL , 200000 ,
    } ,
    {
        "Random Number Seed" , "Seed value for random number generator" ,
        "sim" ,
        CMD_FLOAT ,     "rand" ,        NULL ,          0.0 ,
    } ,
    {
        "X" , "Starting X location" ,
        "y&station" ,
        CMD_INTEGER ,   "x" ,           NULL ,          0 ,
    } ,
    {
        "Y" , "Starting Y location" ,
        "x&station" ,
        CMD_INTEGER ,   "y" ,           NULL ,          0 ,
    } ,
};

main(argc, argv)
    int   argc;
    char *argv[];
{
    int32 leftover_argc;
    char **leftover_argv;
    cmd_process_options(argc, argv, /* args to main() */
                        &leftover_argc, &leftover_argv, /* unrecognized */
                        (CMD_OPTION *)&options, /* options array */
                        sizeof(options), /* size of options array */
                        TRUE /* Verbose mode */
                        );

    /* Pass remaining args to X windows */
    XtInitialize(XtInitialize(argv[0], "Saf", NULL, 0,
                              &leftover_argc, leftover_argv);

    cmd_gripe(leftover_argc, leftover_argv);

    /* Example usage... */
    printf("Reading /saf/terrain/%s\n", options.terrain.value);
}
```

It is also possible to describe command line options in multiple arrays by aggeregating them and then processing the resulting structure. This is useful for keeping command line options associated with modules. For example, one module could may control the application processes:

```
#include "libcmdline.h"
#include <stdext.h> /*common/include/global*/

struct
{
    CMD_TOGGLE_OPTION safsim;
    CMD_TOGGLE_OPTION safstation;
} process_options = {
    {
        "SAFSIM" , "Selects whether to run SAF simulation processes" ,
        NULL ,
        CMD_TOGGLE ,    "sim" ,         "nosim" ,       TRUE ,
    } ,
    {
        "SAFSTATION" , "Selects whether to run SAF workstation processes" ,
        NULL ,
        CMD_TOGGLE ,    "station" ,     "nostation" ,   TRUE ,
    } ,
};

void process_init (options, sizeof_options, map)
    CMD_OPTION **options;
    uint32      *sizeof_options;
    CMD_TYPES ***map;
{
    cmd_aggregate (options, sizeof_options,
                   (CMD_OPTION *)&process_options,
                   sizeof (process_options),
                   map);
}

void process_simul ()
{
    printf ("Simulation process is %s\n",
            process_options.safsim.value ? "On" : "Off");
    printf ("Workstation process is %s\n",
            process_options.safstation.value ? "On" : "Off");
}
```

Another module may control the network:

```
#include "libcmdline.h"
#include <stdext.h> /*common/include/global*/

struct
{
    CMD_TOGGLE_OPTION network;
    CMD_INTEGER_OPTION exercise;
} network_options = {
    {
        "Network" , "Selects whether to use network" ,
        NULL ,
        CMD_TOGGLE ,    "network" ,     "nonet" ,       TRUE ,
    } ,
    {
        "Exercise ID" , "Specifies simulation exercise id" ,
        "network" ,
        CMD_INTEGER ,   "exercise" ,    NULL ,          1 ,
    } ,
};

void network_init (options, sizeof_options, map)
    CMD_OPTION **options;
    uint32      *sizeof_options;
    CMD_TYPES ***map;
{
    cmd_aggregate (options, sizeof_options,
                   (CMD_OPTION *)&network_options,
                   sizeof (network_options),
                   map);
}

void network_simul ()
{
    printf ("Network is %s\n",
            network_options.network.value ? "On" : "Off");
    if (network_options.network.value)
      printf ("Exercise ID is %d\n", network_options.exercise.value);
}
```

The options from these two modules may be combined as follows:

```
#include "libcmdline.h"
#include <stdext.h> /*common/include/global*/

extern void process_init ();
extern void process_simul ();
extern void network_init ();
extern void network_simul ();

main(argc, argv)
    int   argc;
    char *argv[];
{
    int32 leftover_argc;
    char **leftover_argv;
    CMD_OPTION *options;
    uint32 sizeof_options;
    CMD_TYPES **map;

    sizeof_options = 0;

    process_init (&options, &sizeof_options, &map);
    network_init (&options, &sizeof_options, &map);

    cmd_process_aggregate_options(argc, argv, /* args to main() */
                        &leftover_argc, &leftover_argv, /* unrecognized */
                        options, /* options array */
                        sizeof_options, /* size of options array */
                        map, /* option value map */
                        TRUE /* Verbose mode */
                        );

   process_simul ();
   network_simul ();
}
```

3 Functions

The following sections describe each function provided by libcmdline, including the format and meaning of its arguments, and the meaning of its return values (if any).

3.1 cmd_process_options

```
void cmd_process_options(argc, argv, leftover_argc, leftover_argv,
                         options, sizeof_options, verbose)
    int         argc;
    char       *argv[];
    int32      *leftover_argc;
    char      **leftover_argv[];
    CMD_OPTION *options;
    uint32      sizeof_options;
    int32       verbose;
```

'**argc, argv**'

Arguments to **main**

'**leftover_argc, leftover_argv**'

Returns unmatched arguments for processing by other command line parsers (such as **XtInitialize**)

'**options**' Specifies array of known options, returns values of those options

'**sizeof_options**'

Specifies the size of the options array (in bytes)

'**verbose**' Specifies whether to print values of options

cmd_process_options processes command line options, providing a value for each (either a system default, an environment default, or a command line switch value). The options which are not recognized are passed back in **leftover_argc/v**.

3.2 cmd_aggregate

```
int32 cmd_aggregate(aggregate, sizeof_aggregate, piece, sizeof_piece, map)
    CMD_OPTION **aggregate;
    uint32      *sizeof_aggregate;
    CMD_OPTION  *piece;
```

```
        uint32      sizeof_piece;
        CMD_TYPES ***map;
```

'**aggregate**'
Pointer to a variable containing a pointer to the aggregate options array. The user simply needs to declare the variable. **cmd_aggregate** will fill it in properly.

'**sizeof_aggregate**'
A pointer to a variable containing the size of the aggregate options array. The user should set the value of the variable to zero (0) before the first call to **cmd_aggregate**.

'**piece**' An options array to be added to the aggregate array.

'**sizeof_piece**'
The size of **piece**.

'**map**' A pointer to a variable containing a pointer to a map array. The user simply needs to declare the variable. **cmd_aggregate** will fill it in properly.

cmd_aggregate adds a command line options array (**piece**) to a dynamically allocated command line options array (**aggregate**).

3.3 cmd_process_aggregate_options

```
    void cmd_process_aggregate_options(argc, argv, leftover_argc,
                                       leftover_argv, options,
                                       sizeof_options, map, verbose)
        int         argc;
        char       *argv[];
        int32      *leftover_argc;
        char      **leftover_argv[];
        CMD_OPTION *options;
        uint32      sizeof_options;
        CMD_TYPES **map;
        int32       verbose;
```

'**argc, argv**'
Arguments to **main**.

'**leftover_argc, leftover_argv**'
Returns unmatched arguments for processing by other command line parsers (such as **XtInitialize**).

'**options**' Specifies aggregated array of known options. Constructed in calls to **cmd_aggregate**.

'**sizeof_options**'
Specifies the size of the options array (in bytes). Determined in calls to **cmd_aggregate**.

'**map**' Contains a mapping the aggregate options array and each of the pieces.

'**verbose**' Specifies whether to print values of options

cmd_process_aggregate_options processes command line options contained and an aggregate options array. The value for each option (either a system default, an environment default, or a command line switch value) is copied into the option arrays that make up the aggregate array. The options which are not recognized are passed back in **leftover_argc/v**.

3.4 cmd_gripe

```
void cmd_gripe(leftover_argc, leftover_argv)
    int32 leftover_argc;
    char *leftover_argv[];
```

'**leftover_argc**'
Specifies the number of unrecognized options

'**leftover_argv**'
Specifies list of unrecognized options

cmd_gripe print a message listing unrecognized arguments.

4 Options Structures

Libcmdline uses six structures to represent command line options. The first (CMD_OPTION) is a generic structure in which default values are specified and resulting values are returned using the following union:

```
union cmd_types
{
    int32   boolean;
    int32   toggle;
    int32   integer;
    char   *string;
    float32 floating;
};
```

The remaining five structures are identical to this master structure, except that rather than using a union, the default and resulting value fields are of a single simple type. This allows these values to be specified using aggregate initializers in C, for example:

```
CMD_TOGGLE_OPTION safsim =
    {
        "SAFSIM" , "Selects whether to run SAF simulation processes" ,
        NULL ,
        CMD_TOGGLE ,    "sim" ,          "nosim" ,       TRUE ,
    };
```

This type of initialization is not possible in C when the structure contains unions.

The CMD_OPTION structure is defined as follows:

```
typedef struct cmd_option
{
    /* Descriptive name of thing controlled with option */
    char            *name;

    /* Description of purpose of this option */
    char            *help;

    /* If non-NULL, specifies an option which must be selected (if the
     * name is that of an option, the value associated with that option
     * must be non-zero; if the name is that of an anti_option, the
     * value must be zero) for this option to be allowed.
     */
```

```
        char            *dependent_option;

        /* CMD_BOOLEAN, CMD_FLOAT, etc. */
        int32            type;

        /* Command line option (preceded with - on command line) */
        char            *option;

        /* Version of the option which indicates 0.  May specify NULL to
         * indicate no such option when type == INTEGER, STRING, or FLOAT;
         * must give a value when type == BOOLEAN or TOGGLE.
         */
        char            *anti_option;

        /* Default value (Note the mis-spelling because default is reserved) */
        union cmd_types default;

        /* Resulting value (don't bother to initialize) */
        union cmd_types value;
    } CMD_OPTION;
```

Each field is described in the following sections.

4.1 name

The *name* of the option is used for two purposes:

- When processing the command line with the Verbose flag set to TRUE, the name is used in printing the value.
- If the name is not specified (NULL), the option is treated as a continuation of the previous option. In this way, a single option can take more than one value.

4.2 help

The *help* string specified for each option is a description of what the option controls. Help is used when one of the flags passed is -help, -h, or -?. Help strings should kept short to avoid wrapping at the end of the screen

4.3 dependent_option

dependent_option(s) are those which must be specified along with the current option. More than one option can be listed here, tied together with the others using the characters '&' or '|', meaning, respectively, that all or at least one of the dependent options must be given. (Don't use any spaces in this string.)

Refering to a boolean or toggle option in the list of dependencies indicates that the value of that option must be TRUE. Referring to any other kind of option indicates that the value of that option must be different than the default value. Finally, referring to an anti-option, indicates that that option must have a zero (or NULL) value. Note that depending on an option with more than one value (see Section 4.1 [name], page 14), will perform these tests on the first value, but not on any continuations.

Dependencies are not checked until after all command line arguments have been processed, so options may be interdependent (-a depends on -b, and -b depends on -a).

4.4 type

The *type* of an option determines how its value is determined, printed, and stored. The following types are defined:

CMD_BOOLEAN

A True (1) or False (0) value. Simply listing the option on the command line is sufficient to specify a True value. Similarly, listing the anti-option specifies a False value.

CMD_TOGGLE

An On (1) or Off (0) value. The only difference between TOGGLE and BOOLEAN is in the way the value is printed.

CMD_INTEGER

An integer value.

CMD_STRING

A character string.

CMD_FLOAT

A floating point number (in a format recognized by scanf(3)).

For each type, there is a variant of the CMD_OPTION structure. They are as follows:

`CMD_BOOLEAN`
    CMD_BOOLEAN_OPTION

`CMD_TOGGLE`
    CMD_TOGGLE_OPTION

`CMD_INTEGER`
    CMD_INTEGER_OPTION

`CMD_STRING`
    CMD_STRING_OPTION

`CMD_FLOAT`
    CMD_FLOAT_OPTION

4.5 option

The *option* string is the sequence of characters which are specified on the command line with a '-' to select this option. This should be NULL when the option is a continuation of a previous option (see Section 4.1 [name], page 14).

4.6 anti_option

The *anti_option* string is a sequence of characters which can be specified on the command line with a '-', to indicate a 0 or `NULL` value. Note that selecting the anti-option nullifies all continuations of the option as well.

For boolean and toggle options, it is mandatory to specify an anti-option (otherwise environmental defaults could not be overridden on the command line). For other types of options, `NULL` may be specified to indicate its absence.

4.7 default

The default value (misspelled *default* because of conflicts with the C language) is the value which is used unless overridden by the evironment variable or on the command line.

4.8 value

The *value* ultimately selected for each option is placed here. This is a write-only field in libcmdline, any initial value will be overwritten by **`cmd_process_options()`**.

LibCollision

Table of Contents

1 Overview

Libcollision provides a 3D physical model of collision detection. It can detect collisions with other network entities (platforms, missiles, and structures), as well as treelines, buildings, and the ground. This library is also responsible for generating and processing collision PDUs. The library uses a parametrically controlled timing heuristic to filter out redundant collisions (such as when both parties in the collision send one another collision PDUs).

This library handles both the simple detection of intersections with nearby features used for slow moving ground vehicles, as well as the more complex detection of collisions needed by a fast moving vehicle which may jump a considerable distance between ticks. For example, a missile traveling at Mach 1, ticking at 2 Hz will jump about 165 meters each tick. Hence, a ray must be run from the old position to the new position to determine if any features were intersected along the way.

The parameters used by a vehicle (or missile) for collision detection are specified in its configuration file as follows:

```
(SM_Collision (check {trees} {buildings} {ground}
                     {platforms} {missiles})
              (announce {trees} {buildings} {ground}
                        {platforms} {missiles})
              (duration <integer milliseconds>)
              (feature_mass <real kg>)
              (fidelity [high|low])
```

The first parameter, **check**, lists those things for which collision detection is required. This affects performance in two ways:

- Each tick, libcollision checks for collisions with those things listed. Each additional item has some added cost.
- When a collision PDU is received, the collision is only reported to the parent object (see Section 2.2 [coll˘class˘init], page 3) if the colliding entity matches one of the types listed. For example, if a tank is configured to not check for collisions with buildings, then a collision emminating from a network entity of domain **Structure** will not be passed on.

The **announce** parameter lists those collision which should be announced on the network (via a collision PDU) when a collision is detected locally. For example, a missile would list **platforms** in its **check** list, but not in its **announce** list, since the missile will be sending an impact PDU instead of a collision PDU.

The **duration** value is used in a simple heuristic which avoids redundant collision detection. When a collision is detected, the time of that collision is noted. Each similar collision (with the same entity or terrain feature) which occurs within the specified **duration** after the original collision is then ignored. This gives the dynamics software a chance to back the vehicle out of the collision, and protects from redundant damage assessment when both parties in a collision detect it and issue collision PDUs.

The **feature_mass** parameter specifies the mass which will be passed up to the parent object when the software detects a collision with a terrain feature. This is intended to simplify encoding of damage models.

Finally, the **fidelity** parameter (which has a value of **high** or **low**) is used to determine the accuracy (and hence computational expense) of the algorithms used in collision detection. The differences are primarily with respect to the point of intersection.

In the high fidelity model, the point of intersection will be accurately determined, using the parallelepipeds described in libphysdb (width, length, height). For example, a collision with a building will be detected exactly as the front edge of the vehicle touches the building.

In contrast, the low fidelity model detects collisions with terrain features using a bounding cube which is aligned with the compass axes, and is as large as the vehicle's largest dimension. Collisions with other vehicles are detected using a simple spherical model of vehicles.

Note that the low fidelity model *does* run the ray intersection test described above, so it may be used for crude missile simulations.

2 Functions

The following sections describe each function provided by libcollision, including the format and meaning of its arguments, and the meaning of its return values (if any).

2.1 coll_init

```
void coll_init(packet_valve, event_id, exercise_id, sim_addr, protocol)
    PV_VALVE_PTR       packet_valve;
    int32             *event_id;
    uint8              exercise_id;
    SimulationAddress *sim_addr;
    int32              protocol;
```

'packet_valve'
Specifies the packet valve used to send/receive collision PDUs

'event_id'
Specifies a pointer to a static host event counter

'exercise_id'
Specifies the exercise on which to broadcast collision PDUs

'sim_addr'
Specifies simulation address for outgoing event DIS IDs

'protocol'
Specifies protocol in use (0 for SIMNET, DIS_PROTOCOL_VERSION_* for DIS)

coll_init initializes libcollision. Call this before calling any other libcollision functions. The packet_valve is created with a call to pv_create_valve.

2.2 coll_class_init

```
void coll_class_init(parent_class, callback)
    CLASS_PTR     parent_class;
    COLL_CALLBACK callback;
```

'parent_class'
Specifies the parent class (probably safobj_class)

'callback'

Specifies the function to call when collisions occur

coll_class_init creates a handle (index) for attaching collision class information to vehicles. The parent_class is one created with class_declare_class. The callback function should be declared:

```
void callback(vehicle_id, position, coll_type,
              other_id, other_mass, other_velocity)
    int32   vehicle_id;
    float64 position[3];
    uint32  coll_type;
    int32   other_id;
    float64 other_mass;
    float64 other_velocity[3];
```

This is called when a collision occurs, after the collision is announced on the network (provided the type of collision is listed in the announce parameter list). The position sent is the point of collision. The coll_type code is one of the following:

COLL_TREES

Indicates crossing a treeline or canopy edge.

COLL_BUILDINGS

Indicates crossing a building or other structure. If the other structure is represented on the network, the vehicle ID of that structure will be provided.

COLL_GROUND

Indicates a collision with the ground.

COLL_PLATFORMS

Indicates intersecting a platform (vehicle, DI, etc.).

COLL_MISSILES

Indicates intersecting a missile (an entity on the network with a munition type).

If the collided entity exists in the vehicle table, its ID is given in the other_id field. For collisions with terrain features, the other_mass will be that specified in the feature_mass field of the parametric data, and the other_velocity will be zero.

2.3 coll_create

```
void coll_create(vehicle_id, parms)
```

```
        int32                      vehicle_id;
        COLLISION_PARAMETRIC_DATA *parms;
```

'vehicle_id'
Specifies the vehicle ID

'parms' Specifies initial parameters

coll_create creates the collision class information for a vehicle and attaches it to the vehicle's libclass user data.

2.4 coll_destroy

```
    void coll_destroy(vehicle_id)
        int32 vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

coll_destroy frees the collision class information for a vehicle.

2.5 coll_process_pdus

```
    void coll_process_pdus(vehicle_id)
        int32 vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

coll_process_pdus processes any collision PDUs received and enqueued since the last call to this function. This may invoke the callback function passed to **coll_class_init** if a received collision is not one which was already detected locally. This should be called early in a vehicle's tick, preferably before updating the camera (in order to ensure minimal visual latency), and before calling **coll_tick**.

2.6 coll_tick

```
void coll_tick(vehicle_id, ctdb)
    int32 vehicle_id;
    CTDB *ctdb;
```

'vehicle_id'
Specifies the vehicle ID

'ctdb' Specifies the terrain database

coll_tick ticks the collision sub-class. During the tick, if a collision is detected, a packet may be sent and the callback function passed to coll_class_init may be invoked.

2.7 coll_get_current

```
void coll_get_current(vehicle_id, current)
    int32        vehicle_id;
    COLL_CURRENT *current;
```

'vehicle_id'
Specifies the vehicle ID

'current' Returns current collision information

coll_get_current returns a *read-only* list of vehicles which have been collided with recently. The user data attached to each vehicle in the list specifies the simulation clock value at the time of collision. It is safe to iterate over the list of vehicles, provided that iteration does not span more than one tick (i.e., the collision subclass will also be iterating over this list). This function also gives information about the most recent terrain collision. The COLL_CURRENT structure is defined as follows:

```
typedef struct coll_current
{
    /* A list of recent vehicle collisions */
    VTAB_LIST list;

    /* A description of the most recent terrain collision */
    struct
    {
        uint32 coll_type; /* COLL_TREES, etc. */
        uint32 simulation_clock;
```

```
            float64 location[3];
            float64 mass;
        } most_recent_terrain;
    } COLL_CURRENT;
```

2.8 coll_ignore

```
    void coll_ignore(vehicle_id, operation, other_id)
        int32          vehicle_id;
        COLL_IGNORE_OP operation;
        int32          other_id;
```

'**vehicle_id**'

Specifies the vehicle ID

'**operation**'

Specifies whether to add or remove the other vehicle from the ignore list

'**other_id**'

Specifies vehicle to ignore

coll_ignore adds or removes a vehicle to the list of vehicles which should be ignored in collision detection. This can be used, for example, to avoid detecting collisions between platforms and their missiles; or aircraft and carriers.

operation has one of the values:

COLL_ADD Ignore collisions with **other_id**.

COLL_REMOVE

Do not ignore collisions with **other_id**.

3 Access Keys

In addition to the functions just described, libcollision also provides libaccess keys with which many variables can be fetched at once. These keys, and the type of argument they expect are given below:

coll_current

    COLL_CURRENT *arg

ADST-TR-W003268

LibComponents

Table of Contents

1 Overview

Libcomponents is an architectural library which provides a level of abstraction away from specific component interfaces. For example, although a vehicle may have one of several sensor models, the interfaces to these models are all basically identical. Libcomponents allows an application to give commands to its "sensor" without knowing which sensor model is being used. Through the use of libparmgr, libcomponents also provides a facility to change, add, or delete a component model after a vehicle has already been created.

The layering of the software looks something like this:

```
---------------------------- specific layer ----------------------

TrackedHull FWAHull                  BallisticGun
WheeledHull RWAHull                 RocketLauncher    Radar
DIHull  MissileHull  GenericTurret  MissileLauncher   Visual

----------------------------- generic layer ----------------------

      Hull              Turret           Gun          Sensor

-------------------------- architectural layer -------------------

                               Components

------------------------------------------------------------------
```

The software layering diagram shown above has been currently implemented via the ModSAF library structure shown below.

```
----------------- specific component libraries -----------------------

libtracked
libwheeled
librwa
libfwa                              libbalgun              libradar
libmissile       libgenturret       libmlauncher           libvisual

------------------ generic component libraries -----------------------

libhulls          libturrets           libguns          libsensors

-------------------------- architectural library --------------------
```

libcomponents

The parametric data of libcomponents identifies each component being used with a name and a model number. Optionally, some components also list what capabilities that component provides to the vehicle when the component is operational. For example, a T72M component entry might look like this:

```
(SM_Components (hull         SM_TrackedHull        SAFCapabilityMobility)
               (turret       SM_GenericTurret)
               (machine-gun  [SM_BallisticGun | 0] SAFCapabilityFirepower)
               (main-gun     [SM_BallisticGun | 1] SAFCapabilityFirepower)
               (visual       SM_Visual))
```

The specific parameters used by each component are listed separately

```
(SM_TrackedHull (max_speed ...)  ...)
(SM_GenericTurret (max_slew ...) ...)
([SM_BallisticGun | 0] (round_types munition_USSR_30mm) ...)
([SM_BallisticGun | 1] (round_types munition_USSR_125HEAT
                                    munition_USSR_125SABOT) ...)
(SM_Visual (max_detectable ...) ...)
```

The T72M whose component entry was displayed above will require one libhull, one libturret, and one libvisual instantiation. It will require two libgun instantiations: one for the machine gun and one for the main gun. The specific component library instantiations will be instances of the generic component library class. This component class relationship for the libraries currently implemented is shown below.

Instantiations of	Belong to generic	Have a command
of the library:	component class:	interface defined in:
libtracked	hull	libhulls
libwheeled	hull	libhulls
librwa	hull	libhulls
libfwa	hull	libhulls
libmissile	hull	libhulls
libgenturret	turret	libturrets
libbalgun	gun	libguns
libmlauncher	gun	libguns
libvisual	sensor	libsensors

libradar	**sensor**	**libsensors**

An application issues commands to a specific component library through its generic component library. For example, an application will interface to libtracked or libfwa through libhulls so that a tank's movement control commands (which are performed by libtracked) and an airplane's movement control commands (which are performed by libfwa) are both issued via the interface defined by libhulls. Similarly, an application will interface to libvisual or libradar through libsensors.

The generic component libraries define the common set of functions that operate in the specific component libraries. Each generic component library (such as libhulls, libsensors, libturret, and libguns) directs libcomponents to define a component class for itself and tells libcomponents the number of its defined functions. This information enables libcomponents to define a structure that accommodates all the components of an object, plus it allows the object's user data to be allocated enough space to hold the address of each function defined in the generic component library.

The interface to a generic component library is defined in it's public header file (such as libturret.h, libguns.h, libsensors.h, and libhulls.h.). These interfaces allow an application to control (set) or learn about (get) such things as component movement, shooting, and sensing. The application gives a command to an objects's component by passing a macro defined in the generic component library. This macro identifies the function which needs to be called. For example, the macro, HULLS_SET_DIRECTION_SPEED, will result in the invocation of the specific component function named, hulls_set_direction_speed.

When a function is to be called, the vehicle id, component number, and function pointer index need to be passed to libcomponents so that the appropriate subclass data can be accessed. The requested function needs an argument list to handle needed input (such as a direction or speed) and/or returned output (such as a state or setting). The interface structure defined in the generic component library defines the argument list which is passed on to a specific component library.

Libcomponents takes care of the following:

- Creating the components listed;
- Associating names of instantiations with user-data handles, for those libraries which support multiple instantiations;
- Dispatching calls defined by the generic layer, but executed by the specific layer; and,
- Creating or deleting components at run time.

Creating/deleting instantiations also impacts the tasks which use those instantiations. For example, a task which looks for targets wants to use all available sensors. Hence, libcomponents must be able to propagate the information of what is available to the tasks at run time. This is handled as follows: component is deleted/added, libcomponents calls a callback routine in libsafobj, libsafobj calls functions in impacted tasks. This facility is also available to handle damage/repair of components (libcomponent provides a function which the component-instance sub-class will call when it is damaged/repaired).

2 Examples

To initialize libhulls, a generic component class:

```
cmpnt_define_class(SM_classHull, HULLS_NUM_FUNCTIONS);
```

To initialize libtracked, an instance of the hull class:

```
tracked_user_data_handle =
  class_reserve_user_data(parent_class, "tracked", tracked_print);

/* Tell libcomponents we are available. */
cmpnt_define_instance(SM_TrackedHull, 1, &tracked_user_data_handle,
                      tracked_create, tracked_destroy,
                      HULLS_SET_DIRECTION_SPEED_FCN, set_dir_speed,
                      HULLS_SET_VELOCITY_GEAR_FCN, set_vel_gear,
                      HULLS_SET_VELOCITY_DIRECTION_FCN, set_vel_dir,
                      HULLS_SET_VELOCITY_ORIENTATION_FCN, set_vel_ori,
                      HULLS_SET_POSITION_DIRECTION_FCN, set_pos_dir,
                      HULLS_SET_GOAL_CORRIDOR_FCN, set_goal_corr,
                      HULLS_SET_TARGET_ID_FCN, set_target_id,
                      HULLS_SET_TARGET_POSITION_FCN, set_target_position,
                      HULLS_GET_ETA_FCN, get_eta);
```

To get the component number of my hull:

```
extern int32 my_hull;

if ((my_hull = cmpnt_locate(vehicle_id, reader_get_symbol("hull"))) ==
    CMPNT_NOT_FOUND)
  printf("Vehicle %d does not seem to have a hull\n", vehicle_id);
```

To then give a command to that hull (the macro is defined by libhulls; it assembles a `HULLS_INTERFACE` structure, and calls `cmpnt_invoke`):

```
if (my_hull != CMPNT_NOT_FOUND)
  HULLS_SET_DIRECTION_SPEED(vehicle_id, hull, dirvec, speed, 0.0, 0.0);
```

3 Functions

The following sections describe each function provided by libcomponents, including the format and meaning of its arguments, and the meaning of its return values (if any).

3.1 cmpnt_init

```
void cmpnt_init()
```

cmpnt_init initializes libcomponents. Call this before calling any other libcomponents functions.

3.2 cmpnt_define_class

```
void cmpnt_define_class(saf_model_class, num_functions)
    uint32 saf_model_class;
    int32 num_functions;
```

'**saf_model_class**'
Specifies the class being defined

'**num_functions**'
Specifies the number of functions provided by that members of the class

cmpnt_define_class defines a class of component. The purpose of a class is to specify the functions provided by components of this class. The **saf_model_class** is one defined in p_safmodels.h.

3.3 cmpnt_define_instance

```
void cmpnt_define_instance(saf_model, num_handles, handles,
                           create, destroy,
                           function_number, function,
                           function_number, function,
                           ...)

    uint32        saf_model;
    int32           num_handles;
    int32           handles[];
    CMPNT_CREATE    create;
    CMPNT_DESTROY   destroy;
    int32           function_number;
    CMPNT_FUNCTION  function;
```

'**saf_model**'
Specifies the model number of the instance (implies a class)

'**num_handles**'
Specifies the number of user data handles provided by the instance (will be > 1 if the instance is multiply instantiable per vehicle)

'**handles**' Specifies a list of user data handles

'**create**' Specifies the function to call to create an instantiation

'**destroy**' Specifies the function to call to destroy an instantiation

'**function_number**'
Specifies a function code number defined for the class

'**function**'
Specifies a function to call when given that **function_number**

cmpnt_define_instance defines an instance of a component class. The instance has a SAF Model number (which implies a class) and a list of functions. The instance also informs libcomponents of how many times it may appear per vehicle, and the handles used for each appearance.

The **create** and **destroy** functions should be of the form:

```
void create(vehicle_id, user_data_handle, params)
    int32   vehicle_id;
    int32   user_data_handle;
    ADDRESS params;

void destroy(vehicle_id, user_data_handle)
    int32 vehicle_id;
    int32 user_data_handle;
```

Each function should be of the form:

```
void function(vehicle_id, user_data_handle, parameters)
    int32              vehicle_id;
    int32              user_data_handle;
    <CLASS>_INTERFACE  parameters;
```

3.4 cmpnt_class_init

```
void cmpnt_class_init(parent_class, availability_fcn)
    CLASS_PTR          parent_class;
    CMPNT_AVAILABILITY availability_fcn;
```

'parent_class'
: Class of the parent (declared with **class_declare_class**)

'availability_fcn'
: Specifies the function to call when component availability changes

cmpnt_class_init creates a handle for attaching components class information to vehicles. The parent_class is one created with class_declare_class.

The **availability_fcn** should be of the form:

```
void availability(vehicle_id, component_number, is_available)
    int32 vehicle_id;
    int32 component_number;
    int32 is_available;
```

Note that this function is not called at vehicle creation (when, technically, components first become available).

3.5 cmpnt_create

```
void cmpnt_create(vehicle_id, parms, name_symbol)
    int32                      vehicle_id;
    COMPONENTS_PARAMETRIC_DATA *parms;
    char                       *name_symbol;
```

'vehicle_id'
: Specifies the vehicle ID

'parms' Specifies the initial parameter values

'name_symbol'
: Specifies the name of the vehicle being created (such as "vehicle_US_M1").

cmpnt_create creates the components class information for a vehicle and attaches it to the vehicle's libclass user data. It also creates the components listed in its parametric data. The **name_symbol** should be a libreader symbol.

3.6 cmpnt_destroy

```
void cmpnt_destroy(vehicle_id)
    int32 vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

cmpnt_destroy frees the components class information for a vehicle. It also frees any component instantiations.

3.7 cmpnt_available

```
void cmpnt_available(vehicle_id, saf_model, is_available)
    int32  vehicle_id;
    uint32 saf_model;
    int32  is_available;
```

'vehicle_id'
Specifies the vehicle ID

'saf_model'
Specifies the model going on/off line

'is_available'
Specifies whether new availability

cmpnt_available informs libcomponents that the component is/isn't available. Libcomponents passes this information on to its parent (such as safobj) which tells users of that component it is available, or not to use it. Components which have been marked as unavailabile will not be able to be located via **cmpnt_locate**.

3.8 cmpnt_locate

```
int32 cmpnt_locate(vehicle_id, specific_name)
    int32 vehicle_id;
    char *specific_name;
```

'vehicle_id'
Specifies the vehicle ID

'specific_name'
Specifies the name of the desired component

cmpnt_locate finds the component number of a specific component name (the name should be a libreader symbol). This number is then used in calls to **cmpnt_invoke**. The return value **CMPNT_NOT_FOUND** is returned when no instance of the passed name can be found. Components which have been marked as unavailabile by **cmpnt_available** will not be able to be located and will return **CMPNT_NOT_FOUND**.

3.9 cmpnt_get_info

```
void cmpnt_get_info(vehicle_id, component_number, specific_name, saf_model)
    int32   vehicle_id;
    int32   component_number;
    char   **specific_name;
    uint32 *saf_model;
```

'vehicle_id'
Specifies the vehicle ID

'component_number'
Specifies the component

'specific_name'
Returns the name of the component

'saf_model'
Returns the model number of the component (from '**p_safmodels.h**').

cmpnt_get_info gets the name and SAF model number of the specified component.

3.10 cmpnt_invoke

```
void cmpnt_invoke(function_number, vehicle_id,
                    component_number, interface_block)
        int32   function_number;
        int32   vehicle_id;
        int32   component_number;
        ADDRESS interface_block;
```

'**function_number**'

Specifies the function to call

'**vehicle_id**'

Specifies the vehicle ID

'**component_number**'

Specifies the component to invoke upon

'**interface_block**'

Specifies a pointer to a block of parameters

cmpnt_invoke calls the specified function for the specified component. The **interface_block** is defined by the component class header file. Note that the class header file generally provides macros which prepare the parameters and invoke this function.

3.11 cmpnt_locate_bymodel

```
    void cmpnt_locate_bymodel(vehicle_id, saf_model, list_size, list)
        int32  vehicle_id;
        uint32 saf_model;
        int32 *list_size;
        int32  list[];
```

'**vehicle_id**'

Specifies the vehicle ID

'**saf_model**'

Specifies the class and instance part of the SAF model, for instance, **SM_Visual**

'**list_size**'

Specifies the size of the passed **list** array
Returns the number of components found

cmpnt_locate_bymodel finds all components with the specified **saf_model** class & instance. The size of the passed list should be passed in ***list_size**. The number of components found is returned in ***list_size**.

3.12 cmpnt_get_capabilities

```
uint32 cmpnt_get_capabilities(vehicle_id)
    int32 vehicle_id;
```

'**vehicle_id**'

Specifies the vehicle ID

cmpnt_get_capabilities finds all the capabilities enabled on a vehicle as determined by having available components with those capabilities, as specified in the components parametric data.

ADST-TR-W003268

LibCoordinates

Table of Contents

1 Overview

There are many coordinate systems used to describe locations on the earth. Libcoordinates provide a facility to translate between any of the following coordinate systems:

COORD_CHARACTER_STRING

Pretty-printed version of one of the other coordinate systems. You can only convert *to* (not from) this system.

COORD_GCC

Geocentric coordinates. This is a right-handed 3D cartesian system, with Z through the north pole and X through the prime meridian at the equator. It assumes the WGS84 ellipsoid model of the earth. This is the coordinate system used in DIS.

COORD_TCC

Topocentric coordinates. This is a right-handed 3D cartesian system, centered around a given point, with Z going up, positive X east, and positive Y north. The system can either be derived from UTM data (as SIMNET terrain databases have been; often called the flat-earth approach) or from GCC data (often called the curved-earth drop-off approach).

COORD_LATLON

Geodetic coordinates. This is the latitude and longitude system used by the air force, navy, and meteorologists. Note that altitude is not measured from sea level, but rather from the reference ellipsoid. Latitude and longitude can be with reference to the mapping datum used at that location, or with reference to the WGS84 ellipsoid.

COORD_UTM_NE

Universal Transverse Mercator projection coordinates. In this system the earth is sliced into 6 degree wide sections (called *zones*), and the terrain in each is flattened out. A location on a slice is identified with a *northing* which indicates distance from the equator in meters, and an *easting* which indicates distance from the center of the zone. Note that when converting from UTM to Geodetic (which is the first step in getting to GCC), the altitude is not changed. This is not entirely correct, since the UTM system increases the altitude at the edges of a zone, and decreases it in the middle. The algorithms to do this altitude adjustment have not been built into libcoordinates.

COORD_UTM_GRID

MilGrid coordinates. This is a shorthand notation expressing a UTM northing and easting in terms of a map sheet, and an offset into that map sheet.

An added complication to these systems is the fact that different parts of the earth are mapped using different assumptions about the earth's size. First, there is the assumption about the shape of the earth, which is called the *spheriod model.* There are 13 different models currently in use. Second, there is the assumption of the location of the center of the earth, which together with a spheroid is called the *datum* (there are 40 of these). The data file '**coordinates.rdr**' specifies *Molodenskiy* parameters for each datum, which allows the software to translate point mapped with each datum to the corresponding point given the current earth model (WGS84). The file also specifies which datums are used in what parts of the world. Finally, the file gives the map sheet code letters used by the MilGrid system in different places.

This data file is not complete — it only covers a small portion of the earth. The instructions for adding descriptions of new areas to this file are contained in a header at the top of the file. Two shell scripts ('**rect**' and '**bounds**') are maintained within this library to assist in generation of the data file.

2 Examples

To define the TCC for Ft. Knox:

```
#include <libctdb.h> /* For datum definition */

COORD_TCC_PTR knox = coord_define_tcc(COORD_UTM_NE,
                                      4155000, 545000, 16, 'S',
                                      DATUM_CONUSNAD27,
                                      75000, 50000);
```

To get the MilGrid designation of the southwest corner of that database:

```
char buf[20];
int32 ret;
float64 z;

if (ret = coord_convert(COORD_TCC,
                        knox, 0.0, 0.0, 185.0,
                        COORD_UTM_GRID,
                        COORD_DEFAULT_ZONE, 8, buf, &z))
  printf("%s\n", convert_error(ret));
```

To get the GCC coordinate of the southwest corner of that database:

```
int32 ret;
float64 x, y, z;

if (ret = coord_convert(COORD_TCC,
                        knox, 0.0, 0.0, 185.0,
                        COORD_GCC,
                        &x, &y, &z))
  printf("%s\n", convert_error(ret));
```

3 Functions

The following sections describe each function provided by libcoordinates, including the format and meaning of its arguments, and the meaning of its return values (if any).

3.1 coord_init

```
int32 coord_init(directory, flags)
    char  *directory;
    uint32 flags;
```

'**directory**'
Specifies the directory where the constants file is expected

'**flags**' Specifies reader options (see section 'reader_read' in LibReader Programmer's Manual)

coord_init initializes libcoordinates, causing it to read its data file (coordinates.rdr) either from '.' or the specified director. The **reader_flags** are as in **reader_read**. The return value is zero if the read succeeds, or one of the libreader return values (**READER_READ_ERROR READER_FILE_NOT_FOUND**) if it fails.

3.2 coord_define_tcc

```
COORD_TCC_PTR coord_define_tcc(source_system,
                               origin_northing, origin_easting,
                               origin_zone_number, origin_zone_letter,
                               mapping_datum,
                               database_width, database_height)
    COORD_SYSTEM source_system;
    float64      origin_northing;
    float64      origin_easting;
    int32        origin_zone_number;
    int32        origin_zone_letter;
    int32        mapping_datum;
    int32        database_width;
    int32        database_height;
```

'**source_system**'
Specifies the underlying system of the topocentric coordinate system

'**origin_northing**'
Specifies the northing of the southwest corner of the TCC

'**origin_easting**'
Specifies the easting of the southwest corner of the TCC

'**origin_zone_number**'
Specifies the zone number of the southwest corner of the TCC

'**origin_zone_letter**'
Specifies the zone letter of the southwest corner of the TCC

'**mapping_datum**'
Specifies the datum in which the TCC data is mapped (only used if **source_system** is **UTM_NE**)

'**database_width**'
Specifies the east-west size of the TCC system (in meters)

'**database_height**'
Specifies the south-north size of the TCC system (in meters)

coord_define_tcc defines a "topocentric" coordinate system (centered around a given location, at which Z is up, X is east, and Y is north). The **source_system** must be either **COORD_GCC** or **COORD_UTM_NE**. GCC derived TCC systems curve "down" in their Z values as distance increases from the center point. UTM derived TCC systems are completely flat. It is significantly cheaper to convert TCC coordinates to or from the system from which it was derived than to or from the other system. Use the returned pointer in calls to **coord_convert**.

3.3 coord_estimate_tcc

```
COORD_TCC_PTR coord_estimate_tcc(source_system,
                                  latitude_hun_sec, longitude_hun_sec,
                                  origin_utm_grid,
                                  database_width, database_height)
    COORD_SYSTEM source_system;
    int32         latitude_hun_sec;
    int32         longitude_hun_sec;
    char         *origin_utm_grid;
    int32         database_width;
    int32         database_height;
```

'**source_system**'
Specifies the underlying system of the topocentric coordinate system

'latitude_hun_sec'
Specifies the latitude of the southwest corner of the TCC in hundredths of seconds

'longitude_hun_sec'
Specifies the longitude of the southwest corner of the TCC in hundredths of seconds

'origin_utm_grid'
Specifies the UTM grid specification of the southwest corner of the TCC

'database_width'
Specifies the east-west size of the TCC system (in meters)

'database_height'
Specifies the south-north size of the TCC system (in meters)

coord_estimate_tcc makes a reasonably accurate guess at the TCC parameters, based upon commonly known information.

3.4 coord_tcc_gcc_rotation

```
int32 coord_tcc_gcc_rotation(tcc, matrix)
    COORD_TCC_PTR tcc;
    float64       matrix[3][3];
```

'tcc' Specifies the TCC

'matrix' Returns the TCC to GCC rotation matrix

coord_tcc_gcc_rotation fills in the passed 3x3 TCC to GCC rotation matrix based upon the GCC-derived **tcc** (the GCC to TCC matrix is the transpose of the returned matrix). The return value, if non-zero indicates that an error occured. Passing a UTM_NE-derived TCC will yield an error (the correct rotation from UTM to GCC is dependent upon the UTM location).

3.5 coord_convert

```
int32 coord_convert(from_system, values..., to_system, values...)
    COORD_SYSTEM from_system;
    values...;
    COORD_SYSTEM to_system;
    values...;
```

'from_system'
Specifies the system to convert from

'to_system'
Specifies the system to convert to

coord_convert takes two sets of arguments — from_system and to_system. Each system has a unique code and a series of values:

COORD_CHARACTER_STRING
string

COORD_GCC
x y z

COORD_TCC
tcc_ptr x y z

COORD_LATLON
latitude longitude z local_datum

COORD_UTM_NE
zone northing easting z

COORD_UTM_GRID
zone resolution string z

The types of these arguments are shown in the following table:

var	from_type	to_type	units	example
x	float64	float64 *	meters	5000.0
y	float64	float64 *	meters	5000.0
z	float64	float64 *	meters	283.0
zone	int32	int32 *		38 or COORD_DEFAULT_ZONE
northing	float64	float64 *	meters	500000.0 (+N -S)
easting	float64	float64 *	meters	500000.0
resolution	int32	int32	digits	3 --> ES450550
latitude	float64	float64 *	degrees	39.0 (+N -S)
longitude	float64	float64 *	degrees	42.0 (+E -W)
local_datum	int32			TRUE/FALSE
tcc_ptr	COORD_TCC_PTR			
string	char *			

Note: UTM_GRID resolution is ignored for from_system
Explicit zone in UTM_GRID string overrides passed zone
local_datum=FALSE implies to use WGS84, as with GPS

Non-zero return values indicate the following errors:

COORD_SYNTAX
Character string not of correct format

COORD_TOO_LONG
Too many milgrid digits passed

COORD_ZONE_UNKNOWN
Milgrid in unrecognized UTM zone

COORD_GRID_UNKNOWN
Milgrid grid not recognized in UTM zone

COORD_BAD_TCC
Passed TCC pointer invalid

COORD_SYSTEM_UNKNOWN
Either to_system or from_system is not recognized

3.6 coord_error

```
char *coord_error(code)
    int32 code;
```

'code'

coord_error returns a string describing the error code returned by **coord_convert**.

3.7 coord_generate_grid

```
void coord_generate_grid(tcc, min_x, max_x, min_y, max_y,
                         spacing, digits, label_mask,
                         max_thin_segments, max_thick_segments, max_labels,
                         n_thin_segments, thin_segments,
                         n_thick_segments, thick_segments,
                         n_labels, labels)
    COORD_TCC_PTR    tcc;
    int32            min_x, max_x, min_y, max_y;
    int32            spacing;
    int32            digits;
    uint32           label_mask;
    int32            max_thin_segments;
```

```
        int32            max_thick_segments;
        int32            max_labels;
        int32           *n_thin_segments;
        float32          thin_segments[];
        int32           *n_thick_segments;
        float32          thick_segments[];
        int32           *n_labels;
        COORD_GRID_LABEL labels[];
```

'tcc' Specifies the TCC of the grid

'min_x, max_x, min_y, max_y'
Specify the boundaries of the grid in TCC coordinates

'spacing' Specifies the grid line spacing (typically multiples of 10)

'digits' Specifies the number of digits with which to label grid lines

'label_mask'
Specifies the sides to label

'max_thin_segments, max_thick_segments, max_labels'
Specifies the sizes of the three passed arrays

'n_thin_segments, n_thick_segments, n_labels'
Returns the number of returned data items in each array

'thin_segments'
Returns a flat list of n_thin_segments X/Y->X/Y line segments

'thick_segments'
Returns a flat list of n_thick_segments X/Y->X/Y line segments

'labels' Returns a list of labels

coord_generate_grid takes bounding points in TCC coordinates and generates two lists of line segments (thin and thick) which form a grid with the specified spacing, and a list of text strings which can be used to label the grid. Note that up to 4 * max_thin_segments may be stored in the thin_segments array.

label_mask indicates which sides to label with an inclusive OR of the following masks:

- COORD_LABEL_LEFT
- COORD_LABEL_RIGHT
- COORD_LABEL_TOP
- COORD_LABEL_BOTTOM

The special mask COORD_LABEL_ALL is the OR of all of these.

Labels are specifies with the following structure:

```
typedef struct
{
    /* The location of the label in TCC coordinates */
    int32 x, y;
    /* Null-terminated label string */
    char txt[COORD_MAX_GRID_LABEL_TXT];
} COORD_GRID_LABEL;
```

3.8 coord_describe_datums

```
void coord_describe_datums()
```

coord_describe_datums prints descriptions of all known datums (from table read in coord_init).

3.9 coord_format_latlon

```
char *coord_format_latlon(latitude, longitude)
    float64 latitude;
    float64 longitude;
```

'latitude'
Specifies the latitude

'longitude'
Specifies the longitude

coord_format_latlon returns a pointer to a character buffer which has the passed lat/long represented as ASCII text. The buffer is static, so only one invocation per argument list is allowed.

3.10 coord_fixed_point_degrees

```
int32 coord_fixed_point_degrees(degrees)
    float64 degrees;
```

'degrees' Specifies an angle in degrees

coord_fixed_point_degrees returns an integer which is the passed number of degrees in a DDMMSSHH format.

3.11 coord_floating_point_degrees

```
float64 coord_floating_point_degrees(ddmmsshh)
    int32 ddmmsshh;
```

'ddmmsshh'
Specifies the encoded number

coord_floating_point_degrees decodes DDMMSSHH format into simple degrees.

3.12 coord_parse_fixed_point_degrees

```
int32 coord_parse_fixed_point_degrees(string)
    char *string;
```

'string' Specifies a string in DDMMSSHH format

coord_parse_fixed_point_degrees parses character string to generate DDMMSSHH format fixed point integer.

3.13 coord_count_utm_zones

```
void coord_count_utm_zones(tcc_ptr, n_zones, base_zone)
    COORD_TCC_PTR tcc_ptr;
    int32         *n_zones;
    int32         *base_zone;
```

'tcc_ptr' Specifies the TCC

'n_zones' Returns the number of zones

'base_zone'
Returns the number of the lowest zone

coord_count_utm_zones returns the number of utm zones covered by the database TCC, and the number of the lowest zone.

3.14 coord_lookup_datum_by_ne

```
int32 coord_lookup_datum_by_ne(zone, northing, easting)
    int32   zone;
    float64 northing;
    float64 easting;
```

'zone' Specifies the zone

'northing'
Specifies the northing

'easting' Specifies the easting

coord_lookup_datum_by_ne returns the DMA suggested mapping datum for a particular location.

3.15 coord_lookup_zone_letter

```
char coord_lookup_zone_letter(zone, northing)
    int32   zone;
    float64 northing;
```

'zone' Specifies the zone

'northing'
Specifies the northing

Given a zone & a northing, **coord_lookup_zone_letter** looks up the correct letter designation.

3.16 COORD_LATLONG_TO_GRID_ZONE

```
int32 COORD_LATLONG_TO_GRID_ZONE(latitude, longitude)
    float64 latitude;
    float64 longitude;
```

'latitude'
Specifies the latitude

'longitude'
Specifies the longitude

Given a latitude & longitude, the macro COORD_LATLONG_TO_GRID_ZONE determines the UTM zone number in which that coordinates resides. With the exception of a couple of anomalies, the world is broken up into 6 degree wide slices.

3.17 COORD_WEST_LONG_OF_GRID_ZONE

```
float64 COORD_WEST_LONG_OF_GRID_ZONE(zone)
    int32 zone;
```

'zone' Specifies the zone

Given a grid zone (in the range 1 through 60), the macro COORD_WEST_LONG_OF_GRID_ZONE returns the longitude of the western edge of that zone at the equator.

3.18 COORD_EAST_LONG_OF_GRID_ZONE

```
float64 COORD_EAST_LONG_OF_GRID_ZONE(zone)
    int32 zone;
```

'zone' Specifies the zone

Given a grid zone (in the range 1 through 60), the macro COORD_EAST_LONG_OF_GRID_ZONE returns the longitude of the eastern edge of that zone at the equator.

LibCreate

Table of Contents

1 Overview

Libcreate provides the following services for simulated ModSAF entities:

- distributed creation of simulated entities with load balancing between computers of the same **simulatorType** (such as simulator_LL_SAFSIM, a type that indicates the computer's ability to simulate entities)
- application directed handoff of simulated entities from one simulator to another
- fault-tolerant takeover of simulated ModSAF entities belonging to a simulator of a similar **simulatorType** when that simulator crashes or exits

The algorithms used to provide these services are provided in the next chapter.

Libcreate works by attaching **object_changed** and **simulator_gone** handlers (see section 'Events and Event Handlers' in LibPO Programmer's Manual) in the PO database, as well as through two interface functions to deal with new and deleted objects.

When a PO entry indicates that a unit object that has not yet been simulated and that entry's "shouldBeSimulated' field is set at TRUE, then libcreate simulates the unit via the following procedure.

1. Make a call to libsafobj, telling it to go ahead and instantiate the corresponding object as a local vehicle. The unit_name of the entry (such as US_F14D) matches to a .rdr file (such as US_F14D_params.rdr) so that the appropriate configuration files are read. Libsafobj returns a vehicle_id when the object has been instantiated.
2. Set the simulated entity to have needed data (such as exercise number, location, forceID, marking, and appearance). This data is passed to libentity from libcreate and the Unit PO (libPO). Next, build a rotation matrix corresponding to the direction the user wanted the unit to face, and set the simulated entity to point that way.
3. Initialize the hull component of the safobj by having the hull point in the desired direction and by assigning an initial speed of zero.
4. Make the appropriated changes to the PO to reflect that this object has been simulated. (The "simulated" field will be set to TRUE, and the "shouldBeSimulated" field will be set to FALSE.)
5. Make a call to libentity to activate the new entity, that is, have it start broadcasting packets.
6. Finally, set the association between the safobj and the c2obj by mapping the safobj's vehicle_id with the c2obj's unit PO.

2 Algorithms

The following sections describe the algorithmic implmentations of the load-leveling creation, migration, and fault-tolerant takeover services of libcreate. Each algorithm depends on the fact that every ModSAF simulated entity has a corresponding Unit Class Persistent Object (PO) corresponding to that simulated entity. The Unit Class objects provide the shared state variables between simulators that are used to arbitrate who will simulate an entity at what time. Since the persistent Unit Class objects will in the steady state be consistent across all simulators, the decision of what simulator will simulate what entity will be consistent across all simulators. Since the persistent Unit Class objects can survive a simulator crashing, the simulated entities derived from those persistent Unit Class objects can survive a simulator crashing.

Through out each algorithm description, it is important to note that although local inconsistencies concerning a PO object can occur temporarily between simulators, over time each simulator will have consistent information about each object in the database.

The following fields in the Unit Class object are relevant to the creation process:

shouldBeSimulated
TRUE if a Unit Class object should be simulated as a simulation entity

simulated
TRUE if a Unit Class object is simulated as a simulation entity. This field is never true if **shouldBeSimulated** is **TRUE**.

simulator
SimulationAddress of the simulator which currently is or most recently has simulated an entity corresponding to this object

simulationID
VehicleID of the simulated entity corresponding to this object

2.1 Load Leveling Creation

Given an object **A** in which **shouldBeSimulated == TRUE** and **simulated == FALSE**, there are three cases:

1. If **simulator** is not **NULL**, then change the object to have **simulator == B**, where **B** is the simulator registering the least simulation load in it's periodic **simulatorPresentPDU**. Increase

your local notion of that simulator's load based on what increased load you would incur for that vehicle.

2. If **simulator** is you, wait for a small amount of time and then perform the **maybe_create** procedure.
3. If **simulator** is not you, ignore the creation which has been selected for another simulator.

When the **maybe_create** procedure is executed, if you are still the selected **simulator** for that object, and the object still wants to be simulated, simulate the entity and set **shouldBeSimulated = FALSE**, **simulated=TRUE**, and **simulationID** to the id of the newly simulated entity.

If, through missed PO packets, two or more simulators decide to simulate entities corresponding to the same object, all but one of the simulators will end up destroying the entities through the process of migration, described below.

2.2 Migration

Given an object **A** in which **Simulated == TRUE** and **SimulationID** is a local vehicle and **simulator** is not me, this indicates a request for me to release my vehicle to the other **simulator**. To do this, deactivate the vehicle with a **deactivateReason** of **vehicleHandoff**.

Given an object **A** in which **Simulated == TRUE** and **SimulationID** is a remote vehicle and **simulator** is me, this indicates a request for me to take over a vehicle from another simulator. To do this, create a local vehicle immediately, reusing the **simulationID**. Activate this vehicle (i.e., broadcast vehicle appearance packets) upon a short delay or receipt of a **deactivatePDU** for that vehicle with a **deactivateReason** of **vehicleHandoff**, which ever occurs first.

Note that where load-leveling creation does load-leveling at the time that simulated vehicles are created, migration can be used to do load-leveling after creation.

Also note that it is not safe for a program to exit soon after performing a migration of it's units. This is because it is possible to start a migration to a simulator that has crashed. The source simulator performing the migration may be the only simulator to hear about the simulator crashing *after* beginning the migration, and in that case, the source simulator is the only simulator who will know to change the migration destination via the Fault Tolerant Takeover process.

2.3 Fault Tolerant Takeover

Given a simulator S which has timed out of the database, query for all objects U in which simulator == S and simulated == TRUE. For each of those objects, set simulated == FALSE and shouldBeSimulated == TRUE. Perform the choose operation as in (see Section 2.1 [Load Leveling Creation], page 3), and this case reduces to Load Leveling Creation.

2.4 Defeating Load Leveling Creation

Not all simulated vehicles want to be load-leveled across different simulators. For instance, some applications may want all the vehicles in a platoon to be on one simulator. This can be accomplished via creation conventions when creating unit hierarchies (task organizations). If a simulator (or user interface) wants a unit and all it's subordinates simulated on the same simulator, it should mark just the unit with shouldBeSimulated == TRUE. If it wants any subordinates to be potentially created on other simulators, those should be marked as shouldBeSimulated == TRUE. When a simulator goes to choose a simulator for an object where shouldBeSimulated == TRUE, it should scan the entire PO database for any subordinates where shouldBeSimulated == FALSE. All those units should be changed to have the same simulator simulate them, and the subordinates should be modified to have shouldBeSimulated == TRUE once the destination simulator is chosen.

3 Examples

The following code fragments demonstrate usage of all libcreate functions:

```
    /* 1. Acquire a simulation exercise.
     * 2. Acquire a simulation address, sim_addr.
     * 3. Determine the number of vehicles this simulator can
     *    simulate, and take the inverse of that number to calculate
     *    a loading factor.
     * 4. Determine the type of simulator you are, typically a
     *    simulator_LL_SAFSIM (for load-leveling SAFsim).
 * 5. Determine the simulation protocol version being used.
     * 6. Create an active open PO database, po_db.
     * 7. Create a packetvalve, valve, and set it up to process PO
     *    packets.
     */
    cr_init(po_db, valve, protocol,
exercise, sim_addr, simulator_type, loading);

    /*
     * Turn on creation debugging.
     */
    cr_debugging(TRUE);

    /*
     * Change all (past, present, and future) my created vehicles to
     *  a new exercise.
     */
    cr_change_exercise(new_ex);
```

4 Functions

The following sections describe each function provided by libcreate, including the format and meaning of its arguments, and the meaning of its return values (if any). The algorithms for load-leveling creation, migration, and fault-tolerant takeover are described in (see Chapter 2 [Algorithms], page 3).

4.1 cr_init

```
void cr_init(db, sim_exercise_id, sim_address, sim_type,
          loading_factor)
      PO_DATABASE       *db;
      PV_VALVE_PTR       packet_valve;
      int32              sim_protocol_version;
      uint8              sim_exercise_id;
      SimulationAddress  sim_address;
      SimulatorType      sim_type;
      float32            loading_factor;
```

db an open active PO database

packet_valve
the **libpktvalve** valve being used for reading and writing PDUs.

sim_protocol_version
the version of the SimulationProtocol being used for this exercise.

sim_exercise_id
the exercise to simulate entities on. This should match the exercise being listened to for SimulationProtocol packets in the packet valve.

sim_address
simulation address being used by this computer, and it must match the address used when db was opened.

sim_type the type of simulator that can be chosen to simulate entities

loading_factor
the weight that one vehicle has on the simulation as a fraction of full load

cr_init initializes libcreate, causing it to listen for new and changed **UnitClass** objects in the PO database and to respond to those objects with local creation and deletion of entities.

4.2 cr_change_exercise

```
void cr_change_exercise(sim_exercise_id)
        uint8 sim_exercise_id;
```

'sim_exercise_id'
Specifies a new exercise id.

cr_change_exercise causes all locally created entities to change to a new exercise and forces all newly created entities to be created in the specified exercise.

4.3 cr_debugging

```
void cr_dcbugging(flag)
        int32 flag;
```

'flag' Specifies whether or not debugging statements should be printed. Has either TRUE or FALSE value.

cr_debugging turns debugging print statements either on or off, according to the flag. When debugging is on, indications of the creation process are printed to stdout.

4.4 cr_new_object

```
void cr_new_object(db, entry)
        PO_DATABASE *db;
        PO_DB_ENTRY *entry;
```

'db' A Persistent Object database.

'entry' A new Persistent Object entry that just arrived in the database.

cr_new_object should be called for every new PO entry. If the entry is of type objectClassUnit and it should be simulated as a simulation object, this function will cause the appropriate entity to be created.

4.5 cr_destroy_object

```
void cr_destroy_object(db,entry)
        PO_DATABASE *db;
        PO_DB_ENTRY *entry;
```

'db' A Persistent Object database.

'entry' A new Persistent Object entry that is about to be deleted from the PO database.

cr_destroy_object should be called for every PO entry which is about to be deleted. If the entry is of type objectClassUnit and it is being simulated, this function will cause the appropriate entity to be deactivated and destroyed.

LibCTDB

Table of Contents

1 Why Libctdb?

The Compact Terrain Database library, libctdb, is used by an application to access elevation, soil type, and feature data of a SIMNET database. Terrain databases are compiled from S1000 source (or other source formats) into libctdb format. Applications use libctdb to load this database into memory, and then use libctdb functions to access the data therein.

Libctdb functions include:

- Reading the database into memory or cache
- Maintaining useful information about the database, such as its size, minimum and maximum elevation, and UTM zone, northing and easting (its location on the planet)
- Point elevation lookup
- Elevation lookup along a line segment (find high ground, find terrain profile, etc.)
- Soil type lookup
- Vehicle placement (rotation matrix generation)
- Intervisibility calculation (including terrain and vehicle blockage)
- Radar clutter calculation
- Generating graphic representation of the terrain such as contour maps and hypsometric maps, in real time

The CTDB header format includes a flag which indicates whether the database was generated assuming grid squares break along diagonals running from NW to SE or along diagonals running from SW to NE. The library's public funtions take note of this flag at invocation and apply different internal algorithms accordingly see Chapter 6 [Algorithms], page 57.

Libctdb provides several advantages over other terrain access libraries used in the past. Some of the highlights are detailed in the following sections.

1.1 Smaller storage requirements

The libctdb terrain database file format uses several compression methods to minimize storage requirements. The resulting files are one-sixth to one-twentieth the size of files used by previous terrain database libraries. In addition to reducing hardware cost associated with storing these databases, this compression allows more of the database to be stored in main memory (improving performance by reducing disk access).

The largest savings is derived by exploiting the regular nature of terrain database modeling. Most elevation data is obtained from the Defense Mapping Agency (DMA). DMA data is a regular sampling of elevations. The sampling interval is specified as "Level 1" (90 meter spacing) or "Level 2" (30 meter spacing). SIMNET databases are built from this data as follows:

1. The DMA data is down-sampled to 125 meter spacing. This is done using linear interpolation. We refer to a point in this 125 meter grid as a *post*.
2. Each square in the grid is broken into two triangles via a line running northwest to southeast:

```
+----+----+
|\   |\   |
| \  | \  |
|  \ |  \ |
|   \|   \|
+----+----+
|\   |\   |
| \  | \  |
|  \ |  \ |
|   \|   \|
+----+----+
```

   and each triangle is assigned a soil type.
3. The grid is broken into *patches* 500 meters on a side:

```
+-+-+-+-+-+-+-+-+
|       |       |
+ + + + + + + + +
|       |       |
+ + + + + + + + +
|       |       |
+ + + + + + + + +
|       |       |
+-+-+-+-+-+-+-+-+
```

Each post, therefore, has an elevation and two soil types (by convention, the soil types of the two triangles to its southeast). The libctdb format encodes the information regarding each post (elevation, two soil, and some flags) into a 32 bit item. Hence, to store a grid the size of the Ft. Knox database (50 km x 75 km) requires 400 posts x 600 posts x 4 bytes per post = just under 1 MB. The posts are stored in such a way that the post for a particular X,Y location can be found with a few arithmetic operations.

In addition to this regular grid of elevations, some areas of the terrain are modeled more precisely using *microterrain*. Microterrain is a collection of squares and triangles which cover a portion of a patch. Libctdb uses an original algorithm to store these microterrain polygons in a compact fashion.

On the surface of the terrain are "buildings" (which is loosely defined to include pipelines and other opaque non-penetrable structures), trees (individual trees, tree lines, and tree canopies), and linear features (roads and rivers). For each patch, the microterrain and surface features (each encoded in a compact fashion) are stored together.

Adding features to the Ft. Knox database brings the total storage requirement up to 4.5 MB. By contrast, other database formats typically require around 30 MB to store Ft. Knox. The large SAKI database (360 km x 290 km) requires about 30 MB in libctdb format, over 600 MB in other formats. The greater compression is achieved because the SAKI database is mostly desert, and hence relatively free of features.

1.2 More efficient disk cache

Although libctdb terrain databases are relatively small, they may still require more memory than an application can afford to give up. The solution to this problem is the use of a disk cache. Libctdb uses two caches, one for the elevation grid, and another for the patch features.

When an application loads a libctdb database, it specifies the maximum memory which can be used to store the database. This memory is split up between the two caches, such that the probability of a hit is the same for each. This will typically result in more memory allocated for the feature cache, since databases will usually require more storage for features than for elevations.

1.2.1 Elevation cache

The elevation data is stored using a technique called *tiling*, in which the large rectangular grid is broken into equally sized sub-rectangles. Each sub-rectangle (called a *page*) is a square, with 32 posts on a side (4 km). The 32 x 32 posts require 4 Kb of storage. Since the posts are grouped into geographically close areas, and applications will typically be accessing geographically close areas, cache consistency (the probability a needed page is in the cache) will be high for most applications.

The elevation cache is a direct mapped, fixed size cache. That is, when the application needs a page of elevation data which is not currently in the cache, it is loaded into the cache location `page_number MODULUS cache_size`. The page numbers are assigned in east to west (minor), south to north (major) order. Such a caching scheme, runs the risk of *thrashing* (consistently missing) when two different pages which happen to occupy the same cache location are needed simultaneously. For example, if the database is 20 pages wide, and the cache size is 20 pages, running visibility rays due north will cause frequent cache misses. To avoid this problem, the caching code monitors cache performance, and when thrashing is detected, the cache size is lowered by one (within reason).

1.2.2 Feature cache

Feature data presents a different problem. Some patches have very few features, others have very many. The quantity of feature data in a patch is referred to as *feature density.* Measured in 4 byte words, Ft. Knox has feature densities ranging from 1 to 739 for a *patch group* of 4 adjacent patches (grouping patches into sets of four is a technique used to reduce overhead). Hence, using a single cache in which each entry can hold any patch (the technique used for the elevation cache) will result is a great deal of *internal fragmentation* (wasted space). Instead, during compilation, each patch group is assigned to one of 16 caches. Each cache has about the same number of entries, but the size of each entry varies by cache to minimize fragmentation.

1.3 New algorithms

New algorithms have been developed to improve performance of libctdb functions. The key improvements have been the elimination of unnecessary search, the use of more efficient methods for several key routines, and the judicious use of approximation.

1.3.1 Elimination of search

Finding an elevation on the regular grid does not require any search, except in the case where microterrain is present.

In addition to elevation and soil information, each post in the regular grid has boolean flags which indicate the presence of microterrain, buildings, trees, or other features. By checking these flags, libctdb can avoid searching for features it will not find. In the case of NW_SE diagonalization, the flags indicate the presence of features in the square to the southeast, and the case of SW_NE diagonalization, the flags refer to features in the square to the northeast of the post.

Furthermore, each feature is marked with a bit mask, identifying which posts it crosses within its patch. In the following example, each post within the patch is identified by the hexadecimal digits '0' through 'F'. A road is drawn with the symbol '#'.

```
For NW_SE diagonalization:
C-D-E-F-+
|       |   0000 = 0
8 9 A B +
|    ###|   1100 = C
4 5 6#7 +
|  ###  |   0110 = 6
0 1#2 3 +
|  #    |   0010 = 2
+-+-+-+-+

For SW_NE diagonalization:
+-+-+-+-+
|       |   0000 = 0
C D E F +
|    ###|   1100 = C
8 9 A#B +
|  ###  |   0110 = 6
4 5#6 7 +
|  #    |   0010 = 2
0-1-2-3-+
```

Notice that the post mask for a feature turns out to be the same regardless of the diagonalization of the underlying grid. The road covers the triangles associated with posts 1, 5, 6, A, and B. Hence it would be marked with the flags 0x0C62. Twenty-five bits are reserved for this mask to accommodate future database requirements. A function finding the soil type of a point southeast of post 6 and northwest of post 3, could eliminate all roads which do not have bit 6 set. In other words, the domain of feature searches can be limited to very likely candidates with only one arithmetic operation per feature.

Another critical algorithm which has been coded to minimize search is the intervisibility algorithm. Using a technique borrowed from computer graphics called *digitalization*, *rasterization*, or *scan conversion*, the triangle edges which cross a line of sight ray can be predicted almost exactly. In the following example, the line of sight ray is show on the left as a series of '.', the posts determined to be involved are shown in the middle as '#', and the edges tested are shown on the right. (Interior posts removed for clarity.)

```
+ + +.+ +     + + $ + +     + + $ + +     0100 = 4
     .                          |\
+   .   +     +   $   +     +   $-  +     0100 = 4
    .                           |\
+   .   +     + $ $   +     + $-$-  +     0110 = 6
   .                          |\|\
+  .    +     + $     +     + $-    +     0010 = 2
   .                          |\
+ . + + +     + + + + +     + + + + +

Sight Line    Posts         Edges
```

As the posts are determined by the rasterization algorithm, a mask can be assembled which encodes which triangles line of sight crosses (in the previous example it would be '0x4462'). This mask can be compared against the masks associated with features such as trees, buildings, and microterrain, to reduce the search domain. Also the feature flags associated with the posts traversed are used to eliminate entire classes of features which could not possibly have interfered with visibility on the patch being tested.

1.3.2 Efficient implementations

Many often-invoked routines have been written using a mathematical construct known as *parametric equations*, in which coordinates of a line are described by simultaneous equations of a common parameter, rather than one in terms of the other. A line segment would be defined,

$$x = f(t), \ y = g(t), \ 0 \leq t \leq 1$$

rather than,

$$y = f(x), \ X_{min} \leq x \leq X_{max}, \ Y_{min} \leq y \leq Y_{max}.$$

Not only is it less expensive to solve for t than for x and y in most cases, but in the case of intersecting finite line segments, the point of intersection can usually be determined to be out of bounds ($t < 0$, or $t > 1$) before it is actually computed (t will typically be computed via a division, hence the range of t can be determined by comparing the magnitude of the numerator versus that of the denominator).

In addition to line/line intersection tests, parametric equations are also used to improve performance of linear interpolation and point/line proximity tests.

1.3.3 Judicious approximation

In some cases, reasonably accurate approximations are used rather than computing the exact answer. The most notable example of this is the use of single precision square root, `fsqrt()`, rather than the float64 precision version, `sqrt()`. Also, the vehicle placement code uses an approximation to guarantee the rotation matrix generated is orthonormal, avoiding a square root and an expensive matrix operation.

1.4 Optimized for RISC processors

The libctdb functions have been extensively profiled and tuned to yield maximum performance on RISC platforms. The types of optimizations which help RISC platforms are foreign to many programmers, and hence are rarely performed. They include:

- Avoiding conversions between different numeric formats (integer, single precision floating point, and float64 precision floating point).
- The use of float64 precision floating point rather than single precision for temporary variables to avoid type conversions during evaluation.
- Using many local variables, rather than small, fixed size arrays to make maximum use of CPU registers.
- Using multiplication, rather than division, wherever possible.
- Judicious use of static global variables to reduce parameter passing overhead in function invocations.
- Use of array syntax (`a[i]`), rather than pointer syntax (`*(a+i)`), in cases where the latter would prevent certain optimizations by the compiler.
- Use of `if ... else if ... else if` chains rather than `switch case` statements when the number of choices is small.

Performance tests were used to confirm choices of syntax in all cases.

1.5 Improved intervisibility model

In addition to execution speed improvements, the fidelity of the intervisibility model has been significantly improved. The new model incorporates:

- Light transmittance model used for modeling cumulative effect of intervening trees, with tree opacity a caller specified parameter.
- Use of both target width and height (relative to the viewer) for determination of visible area (older models used height only).
- Accurate model of building sizes and locations (older models used only compass aligned bounding box).
- Determination of visibility as an analog value in the range 0 to 1, rather than a digital Visible/Partial/Blocked result.
- Accurate model of vehicle-blocking-vehicle visibility, incorporating width and height (relative to the viewer) of intervening vehicles.
- Reasonably fast, extremely high fidelity rasterization model available, which can determine exactly whăt parts of a vehicle are visible.

1.6 More thorough documentation

In addition to this document, libctdb is documented via extensive in-line comments, and a documented header file.

2 Examples

The following programs demonstrate the use of selected libctdb functions. The programs were intentionally kept brief for clarity. In general, a program should declare a variable of type 'CTDB' to hold header information, call ctdb_read(...) to initialize this structure, and then call other libctdb functions with a pointer to this variable as their first argument.

2.0.1 Elevation Lookup

The following program reads a libctdb format database, prints information in the database header, and finds the elevation at the center of the database.

```
/*
 * File: dbcenter.c
 *
 * Compiled as follows:
 *
 *  cc -o dbcenter dbcenter.c -I/common/include/libinc -L/common/lib \
 *     -lctdb -lm
 */

/* Include this header in all files referencing libctdb functions,
 * constants, or data structures.
 */
#include <libctdb.h>

main(argc, argv)
    int32 argc;
    char *argv[];
{
    CTDB ctdb; /* Allocate storage on the stack for the database
                * header.  Most of the memory used by libctdb
                * will be allocated dynamically when the database
                * is read.
                */
    float64 mid_x, mid_y, mid_z;

    if (argc != 2)
    {
        printf("Usage: %s <file-name>\n", argv[0]);
        exit(0);
    }

    /* Read the database.  Allow the library no more than 1MB of
```

```
     * storage.  Note that if an error occurs, libctdb will call
     * exit(1).
     */
    ctdb_read(&ctdb, argv[1], 1<<20);

    /* Print a description of the database.
     */
    ctdb_print_description(&ctdb);

    /* Find the midpoint of the database.
     */
    mid_x = (ctdb.max_x + ctdb.min_x) / 2.0;
    mid_y = (ctdb.max_y + ctdb.min_y) / 2.0;

    /* Find the elevation at that point.
     */
    mid_z = ctdb_lookup_elevation(&ctdb, mid_x, mid_y);

    /* Print the result.
     */
    printf("Center of database: <%f %f %f>\n",
           mid_x, mid_y, mid_z);
}
```

2.0.2 Visibility Testing

The following program reads a libctdb format database, and determines visibility between two user specified points. Libcoord is used to translate the user specified coordinates from UTM to X,Y. Libreader is used to read the libcoord database.

Note that information in the CTDB data structure can be used to initialize libcoord.

```
/*
 * File: ckvis.c
 *
 * Compiled as follows:
 *
 *  cc -o ckvis ckvis.c -I/common/include/libinc -L/common/lib \
 *     -lctdb -lcoord -lreader -ll -lm
 */

/* Include this header in all files referencing libctdb functions,
 * constants, or data structures.
 */
#include <libctdb.h>
```

```
/* Other header files */
#include <libreader.h>
#include <libcoord.h>

/* Typical values */
#define EYE_HEIGHT    2.0
#define TARGET_HEIGHT 2.0
#define TARGET_WIDTH  4.0
#define TREE_OPACITY  0.5 /* Must be between 0.0 and 1.0 */

main(argc, argv)
    int32 argc;
    char *argv[];
{
    CTDB ctdb; /* Allocate storage on the stack for the database
                * header.  Most of the memory used by libctdb
                * will be allocated dynamically when the database
                * is read.
                */
    DATA_UNION utm_file;
    int32 ret;
    float64 eye_x, eye_y, eye_z;
    float64 targ_x, targ_y, targ_z;
    float64 vis;

    if (argc != 4)
    {
        printf("Usage: %s <file-name> <eye UTM> <target UTM>\n",
               argv[0]);
        exit(0);
    }

    /* Read the database.  Allow the library no more than 1MB of
     * storage.  Note that if an error occurs, libctdb will call
     * exit(1).
     */
    ctdb_read(&ctdb, argv[1], 1<<20);

    /* Initialize libcoord.
     */
    if (reader_read_file("/saf/config/utm.lisp", &utm_file) != 0)
    {
        printf("Error reading libcoord database\n");
        exit(0);
    }
    coord_set_milgrid_table(utm_file.array);
    simnet_origin_northing =    ctdb.origin_northing;
    simnet_origin_easting =     ctdb.origin_easting;
    simnet_origin_zone =        ctdb.origin_zone_number;
    simnet_origin_zone_letter = ctdb.origin_zone_letter;
```

```
simnet_datum =              ctdb.datum;

/* Convert user supplied coordinates.
 */
ret = convert_coordinates(COORD_SYSTEM_UTMGRID,
                          COORD_DEFAULT_ZONE, 0, argv[1],
                          COORD_SYSTEM_SIMNET,
                          &eye_x, &eye_y);
if (ret)
{
    printf("Error converting UTM %s: %s\n",
           argv[1], convert_error(ret));
    exit(0);
}

ret = convert_coordinates(COORD_SYSTEM_UTMGRID,
                          COORD_DEFAULT_ZONE, 0, argv[2],
                          COORD_SYSTEM_SIMNET,
                          &targ_x, &targ_y);
if (ret)
{
    printf("Error converting UTM %s: %s\n",
           argv[2], convert_error(ret));
    exit(0);
}

/* Find the elevation at each point.
 */
eye_z = ctdb_lookup_elevation(&ctdb, eye_x, eye_y);
targ_z = ctdb_lookup_elevation(&ctdb, targ_x, targ_y);

/* Compute visibility using typical values.
 */
vis = ctdb_point_to_point(&ctdb,
                          eye_x, eye_y, eye_z,
                          targ_x, targ_y, targ_z,
                          targ_z+TARGET_HEIGHT,
                          TARGET_WIDTH,
                          TREE_OPACITY,
                          0.0, /* Compute even if
                                * barely visible.
                                */
                          0 /* No vehicles in the
                             * way.
                             */
                          );

/* Print the result.
 */
printf("%5.2f%%\n", 100.0 * vis);
```

```
    }
```

3 Functions

The following sections describe each function provided by libctdb, including the format and meaning of its arguments, and the meaning of its return values (if any). Descriptions of how each function performs its task can be found in the algorithms section. See Chapter 6 [Algorithms], page 57.

3.1 ctdb_apparent_shape

```
     void ctdb_apparent_shape(x0, y0, z0, x1, y1, z1,
                              rotation, length, width, height,
                              app_loc, zh, x_min, x_max, z_min, z_max,
                              proj, dist, object_correction,
                              raster)
         float64      x0, y0, z0;
         float64      x1, y1, z1;
         float64      rotation[3][3];
         float64      length, width, height;
         float64      app_loc[3];
         float64     *zh;
         float64     *x_min;
         float64     *x_max;
         float64     *z_min;
         float64     *z_max;
         float64      proj[3][3];
         float64     *dist;
         float64     *object_correction;
         uint32       raster[32];
```

'x0, y0, z0'
Specifies the location of the viewer

'x1, y1, z1'
Specifies the location of the target

'rotation'
Specifies the 3x3 rotation matrix of the target

'length' Specifies the length of the target (its Y dimension)

'width' Specifies the width of the target (its X dimension)

'height' Specifies the height of the target (its Z dimension)

'app_loc' Returns the apparent location of the target

'zh' Returns the z-value at the top in the target

'x_min, x_max, z_min, z_max'
Returns the extents of the target (in viewer coordinates)

'proj' Returns an orientation matrix for the target's projection coordinate system. In this coordinate system, Y runs down the line of sight, Z runns perpendicular to Y and the world XY plane, and X runs perpendicular to Y and Z.

'dist' Returns the distance to the origin of the target's coordinate system

'object_correction'
Returns the ratio: distance to the nearest point on the target divided by the distance to the target

'raster' Returns a rasterized outline of the target

ctdb_apparent_shape is similar to ctdb_apparent_size, except that, in addition to returning the "size" of the distant object, it returns the raster giving a bitmap of the outline of the vehicle as seen from the eye point. In addition, it returns the apparent leftmost, rightmost, top and bottom of the object in "projection" coordinates. This is the frame of reference for the raster.

The value of object_correction returned is used by the caller to pass to subsequent calls to ctdb_ptop_raster. This value is a ratio, the distance from the viewpoint to the nearest vertex of the target divided by the distance to the target.

The value of zh returned is the z-coordinate of the highest vertex of the target as as seen from the viewer's perspective, expressed in *world coordinates.*

See Section 3.2 [ctdb'apparent'size], page 16.

3.2 ctdb_apparent_size

```
void ctdb_apparent_size(x0, y0, z0, x1, y1, z1,
                        rotation, length, width, height,
                        app_loc, app_width, app_height)
    float64     x0, y0, z0;
    float64     x1, y1, z1;
    float64     rotation[3][3];
    float64     length, width, height;
    float64     app_loc[3];
    float64    *app_width;
    float64    *app_height;
```

'x0, y0, z0'

Specifies the location of the viewer

'x1, y1, z1'
Specifies the location of the target

'rotation'
Specifies the 3x3 rotation matrix of the target

'length' Specifies the length of the target (its Y dimension)

'width' Specifies the width of the target (its X dimension)

'height' Specifies the height of the target (its Z dimension)

'app_loc' Returns the apparent location of the target

'app_width, app_height'
Returns the apparent dimensions of the target

ctdb_apparent_size finds the apparent size of the perpendicular parallelepiped described by length, width, height, given the eye point x0,y0,z0, the target location x1,y1,z1 and target rotation. The apparent width and height of the object are returned, as well as a location corrected to be at the bottom center of the apparent location.

This function is provided to allow extremely accurate intervisibility tests. Ideally this should be run for the target, as well as for each potentially blocking vehicle. For applications needing a more coarse measure of visibility, it is probably not necessary to use this function.

See Section 3.1 [ctdb'apparent'shape], page 15.

3.3 ctdb_contour_image

```
void ctdb_contour_image(ctdb,
                        low_x, low_y, width, height,
                        scale_mpp,
                        dirt_pixel, contour_pixel,
                        img)
    CTDB     *ctdb;
    int32     low_x;
    int32     low_y;
    int32     width;
    int32     height;
    float64   scale_mpp;
    int32     dirt_pixel;
    int32     contour_pixel;
    char      img[];
```

'ctdb' Pointer to initialized CTDB structure.

'low_x, low_y, width, height'
Specifies screen extents, in meters.

'scale_mpp'
Specifies screen scale in meters per pixel.

'dirt_pixel, contour_pixel'
Specifies pixel values to use for background and contours, respectively.

'img' Returns Z-pixmap format image of terrain contours.

ctdb_contour_image generates a Z-pixmap format image of the contours of the terrain bound by low_x,low_y to low_x+width,low_y+height, using dirt_pixel as a background, and contour_pixel as the foreground. The size of the image is computed from scale_mpp (meters per pixel). This routine may be faster than ctdb_contour_segments for extremely large terrain areas, although its use at application side is much more complicated.

See Section 3.5 [ctdb˙contour˙segments], page 19.

3.4 ctdb_contour_route

```
void ctdb_contour_route(ctdb, x0, y0, x1, y1,
                        max_deviation, direction,
                        sin_significant_slope, n_route, route)
    CTDB        *ctdb;
    float64      x0, y0;
    float64      x1, y1;
    float64      max_deviation;
    int32        direction;
    float64      sin_significant_slope;
    int32       *n_route;
    ROUTE_POINT route[];
```

'ctdb' Pointer to initialized CTDB structure.

'x0, y0' Specifies start of route.

'x1, y1' Specifies end of route.

'max_deviation'
Specifies the maximum deviation from the original route

'direction'
Specifies whether valleys or crests are desired

'sin_significant_slope'

Specifies the sin of the minimum slope angle which should be considered significant

'n_route' Specifies size of route array. Returns number of points in route.

'route' Returns the contoured route.

ctdb_contour_route generates a route between points x0,y0 and x1,y1 which follows the contour of the terrain. The route either attempts to stay high (direction==CTDB_CREST) or low (direction==CTDB_VALLEY), but will not deviate more than max_deviation from the original route. The route is returned in route[] and the length of the route is returned in n_route. Upon invokation, n_route should contain the maximum number of points which will fit in route. When the routine returns, route[0] will be at x0,y0, and route[(*n_route)-1] will be at x1,y1.

See Section 3.23 [ctdb˙profile˙vector], page 32.

3.5 ctdb_contour_segments

```
void ctdb_contour_segments(ctdb,
                           interval,
                           x_low, y_low, x_high, y_high,
                           draw_fcn)
    CTDB   *ctdb;
    float64 interval;
    int32   x_low, y_low, x_high, y_high;
    void    (*draw_fcn)(/* int32   seg_cnt;
                           float64 segs[CTDB_MAX_CONTOUR_PTS];
                           float64 elevs[CTDB_MAX_CONTOUR_PTS/4]
                         */);
```

'ctdb' Pointer to initialized CTDB structure.

'interval'

Specifies distance between contour lines, in meters.

'x_low, y_low, x_high, y_high'

Specifies screen extents, in meters.

'draw_fcn'

Called periodically to draw a group of line segments.

ctdb_contour_segments calls the passed draw_fcn for each contour line segment in the area bound by x_low,y_low to x_high,y_high at the specified interval. seg_cnt is the number of elements in the segs[] array (x0, y0, x1, y1, in meters). seg_cnt/4 is the the number of elements in the elevs[] array (one elevation for each line segment in the segs[] array).

See Section 3.3 [ctdb˙contour˙image], page 17.

3.6 ctdb_find_ground_intersection

```
     int32 ctdb_find_ground_intersection(ctdb, x0, y0, z0, x1, y1, z1,
                                        pt_ret, tests,
                                        n_veh,
                                        veh, ignore0, ignore1, hit_veh)
        CTDB                    *ctdb;
        float64                  x0, y0, z0;
        float64                  x1, y1, z1;
        float64                  pt_ret[3];
        uint32                   tests;
        int32                    n_veh;
        CTDB_VEHICLE_LOCATION    veh[];
        int32                    ignore0, ignore1;
        int32                   *hit_veh;
```

'ctdb' Pointer to initialized CTDB structure.

'x0, y0, z0'
Specifies the start of a 3D line

'x1, y1, z1'
Specifies the end of a 3D line

'pt_ret' Returns the first intersection of 3D line with ground polygon

'tests' Specifies the intersection tests to be performed

'n_veh' Specifies the number of vehicles in the veh[] array.

'veh' Specifies locations of other vehicles which may block visibility.

'ignore0, ignore1'
Specifies 0, 1, or 2 vehicles in the veh[] array which should not be checked.

'hit_veh' Returns the first vehicle which blocked the ray.

ctdb_find_ground_intersection locates the point at which the ray from <x0 y0 z0> to <x1 y1 z1> crosses through the terrain grid, a microterrain polygon, a building, or a tree line (including those which surround canopies). The point of intersection (+/- 1 meter along the ray when testing ground polygons) is returned in pt_ret. If no such intersection can be found, the routine returns 0, otherwise one of the following constants:

`CTDB_HIT_GROUND`
Indicates a ground polygon was intersected.

`CTDB_HIT_BUILDING`
Indicates the ray passed under the roofline of a building.

`CTDB_HIT_TREELINE`
Indicates the ray passed through a treeline, or under the edge of a tree canopy.

`CTDB_HIT_VEHICLE`
Indicates the ray crossed a vehicle in the `veh` array.

`CTDB_HIT_WATER`
Indicates the ray crossed a linear feature which isn't a road.

The endpoints are clipped to database boundarie If the return value is `CTDB_HIT_VEHICLE`, the index of that vehicle in the veh array is returned in `*hit_veh`. Note that individual trees are not modeled accurately enough for tests of them to be meaningful.

The `tests` parameter should be an inclusive OR of the same constants which are returned (`CTDB_HIT_GROUND`, etc.). Depending upon which of these bits are set, the various types of features will be tested for intersection with the ray.

3.7 ctdb_get_treeline_segment

```
void ctdb_get_treeline_segment(ctdb, pt0, pt1)
    CTDB     *ctdb;
    float64  pt0[2];
    float64  pt1[2];
```

'`ctdb`' Pointer to initialized CTDB structure.

'`pt0, pt1`' Return the endpoints of the intersected treeline segment.

ctdb_get_treeline_segment() must be called *after* a call to ctdb_find_ground_intersection(). If a treeline was intersected in the call to ctdb_find_ground_intersection(), then the endpoints of that segment of the treeline (in meters from the lower left coner of the database) are returned in pt0[] and pt1[]. If there was no treeline intersection in the last call to ctdb_find_ground_intersection(), zeros are returned in pt0[] and pt1[].

3.8 ctdb_find_high_ground

```
float64 ctdb_find_high_ground(ctdb, x0, y0, x1, y1, xy_at_high)
    CTDB     *ctdb;
    float64  x0, y0;
    float64  x1, y1;
    float64 *xy_at_high;
```

'ctdb' Pointer to initialized CTDB structure.

'x0, y0' Specifies the start of the search line

'x1, y1' Specifies the end of the search line

'xy_at_high'
Returns the X and Y coordinates of the high point (applications may pass NULL, if they are not interested in this value).

ctdb_find_high_ground returns the elevation of the highest point between x0,y0 and x1,y1. If non-NULL, xy_at_high will be filled in with the x and y values of this point. Hence, to get the 3D point, usage would be: vec[Z] = ctdb_find_high_ground(ctdb,x0,y0,x1,y1,vec); The ray from x0,y0 to x1,y1 is clipped to terrain boundaries, if the ray does not cross the terrain, the returned elevation will be 0.0, and the returned point will by x0,y0.

3.9 ctdb_hypso_bitmap

```
void ctdb_hypso_bitmap(ctdb, low_x, low_y, width, height,
                       scale_mpp, min_z, max_z, bitmap)
    CTDB    *ctdb;
    int32    low_x;
    int32    low_y;
    int32    width;
    int32    height;
    float64  scale_mpp;
    float64  min_z;
    float64  max_z;
    char     bitmap[];
```

'ctdb' Pointer to initialized CTDB structure.

'low_x, low_y, width, height'
Specifies screen extents, in meters.

'scale_mpp'
Specifies screen scale in meters per pixel.

'min_z, max_z'

Specifies the minimum and maximum altitudes which should be represented (ctdb->min_z and ctdb->max_z are good candidates).

'bitmap' Returns an XY-bitmap format image of the hypsometric map.

ctdb_hypso_bitmap generates an XY-bitmap format hypsometric image of the terrain bound by low_x,low_y to low_x+width,low_y+height, using dithering. Points off the terrain database are set to 0. The size of the image is computed from scale_mpp (meters per pixel). min_z and max_z are passed in by the caller. points at or below min_z use an empty dither, and points at or above max_z use a solid dither. Points in between use dither patterns with density proportional to elevation. The returned bitmap is suitable for a call to **XPutImage()** or **XCreatePixmapFromBitmapData()**.

NOTE: This function is conditionally compiled only if the the CFLAGS has -DHYPSO. This avoids a dependency on libdither if libctdb is not being used for hypsometric map drawing.

3.10 ctdb_hypso_image

```
void ctdb_hypso_image(ctdb, low_x, low_y, width, height,
                      scale_mpp,
                      ncells, cmap, off_terrain_pixel,
                      img, min_z, max_z)
    CTDB      *ctdb;
    int32      low_x;
    int32      low_y;
    int32      width;
    int32      height;
    float64    scale_mpp;
    int32      ncells;
    int32      cmap[];
    int32      off_terrain_pixel
    char       img[];
    float64   *min_z;
    float64   *max_z;
```

'ctdb' Pointer to initialized CTDB structure.

'low_x, low_y, width, height'
Specifies screen extents, in meters.

'scale_mpp'
Specifies screen scale in meters per pixel.

'ncells' Specifies the number of cells available in the color map.

'cmap' Specifies pixel values for elevation ranges.

'off_terrain_pixel'
Specifies pixel value for points off the terrain database.

'img' Returns a Z-pixmap format image of the hypsometric map.

'min_z, max_z'
Returns the minimum and maximum altitudes in the area (useful for assigning colors to the colormap).

ctdb_hypso_image generates a Z-pixmap format hypsometric image of the terrain bound by low_x,low_y to low_x+width,low_y+height, using the pixel values specified in the cmap[0..ncells-1] array. Points off the terrain database are set to the off_terrain_pixel value. The size of the image is computed from scale_mpp (meters per pixel). *min_z and *max_z return the elevations corresponding to cmap[0] and cmap[ncells-1], respectively. It is expected that the caller can use this information to change the color map appropriately.

NOTE: This function is conditionally compiled only if the the CFLAGS has -DHYPSO. This avoids a dependency on libdither if libctdb is not being used for hypsometric map drawing.

3.11 ctdb_lookup_elevation

```
float64 ctdb_lookup_elevation(ctdb, x, y)
    CTDB   *ctdb;
    float64 x;
    float64 y;
```

'ctdb' Pointer to initialized CTDB structure.

'x, y' Specifies a point on the database.

ctdb_lookup_elevation finds the elevation at the specified point on the terrain database based upon the elevation grid and microterrain. x and y are in meters. The point is tested to make sure it is not off the terrain database (it is not necessary for the application to make this check), and 0.0 is returned in that case.

See Section 3.13 [ctdb'lookup'max'elevation], page 25.

3.12 ctdb_lookup_feature_info

```
int32 ctdb_lookup_feature_info(ctdb, x, y,
                               soil_ret, width_ret, direction_ret)
    CTDB    *ctdb;
    float64  x;
    float64  y;
    int32   *soil_ret;
    float64 *width_ret;
    float64  direction_ret[2];
```

'ctdb' Pointer to initialized CTDB structure.

'x, y' Specifies a point on the database.

'soil_ret'
Returns the soil type of the feature

'width_ret'
Returns the width of the feature

'direction_ret'
Returns the direction of the feature

ctdb_lookup_feature_info takes a point and looks to see if there is a linear feature under it. If so it returns 1 and fills in the the type, width, and direction variables passed to it. Otherwise it returns 0 and type, width, and direction are undefined.

3.13 ctdb_lookup_max_elevation

```
float64 ctdb_lookup_max_elevation(ctdb, x, y, check_canopies)
    CTDB    *ctdb;
    float64  x;
    float64  y;
    int32    check_canopies;
```

'ctdb' Pointer to initialized CTDB structure.

'x, y' Specifies a point on the database.

'check_canopies'
Indicates whether the altitude returned should clear tree canopies.

ctdb_lookup_max_elevation finds the elevation at the specified point on the terrain database based upon the elevation grid, microterrain, buildings, and if check_canopies is 'TRUE', tree canopies. x and y are in meters. The point is tested to make sure it is not off the terrain database (it is not necessary for the application to make this check), and 0.0 is returned in that case.

3.14 ctdb_lookup_soil

```
int32 ctdb_lookup_soil(ctdb, x, y)
    CTDB   *ctdb;
    float64 x;
    float64 y;
```

'ctdb' Pointer to initialized CTDB structure.

'x, y' Specifies a point on the database.

ctdb_lookup_soil finds the soil type at the specified point on the terrain database based upon the elevation grid and microterrain. x and y are in meters. The point is tested to make sure it is not off the terrain database (it is not necessary for the application to make this check), and 0 is returned in that case.

3.15 ctdb_place_vehicle

```
void ctdb_place_vehicle(ctdb, x, y, length, width, dx, dy,
                              z, rotation, soil)
    CTDB     *ctdb;
    float64   x;
    float64   y;
    float64   length;
    float64   width;
    float64   dx;
    float64   dy;
    float64  *z;
    float64   rotation[3][3];
    int32    *soil;
```

'ctdb' Pointer to initialized CTDB structure.

'x, y' Specifies a point on the database.

'length, width'
Specifies the length (Y dimension) and width (X dimension) of the vehicle.

'dx, dy' Specifies the direction of the vehicle as a cosine/sine pair. north = <0 1>, south = <0 -1>, east = <1 0>, west = <-1 0>

'z' Returns the elevation at the point.

'rotation'
Returns a 3x3 rotation matrix for the vehicle.

'soil' Returns the soil type at the point.

ctdb_place_vehicle finds the elevation, hull-to-world rotation matrix, and soil type for a vehicle siting at the specified point, facing the specified direction. x, y, length, and width are in meters. <dx dy> is assumed to be a unit vector pointing in the direction of the vehicle. The elevation is returned in *z, the rotation matrix is filled in, and the soil type is returned in *soil. The point is tested to make sure it is not off the terrain database (it is not necessary for the application to make this check); points off the database have an elevation of 0.0, a flat rotation matrix pointing in the specified direction, and a soil type of 0.

3.16 ctdb_point_on_database

```
int32 ctdb_point_on_database(ctdb, x, y)
    CTDB   *ctdb;
    float64 x;
    float64 y;
```

'ctdb' Pointer to initialized CTDB structure.

'x, y' Specifies the point to check.

ctdb_point_on_database returns 1 if the point is on the database, 0 if it is not. All libctdb functions make this check internally.

3.17 ctdb_point_on_ground

```
int32 ctdb_point_on_ground(ctdb, x, y, radius, soil_ret)
    CTDB   *ctdb;
    float64 x;
    float64 y;
    float64 radius;
    int32  *soil_ret;
```

'ctdb' Pointer to initialized CTDB structure.

'x, y' Specifies the center of the circle.

'radius' Specifies the radius of the circle.

'soil_ret'

Returns the soil type at the center of the circle.

ctdb_point_on_ground returns 1 if the circle centered at the passed location, with the specified radius is completely ground (no trees or buildings cross the circle). The soil type at the passed point is also returned so the caller can check for undesired soil types. This function is used for initializing vehicles in reasonable locations.

See Section 3.15 [ctdb˙place˙vehicle], page 26.

3.18 ctdb_get_buildings

```
int32 ctdb_get_buildings(ctdb, x, y, radius, max_count, buildings)
    CTDB                 *ctdb;
    float64               x;
    float64               y;
    float64               radius;
    int32                 max_count;
    CTDB_FEATURE_OUTLINE buildings[];
```

'**ctdb**' Pointer to initialized CTDB structure.

'**x, y**' Specifies the center of the circle.

'**radius**' Specifies the radius of the circle.

'**max_count**'
Specifies the maximum number of buildings to count

'**buildings**'
If non-NULL, returns list of building outlines

ctdb_get_buildings returns the number of buildings within the specified radius of the passed point. If *any corner* of a building is within the radius, the building will be included (note that if the circle falls completely within a building, that building will not be detected). If non-**NULL**, **buildings** will be filled with the list of buildings (where **max_count** specifies the number which will fit in the passed array). The **max_count** parameter can also be used when **buildings** is **NULL** to indicate the maximum number to count. For example, if the question being asked is "are there more than 10 buildings in this area?", then the number 11 may be passed in, so that the search can be stopped as soon as possible.

The buildings are returned using the following structure:

```
typedef struct ctdb_feature_outline
{
    int32   n_verts;
```

```
        float64 verts[10][3];
    } CTDB_FEATURE_OUTLINE;
```

n_verts returns the number of vertices in the outline of the building, and verts returns the 3D coordinate of each vertex (following the roofline of the building).

3.19 ctdb_point_thru_point

```
    float64 ctdb_point_thru_point(ctdb, x0, y0, z0, x1, y1, zl, zh,
                                  width, tree_opacity,
                                  minimum_visibility, range,
                                  n_veh,
                                  veh, ignore0, ignore1)
        CTDB                     *ctdb;
        float64                   x0, y0, z0;
        float64                   x1, y1, zl, zh;
        float64                   width;
        float64                   tree_opacity;
        float64                   minimum_visibility;
        float64                   range;
        int32                     n_veh;
        CTDB_VEHICLE_LOCATION     veh[];
        int32                     ignore0, ignore1;
```

'ctdb' Pointer to initialized CTDB structure.

'x0, y0, z0'
Specifies the eye point.

'x1, y1, zl, zh'
Specifies the target location, and the bottom and top of the target.

'width' Specifies target width.

'tree_opacity'
Specifies reduction of visibility resulting from trees.

'minimum_visibility'
Specifies a threshold below which 0 may be returned.

'range' Specifies the range of the radar sensitive to clutter.

'n_veh' Specifies the number of vehicles in the veh[] array.

'veh' Specifies locations of other vehicles which may block visibility.

'ignore0, ignore1'
Specifies 0, 1, or 2 vehicles in the veh[] array which should not be checked.

ctdb_point_thru_point is a special version of intervisibility for use with radar. Within the passed range, objects behind the target block visibility to the same extent as objects in front of it.

See Section 3.20 [ctdb˙point˙to˙point], page 30.
See Section 3.24 [ctdb˙ptop˙raster], page 32.
See Section 3.27 [ctdb˙vehicle˙blockage], page 35.
See Section 3.2 [ctdb˙apparent˙size], page 16.

3.20 ctdb_point_to_point

```
float64 ctdb_point_to_point(ctdb, x0, y0, z0, x1, y1, zl, zh,
                            width, tree_opacity,
                            minimum_visibility, n_veh,
                            veh, ignore0, ignore1)
    CTDB                    *ctdb;
    float64                  x0, y0, z0;
    float64                  x1, y1, zl, zh;
    float64                  width;
    float64                  tree_opacity;
    float64                  minimum_visibility;
    int32                    n_veh;
    CTDB_VEHICLE_LOCATION    veh[];
    int32                    ignore0, ignore1;
```

'ctdb' Pointer to initialized CTDB structure.

'x0, y0, z0'
Specifies the eye point.

'x1, y1, zl, zh'
Specifies the target location, and the bottom and top of the target.

'width' Specifies target width.

'tree_opacity'
Specifies reduction of visibility resulting from trees.

'minimum_visibility'
Specifies a threshold below which 0 may be returned.

'n_veh' Specifies the number of vehicles in the veh[] array.

'veh' Specifies locations of other vehicles which may block visibility.

'ignore0, ignore1'
Specifies 0, 1, or 2 vehicles in the veh[] array which should not be checked.

ctdb_point_to_point performs an intervisibility check starting at the point x0,y0 and proceeding to the point x1,y1. z0 is the eye point of the viewer and zl and zh are the bottom and top of the target. width is the width of the target, and is used when comparing against individual trees and buildings, otherwise a zero-width target is assumed. All these values are in meters. tree_opacity encapsulates the effect trees have on the visual system being modeled. An opacity of 0 indicates trees have no effect; 1 indicates trees completely block visibility. The effect of multiple trees is combined using a simple light transmittance model. The visible target area (adjusted for tree opacity) is returned as a floating point number in the range 0.0 to 1.0 (0.0 for complete blockage, 1.0 for complete visibility). Since visibility can only get smaller as more features are tested, knowing the minimum visibility interesting to the application can greatly enhance the speed of calculation. Even very small values (such as 0.05) can greatly increase speed. If the visibility measure drops below this visibility, 0.0 will be returned. The ray from x0,y0 to x1,y1 is clipped to terrain boundaries, visibility is 1.0 in areas off the terrain database.

The last set of parameters are only examined if n_veh is non-zero; hence they need not be passed unless vehicle blocking vehicle intervisibility calculations are required. If n_veh is non-zero, ctdb_point_to_point first calls ctdb_vehicle_blockage with the arguments specified (see Section 3.27 [ctdb'vehicle'blockage], page 35, for an explanation of these arguments). Note that since a target could be partially blocked by terrain and partially blocked by vehicles, two separate calls (one to ctdb_point_to_point with n_veh=0, and one to ctdb_vehicle_blockage) will not necessarily yield the same result as one call to ctdb_point_to_point with n_veh>0.

See Section 3.19 [ctdb'point'thru'point], page 29.
See Section 3.27 [ctdb'vehicle'blockage], page 35.
See Section 3.24 [ctdb'ptop'raster], page 32.
See Section 3.2 [ctdb'apparent'size], page 16.

3.21 ctdb_print_description

```
void ctdb_print_description(ctdb)
    CTDB *ctdb;
```

'ctdb' Pointer to initialized CTDB structure.

ctdb_print_description prints a description of the passed ctdb database including min/max information and memory usage statistics.

3.22 ctdb_print_stats

```
void ctdb_print_stats(ctdb)
    CTDB *ctdb;
```

'ctdb' Pointer to initialized CTDB structure.

ctdb_print_stats prints cache performance statistics.

3.23 ctdb_profile_vector

```
int32 ctdb_profile_vector(ctdb, x0, y0, x1, y1, n_prof, prof)
    CTDB    *ctdb;
    float64  x0, y0;
    float64  x1, y1;
    int32   *n_prof;
    float64  prof[][3];
```

'ctdb' Pointer to initialized CTDB structure.

'x0, y0' Specifies start of vector.

'x1, y1' Specifies end of vector.

'n_prof' Specifies size of prof array. Returns number of point in profile.

'prof' Returns exact 3D profile of vector, conforming to each grid or microterrain edge.

ctdb_profile_vector generates a sequence of 3D points along the specified ray from x0,y0 to x1,y1 which exactly follow the contour of the terrain. The points are returned in prof[] and the number of points is returned in n_prof. Upon invocation, n_prof should contain the maximum number of points which will fit in prof[]. If more profile points are necessary than will fit in prof[], the function returns 0, otherwise it returns 1. ctdb_profile_vector clips the specified ray to terrain database boundaries, hence the points x0,y0 and x1,y1 will not necessarily be the first and last points in the prof[] array (although they usually will be).

3.24 ctdb_ptop_raster

```
float64 ctdb_ptop_raster(ctdb, x0, y0, z0, x1, y1, zl, zh,
                         width, ftype, object_correction, tree_opacity,
                         minimum_visibility, grid,
                               x_min, x_max, z_min, z_max, actual_visible_area,
                               n_veh, veh, ignore0, ignore1)
       CTDB                      *ctdb;
       float64                    x0, y0, z0;
       float64                    x1, y1, zl, zh;
       float64                    width;
       int32                      ftype;
       float64                    object_correction;
       float64                    tree_opacity;
       float64                    minimum_visibility;
       CTDB_RASTER                grid;
       float64                   *x_min;
       float64                   *x_max;
       float64                   *z_min;
       float64                   *z_max;
       float64                   *actual_visible_area;
       int32                      n_veh;
       CTDB_VEHICLE_LOCATION      veh[];
       int32                      ignore0, ignore1;
```

'ctdb' Pointer to initialized CTDB structure.

'x0, y0, z0'
Specifies the eye point.

'x1, y1, zl, zh'
Specifies the target location, and the bottom and top of the target.

'width' Specifies target width.

'ftype' If viewing a feature, specifies the type of feature being viewed.

'object_correction'
Specifies target scale factor (see below).

'tree_opacity'
Specifies reduction of visibility resulting from trees.

'minimum_visibility'
Specifies a threshold below which 0 may be returned.

'grid' Specifies the exact profile of the target.

'x_min, x_max, z_min, z_max'
Specifies the size of the target in various dimensions.

'actual_visible_area'
Returns actual visible area.

'n_veh' Specifies the number of vehicles in the veh[] array.

'veh' Specifies locations of other vehicles which may block visibility.

'ignore0, ignore1'
Specifies 0, 1, or 2 vehicles in the veh[] array which should not be checked.

ctdb_ptop_raster is the same as ctdb_point_to_point except that it takes an additional argument, CTDB_RASTER grid, a pre-computed raster supplied by the client. It also requires the frame of reference (x_min, x_max, z_min, z_max) in "projection" coordinates of the raster. These are adjusted according to the result of the intervis computation and passed back to the caller to reticulate the object.

Object correction must be supplied by the caller and is computed by a call to **ctdb_apparent_shape**. This value is used as a scale factor to "pull back" the shape of the target along parametric lines to the eye point to enable intervis calculations in two steps for terrain features as targets.

See Section 3.20 [ctdbˆpointˆtoˆpoint], page 30.
See Section 3.19 [ctdbˆpointˆthruˆpoint], page 29.
See Section 3.27 [ctdbˆvehicleˆblockage], page 35.
See Section 3.2 [ctdbˆapparentˆsize], page 16.
See Section 3.1 [ctdbˆapparentˆshape], page 15.

3.25 ctdb_read

```
void ctdb_read(fname, ctdb, memory_limit)
    char   *fname;
    CTDB   *ctdb;
    int32   memory_limit;
```

'fname' File name of database. Can be any valid unix path in the style of open(2).

'ctdb' Pointer to uninitialized 'CTDB' type variable.

'memory_limit'
Maximum amount of memory which should be used, in bytes.

ctdb_read loads the ctdb format terrain database from the named file, and puts the resulting information into the user-supplied CTDB structure. This structure will be needed for calls to almost all libctdb functions.

If the database needs less than the allowed amount of memory, only the amount needed will be allocated. If the database is larger than this amount, a cache will be used, sized to fit within this limit. Note that the special value '0' may be used to indicate an unlimited amount of memory (usually a bad idea, since the libctdb cache will typically perform much better than virtual memory would).

It is assumed that loading databases occurs during the initialization phase of a program, and hence if an error is detected, libctdb will invoke `exit(1)`.

3.26 ctdb_reread

```
void ctdb_reread(fname, ctdb, memory_limit)
    char   *fname;
    CTDB   *ctdb;
    int32   memory_limit;
```

'`fname`' File name of database. Can be any valid unix path in the style of `open(2)`.

'`ctdb`' Pointer to initialized '`CTDB`' type variable.

'`memory_limit`'
Maximum amount of memory which should be used, in bytes.

ctdb_read loads the ctdb format terrain database from the named file, and puts the resulting information into the user-supplied CTDB structure, as in **ctdb_read**. However, it is assumed that the passed CTDB structure already contains a database, and that the memory used by the original database must be either reused or freed.

See Section 3.25 [ctdb˙read], page 34.

3.27 ctdb_vehicle_blockage

```
float64 ctdb_vehicle_blockage(ctdb, x0, y0, z0, x1, y1, zl, zh,
                              width, n_veh, veh, ignore0, ignore1)
    CTDB                     *ctdb;
    float64                   x0, y0, z0;
    float64                   x1, y1, zl, zh;
    float64                   width;
    int32                     n_veh;
    CTDB_VEHICLE_LOCATION     veh[];
    int32                     ignore0, ignore1;
```

'`ctdb`' Pointer to initialized CTDB structure.

'`x0, y0, z0`'
Specifies the eye point.

'x1, y1, zl, zh'
Specifies the target location, and the bottom and top of the target.

'width' Specifies target width.

'n_veh' Specifies the number of vehicles in thc vch[] array.

'veh' Specifies locations of other vehicles which may block visibility.

'ignore0, ignore1'
Specifies 0, 1, or 2 vehicles in the veh[] array which should not be checked.

ctdb_vehicle_blockage performs a visibility check from the eye point to the target point, comparing against the vehicles passed in the veh[] array. The locations and width are as in ctdb_point_to_point. n_veh is the size of the veh[] array. Vehicles in the veh[] array are only checked if the 'used' field is set. The height and width are assumed to be projected onto the viewing plane of the observer. Two vehicles can be eliminated from the search by setting ignore0 and ignore1 to their indices in the veh[] array. To avoid eliminating vehicles, pass CTDB_DONT_IGNORE for ignore0 and ignore1. As in ctdb_point_to_point, the return value is in the range 0.0 for full blockage, to 1.0 for full visibility.

See Section 3.2 [ctdb'apparent'size], page 16.
See Section 3.20 [ctdb'point'to'point], page 30.

4 Porting Guide

The libctdb library has been compiled on the following platforms:

- Mips
- Apollo
- Hewlett-Packard

The code is written to be extremely portable, but on other platforms some modification may be necessary. The most likely problem is the lack of the single precision square root function `fsqrt()`. The first definitions in the header file '`libctdb.h`' allow correction of this problem:

```
#ifdef apollo
#define fsqrt sqrt
#endif
```

This can be extended to correct similar problems on other machines.

Another potential problem is the dependency on the common library '`libdither`'. This library is only used if the hypsometric map generating functions `ctdb_hypso_image` or `ctdb_hypso_bitmap` will be used on the target platform. If these functions will be used, the preprocessor symbol `HYPSO` must be defined. This can be achieved in the '`Makefile`' with the line:

```
EXTRA_CFLAGS = -DHYPSO
```

Omitting the `HYPSO` definition will cause the hypsometric mapping functions to be omitted from the '`libctdb.a`' archive, and hence the common include file '`libdither.h`' will not be needed.

4.1 Platform specific optimization

The libctdb source code has been written in a manner which works best when compiled with an optimizing compiler. For almost all compilers, optimization is enabled with the compiler flag '`-O`'. This should be one of the arguments passed to '`cc`' when this library is compiled.

Other heuristics have been applied as well (see Section 1.4 [Optimized for RISC], page 7). In general, these conventions will not have a significant impact on non-RISC architectures. If porting to a new platform on which performance is a primary concern, the use of a profiler is highly encouraged. '`test.c`' in the libctdb library is good test program for use with a profiler. The mechanics of profiling differ from platform to platform, but the following rules should generally apply:

- Always profile an optimized executable. The execution profile of non-optimized code is generally nothing at all like the profile of optimized code.
- Do not change the calling sequences of functions to be different for a particular platform. This inhibits portability and reuse of code.
- Never make a change to the code without testing to see if it actually improved performance. The '`test`' program can be used to confirm average execution times for most functions.
- If a change is specific to a particular platform, it should be coded using '`#ifdef`' syntax. For example, if a target platform performs significantly better using `unsigned short` rather than `int` for iteration variables, an appropriate coding would be:

  ```
  /* Optimization */
  #ifdef my_machine
  #define ITERATOR_TYPE uint32 short
  #else
  #define ITERATOR_TYPE int
  #endif
  ...
  {
      ITERATOR_TYPE i;
      ...
  }
  ```

  This allows future porters to easily decide whether this optimization improves performance on their target platform.

ADST-TR-W003268

LibDelObj

Table of Contents

1 Overview

LibDelObj provides a simple object deletion editor for the GUI. The user can select objects to delete (which are marked with big red X's), then can delete them by clicking "Done". The library takes care of some of the complexities of deleting object, such as removing text associated with deleted objects.

2 Usage

The software library 'libdelobj.a' should be built and installed in the directory '/common/lib/'. You will also need the header file 'libdelobj.h' which should be installed in the directory '/common/include/libinc/'. If these files are not installed, you need to do a '**make**' in the libdelobj source directory. If these files are already built, you can skip the section on building libdelobj.

2.1 Building Libdelobj

The libdelobj source files are found in the directory '/common/libsrc/libdelobj'. 'RCS' format versions of the files can be found in '/nfs/common_src/libsrc/libdelobj'.

If the directory 'common/libsrc/libdelobj' does not exist on your machine, you should use the '**genbuild**' command to update the common directory hierarchy.

To build and install the library, do the following:

```
# cd common/libsrc/libdelobj
# co RCS/*,v
# make install
```

This should compile the library 'libdelobj.a' and install it and the header file 'libdelobj.h' in the standard directories. If any errors occur during compilation, you may need to adjust the source code or 'Makefile' for the platform on which you are compiling. libdelobj should compile without errors on the following platforms:

- Mips
- SGI Indigo
- Sun Sparc

2.2 Linking with Libdelobj

Libdelobj can be linked into an application program with the following link time flags: 'ld [source .o files] -L/common/lib -ldelobj [other libraries]'. If your compiler does not support '-L' syntax, you can use the archive explicitly: '`ld [source .o files] /common/lib/libdelobj.a`'.

Libdelobj depends directly on the following libraries: libsafgui, libtactmap, libcoordinates, libsensitive, libcallback, libpo, libeditor, and libreader.

3 Functions

The following sections describe each function provided by libdelobj, including the format and meaning of its arguments, and the meaning of its return values (if any).

3.1 delobj_init

```
void delobj_init()
```

delobj_init initializes libdelobj. Call the before any other libdelobj function.

3.2 delobj_init_gui

```
int32 delobj_init_gui(data_path, reader_flags,
                      gui, tactmap, tcc, map_erase_gc,
                      sensitive, refresh_event, db)
    char                *data_path;
    uint32               reader_flags;
    SGUI_PTR             gui;
    TACTMAP_PTR          tactmap;
    COORD_TCC_PTR        tcc;
    GC                   map_erase_gc;
    SNSTVE_WINDOW_PTR    sensitive;
    CALLBACK_EVENT_PTR   refresh_event;
    PO_DATABASE         *db;
```

'data_path'
: Specifies the directory where data files are expected

'reader_flags'
: Specifies flags to be passed to reader_read when reading data files

'gui'
: Specifies the SAF GUI

'tactmap'
: Specifies the tactical map

'tcc'
: Specifies the map coordinate system

'map_erase_gc'
: Specifies the GC which can erase things from the tactical map

'sensitive'
: Specifies the sensitive window for the tactical map

'**refresh_event**'
Specifies the event which fires when the map is refreshed

'**db**' Specifies the persistent object database

delobj_init_gui create the object deletion tool. The data file ('**delobj.rdr**') is read either from '.' or the specified data path, depending upon the **reader_flags**. The **reader_flags** are as in **reader_read**. The return value is zero if the read succeeds, or one of the libreader return values: **READER_READ_ERROR**, **READER_FILE_NOT_FOUND**.

ADST-TR-W003268

LibDetonation

Table of Contents

1 Overview

Libdetonation provides a model of proximity detonation. It can detect detonations due to proximity with other network entities (platforms, missiles, and structures). Proximity detonation due to the ground, buildings, and other terrain features is not yet supported. This library will generate impact PDUs, if told to do so for a given vehicle.

This library determines that a detonation should occur if the distance to the target, as measured along the secant between positions of the ticked vehicle (usually a missile) during consecutive ticks achieves a local minimum (and is less than the **detonation_radius** parameter specified for the model instance.) In performing this calculation, the position of the target is projected forward or backward in time ("Anti-RVA") to find the point on its trajectory closest to the point where the local minimum occurred. This estimated location is passed back via the registered **DET_CALLBACK** function. If the local minimum has been passed and the distance to the target is greater than **detonation_radius**, then a near-miss is declared.

There are two models used for selecting potential target entities: low-fidelity and high-fidelity. When low-fidelity detonation is used, a list of potential targets must be supplied by the simulation. These are the ONLY vehicles which will trigger detonation. When high-fidelity is used, libdetonate builds a suitable list of nearby vehicles to check. This is considerably more expensive.

In addition to proximity detonations which are within the detonation radius specified by the parametric data, libdetonation also informs clients of near-misses with vehicles on the list of detonation candidates.

The parameters used by a vehicle (missile) for detonation detection are specified in its configuration file as follows:

```
(SM_Detonation (check {trees} {buildings} {ground}
                      {platforms} {missiles})
               (detonation_radius <real meters>)
               (fidelity [high|low])
```

The first parameter, **check**, lists those things for which detonation detection is required. This affects performance when high-fidelity detonation is enabled.

The **detonation_radius** parameter specifies the maximum proximity which will trigger a detonation for this vehicle (missile.)

Finally, the **fidelity** parameter (which has a value of **high** or **low**) is used to determine the method for selecting potential targets. It also affects the expense of the algorithms used in determining if detonation should occur.

In the high fidelity model, entire classes of entities are checked for proximity, while in the low fidelity model, a list of vehicles (targets) must be built using the function **det_add_target()**.

det_tick ticks the detonation sub-class. During the tick, if a detonation is detected, a packet can be sent and the callback function (if any) registered with **det_class_init** will be invoked. The detonation tick input includes the vehicle id of the missile and a pointer to the terrain database on which the missile is simulated. The detonation tick processing of a missile includes the follow steps:

1. Set the libdetonation variable, old_pos, to the location the missile was at last tick (detonation->last_pos). This will be referred to as old position.
2. Get libentity information(ent_position, ent_velocity, ent_stationary) and store it as new_pos, velocity, and stationary. Set the detonation->last_pos field to new_pos (the newly retrieved ent_position).
3. Call the function **ent_get_physdb(vehicle_id)** to get the missile's physical data, which includes the missile dimensions. Find an upper bound size by using the maximum of length, width, and height. This upper bound is needed when calculating the range to pass to the position based vehicle table.
4. If the missile is stationary it is not necessary to check for detonations with terrain so omit DET_BUILDINGS, DET_GROUND, and DET_TREES from the missile's detect checklist. If the detect checklist indicates DET_PLATFORMS, DET_MISSILES, or DET_BUILDINGS add the relevant types (VTAB_VEHICLE, VTAB_MISSILE, VTAB_STRUCTURE) to the missile's list of types-to-check.
5. If the missile's detect checklist still contains some types and high fidelity processing is indicated for this missile, create a binary tree vtab structure named "to_check" to pass to libpdtab so a list of potential target vehicles within a specified range can be generated.
6. Call the function **pbt_get_vehicles(cx, cy, range, to_check)** to get the to_check list (a list of vehicles that are within range). The missile locus (cx, cy) was determined by interpolating a location midway between the old and new positions. The range was determined by the distance between the old and new positions, the upper bound size, and the >error threshold between actual locations and locations used by libpbtab (the position-based vehicle table).
7. Remove the vehicle id of this missile from the to_check list.
8. If the count of vehicles in the list is zero, then remove the types DET_PLATFORMS and DET_MISSILES from the types-to-check list. This means there is no need to check the vehicle table at all.

9. If there are no types left in the types-to-check list, exit.
10. If this missile is not stationary, then perform whichever type of fidelity processing is indicated for this missile (either high fidelity or low fidelity processing). The processing for these fidelity types differs in the choice of possible targets. Basically, the processing consists of checking whether the distance squared to each vehicle in the list of potential targets has achieved a local minimun and whether this local minimum has occurred at a distance which is less than the minimum detonation radius. Then of these distances, pick the minimum. A more detailed description of the detonation detection algorithm is presented below.

The detonation detection software performs the following actions:

1. For each possible target, determine if it is a candidate for detenotation. A target is a candidate for detonation when the missile's closest point to target has just been passed. This is calculated by comparing the missile's current distance to target with the missile's old position distance to target.
2. For a candidate, calculate its potential "hit_point". A "hit_point" is computed to be the point on the line segment from old_pos to new_pos which is closest to the current location of the target, veh[vcount].location. It is not decided whether the target was hit yet. Find the point on the projected path of the target that is closest to the computed "hit_point" to determine if it is within the detonation radius. To do this, project the position of the target vehicle using its current velocity ("anti-RVA") and find the point on its trajectory ("projected_pos") that is closest to "hit_point." If this "projected_pos" is within range, classify this detonation as a hit. If this "projected_pos" is outside the detonation radius, classify this detonation as a near miss.

The three functions that can be called to handle the different "hit" classifications are: vehicle_detonation_detected, vehicle_near_miss_detected, and terrain_detonation_detected.

The `vehicle_detonation_detected` function does the following:

1. Determine the detection type of the hit object (either DET_PLATFORMS, DET_MISSILES, or DET_BUILDINGS).
2. Invoke the detonation callback function registered with `det_class_init`. A callback function is used to inform another safobj sub-class library (such as libmissile) of the detonation. That library might need to modify private data (for example, saving the missile's detonation position and the target id) and/or perform an action (for example, send an impact PDU onto the simulation network).

The `vehicle_near_miss_detected` function does the following:

1. When DEBUG_DETONATION is ON, print a message saying that the object passed close to this target and print the distance squared between the missile and target.
2. Invoke the callback function registered with **det_class_init**. Since the detonation library really only figures out if the missile got within a given radius of the target, libmissile needs to decide if it was legitimate for the the missile to be aware of this condition. For example, "Does the missile have an active radar?" or "If the missile depends on reflected radar emissions from the launching aircraft, is the aircraft still alive and the radar still locked on?"

The **terrain_detonation_detected** function does the following:

1. When DEBUG_DETONATION is ON, print a message saying that the object detected a detonation with either trees, building, or ground. Also print the impact location.
2. Invoke the callback function registered with **det_class_init** with the other id field set at 0.

2 Functions

The following sections describe each function provided by libdetonation, including the format and meaning of its arguments, and the meaning of its return values (if any).

2.1 det_init

```
void det_init()
```

det_init initializes libdetonation. Call this before calling any other libdetonation functions.

2.2 det_class_init

```
void det_class_init(parent_class, callback)
    CLASS_PTR     parent_class;
    DET_CALLBACK callback;
```

'**parent_class**'

Specifies the parent class

'**callback**'

Specifies the function to call when detonations occur

det_class_init creates a handle for attaching detonation class information to vehicles. The **parent_class** is one created with **class_declare_class**. The **callback** function should be declared:

```
void callback(vehicle_id, position, det_type, other_id, target_position)
    int32   vehicle_id;
    float64 position[3];
    uint32  det_type;
    int32   other_id;
    float64 target_postion[3];
```

This is called when a detonation occurs. The **position** sent is the point of detonation and **target_postion** is the estimated position of the target at *that* time. The **det_type** code is one of the following:

DET_TREES
: Indicates proximity to a treeline or canopy edge.

DET_BUILDINGS
: Indicates proximity to a building or other structure. If the other structure is represented on the network, the vehicle ID of that structure will be provided.

DET_GROUND
: Indicates a detonation due to proximity to the ground.

DET_PLATFORMS
: Indicates proximity to a platform (vehicle, DI, etc.).

DET_MISSILES
: Indicates proximity to a missile (an entity on the network with a **munition** type).

DET_NEAR_MISS
: Indicates a near-miss of a target on the detonation list

If the collided entity exists in the vehicle table, its ID is given in the **other_id** field. For detonations with terrain features, the **other_id** will be zero.

2.3 det_create

```
void det_create(vehicle_id, parms)
    int32                      vehicle_id;
    DETONATION_PARAMETRIC_DATA *parms;
```

'vehicle_id'
: Specifies the vehicle ID

'parms' Specifies initial parameters

det_create creates the detonation class information for a vehicle and attaches it to the vehicle's libclass user data.

2.4 det_destroy

```
void det_destroy(vehicle_id)
    int32 vehicle_id;
```

'**vehicle_id**'
: Specifies the vehicle ID

det_destroy frees the detonation class information for a vehicle.

2.5 det_add_target

```
void det_add_target(vehicle_id, target_id)
    int32 vehicle_id;
    int32 target_id;
```

'**vehicle_id**'
: Specifies the vehicle ID

'**target_id**'
: Specifies the vtab ID of the target to be added

det_add_target adds a target to the list of detonation candidates for a missile.

2.6 det_delete_target

```
void det_delete_target(vehicle_id, target_id)
    int32 vehicle_id;
    int32 target_id;
```

'**vehicle_id**'
: Specifies the vehicle ID

'**target_id**'
: Specifies the vtab ID of the target to be deleted

det_delete_target deletes a target from the list of detonation candidates for a missile.

2.7 det_clear_list

```
void det_clear_list(vehicle_id)
    int32 vehicle_id;
```

'vehicle_id'
: Specifies the vehicle ID

det_clear_list clears the list of detonation candidates for a missile.

2.8 det_tick

```
void det_tick(vehicle_id, ctdb)
    int32 vehicle_id;
    CTDB *ctdb;
```

'vehicle_id'
: Specifies the vehicle ID

'ctdb' Specifies the terrain database

det_tick ticks the detonation sub-class. During the tick, if a detonation is detected, a packet may be sent and the callback function passed to det_class_init may be invoked.

ADST-TR-W003268

LibDI

Table of Contents

1 Overview

Libdi implements an instance of the hull class of components. It provides a low-fidelity model of DI vehicle dynamics. Capabilities are modeled only to the second order (maximum velocity, maximum acceleration), and they depend upon the soil type. Unlike previous models, the parameters for each soil type are specified in a data file, so the software does not need to be modified to accommodate new types of terrain.

The parameters of a DI vehicle are specified in its configuration file as follows:

```
(DI (soils (<integer soil type> (max_speeds <float forward KPH>
                                                 <float reverse KPH>)
                                     (max_accel <float mps2>)
                                     (max_decel <float mps2>)
                                     (max_turn  <float dps>)
                                     (max_climb <float degrees>)
                                     (dust_speeds <float smallKPH>
                                                  <float mediumKPH>
                                                  <float largeKPH>))
                (<integer soil type> (max_speeds <float KPH>
                                                 <float reverse KPH>)
                                     (max_accel <float mps2>)
                                     (max_decel <float mps2>)
                                     (max_turn  <float dps>)
                                     (max_climb <float degrees>)
                                     (dust_speeds <float smallKPH>
                                                  <float mediumKPH>
                                                  <float largeKPH>))
                 ...)
         (fuel_usage (<float speed1> <float speed2> ...)
             (<float rate1>  <float rate2> ...))
)
```

The parameters specified for soil type 0 (or the first soil type, if no 0 type is provided) are used as a default when on a soil type not in the list.

To indicate that the vehicle should not kick up any dust on a kind of soil, specify a speed which is higher than the maximum the vehicle can travel across that soil.

The fuel_usage table consists of a list of speeds in kilometers per hour, with a list of corresponding consumption rates in liters per hour. If an older vehicle parameter file is used, with a scalar fuel_usage figure, it will ignore it and use the internal default corresponding to 8 kilometers per liter. The minimum table size is one speed/rate pair.

Applications interface to the DI model primarily through the libhulls interface. The most efficient interface for controlling vehicle motion is HULLS_SET_DIRECTION_SPEED. All interfaces use only two dimensions of the provided parameters. Also, when a direction vector is given it is *not* necessary to make that vector a unit vector. Libdi will do the normalization only if it is necessary (for example, if the vehicle is already pointing the right way, no normalization is needed).

Libdi supports only one instantiation per vehicle (i.e., a vehicle may not have more than one DI hull).

The libhulls library defines a common set of functions (and the semantics of those functions) which are invoked on instances of the hulls class (such as those instantiated by libtracked or libfwa). It is possible to modify the DI model by changing an exisiting hulls interface function or by adding a completely new function.

To modify an existing libdi interface function would require the following actions:

1. If the change occurs only in the function body, only change the function code in the libdi library. If the change occurs to the function's argument list, change the function code in both the libdi library and the hulls interface structure definition found in libhulls.h. Also to maintain the common hulls interface, change the code for the modified function in any other hull specific component libraries (such as libfwa or libmissile).
2. Recompile ModSAF.

To add an additional libdi function to the current model would require the following actions:

1. Write the function as part of the libdi library. The function is written in the code which manages the libdi class information attached to each vehicle (di_class.c).
2. Add the function and its declaration to any of the other hull specific component libraries. This maintains the common hulls interface.
3. In the libdi source code that handles libhull initialization processing, include a **function_number, function** entry identifying the new function for the **cmpnt_define_instance** function and every other hull instance library (libfwa, libmissile, etc.).
4. In libhulls.h, add an entry to identify the new macro and associate it with a function code number. This new addition means that the number of hulls functions must be incremented by one. The hulls interface structure definition that appears in libhulls.h must include a structure to define the new function's argument list.
5. Recompile ModSAF.

To replace this DI model with a completely different one would require the following actions:

1. Decide on the get functions and set functions that would be required in the new model. Try to map these needed functions to the existing hulls interface. A function can map if its argument list can remain the same. Functions that can not map must be added to the hulls interface.
2. For those functions that can map to the existing hulls interface but whose code body you want to change, edit the code for the function in the libdi source file that contains the code to manage the libdi class information (di_class.c).
3. For those functions that can't map to the existing hulls interface, add an additional function to the hulls interface. The addition procedure was described above.
4. Recompile ModSAF.

2 Examples

To get the component number of my hull:

```
extern int32 my_hull;

if ((my_hull = cmpnt_locate(vehicle_id, reader_get_symbol("hull"))) ==
    CMPNT_NOT_FOUND)
  printf("Vehicle %d does not seem to have a hull\n", vehicle_id);
```

To then give a command to that hull:

```
if (my_hull != CMPNT_NOT_FOUND)
  HULLS_SET_DIRECTION_SPEED(vehicle_id, hull, dirvec, speed, 0.0, 0.0);
```

3 Functions

The following sections describe each function provided by libdi, including the format and meaning of its arguments, and the meaning of its return values (if any).

3.1 disinf_init

```
void disinf_init()
```

disinf_init initializes libdi. Call this before calling any other libdi functions.

3.2 disinf_class_init

```
void disinf_class_init(parent_class)
    CLASS_PTR parent_class;
```

'**parent_class**'
Class of the parent (declared with **class_declare_class**).

disinf_class_init creates a handle for attaching DI class information to vehicles. The parent_class is one created with class_declare_class.

3.3 disinf_tick

```
void disinf_tick(vehicle_id, ctdb)
    int32  vehicle_id;
    CTDB  *ctdb;
```

'**vehicle_id**'
Specifies the vehicle ID

'**ctdb**' Specifies the terrain database the vehicle is operating on.

disinf_tick ticks the DI hull dynamics model.

3.4 disinf_collision

```
    void disinf_collision(vehicle_id, position, coll_type,
                          other_id, other_mass, other_velocity)
        int32   vehicle_id;
        float64 position[3];
        uint32  coll_type;
        int32   other_id;
        float64 other_mass;
        float64 other_velocity[3];
```

'**vehicle_id**'
    Specifies the vehicle ID

'**position**'
    Specifies the position of impact in world coordinates

'**coll_type**'
    Specifies the type of collision

'**other_id**'
    Specifies the vehicle ID of the other party (or 0 if terrain)

'**other_mass**'
    Specifies the mass of the other party

'**other_velocity**'
    Specifies the velocity of the other party

disinf_collision tells the DI hull dynamics model that a collision occured. The coll_type should be one of the libcollision constants:

COLL_TREES
    Indicates crossing a treeline or canopy edge.

COLL_BUILDINGS
    Indicates crossing a building or other structure. If the other structure is represented on the network, the vehicle ID of that structure should be provided.

COLL_GROUND
    Should not be checked for ground vehicles.

COLL_PLATFORMS
    Indicates intersecting a platform (vehicle, DI, etc.).

COLL_MISSILES
    Indicates intersecting a missile (an entity on the network with a **munition** type.

3.5 disinf_damage

```
void disinf_damage(vehicle_id, damage)
    int32 vehicle_id;
    int32 damage;
```

'**vehicle_id**'
Specifies the vehicle ID

'**damage**' Specifies whether the DI dynamics should simulate being damaged

disinf_damage tells the DI hull dynamics model that it is damaged (or not) depending on the boolean value of the **damage** flag.

LibDISConst

Table of Contents

1 Overview

LibDISConst performs conversions between SIMNET and DIS constants for object type (guise) specification, appearance modifiers, and capabilities. The object type translations are specified in a data file ('disconst.rdr'). The appearance and capabilities translations are hard-coded (specification of these translations via a data file would be very difficult, since SIMNET and DIS are not very similar in the way they specify these items).

Internally, the translations are stored in two formats:

- The SIMNET to DIS translations are stored in a libOTMatch format database (see section 'Overview' in LibOTMatch Programmer's Manual).
- The DIS to SIMNET translations are stored in a n-ary tree, each node of which contains an array of sub-trees which are selected by the value of a field of the DIS entity type. Since all of the fields except country are within the range 0-255 (and most are less than 20), this selection is done using a direct map (no searching). Country codes are mapped down to a few small integers using a second-level map, to allow a direct map scheme to be used within the search tree for countries as well.

The source data for the translations comes from the data file '**disconst.rdr**'. It contains the following sections:

Traversal Instructions

Although all DIS entities are described using the same taxonomy (**Kind**, **Domain**, **Country**, **Category**, **Sub-category**, **Specific**, **Extra**), not all entity kinds use all the fields. Also, different entity kinds traverse the fields in different orders (for platforms, **Domain** selects the meaning of **Category**; for munitions, **Category** selects the meaning of **Domain**). Thus, the first thing specified in the data file is a default order of traversal, and special traversals for selected kinds. The traversal instructions take the general form:

```
(
 (<DIS Kind for which instructions apply>
  DISKind <field> <field> <field> ...)
 ...
)
```

The first traversal instruction is the default.

Country Codes

For compactness, and to allow both DIS 1.0 and DIS 2.0 object specifications within the same data file, the countries are stored internally using a lookup table. The translations refer to the countries by name. This name is used as the key into a lookup table. The lookup table takes the general form:

```
(
 ("<country name>" <DIS 1.0 code number> <DIS 2.0 code number>)
 ...
)
```

All countries referenced in the next section *must* appear in this lookup table.

Translations

Next come the SIMNET to DIS translations (the software deduces the DIS to SIMNET translations from this list at startup). The translation table takes the general form:

```
(
 (<SIMNET Object Type> (<Kind> <Domain> "<Country>"
                        <Category> <Sub-category> <Specific>
                        <Extra>))
 ...
)
```

All the fields except `Country` are small integers (`Kind` and platform `Domain` macros are supplied at the top of the file). `Country` should be a string which matches one of the countries in the Counry Code table, above.

Order is significant in two cases:

1. If more than one SIMNET object type corresponds to a single DIS entity type, the first one listed will determine the resulting DIS->SIMNET translation.
2. If a DIS entity type which is not in the table is translated to a SIMNET object type, it will default to the first object encountered which matches the most fields earliest in the traversal order (analogous to the libOTMatch defaulting scheme).

Thus, more common object types should be listed earlier in the data file.

The data file is complete as of the initial writing of this document. There are many defined SIMNET object types for which no obvious DIS equivalent exists (including almost all the structures). These problem translations are noted in the data file with comments.

To print the translation table in a format similar to the DIS standard, `make test`, then run `test 1`. The test program can also peform conversions from the command line (run `test` with no arguments for usage instructions).

1.1 Examples

The test program '`test.c`' gives examples of how to initialize libDISConst, and perform object type conversions.

2 Functions

The following sections describe each function provided by libDISConst, including the format and meaning of its arguments, and the meaning of its return values (if any).

2.1 disconst_init

```
int32 disconst_init(data_path, reader_flags, dis_version)
    char  *data_path;
    uint32 reader_flags;
    int32  dis_version;
```

'**data_path**'
Specifies directory where data file is expected

'**reader_flags**'
Specifies libreader format file reading flags

'**dis_version**'
Specifies version of DIS in use

disconst_init initializes libdisconst, causing it to read its data file ('**disconst.rdr**') from the passed directory (or '.') using the specified **reader_flags**. The return value is 0 for success, or one of the libreader **reader_read** return values.

2.2 disconst_print

```
void disconst_print()
```

disconst_print prints the DIS->SIMNET translation tree in a manner similar to the DIS specification.

2.3 disconst_guise_simnet_to_dis

```
void disconst_guise_simnet_to_dis(simnet, dis, warhead)
    ObjectType       *simnet;
```

```
        DIS_ENTITY_TYPE  *dis;
        DIS_WARHEAD_TYPE *warhead;
```

'simnet' Specifies SIMNET object type

'dis' Returns DIS entity type

'warhead' Returns DIS warhead type

disconst_guise_simnet_to_dis converts a SIMNET guise to its DIS equivalent. If the **warhead** is not desired, pass **NULL**.

2.4 disconst_guise_dis_to_simnet

```
    void disconst_guise_dis_to_simnet(dis, warhead, simnet)
        DIS_ENTITY_TYPE  *dis;
        DIS_WARHEAD_TYPE *warhead;
        ObjectType       *simnet;
```

'dis' Specifies DIS entity type

'warhead' Specifies DIS warhead (or **NULL**)

'simnet' Returns SIMNET object type

disconst_guise_dis_to_simnet converts a DIS guise to its SIMNET equivalent. If the warhead is not relevant, pass **NULL** for **warhead**.

2.5 disconst_appearance_simnet_to_dis

```
    void disconst_appearance_simnet_to_dis(dis_type, simnet, dis)
        DIS_ENTITY_TYPE       *dis_type;
        uint32                *simnet;
        DIS_ENTITY_APPEARANCE *dis;
```

'dis_type'
Specifies DIS entity type

'simnet' Specifies SIMNET appearance bits

'dis' Returns DIS appearance bits

disconst_appearance_simnet_to_dis converts a SIMNET appearance to its DIS equivalent.

2.6 disconst_appearance_dis_to_simnet

```
void disconst_appearance_dis_to_simnet(dis_type, dis, simnet)
    DIS_ENTITY_TYPE       *dis_type;
    DIS_ENTITY_APPEARANCE *dis;
    uint32                *simnet;
```

'dis_type'
Specifies DIS entity type

'dis' Specifies DIS appearance bits

'simnet' Returns SIMNET appearance bits

disconst_appearance_dis_to_simnet converts a DIS appearance to its SIMNET equivalent.

2.7 disconst_capabilities_simnet_to_dis

```
void disconst_capabilities_simnet_to_dis(simnet, dis)
    VehicleCapabilities      *simnet;
    DIS_ENTITY_CAPABILITIES *dis;
```

'simnet' Specifies SIMNET capabilities

'dis' Returns DIS capabilities

disconst_capabilities_simnet_to_dis converts a SIMNET capabilities record to its DIS equivalent.

2.8 disconst_capabilities_dis_to_simnet

```
void disconst_capabilities_dis_to_simnet(dis, simnet)
    DIS_ENTITY_CAPABILITIES *dis;
    VehicleCapabilities      *simnet;
```

'dis' Specifies DIS capabilities

'simnet' Returns SIMNET capabilities

disconst_capabilities_dis_to_simnet converts a DIS capabilities record to its SIMNET equivalent.

ADST-TR-W003268

LibDither

Table of Contents

1 Overview

Libdither uses a fairly standard method for generating shading dithers.

A program using libdither will first make a dither-generating matrix using the function `dither_matrix()`. The function generates a `NxN` dither matrix where `N` is a power of 2. The algorithm is based on the Judice, Jarvice, Ninke recurrence relation detailed in Foley, vanDam, et. al. (II edition, p. 571). The dither matrix can be used to generate dither bitmaps. For example, the 2x2 dither matrix:

$$\begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix}$$

can be used to generate the following bitmaps:

```
00 10 10 11 11
00 00 01 01 11
```

These bitmaps can be generated explicitly, or they can be generated on an as-needed basis, and copied or OR'd into other bitmaps.

Macros are provided to correctly declare the matrix (`DITHER_MATRIX`) and bitmap (`DITHER_BITMAPS`) data structures, as well as a macro which computes the number of unique bitmaps which can be generated from a dither matrix of a given size (`DITHER_COUNT`).

2 Examples

The following example program (called '`xtest.c`' in the libdither source directory) fills the root window with a number of 4x4 rectangles. Each rectangle is stippled with a dither corresponding to its distance from the center of the screen. A 16x16 dither pattern is used.

Depending upon whether `#define DO_RECTS` is present, the program will either use `dither_bitmaps()` to generate stipples for a GC, or it will use `dither_or()` to generate one large bitmap. The second version runs about 15 times as fast, due to the reduction X traffic.

```
/*
#define DO_RECTS
*/

#ifndef DO_RECTS
#define DO_BITMAP
#endif

#include "libdither.h"

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <stdio.h>
#include <math.h>

#define SIZE 16

#ifdef DO_RECTS
main()
{
    Display *dpy;
    int32 screen;
    Window root;
    GC gc;
    int32 i, x, y;
    int32 w, h, Hw, Hh;
    float64 d;
    DITHER_MATRIX(mat,SIZE);
    DITHER_BITMAPS(bits,SIZE);
    Pixmap pix[DITHER_COUNT(SIZE)];

    /* Make the dither generating matrix */
    dither_matrix(SIZE, mat);

    /* Make DITHER_COUNT(SIZE) bitmaps from the matrix */
    dither_bitmaps(SIZE, mat, bits);
    /* Open the display device */
    dpy = XOpenDisplay(NULL);
    if (!dpy)
      exit (2);

    /* Find the screen and root window */
    screen = DefaultScreen(dpy);
    root = RootWindow(dpy, screen);

    /* Generate DITHER_COUNT(SIZE) pixmaps from the bitmaps */
    for(i=0;i<DITHER_COUNT(SIZE);i++)
      pix[i] = XCreateBitmapFromData(dpy, root, bits[i], SIZE, SIZE);

    /* Create a GC for filling in the rectangles */
    gc = XCreateGC(dpy, root, 0, NULL);
    XSetForeground(dpy, gc, BlackPixel(dpy, screen));
    XSetBackground(dpy, gc, WhitePixel(dpy, screen));
    XSetFillStyle(dpy, gc, FillOpaqueStippled);

    /* Find the size of the screen */
    w = DisplayWidth(dpy,screen);
    h = DisplayHeight(dpy,screen);
    Hw = w/2;
    Hh = h/2;

    /* Find the distance that the furthest rectangle will fall from the
     * center.
     */
    d = sqrt(Hw*Hw + Hh*Hh);

    /* Fill in a bunch of 4x4 rectangles, stippled according to their
     * distance from the center of the screen.
     */
    for (y=0;y<h;y+=4)
      for (x=0;x<w;x+=4)
      {
          i = (DITHER_COUNT(SIZE)-1) *
              sqrt((Hw-x)*(Hw-x) + (Hh-y)*(Hh-y))/d;
          XSetStipple(dpy, gc, pix[i]);
          XFillRectangle(dpy, root, gc, x, y, 4, 4);
      }

    /* Close the display */
    XCloseDisplay(dpy);
}
#endif
#ifdef DO_BITMAP
main()
{
    XImage *image;
```

```
Display *dpy;
int32 screen;
Window root;
GC gc;
int32 i, x, y;
int32 w, h, Hw, Hh;
float64 d;
DITHER_MATRIX(mat,SIZE);
char *bitmap;

/* Make the dither generating matrix */
dither_matrix(SIZE, mat);

/* Open the display device */
dpy = XOpenDisplay(NULL);
if (!dpy)
  exit (2);

/* Find the screen and root window */
screen = DefaultScreen(dpy);
root = RootWindow(dpy, screen);

/* Find the size of the screen *'
w = DisplayWidth(dpy,screen);
h = DisplayHeight(dpy,screen);

/* Allocate a bitmap big enough to cover the whole screen */
bitmap = (char *)malloc(h * ((w+7)/8));
bzero(bitmap, h * ((w+7)/8));

/* Create an XYBitmap format XImage */
image = XCreateImage(dpy, NULL, 1, XYBitmap, 0, bitmap,
                     w, h, 8, (w+7)/8);

/* Find the distance that the furthest rectangle will fall from the
 * center.
 */
Hw = w/2;
Hh = h/2;
d = sqrt(Hw*Hw + Hh*Hh);

/* Fill in a bunch of 4x4 rectangles, stippled according to their
 * distance from the center of the screen.
 */
for (y=0;y<h;y+=4)
  for (x=0;x<w;x+=4)
  {
      i = (DITHER_COUNT(SIZE)-1) *
          sqrt((Hw-x)*(Hw-x) + (Hh-y)*(Hh-y))/d;
      dither_or(SIZE, i, mat, bitmap, w, h, x, y, 4, 4);
      }

    /* Create a GC for copying the image */
    gc = XCreateGC(dpy, root, 0, NULL);
    XSetForeground(dpy, gc, BlackPixel(dpy, screen));
    XSetBackground(dpy, gc, WhitePixel(dpy, screen));

    /* Copy the bitmap to the screen */
    XPutImage(dpy, root, gc, image, 0, 0, 0, 0, w, h);

    /* Close the display */
    XCloseDisplay(dpy);
}
#endif
```

3 Functions

The following sections describe each function provided by libdither, including the format and meaning of its arguments, and the meaning of its return values (if any).

3.1 dither_matrix

```
void dither_matrix(size, mat)
    int32          size;
    DITHER_MATRIX(mat, size);
```

'size'   Specifies the size of the dither matrix

'mat'   Returns the dither-generating matrix

dither_matrix creates a dither matrix which can then be passed to dither generating functions.

3.2 dither_bitmaps

```
void dither_bitmaps(size, mat, bits)
    int32          size;
    DITHER_MATRIX( mat, size);
    DITHER_BITMAPS(bits, size);
```

'size'   Specifies the size of the dither matrix

'mat'   Specifies the dither generating matrix

'bits'   Returns the bitmaps

dither_bitmaps creates **DITHER_COUNT(size)** bitmaps from the dither generating matrix (created with **dither_matrix()**). **bits[i] 0<=i<DITHER_COUNT(size)**, is the ith bitmap, and can be passed to X-windows functions such as **XPutImage()**, or **XCreatePixmapFromBitmapData()**.

3.3 dither_or

```
void dither_or(size, which_dither, mat, bitmap, bitmap_width,
                      bitmap_height, x0, y0, area_width, area_height)
        int32          size;
        int32          which_dither;
        DITHER_MATRIX(mat, size);
        char          *bitmap;
        int32          bitmap_width;
        int32          bitmap_height;
        int32          x0, y0;
        int32          area_width, area_height;
```

'size' Specifies the size of the dither

'which_dither'
Specifies which dither to generate

'mat' Specifies the dither-generating matrix

'bitmap' Returns the bitmap

'bitmap_width, bitmap_height'
Specifies the size of the bitmap

'x0, y0' Specifies the starting location within the bitmap

'area_width, area_height'
Specifies the area to dither within the bitmap

dither_or OR's in an area_width x area_height section of the which_ditherth dither generated from the matrix mat. The upper left corner of the dither is at x0, y0 of the bitmap.

See ⟨undefined⟩ [dither'copy], page ⟨undefined⟩.

3.4 dither_copy

```
    void dither_copy(size, which_dither, mat, bitmap, bitmap_width,
                      bitmap_height, x0, y0, area_width, area_height)
        int32          size;
        int32          which_dither;
        DITHER_MATRIX(mat, size);
        char          *bitmap;
        int32          bitmap_width;
        int32          bitmap_height;
        int32          x0, y0;
        int32          area_width, area_height;
```

'size' Specifies the size of the dither

‘which_dither’
Specifies which dither to generate

‘mat’ Specifies the dither-generating matrix

‘bitmap’ Returns the bitmap

‘bitmap_width, bitmap_height’
Specifies the size of the bitmap

‘x0, y0’ Specifies the starting location within the bitmap

‘area_width, area_height’
Specifies the area to dither within the bitmap

dither_copy works just like **dither_or**, except that zero bits in the dither are set to zero in the bitmap. Hence, **dither_copy** runs only about half the speed of **dither_or**.

See ⟨undefined⟩ [dither˙or], page ⟨undefined⟩.

LibDr

Table of Contents

1 Overview

Because of limitations in network bandwidth, the DIS world uses the concept of dead reckoning (DR) to lessen the network traffic of DIS entities. The theory is that if the (DR) algorithm used by a particular entity is known, the position (and orientation if the algorithm specifies it) of a remote entity can be approximated until the next actual update is received. On the sender's side, the sender calculates the actual state of the entity and then compares it against the approximated state that the world assumes as truth. If the two states deviate by more than a specified set of thresholds, the sender must send out a packet updating its state for the rest of the players.

Libdr provides a set of generic routines which can be combined to do various types of dead reckoning algorithms. It assumes a cartesian coordinate system, such as GCC or SIMNET's Level Metric, but is independent of units; the parameters – position, velocity, time, etc – must be specified consistently, but the user may choose the units. In other words, if position is specified in meters, and time in seconds, the velocity must be specified in meters/second. It can not be kilometers/second or meters/millisecond. Position, velocity, and acceleration must be also be specified with respect to the same coordinate system (WORLD vs BODY).

In addition to the dead reckoning routines, libdr also provides a set of routines to do positional and rotational threshold checking. These routines can be used in conjunction with the dead reckoning routines to determine whether or not a entity has exceeded its thresholds, requiring that a packet be sent out to update its position in the world.

All linear dead reckoning routines are supported for both float32 and float64 vectors. Angular dead reckoning routines using 3 x 3 matrices for rotation are supported for both float32 and float64 matrices. Angular dead reckoning routines using quaternions for rotation are currently supported for float64 quaternions only. Threshold routines for rotational checking are supported for both float64 and float32 matrices. Threshold routines for rotational checking using quaternions and positional checking are supported for float64 only. Elapsed_time and position are always represented using float64's. In addition, the angular dead reckoning routines support both WORLD_to_BODY and BODY_to_WORLD matrices as well as quaternions. The naming convention for these functions is:

32 bit or 64 bit floating point

Name contains 32 or 64 at or near end

body or world

Name ending in body or `world` indicates that rotations are specified as `body_to_world` or `world_to_body` respectively.

matrix or quat

Name contains matrix or quat at or near end indicates that rotations are specified using a 3 x 3 matrix or quaternion respectively.

For example, the 64 bit, first order angular dead reckoning function using world_to_body matrices is dr_first_order_angular_dr_matrix64_world.

The Functions Chapter specifies the versions which are currently supported. The prototype types described in that section use generic terms as follows:

scalar s Either float32 s or float64 s

vector v Either float32 v[3] or float64 v[3]

matrix m Either float32 m[3][3] or float64 m[3][3]

quat q quaternion which contains 2 parts, a scalar, referred to as qscalar and a vector, referred to as qvector

rotation r
Either matrix or quat

2 Examples

The program 'dr_test', found in 'dr_test.c' in the libdr source directory, demonstrates some possible uses for libdr. It can be compiled with the command 'make dr_test'. Due to the nature of the test cases, they do not determine their own success or failure. Their output must be analyzed by the tester with the help of a scientific calculator.

The first order and second order linear dead reckoning routines can be easily combined to form the following algorithms, defined in the DIS 2.0 Standard. The 3 letter naming conventions can be interpreted as:

```
F/R : Fixed Rotation or 1st Order Rotation
P/V : 1st Order Position or 2nd Order Position (Velocity)
W/B : World or Body Coordinates

FPW : Fixed Rotation, First Order Position, World Coordinates
      Defined as Algorithm 2 in the DIS 2.0 Standard
      (DR algorithm used in SIMNET)

RPW : First Order Rotation, First Order Position, World Coordinates
      Defined as Algorithm 3 in the DIS 2.0 Standard

RVW : First Order Rotation, Second Order Position, World Coordinates
      Defined as Algorithm 4 in the DIS 2.0 Standard

FVW : Fixed Rotation, Second Order Position, World Coordinates
      Defined as Algorithm 5 in the DIS 2.0 Standard

FPB : Fixed Rotation, First Order Position, Body Coordinates
      Defined as Algorithm 6 in the DIS 2.0 Standard

FVB : Fixed Rotation, Second Order Position, Body Coordinates
      Defined as Algorithm 9 in the DIS 2.0 Standard
```

The following code fragment illustrates how the routines can be combined to execute the RPW algorithm. The 'RPW' is specified in the DIS 2.0 standard as DR algorithm 3. For this algorithm, first order linear positional dead reckoning as well as first order rotational dead reckoning is applied. Initial velocity is specified in WORLD Coordinates. Example is for float64's using an initial BODY_to_WORLD orientation matrix.

```
/*
 * Given elapsed_time, initial_position, and velocity, a
 * current_position position is calculated and returned in
 * 'current_position'
 */
float64 elapsed_time;        /* time from when initial_position was
                                accurate until now                 */
float64 initial_position[3]; /* position before DR applied         */
float64 velocity[3];         /* linear velocity, WORLD coords      */
float64 current_position[3]; /* position after DR applied          */

dr_first_order_linear_dr64 ( elapsed_time, initial_position, velocity,
                             current_position );

/*
 * The angular velocity vector only changes when a new packet arrives.
 * To save compute cycles, the omega and axis of rotation that is
 * extracted from the vector should be cached.
 *
 * Given an angular velocity vector, omega (the magnitude) and axis of
 * rotation are determined and returned in 'omega' and 'axis'.
 */
float64 angular_velocity[3]; /* angular velocity, WORLD coords   */
float64 omega;               /* unpacked angular velocity        */
float64 axis[3];             /* axis of rotation                 */

dr_unpack_angular_velocity64 ( angular_velocity, &omega, axis );

/*
 * Given an initial BODY_to_WORLD matrix, elapsed_time, omega, and
 * axis of rotation, a current BODY_TO_WORLD matrix is calculated
 * and returned in 'current_body_to_world'.
 */

float64 initial_body_to_world[3][3]; /* BODY_to_WORLD matrix before DR */
float64 current_body_to_world[3][3]; /* BODY_to_WORLD matrix after DR  */

dr_first_order_angular_dr_matrix64_body ( elapsed_time, omega, axis,
                                          initial_body_to_world,
                                          current_body_to_world );
```

3 Functions

The following sections describe each function provided by libdr, including the format, meaning of its arguments, and meaning of its return values (if any).

3.1 dr_first_order_linear_dr

```
    void dr_first_order_linear_dr ( elapsed_time, initial_position,
                                    initial_velocity, current_position )
        float64     elapsed_time;
        vector     initial_position[3];
        vector     initial_velocity[3];
        vector     current_position[3];
```

'elapsed_time'
: Time elapsed between **initial_position** and **current_position**

'initial_position'
: Position to dead reckon from.

'initial velocity'
: Velocity to apply to **initial_position** to calculate **current_position**.

'current_position'
: New position after dead reckoning; return value.

Available formats:

- **dr_first_order_linear_dr64**
- **dr_first_order_linear_dr32**

dr_first_order_linear_dr64 performs first order linear dead reckoning using the equation:

```
  p = p_0 + (v_0 * t)

  p   : current_position          p_0 : initial_position
                                  v_0 : initial_velocity
                                  t   : elapsed_time
```

3.2 dr_second_order_linear_dr

```
void dr_second_order_linear_dr ( elapsed_time, initial_position,
                                 initial_velocity, initial_acceleration,
                                 current_position, current_velocity )
    float64    elapsed_time;
    vector     initial_position[3];
    vector     initial_velocity[3];
    vector     initial_acceleration[3];
    vector     current_position[3];
    vector     current_velocity[3];
```

'**elapsed_time**'
: Time elapsed between **initial_position/velocity** and **current_position/velocity**

'**initial_position**'
: Position to dead reckon from.

'**initial velocity**'
: Velocity to apply to **initial_position** to calculate

'**initial_acceleration**'
: Velocity to apply to **initial_position** to calculate **current_position** and **current_velocity**

'**current_position**'
: New position after dead reckoning; return value.

'**current_velocity**'
: New velocity after dead reckoning; return value.

Available formats:

- **dr_second_order_linear_dr64**
- **dr_second_order_linear_dr32**

dr_second_order_linear_dr64 performs second order linear dead reckoning using the equation:

```
p = p_0 + (v_0 * t) + (0.5*a_0*(t^2))
v = v_0+ a_0

p   : current_position          p_0 : initial_position
v   : current_velocity          v_0 : initial_velocity
                                a_0 : initial_acceleration
                                t   : elapsed_time
```

3.3 dr_first_order_angular_dr

```
void dr_first_order_angular_dr ( elapsed_time, omega, axis,
                                 initial_rotation, current_rotation )
    float64  elapsed_time;
    scalar   omega;
    vector   axis;
    rotation initial_rotation;
    rotation current_rotation;
```

'elapsed_time'
Time elapsed between initial_rotation and current_rotation

'omega' Magnitude of angular velocity.

'axis' Axis of rotation of angular velocity

'initial_rotation'
Rotation to dead reckon from

'current_rotation'
New rotation after rotational dead reckoning; return value.

Available formats:

- dr_first_order_angular_dr_matrix64_body
- dr_first_order_angular_dr_matrix32_body
- dr_first_order_angular_dr_matrix64_world
- dr_first_order_angular_dr_matrix32_world
- dr_first_order_angular_dr_quat64_world

dr_first_order_angular_dr performs first order angular dead reckoning using either cosine matrices or quaternions. The change in rotation for a given time frame is represented by the following quaternion:

```
h_angle  : half_angle of rotation
w        : omega
t        : elapsed_time
axis     : axis of rotation
quat     : intermediate quaternion for change in rotation (qscalar, qvector)
s_angle  : sin (h_angle)

h_angle    = 0.5*wt
qscalar    = cos (h_angle)
qvector[0] = s_angle * axis[0]
```

```
qvector[1] = s_angle * axis[1]
qvector[2] = s_angle * axis[2]
```

If the input rotation is a quaternion, the initial_quaternion is concatenated with the newly calculated quaternion to return a quaternion representing the dead reckoned rotation. If the input rotation is a matrix, the newly calculated quaternion is first converted to a direction cosine matrix. Two matrices are then concatenated to return a matrix representing the dead reckoned rotation.

3.4 dr_unpack_angular_velocity

```
extern void dr_unpack_angular_velocity ( angular_velocty, omega, axis )
    vector      angular_velocity;
    scalar     *omega;
    vector      axis;
```

'angular_velocity'
Angular velocity vector in Body Coordinates.

'omega' Pointer to Magnitude of angular velocity; return value.

'axis' Axis of rotation of angular velocity; return value.

Available formats:

- **dr_unpack_angular_velocity64**
- **dr_unpack_angular_velocity32**

dr_unpack_angular_velocity takes an angular velocity vector and extracts its magniitude, **omega**, and its axis of rotation, **axis**. This action need only be done whenever the angular velocity changes, not each time one performs angular dead reckoning. Therefore, it is suggested that this routine be called only when necessary and the values for omega and axis be cached for efficiency. For remote entity rotational dead reckoning, this would need to be called for a particular entity whenever a new packet arrived for it.

3.5 dr_check_location_thresholds

```
int32 dr_check_location_thresholds ( thresh_sq, current_position
                                     dr_position )
    scalar   thresh_sq;
```

```
        vector   current_position;
        vector   dr_position;
```

'thresh_sq;'
: Square of the position threshold. Must be in units consistent with those used for position.

'current_position'
: Actual position of the local entity

'dr_position'
: Dead reckoned position of the local entity since the last packet was sent

dr_check_location_thresholds compares the current_position with that of dr_position. If 'current' deviates from 'dr' by more than the specified threshold, it returns TRUE (1) indicating that the threshold has been exceeded and therefore, a packet should be sent. Otherwise, it returns FALSE (0). The algorithm actually requires the square of the threshold. Therefore, for efficiency, the routine takes the threshold squared as a parameter rather than just the threshold value and assumes the user can cache the squared value instead of recomputing it every time.

3.6 dr_check_orientation_threshold

```
    int32 dr_check_orientation_threshold ( thresh, current_rotation,
                                           dr_rotation )
        scalar   thresh;
        rotation current_rotation;
        rotation dr_rotation;
```

'thresh'
: For efficieny purposes, thresh has different representations depending on whether or not quaternions or matrices are used. Given a rotational threshold in radians, thresh should be one of the following:

```
thresh = cos (0.5 * rotation threshold)         ; for quaternions
thresh = 2 * cos (rotation threshold) + 1       ; for matrices
```

'current_rotation'
: Rotation representing the actual orientation of the entity

'dr_rotation'
: Rotation representing the dead reckoned approximation of the orientation of the entity.

Available formats:

- **dr_check_orientation_threshold_matrix64**
- **dr_check_orientation_threshold_matrix32**
- **dr_check_orientation_threshold_quat**

dr_check_orientation_threshold compares the current_rotation with that of the dr_rotation. If '**current**' deviates from '**dr**' by more than the specified threshold, it returns TRUE (1) indicating that the threshold has been exceeded and therefore, a packet should be sent. Otherwise, it returns FALSE (0). As with **dr_check_location_thresholds**, the algorithms utilize the threshold information in a manner which can be precalculated and stored for future use. Details of the format are indicated above under '**thresh**'.

3.7 dr_quat_transform

```
void dr_quat_transform ( src, rotation, dest )
    vector src;
    quat   rot;
    vector dest;
```

'**src**' Initial vector that is to be transformed.

'**rot**' Quaternion which represents the rotation by which the vector is to be rotated.

'**dest**' Resulting vector after transformation has been applied; return value.

dr_quat_transform rotates a vector by the rotation specified in the quaternion. Ultimately, this belongs in a quaternion equivalent of libvecmat.

Libechelondb

Table of Contents

1 Overview

Libechelondb provides a database of named standard military echelon organizations (also referred to as *Units*), which can be used as templates or parts of templates for unit creation. Libechelondb uses a database format which is accessed using libotmatch (see section 'Overview' in LibOTMatch Programmer's Manual). A GUI for unit creation can access this library to allow the initialization of a unit to expand into the initialization of an entirely instantiated unit hierarchy. Given a unit to be created, libechelondb only supplies the information used to create the unit and its subordinates. It does not actually create the unit persistent object or its subordinate persistent objects.

The types of information stored in the echelon database are as follows:

The collection of subunits in a unit

> The subunits can be recursive references to other libechelondb units. For example, a platoon can contain several vehicles, while a company can contain several command vehicles and several platoons.

The way vehicle designations are generated for each vehicle or unit

> For example, a company might be designated as "A ", the first platoon in that company might be designated as "A1 ", and the second vehicle in the first platoon might be designated "A12".

The order of promotion between units

> This ordering can also be used to identify unique members in a formation of units. This information is stored implicitly in the data file by the ordering of the subunits.

The echelon database is stored in the data file '`echelondb.rdr`'. The format of this data file is as follows:

```
(
  (<unit identifier> (<subunit identifier1>
                      <subunit identifier2>
                      <subunit identifier3>
                      ...))
  <more unit definitions>
  ...
)
```

`<unit identifier>` is the object type representing the unit.

A `<subunit identifier>` has the following format:

```
([leaf|tree] [<vehicle identifier> | <unit identifier>] <marking pattern>)
```

leaf means that this is a terminal node in a unit hierarchy. **tree** means that this subunit should be reexpanded into other subunits by means of a recursive query into the database. A unit identifier could be specified as a leaf node, which would imply a unit hierarchy containing a command unit which does not have vehicles in it. This could be used to support the representation of aggregate simulation, or provide empty units for later task organization.

The **<vehicle identifier>** or **<unit identifier>** specifies the object type of the unit at this level in the hierarchy. Vehicle object types indicate physical vehicles within the unit hierarchy, while unit object types indicate conceptional aggregate units (with or without vehicles subordinate to them).

The **<marking pattern>** is a three character string used to generate markings for a particular unit in the hierarchy. In such a string, the character ? indicates to inherit the character at this position from the superior unit. Thus, a vehicle with a marking pattern of "??4" will receive a marking of "A24" when present in a platoon labeled "A2 ".

An example of a data file which supports platoons, companies and battalions of M1 tanks is as follows:

```
(
 (unit_US_M1_Platoon ((leaf vehicle_US_M1 "??1")
                      (leaf vehicle_US_M1 "??2")
                      (leaf vehicle_US_M1 "??3")
                      (leaf vehicle_US_M1 "??4")))

 (unit_US_M1_Company ((leaf vehicle_US_M1 "?66")
                      (leaf vehicle_US_M1 "?65")
                      (tree unit_US_M1_Platoon "?1 ")
                      (tree unit_US_M1_Platoon "?2 ")
                      (tree unit_US_M1_Platoon "?3 ")))

 (unit_US_M1_BattalionHQ ((leaf vehicle_US_M1 "HQ1")
                          (leaf vehicle_US_M1 "HQ2")))

 (unit_US_M1_Batallion ((tree unit_US_M1_BattalionHQ "H  ")
                        (tree unit_US_M1_Company "1  ")
                        (tree unit_US_M1_Company "2  ")
                        (tree unit_US_M1_Company "3  ")))
)
```

The algorithm used within libechelondb for queries is as follows. Queries are initiated by a call to echdb_expand, passing in an array of the ECHDB_DATA structure to be filled out. The format of the ECHDB_DATA data structure, which will be filled out by calls to echdb_expand, is as follows:

```
typedef struct echdb_data
{
    ObjectType type;
    int32      superior;
    int32      promotion_index;
    char       designation[4];
} ECHDB_DATA;
```

type is the vehicle or unit object type of the unit.

superior is the array index of the unit which is superior to this unit. The topmost unit, which will be the zeroth element of the array, will have a superior of -1.

promotion_index is a small integer representing the order of promotion for all the units directly subordinate to the superior of this unit. All units directly subordinate to a particular superior unit will have a unique promotion_index.

designation is a NULL terminated 3-character string which represents the designation or "bumper-number" of the unit.

Queries into the database by echdb_expand use otm_query (see section 'otm_query' in LibOT-Match Programmer's Manual) to access the data for a particular unit. First, the input unit is used to fill out the zeroth ECHDB_DATA element. Then, if a unit is found via a direct otm_query match, the data returned by otm_query is used to fill out other ECHDB_DATA elements. Recursive queries are performed to expand any elements marked as tree nodes.

1.1 Examples

The test program 'test.c' demonstrates initialization of libphysdb and all the libraries it depends on. The output of a sample run is as follows:

```
crimson-> test unit_US_M1_Company "A  "

Expansion of unit_US_M1_Company yields 18 subunits:
  0:        unit_US_M1_Company, marking "A  ", index 0, superior is  -1
  1:              vehicle_US_M1, marking "A66", index 0, superior is   0
  2:              vehicle_US_M1, marking "A65", index 1, superior is   0
```

```
 3:       unit_US_M1_Platoon, marking "A1 ", index 2, superior is  0
 4:       unit_US_M1_Platoon, marking "A2 ", index 3, superior is  0
 5:       unit_US_M1_Platoon, marking "A3 ", index 4, superior is  0
 6:            vehicle_US_M1, marking "A11", index 0, superior is  3
 7:            vehicle_US_M1, marking "A12", index 1, superior is  3
 8:            vehicle_US_M1, marking "A13", index 2, superior is  3
 9:            vehicle_US_M1, marking "A14", index 3, superior is  3
10:            vehicle_US_M1, marking "A21", index 0, superior is  4
11:            vehicle_US_M1, marking "A22", index 1, superior is  4
12:            vehicle_US_M1, marking "A23", index 2, superior is  4
13:            vehicle_US_M1, marking "A24", index 3, superior is  4
14:            vehicle_US_M1, marking "A31", index 0, superior is  5
15:            vehicle_US_M1, marking "A32", index 1, superior is  5
16:            vehicle_US_M1, marking "A33", index 2, superior is  5
17:            vehicle_US_M1, marking "A34", index 3, superior is  5
```

2 Functions

The following sections describe each function provided by libechelondb, including the format and meaning of its arguments, and the meaning of its return values (if any).

2.1 echdb_init

```
int32 echdb_init(directory, flags)
    char  *directory;
    uint32 flags;
```

'directory'
Specifies the directory where the echdb file is expected

'flags' Specifies reader options (see section 'reader_read' in LibReader Programmer's Manual)

echdb_init initializes libechdb, causing it to read its data file 'echdb.rdr' from the specified directory. The flags are as in reader_read. The return value is zero if the read succeeds, or one of the libreader return values: READER_READ_ERROR READER_FILE_NOT_FOUND.

Note that the libreader function reader_init, the librdrconst function rdc_init, and the libotmatch function otm_init must be called before this function.

2.2 echdb_expand

```
int32 echdb_expand(input, input_designation, output_length, output)
    ObjectType  input;
    char        input_designation[];
    int32       output_length;
    ECHDB_DATA  output[];
```

'input' Specifies the unit to look up in the database

'input_designation'
Specifies the ASCII character designation string for the top level unit. Designations for lower level units will be derived from this string.

'output_length'
Specifies the maximum number of outputs to fill out.

'output' Specifies an array of ECHDB_DATA records to fill out the results of the query.

echdb_expand looks up the supplied `input` in the echelon database and fills out the units in the `output` structure. A maximum of **`output_length`** units will be filled out. **echdb_expand** returns the number of entries in `output` that have been filled out by the call. `input` is *always* returned as the zeroth element of `output`. If `input` is not found in the echelon database, 1 will therefore be returned.

ADST-TR-W003268

LibEditor

Table of Contents

1 Overview

Libeditor provides a facility to build flexible data-driven editors with a minimum of effort. The library is an integral part of the ModSAF user interface architecture.

The programming paradigm used is as follows:

- The application defines a data structure which is to be edited. This data structure is fixed-size, although it may contain one variable length field (such as the way a protocol PDU cannot be larger than the ethernet packet, but it may contain a variable amount of valid information).
- The application defines a data file which expresses five things about the data which is edited:
  - The name of the editor
  - The memory layout of fields in the structure
  - The abstract type of those fields which may be edited
  - The way to initialize each field
  - The fields which impact the way an item is rendered
- The application passes this information to libeditor at startup, at which time a user interface is constructed. Optionally, the application may also pass functions which are called to display the item being edited, and to process the edited item when the editor is exited.
- The application starts the editor when needed.

The following sections describe the five main sections of an editor definition file.

1.1 Name Definition

The first thing which must be specified for an editor is its name. This should be a short description of what the editor does. For example:

```
(name "Text Editor")
```

1.2 Structure Definition

The structure is defined by listing each member of the structure by name, and then specifying its data type, and optionally its length. Padding is specified by using the reserved label padding and specifying how many bits (must be a multiple of 8).

For example, the structure:

```
struct foo
{
    int16 bar;
    int16 _padding1;
    uint32 baz[3];
};
```

Would be defined with the expression:

```
(struct (bar     int16)
        (padding 16)
        (baz     uint32 3)
        )
```

The data types which are recognized are as follows:

```
int8 uint8 int16 uint16 int32 uint32 float32 float64
PointDescription MunitionQuantity ObjectIDType
```

When the editor is created, the software compares the size of the structure specified by this data file to the size passed in, to help identify inconsistencies (usually padding problems).

Note that C compilers pad structures for efficient access to data over the bus. In general, the following rules should be assumed:

- The byte offset to a primitive type which is n bytes long will be a multiple of n. For example, the structure:

  ```
  struct
  {
      int8  bar;
      int16 baz;
  } foo;
  ```

  has a byte of implicit padding between bar and baz so that baz may start on a 2 byte offset into the structure.

- The byte offset to a structure within a structure must conform to the strictest alignment requirements of the sub-structure's members. For example, the structure:

  ```
  struct
  {
      int8 bar;
  ```

```
    struct
    {
        int16 baz;
        int32 bax;
    } biz;
} foo;
```

has 3 bytes of implict padding between **bar** and the sub-structure **biz**, and 2 bytes of implicit padding between **baz** and **bax**.

- The total size of a structure must be a multiple of its member's strictest alignment requirement. This is to ensure structures can be placed into arrays. For example, the structure:

  ```
  struct
  {
      int32 bar;
      int16 baz;
  } foo;
  ```

  has 2 bytes of implicit padding after **baz**, so that the entire structure is a multiple of 4 bytes (the requirement of bar).
- Bitfields generally require 4 byte alignment.

These rules are enforced to varying degrees on different hardware platforms. To ensure portability, always declare padding explicitly:

```
struct
{
    int8   bar;
    uint8  _pad1;
    uint16 _pad2;
    int32  baz;
} foo;
```

The alignment requirements of libeditor's basic types are as follows:

int8	1
uint8	1
int16	2
uint16	2
int32	4
uint32	4
float32	4
float64	8

`PointDescription`
4

`MunitionQuantity`
4

`ObjectIDType`
2

1.3 Editor Definition

An editor is defined by a series of *editables*. Each editable has a name, a type, a storage location (which is a reference back to a name listed in the **struct** part of the data file), and other configuration data. For example, the editor definition for a text object editor might look like this:

```
(editor ("Location" PLACE location)
        ("Color" CHOOSE_ONE color HIDE EDT_OVERLAY_COLORS)
        ("Overlay" OVERLAY overlay)
        ("Text" STRING text 5 length)
        )
```

There are currently 16 different types of editables, which are described in the following sections.

Note that in the usage syntax used below, [] indicates an optional field, | indicates a choice between two or more values, and {} indicates a list of values.

1.3.1 ALTITUDE

Usage

```
(<name> ALTITUDE <storage>)
```

Internal Representation

Meters.

Description

An altitude. The user can specify distance in meters, kilometers, feet, miles, or nautical miles.

1.3.2 ANGLE

Usage

```
(<name> ANGLE <storage> [<ccw bound> <cw bound>])
```

Internal Representation

Zero refers to North, and the value increases in a counter-clockwise direction. The units are either:

- BAMs, if stored in an integer type
- Radians, if stored in a floating point type

Description

An angle. User has choice of specifying in degrees, mils, or compass units. If specified, the counter-clockwise bound (<ccw bound>) and clockwise bound (<cw bound>), which should be an integer in degrees, are represented on the display but not enforced.

1.3.3 CHOOSE_ONE

Usage

```
(<name> CHOOSE_ONE <storage> SHOW|HIDE
        { (<choice name> <choice value> [<color name>]) }
        )
```

Internal Representation

Integer or floating point value of current choice.

Description

A choice of one item from a list of choices. The editable can be configured such that all choices are presented (SHOW), or with only the current choice shown (HIDE). If a color name is given, that color will be used for the background color of the choice (this is used in the pre-defined macro EDT_OVERLAY_COLORS for a list of standard overlay colors).

Note that it is possible to use CHOOSE_ONE to specify a choice from a number of libreader symbols, as opposed to integer or floating-point values. To do this, the structure definition in the program would be specified as:

```
struct foo
{
  ...
  char *symbol;
  ...
}
```

and the structure definition in the editor definition file would be specified as:

```
(struct
  ...
  (symbol uint32)
  ...
)
```

Then, strings may be specified for <choice value>, (as well as in initialization section of the editor definition file), as follows:

```
(editor
   ...
   ("Symbol" CHOOSE_ONE symbol HIDE
             ("Label1"  "symbol-value1")
             ("Label2"  "symbol-value2")
             ...
   )
)
```

The CHOOSE_ONE editable type also will correctly handle the case where the target structure type is an array of characters, and the value is a string:

```
(struct
  ...
  (symbol uint8 30)
  ...
)
...
(initial
  ...
  (symbol CONSTANT "symbol-value1")
  ...
)
```

1.3.4 CHOOSE_SOME

Usage

```
(<name> CHOOSE_SOME <value storage> <length storage> SHOW|HIDE
        { (<choice name> <choice value> [<color name>]) }
        )
```

Internal Representation

Array of integer or floating point values which were selected, and the number of elements of that array which are used.

Description

A choice of several items from a list of choices. The editable can be configured such that all choices are presented (`SHOW`), or with only the current choice shown (`HIDE`). If a color name is given, that color will be used for the background color of the choice. The values of all the selected values are placed in successive position in the **`<value storage>`** array. The number of items chosen is placed in the **`<length storage>`**.

Note that `CHOOSE_SOME` can be used to choose values from a number of libreader symbols, as in `CHOOSE_ONE` (see Section 1.3.3 [CHOOSE'ONE], page 5).

1.3.5 DATE

Usage

```
(<name> DATE <storage>)
```

Internal Representation

Seconds since 1970 ("unix" time).

Description

A date. The user specifies the month, day, and year.

1.3.6 DRAW_AREA

Usage

```
(<name> DRAW_AREA <storage>)
```

Internal Representation

An X widget

Description

A drawing area widget is returned in the storage area.

1.3.7 DISTANCE

Usage

```
(<name> DISTANCE <storage>)
```

Internal Representation

Meters.

Description

A distance. The user can specify distance in meters, kilometers, feet, miles, or nautical miles. It can also be specified by dragging on the map.

1.3.8 FUEL

Usage

```
(<name> FUEL <storage> [<requisite>])
```

Internal Representation

Liters.

Description

A quantity of fuel. The user can specify liters, gallons, or pounds. If a `requisite` is specified, the Fuel display will only appear if that structure member has a value.

1.3.9 LABEL

Usage

```
(<name> LABEL <storage>)
```

Internal Representation

A `NULL`-terminated string.

Description

An output-only string. This provides a facility for editors to provide descriptive text to the user.

1.3.10 LINE

Usage

```
(<name> LINE <value storage> <length storage>)
```

Internal Representation

Array of `PointDescriptions`.

Description

A multi-segment line. The user clicks out a line on the map, and can edit it in a variety of ways. The points are placed in successive slots of the `<value storage>` array (which must be declared type `PointDescription`). The number of points is placed in the `<length storage>`.

1.3.11 MUNITIONS

Usage

```
(<name> MUNITIONS <storage> [<requisite>])
```

Internal Representation

Array of `MunitionQuantitiess`.

Description

A list of munitions. The user may change the value associated with each munition, but not the type. A `U` is entered for unlimited supplies. When a `U` is entered, the internal representation becomes the negative of the current value. The last current value becomes the amount of the munition, but the negative signifies it is never decremented. The munitions are placed in successive slots of the `<storage>` array (which must be declared type `MunitionQuantity`). Unused slots use the invalid munition code 0. If a `requisite` is specified, the Munition display will only appear if that structure member has a value.

1.3.12 OBJECT

Usage

```
(<name> OBJECT <storage> [ONESHOT|NOCANCEL] [POINTSTYLE <style>]
 {objectClass...})
```

Internal Representation

ObjectID.

Description

A persistent object. The user may select any object on the map which is in one of the specified classes. If no object is chosen, the value 0/0/0 will be stored. If `ONESHOT` is specified, the value will be set back to 0/0/0 immediately after the render function is called (this allows an editor to pick many objects, which the application then keeps track of). If either `ONESHOT` or `NOCANCEL` is specified, no `Cancel Choice` button will be provided. If `POINTSTYLE` is specified, then the next value should be the type of point which is created by the system in response to a user map click (provided `objectClassPoint` appears in the list of classes. The `<storage>` must be an array of three `int16s`.

1.3.13 OVERLAY

Usage

```
(<name> OVERLAY <storage>)
```

Internal Representation

ObjectID.

Description

An overlay. The user may select an existing overlay, or create a new one. Attributes of the overlay may also be edited. If no overlays exist when the editor comes up, one is created. The **<storage>** must be an array of three **int16s**.

1.3.14 OVERLAY_DISPLAY

Usage

```
(<name> OVERLAY_DISPLAY <value storage> <length storage> SHOW|HIDE)
```

Internal Representation

Array of ObjectIDs of overlays that are selected, and the number of elements of that array which are used.

Description

A choice of several overlays selected from a list of overlays. The editable can be configured such that all choices are presented (SHOW), or with only the current choice shown (HIDE). The objectIDs of all the selected overlays are placed in successive positions in the **<value storage>** array. The **<value storage>** array must be declared as an array of ObjectIDs. The number of items chosen is placed in **<length storage>**.

1.3.15 PLACE

Usage

```
(<name> PLACE <storage>)
```

Internal Representation

Two-dimensional TCC location (either two integer, or two floating point numbers).

Description

A place. The user may type a location in X/Y, Latitude/Longitude, or UTM coordinates, or may select a location on the map. The **<storage>** must be an array of at least two elements.

1.3.16 SCALE

Usage

```
(<name> SCALE <storage> [SHOW|HIDE] [<min label> <max label>
        [<min val> <max val> [<units>]]])
```

Internal Representation

Number.

Description

A number. The editable can be configured such that the current value is shown numerically and graphically (**SHOW**), or only graphically (**HIDE**). The user can adjust the value by moving an indicator, or (if **SHOW**) by typing a value (much like the **ANGLE** widget).

The minimum and maximum ends of the scale are labeled with the **<min label>** and **<max label>** strings, if specified. The scale defaults to a range of 0.0 to 1.0, unless a different **<min val>** and **<max val>** are specified (these should be floating point numbers). Finally, for scales configured to **SHOW** their value, if a string is specified for **<units>** (such as "%" or "**deg/sec**"), this will be displayed to the right of the value.

1.3.17 SORT

Usage

```
(<name> SORT <value storage> <length storage> [<omit storage>]
        { (<choice name> <choice value>) }
        )
```

Internal Representation

Array of integer or floating point values in sorted order, the number of elements of that array which are used, and (optionally) the index of the first element which is to be omitted.

Description

A sorted list. The user is presented with a list of choices and may move them up or down relative to one another. If an **<omit storage>** is provided, an *omit* line will be presented which may be exploited by the user to indicate that some choices are to be omitted (whatever that might mean to an application).

1.3.18 SPEED

Usage

```
(<name> SPEED <storage>)
```

Internal Representation

Meters/Second.

Description

A speed. The user may specify a value in meters/second, km/hour, feet/second, miles/hour, knots or mach.

1.3.19 STEALTH

Usage

```
(<name> STEALTH <storage>)
```

Internal Representation

Vehicle ID.

Description

A stealth or stealth preview. The user may select a stealth from the map. If no object is chosen, the value 0 will be stored. The `<storage>` must be an `int32`.

1.3.20 STRING

Usage

```
(<name> STRING <value storage> <lines> [<length storage>])
```

Internal Representation

A `NULL`-terminated string.

Description

A string. The user may type up to the number of characters which will fit in the passed `<value storage>` (which must be an array), leaving at least one element for `NULL` termination. The `<lines>` attribute indicates how many lines should be presented for text entry (multi-line text windows are scrollable). If specified, the `<length storage>` will be filled with the length of the string, including the `NULL` terminator.

1.3.21 TIME

Usage

```
(<name> TIME <storage>)
```

Internal Representation

Seconds.

Description

An absolute or relative time. The user may specify a time with up to second resolution. It is up to the application to decide whether to treat this time as a relative or absolute quantity.

1.3.22 TOGGLE

Usage

```
(<name> TOGGLE <storage>)
```

Internal Representation

0 or -1

Description

A True or False value. The user is given a toggle button which can be set on or off.

1.3.23 VECTOR

Usage

```
(<name> VECTOR <storage>)
```

Internal Representation

Two two-dimensional TCC locations (either four integers, or four floating point numbers).

Description

A map click-and-drag. The user clicks on the map for the first location and drags out to the second location. This is provided to facilitate building tools which use such gestures as their input.

1.3.24 VEHICLE

Usage

```
(<name> VEHICLE <storage>)
```

Internal Representation

Vehicle ID. If storage is an array of three `int16s`, the full 48 `VehicleID` will be stored, otherwise the 32 bit hashed id will be used.

Description

A vehicle. The user may select a vehicle from the map. If no object is chosen, the value 0 will be stored. The **<storage>** must be an **int32**.

1.4 Initialization Rules

Each field in the **struct** part of the editor *must* be initialized. For example:

```
(initial (foo CONSTANT 0)
         (bar FORCE "You must provide a bar")
         (baz REFERENCE bar)
         )
```

The initialization method must be specified as one of the following:

CONSTANT **(<storage> CONSTANT {<values>})**
A constant value.

FUNCTION **(<storage> FUNCTION <function>)**
The system initializes the value by calling a function. The function names currently recognized are:

current_date
Gets the current date. Only may be used with **DATE** editable.

current_time
Gets the current time. Only may be used with **TIME** editable.

FORCE **(<storage> FORCE <help message>)**
Forces the user to provide a value. The help message is displayed at the bottom of the screen in a distracting manner.

REFERENCE
(<storage> REFERENCE <storage>)
References another storage location. If that storage location is initialized with **FORCE**, the value will be copied *after* the user has give the referent value.

1.5 Rendering Information

The last part of an editor definition is the render list. It specifies the names of storage locations which, when changed, should cause the object to be redrawn (the application provides a drawing function to libeditor at create time). For example:

```
    (render foo bar)
```

In addition, the application may use the following reserved words in the render list:

APPLY Indicates that an 'Apply' button should be provided which the user can click on to trigger a redraw.

REVERT Indicates that an 'Revert' button should be provided which the user can click on to revert to initial values, and trigger a redraw.

EXPOSE Indicates that the application needs to redraw when the map is refreshed or exposed (this would be true of anything which drew directly on the map widget).

NOINIT Indicates that the editor should only be initialized once. Thereafter, the editor will resume with exactly the configuration it was left with when last exited. This is useful for editors which always operate on a single set of data.

2 Usage

The software library 'libeditor.a' should be built and installed in the directory '/common/lib/'. You will also need the header file 'libeditor.h' which should be installed in the directory '/common/include/libinc/'. If these files are not installed, you need to do a 'make' in the libeditor source directory. If these files are already built, you can skip the section on building libeditor.

2.1 Building Libeditor

The libeditor source files are found in the directory '/common/libsrc/libeditor'. 'RCS' format versions of the files can be found in '/nfs/common_src/libsrc/libeditor'.

If the directory 'common/libsrc/libeditor' does not exist on your machine, you should use the 'genbuild' command to update the common directory hierarchy.

To build and install the library, do the following:

```
# cd common/libsrc/libeditor
# co RCS/*,v
# make install
```

This should compile the library 'libeditor.a' and install it and the header file 'libeditor.h' in the standard directories. If any errors occur during compilation, you may need to adjust the source code or 'Makefile' for the platform on which you are compiling. libeditor should compile without errors on the following platforms:

- Mips
- SGI Indigo
- Sun Sparc

2.2 Linking with Libeditor

Libeditor can be linked into an application program with the following link time flags: 'ld [source .o files] -L/common/lib -leditor [many other ModSAF libraries]'. If your compiler does not support '-L' syntax, you can use the archive explicitly: 'ld [source .o files] /common/lib/libedi

Libeditor depends on libcallback, libcoordinates, libpo, libquad, libreader, libsafgui, libsensitive, and libtactmap.

2.3 Examples

The test program, '`test.c`', and its data file, '`test.rdr`', give a complete example of how to define editors. See those files for example usage.

3 Functions

The following sections describe each function provided by libeditor, including the format and meaning of its arguments, and the meaning of its return values (if any).

3.1 edt_init

```
void edt_init(directory, reader_flags)
    char  *directory;
    uint32 reader_flags;
```

'directory'

Specifies directory where configuration file can be found

'reader_flags'

Specifies reader options (see section 'reader_read' in LibReader Programmer's Manual)

edt_init initializes libeditor, causing it to read its data file ('editor.rdr') from the passed directory. The return value is zero if the read succeeds, or one of the libreader return values (READER_READ_ERROR, READER_FILE_NOT_FOUND) if it fails.

3.2 edt_set_sensitive_class

```
void edt_set_sensitive_class(gui, type, class)
    SGUI_PTR           gui;
    EDT_SENSITIVE_TYPE type;
    SNSTVE_CLASS       *class;
```

'gui' Specifies the GUI

'type' Specifies which sensitive class is being specified

'class' Specifies the class which represents that type for this GUI

edt_set_sensitive_class sets the class structure used by *all* editors to interact with a class of senstitive objects. This class should be initialized by the caller with SNSTVE_INIT_CLASS (see section 'Class Definition' in LibSensitive Programmer's Manual).

Since many different libraries define objects which editors are interested in, and since most of those libraries will themselves depend on libeditor, it makes the most sense to have this information set globally, from the top-down. In cases where no sensitive class information has been given for a particular class, the editors will merely not support map interaction with those objects.

The type should be one of the following:

Terrain Classes
EDT_ROADS

Persistent object classes
(Note that user_data should point to ALLOCd ObjectID structure.)
EDT_POINT, EDT_LINE, EDT_SECTOR, EDT_TEXT, EDT_UNIT, EDT_TASK, EDT_TASKFRAME

Simulation classes
(Note that user_data should be vehicle ID)
EDT_VEHICLE, EDT_STEALTH

3.3 edt_offer_default_object

```
void edt_offer_default_object(gui, obj_id)
    SGUI_PTR  gui;
    ObjectID *obj_id;
```

'gui' Specifies the GUI

'obj_id' Specifies the object ID

edt_offer_default_object offers a default object to the next editor which is resumed. This object will be accepted if the editor is forcing the selection of an object, and the offered object is of a type acceptable for the OBJECT editable.

3.4 edt_create

```
EDT_EDITOR_PTR edt_create(definition, sizeof_structure,
                          render_fcn, render_arg,
                          exit_fcn, exit_arg, leave_mode,
                          gui, tactmap, tcc, map_erase_gc,
                          sensitive, refresh_event, db)
    READER_UNION        *definition;
```

```
        uint32              sizeof_structure;
        EDT_RENDER_FUNCTION render_fcn;
        ADDRESS             render_arg;
        EDT_EXIT_FUNCTION   exit_fcn;
        ADDRESS             exit_arg;
        int32               leave_mode;
        SGUI_PTR            gui;
        TACTMAP_PTR         tactmap;
        COORD_TCC_PTR       tcc;
        GC                  map_erase_gc;
        SNSTVE_WINDOW_PTR   sensitive;
        CALLBACK_EVENT_PTR  refresh_event;
        PO_DATABASE        *db;
```

'**definition**'
Specifies the definition of the editor in libreader format

'**sizeof_structure**'
Specifies the size of the data structure which is edited

'**render_fcn, render_arg**'
Specifies a function (and argument) which is called to render the data being edited (may be NULL)

'**exit_fcn, exit_arg**'
Specifies a function (and argument) which is called when the editor is exited (may be NULL)

'**leave_mode**'
Specifies whether the current SGUI mode whould be exited when the editor is exited

'**gui**' Specifies the GUI

'**tactmap**' Specifies the tactical map

'**tcc**' Specifies the mapping coordinate system

'**map_erase_gc**'
Specifies the GC used to erase things from the map (provided by libtactmap)

'**sensitive**'
Specifies the sensitive window

'**refresh_event**'
Specifies the event which fires when the map is refreshed

'**db**' Specifies the PO database

edt_create creates an editor. The design of the editor is specified in the passed definition. The software ensures at create time that the data described in the definition file correlates with the intended structure by comparing the sizes. The editor will load up with the initial values specified in the data file, but will remain unmanaged until started by **edt_state**. The **leave_mode** flag indicates whether the editor should exit the passed mode when the user clicks 'done' or 'abort'.

The **render** function should be declared as follows:

```
void render(editor, transient, old_data, new_data, arg)
    EDT_EDITOR_PTR editor;
    int32          transient;
    ADDRESS        old_data;
    ADDRESS        new_data;
    ADDRESS        arg;
```

The **transient** flag indicates whether the rendering is the result of a transient operation (such as draging the mouse). Libeditor guarantees that after a transient operation is finished, render will be called once more with the **transient** flag set to FALSE..

The **old_data** and **new_data** are old and new versions of the data being editing. The old version should be used to erase the existing rendition, and the new version should be used to draw.

The **arg** is whatever was passed to **edt_create** as the **render_arg**.

The **exit** function should be declared as follows:

```
void exit_function(editor, data, arg, status)
    EDT_EDITOR_PTR  editor;
    ADDRESS         data;
    ADDRESS         arg;
    EDT_EXIT_STATUS status;
```

The **data** is the final version of the data which was edited. If the user clicked 'Abort', this will be the same as the initial version.

The **arg** is whatever was passed to **edt_create** as the **exit_arg**.

The **status** is one of the following:

EDT_DONE The user clicked 'Done' or the user clicked on the arrow icon and no forced choices were outstanding.

EDT_ABORT

The user clicked 'Abort' or the user clicked on the arrow icon and forced choices were outstanding.

3.5 edt_create_defaults_editor

```
EDT_EDITOR_PTR edt_create_defaults_editor(gui, exit_fcn, exit_arg)
    SGUI_PTR          gui;
    EDT_EXIT_FUNCTION exit_fcn;
    ADDRESS           exit_arg;
```

'gui' Specifies the GUI

'exit_fcn, exit_arg'
Specifies the function to call at exit

edt_create_defaults_editor creates the libeditor user preferences (defaults) editor. This editor allows customization of default units for many editable types.

3.6 edt_create_profile_menu

```
void edt_create_profile_menu(gui, dialog_parent, dir)
    SGUI_PTR gui;
    Widget   dialog_parent;
    char     dir[];
```

'gui' Specifies the GUI

'dialog_parent'
Specifies the widget which should parent libxfile dialogs

'dir' Specifies the directory where profiles are stored

edt_create_profile_menu create the user profile menu, which allows the user to save/load the contents of the user preferences (defaults) editor.

3.7 edt_load

```
void edt_load(editor, data, size)
    EDT_EDITOR_PTR editor;
```

```
        ADDRESS         data;
        uint32          size;
```

'editor' Specifies the editor

'data' Specifies the data

'size' Specifies the size of the data

edt_load loads the passed data into the editor. If you pass NULL for the data, it reloads the editor with the initial values.

3.8 edt_load_reader

```
    void edt_load_reader(editor, count, data)
        EDT_EDITOR_PTR editor;
        int32          count;
        READER_UNION  *data;
```

'editor' Specifies the editor

'count' Specifies the number of initialization parameters (typically read_data.array[0].integer - 2)

'data' Specifies the initialization parameters (typically &read_data.array[2])

edt_load_reader loads the passed libreader format data into the editor for subsequent initialization. This initialization overrides the initialization parameters originally given the editor. Fields which are not specified retain their original initialization methods and values.

3.9 edt_set_field

```
    void edt_set_field(editor, member_offset, addr, size)
        EDT_EDITOR_PTR editor;
        uint32         member_offset;
        ADDRESS        addr;
        uint32         size;
```

'editor' Specifies the editor

'member_offset'
Specifies the byte-offset of the field which is to be changed

'addr' Specifies the address of the new value

'size' Specifies the size of the new value

edt_set_field initializes a single field of a running editor. as though the user had specified that field. The format of the provided value must the be same as the storage format of that field in the edited data structure.

3.10 edt_apply_offset

```
void edt_apply_offset(editor, member_offset, index, dx, dy)
    EDT_EDITOR_PTR editor;
    uint32         member_offset;
    int32          index;
    int32          dx, dy;
```

'editor' Specifies the editor

'member_offset'
Specifies the byte-offset of the field which is to be changed

'index' If changing a line, specifies the index of the point to be changed

'dx, dy' Specifies the amount of change (in pixels)

edt_apply_offset incorporates the user's initial drag into the currently edited graphic, as though the drag had occurred in the context of the editor. The **member_offset** identifies which member of the structure the offset should be applied to (if not a PLACE or LINE editable, nothing will happen). The **index** field specifies (for LINEs only) which vertex should be offset; -1 indicates movement of the whole object.

3.11 edt_state

```
void edt_state(editor, mode, state)
    EDT_EDITOR_PTR  editor;
    SGUI_MODE_PTR   mode;
    SGUI_MODE_STATE state;
```

'editor' Specifies the editor

'mode' Specifies the mode associated with the state (used for display of help messages) '**state**' Specifies the new state

edt_state sets the state of an editor to one of the libSAFGUI states (ACTIVE, SUSPENDED, RESUMED, INACTIVE).

See section 'sgui_add_mode' in LibSAFGUI Programmer's Manual.

3.12 edt_query

```
int32 edt_query(editor, value)
    EDT_EDITOR_PTR editor;
    ADDRESS        value;
```

'**editor**' Specifies the editor

'**value**' Returns the editor values

edt_query gets the current values of the object being edited. The return value is 1 if any forced choices are still unresolved, 0 otherwise. This may be called when the editor is inactive.

3.13 edt_name

```
char *edt_name(editor)
    EDT_EDITOR_PTR editor;
```

'**editor**' Specifies the editor

edt_name returns the name of the editor.

3.14 edt_get_references

```
void edt_get_references(editor, data, refcount, references)
    EDT_EDITOR_PTR editor;
    ADDRESS        data;
    uint8         *refcount;
    ObjectID       references[];
```

'editor' Specifies the editor

'data' Specifies the data which contains the references

'refcount'
Returns the number of ObjectID references made by the editor

'references'
Returns the ObjectID value for each reference made in the passed data

edt_get_references finds all the ObjectID's referenced in the passed data structure and copies them into the passed array. This can be used to fill in the **references** list in a Task Class object.

3.15 edt_format

```
uint32 edt_format(gui, output, format, args...)
    ADDRESS gui;
    char    output[];
    char    format[];
    args...
```

'gui' Specifies the GUI (declared as an ADDRESS so libraries needn't specifically depend upon libSAFGUI)

'output' Returns the formatted string

'format' Specifies the output format

'args...' Specifies format arguments

edt_format is like **sprintf**, in that it generates a formatted output based upon a format string, and a set of arguments. Whereas **sprintf** uses **%** to specify output tokens, **edt_format** uses **#**. Values are output in a manner appropriate given the user's GUI preferences (UTMs vs Lat/Long, for example). The return value is the length of the formatted string. The recognized tokens are as follows:

Altitude

Token	**#h**
Argument	**float64**
Units	Meters
Example	"28000 feet"

Angle

Token	**#a**

Argument **float64**

Units Radians

Example **"35 degrees"**

Distance

Type **#d**

Argument **float64**

Units Meters

Example **"150 feet"**

Fuel

Type **#f**

Argument **float64**

Units Liters

Example **"2000 lbs"**

Location

Type **#l**

Argument **COORD_TCC_PTR float64 float64**

Units Meters (X, Y)

Example **"16SES45005500"**

Object

Type **#o**

Argument **PO_DATABASE * ObjectID ***

Example **"Task Frame 'Fly On Route'"**

Speed

Type **#s**

Argument **float64**

Units Meters/Second

Example **"mach 1.1"**

Vehicle

Type **#v**

Argument **VehicleID ***

Example **"A11 (15)"**

Also, the following tokens are provided to avoid the need to use both **edt_format** and **sprintf**:

Float

	Token	#G
	Argument	float64
	Same as	%g
Integer		
	Token	#D
	Argument	int32
	Same as	%d
String		
	Token	#S
	Argument	char *
	Same as	%s

3.16 edt_pvd_defaults

```
void edt_pvd_defaults(gui, result)
    SGUI_PTR           gui;
    EDT_PVD_DEFAULTS *result;
```

'gui' Specifies the GUI

'result' Returns the settings

edt_pvd_defaults returns user preferences (defaults) editor choices related to PVD operation.

3.17 edt_color_choice

```
void edt_color_choice(editor, pretty_name, value, color)
    EDT_EDITOR_PTR editor;
    char          *pretty_name;
    int32          value;
    Pixel          color;
```

'editor' Specifies the editor

'pretty_name'
Specifies a libreader symbol, which identifies the editable

'value' Specifies the value associated with the choice to be colored 'color' Specifies the desired color.

edt_color_choice sets the foreground color of a button in a CHOOSE_ONE or CHOOSE_SOME editable. The pretty_name should be a libreader symbol which corresponds to the name of the editable in the editor section of the definition file (see Section 1.3 [Editor Definition], page 4).

3.18 edt_editor_empty

```
int32 edt_editor_empty(editor)
    EDT_EDITOR_PTR  editor;
```

'editor' Specifies the editor

edt_editor_empty returns TRUE it the editor has no editables.

4 Events

The following sections describe each event provided by libeditor.

4.1 edt_defaults_callback

```
CALLBACK_EVENT_PTR edt_defaults_callback;
```

The **edt_defaults_callback** event fires in response to the user changing the settings of the defaults editor.

The handler should be prototyped as follows:

```
void handler(gui)
    SGUI_PTR gui;
```

See Section 3.16 [edt˙pvd˙defaults], page 29.

5 X Resource Definitions

Many attributes of the fields which make up an editor are specified via the X resource database. These can be overridden for individual fields in individual editors, to customize the interface beyond its default appearance.

In general, a resource override should be formatted as follows:

```
*.SAFGUI.*.Editor.*.Editor Name.*.Field Name.*.[Component Name].Attribute: Value
```

For example, to change the *label* on the *object* button for the *unit* field in the *mission assignment* editor:

```
*.SAFGUI.*.Editor.*.Mission Assignment.*.Unit.*.ObjectButton.labelString: \
Select Unit from Map
```

To change the help string associated with this editable:

```
*.SAFGUI.*.Editor.*.Mission Assignment.*.Unit.*.Help: \
Select a unit from the map (valid choices are hot).
```

As these examples show, the component name of the field is only needed when the resource would otherwise be ambiguous (it is needed for the first example because both the `ObjectButton` and the `Cancel Choice` members have a `labelString` attribute).

6 Defining New Types

One design goal of libEditor is that the user is never confronted with raw data types (such as "a number"). It is much better if the input is requested in a way specifically designed for the parameter being specified (such as "a speed"). It seems likely that as the program expands, we will find continue to find more types of quantities that can be edited. This chapter give step by step instructions for adding a new type.

Note that it if you choose to use a built-in Motif widget class (such as xmToggleButtonWidgetClass), you must call the function **edt_focus_fix** on the created widget. This is to correct a problem with Motif focus management (clicking in a widget which already has the focus pushes the focus into an adjacent widget). Custom widgets (such as the edt_dateWidgetClass) do not need this function called on them at create time, because they explicitly prevent the problem in their implementation using a construct like:

```
if (!_XmFocusIsHere((Widget)w))
  XmProcessTraversal((Widget)w, XmTRAVERSE_CURRENT);
```

in their focus-accepting methods.

The procedure described is relatively simple, since it does not require map input, a unique input widget (it is assembled from text & toggles), or any special map display methods. To create a widget with these features, first follow the instructions below, then find another editable (such as PLACE or ANGLE) to use as an example for the other features.

1. Name the type. Choose a name which is short and descriptive.
2. Add the necessary types and prototypes to '**libedt_local.h**'. Each widget type has its own section in this file, add the new one just before the comment:

   ```
   /* ADD NEW TYPES HERE */
   ```

3. Define any necessary enumerated types and prototype the **create** and **set_value** functions.
4. If the editable supports different units, add them to the **EDT_DEFAULTS** data structure.
5. Add the name of the new type to the **EDT_CLASS** enumerated type (put it in alphabetical order).
6. If the editable will support any special configuration parameters (display options, etc.), create a **EDT_<class>_CONFIG** structure to hold them, just prior to the comment:

   ```
   /* ADD NEW CONFIGS HERE */
   ```

   Then add that to the **config** member of the **EDT_EDITABLE** structure.

7. Update the file '**editor.rdr**' to include the new units. Be sure to add the new field to the **struct**, **editor**, **initial** and **render** lists.
8. Update the functions **save_profile** and **load_profile** in '**edt_profile.c**' to include the new units.
9. Next, add co‹e to '**edt_init.c**' to recognize and create the new type. Find the comment:

   ```
   /* ADD NEW SYMBOLS HERE */
   ```

   and add a variable to the list (in alphabetical order) for the new type. Add the initialization for that type, just below. Finally, add the new type to the **if-else-if** chain a little farther down.
10. If the editable requires a certain type of storage (must be an integer, for example), add a check for that before the comment:

    ```
    /* ADD NEW CHECKS HERE */
    ```
11. If the editable supports special configuration, add the necessary parsing before the comment:

    ```
    /* ADD NEW CONFIG PARSING HERE */
    ```
12. Add a **case** to the **switch** in the function **create_widget** to create the new widget type. Put it in alphabetical order.
13. Next, add code to '**edt_state.c**' to initialize the new type. Find the comment:

    ```
    /* ADD NEW INITIALIZATION HERE */
    ```

    and add a **case** to the **switch** to initialize the new type. Note that at this point the values have already been converted into every necessary format, so just use the one which is appropriate for the new editable.
14. Create a file in which to define the new widget '**edt_<type>.c**'. This is fairly easy if you start with a similar widget definition file as a prototype, and use case-insenstive string replace to customize it.
15. Add necessary resources for this new widget to '**editor.xrdb**'. This should include a help string, and probably some layout information (again, find a similar widget and copy it).
16. Add a description of the new editable type to the *Editor Definition* section of '**libeditor.texinfo**'.
17. Add the new file to the '**Makefile**'.
18. Test the new editable by adding it to the test editor defined in '**test.rdr**' and '**test.c**'.

ADST-TR-W003268

LibEntity

Table of Contents

1 Overview

Libentity provides a uniform interface to all network entities represented within SAF. Entity is a sub-class of each vehicle.

In addition to the bookkeeping functions to create, destroy, activate, etc., libentity provides a collection of get and set functions for each of the entity state variables. This functions act as a lazy evaluation buffer, which prevents conversions to or from network representation until absolutely necessary, and then saves those converted values until they once again become out of date.

Note that libentity currently only supports getting DIS style data from an entity. In the future, libentity may be modified to accept the setting of DIS style data for local vehicles, causing SIMNET style data to be derived from the DIS data.

The entity sub-class of the vehicle is also responsible for maintaining that vehicle's location in the position-based table. For local vehicles, this update occurs whenever the position is changed. For remotes, this update occurs either (1) when a packet is received which modifies the remote vehicle's position, or (2) when the RVA'd position of the vehicle exceeds a tolerable error threshold from the position represented in the table.

Lazy evaluation of remote vehicle RVA is implemented to guarantee the following:

- Each remote vehicle is RVA'd no more often than once every full loop through the scheduler.
- Remotes are not RVA'd unless get_position is explicitly called, or RVA is necessary to update the position-based vehicle table.

The parameters for an entity are used primarily for network interactions. They are as follows:

```
(SM_Entity (length_threshold <real percent>)
           (width_threshold <real percent>)
           (height_threshold <real percent>)
           (rotation_threshold <real degrees>)
           (turret_threshold <real azimuth degrees>)
           (gun_threshold <real elevation degrees>)
           (vehicle_class <int class>)
           (guises <int primary guise>  ;; What this thing really is
                   <int secondary guise> ;; A similary thing with an
                                         ;;  opposite country code
           (send_dis_deactivate <true | false>)
            )
```

```
    )
```

The thresholds are used for dead reckoning, to indicate when a packet should be transmitted. They are typically 10% and 3 degrees. The `vehicle_class` should be either `vehicleClassSimple` or `vehicleClassTank`, depending upon whether the vehicle has a turret. Two guises must be provided – the primary guise, which should be accurate; and the secondary guise which should be a similar vehicle with a different country code (used in relative battle scheme battles). `send_dis_deactivate` is a boolean value indicating whether or not the entity should send a `DIS_DEACTIVATE_REQUEST` pdu when the entity deactivates (under DIS). This will typcially be `true` for all vehicles except missiles, since missile impacts provide an implicit deactivate under DIS.

2 Examples

To set an entity's position:

```
#include <libentity.h>
...
    float64 pos;
    ...
    ent_set_position(vehicle_id, pos);
```

To get an entity's position:

```
#include <libentity.h>
...
    float64 pos;
    ent_get_position(vehicle_id, pos);
```

To get an entity's position and velocity via libaccess:

```
#include <libaccess.h>
#include <libentity.h> /* To get key prototype */
#include <stdext.h> /* To get A_END */
...
    float64 pos[3];
    float64 vel[3];

    access_get(vehicle_id,
               ent_position, pos,
               ent_velocity, vel,
               A_END);
```

3 Functions

The following sections describe each function provided by libentity, including the format and meaning of its arguments, and the meaning of its return values (if any).

3.1 ent_init

```
void ent_init(packet_valve, tcc, protocol, use_atlas_variations)
    PV_VALVE_PTR  packet_valve;
    COORD_TCC_PTR tcc;
    int32         protocol;
    int32         use_atlas_variations;
```

'packet_valve'
: Specifies the packet_valve to use when sending packets.

'tcc'
: Specifies the terrain coordinate system

'protocol'
: Specifies protocol in use (0 for SIMNET, DIS_PROTOCOL_VERSION_* for DIS)

'use_atlas_variations'
: Specifies whether to use the Atlas Elektronik protocol variations when communicating with a DIS protocol. This impacts conversions of locations and velocities.

ent_init initializes libentity. Call this before calling any other libentity functions. The packet_valve is created with a call to pv_create_valve.

3.2 ent_set_minimum_pbt_error

```
void ent_set_minimum_pbt_error(error_threshold)
    float64 error_threshold;
```

'error_threshold'
: Specifies the maximum error allowed between a vehicle's actual location and that used in the position-based table.

ent_set_minimum_pbt_error reduces the allowable error (in meters) in the position-based table. Only those remote vehicles which exceed this error will be RVA'd and updated in the position based table. Calls which pass larger values than the current threshold are ignored. Until this error value is set, no RVA will be done on behalf of the position-based table (RVA will be done, however, for those vehicles which have their position queried).

3.3 ent_get_minimum_pbt_error

```
float64 ent_get_minimum_pbt_error()
```

ent_get_minimum_pbt_error returns the current error threshold being maintained by libentity for the position-based table.

See ⟨undefined⟩ [ent˙set˙minimum˙pbt˙error], page ⟨undefined⟩.

3.4 ent_set_battle_scheme

```
void ent_set_battle_scheme(battle_scheme)
    uint8 battle_scheme;
```

'**battle_scheme**'
Specifies the new battle scheme (**battleSchemeRelative** or **battleSchemeAbsolute**).

ent_set_battle_scheme sets the battle scheme used to generate vehicle guises for local vehicles. The **PARAMETRIC_DATA** used by an entity specifies two guises which are used to describe the vehicle, one which actually represents the vehicle, and a similar one on the other side. For example, a T72-M tank might be defined with the guises:

```
(guises vehicle_USSR_T72M vehicle_US_M1)

or,

(guises vehicle_USSR_T72M vehicle_Germany_LEO2)
```

The relationship of this data to what goes out on the network is a function of both the global battle scheme and the force of the given vehicle. The rules are as follows:

- For relative battle scheme, if the force is:

distinguished, other
The guises are used just as they appear in the **PARAMETRIC_DATA**.

observer A US or German object type is selected for both guises.

target A USSR object type is selected for both guises.

- For absolute battle scheme, if the force is:

distinguished, other
The first guise listed in the **PARAMETRIC_DATA** is used for both guises.

observer A US or German object type is selected for the distinguished guise, and a USSR object type is selected for the other.

target A USSR object type is selected for the distinguished guise, and a US or German object type is selected for the other.

3.5 ent_class_init

```
void ent_class_init(parent_class)
    CLASS_PTR parent_class;
```

'parent_class'
Specifies the parent class of entity (probably safobj_class).

ent_class_init creates a handle for attaching Entity class information to vehicles. The **parent_class** is one created with **class_declare_class**.

3.6 ent_create

```
void ent_create(vehicle_id, parms, deactivate_fcn)
    int32                       vehicle_id;
    ENTITY_PARAMETRIC_DATA *parms;
    void                     (*deactivate_fcn)(/* int32 vehicle_id */);
```

'vehicle_id'
Specifies the vehicle_id of the vehicle to be created.

'parms' Specifies parametric data for the entity.

'deactivate_fcn'
Specifies the function to call if the remote vehicle is deactivated.

ent_create creates the Entity class information for a vehicle and attaches it to the vehicle's libclass user data. All vehicle start inactive. Local vehicles become active (start broadcasting appearance packets) when ent_activate is called. Remote vehicles become active when their first appearance packet is received. The argument 'parms' is only necessary for local vehicles. You may pass NULL for this argument for remotes.

The deactivate_fcn is called when remote vehicles receive a deactivate request packet. It is up to the creator of the instance to act on the deactivate.

Libentity keeps a five minute history of vehicles which it has deactivated (for whatever reason). When **ent_create** is called with a **vehicle_id** which corresponds to a vehicle in this history, the new vehicle is automatically initialized with the values from the previous incarnation. The position is RVA'd forward to where the vehicle would be at the current time, had it not ceased to exist. This functionality is provided to simplify simulation handoff. In other cases, the caller would normally set all the fields of the entity appearance to override these default values.

This can be called for Any vehicle.

3.7 ent_destroy

```
void ent_destroy(vehicle_id)
    int32 vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID.

ent_destroy frees the Entity class information for a vehicle. For local active vehicles, a deactivate will automatically be sent.

This can be called for Any vehicle.

3.8 ent_tick

```
void ent_tick(vehicle_id)
    int32 vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID.

ent_tick for remotes, manages timeouts and RVA. For locals, sends current appearance of vehicle on network if warranted by RVA thresholding. This should be called no more often than once per 67ms.

This can be called for Any vehicle.

3.9 ent_packet_received

```
void ent_packet_received(vehicle_id, packet)
    int32      vehicle_id;
    PV_PACKET *packet;
```

'vehicle_id'
Specifies the vehicle ID.

'packet' Pointer to the packet received.

ent_packet_received calls this function when a new appearance packet or a deactivate packet is received off the network. Tbe packet must be allocated using pv_buffer_allocate.

Receipt of an appearance packet will cause the update of the position-based table component of the vehicle.

This can be called for Remote vehicles only.

3.10 ent_set_exercise_id

```
void ent_set_exercise_id(vehicle_id, exercise_id)
    int32 vehicle_id;
    uint8 exercise_id;
```

'vehicle_id'
Specifies the vehicle ID.

'exercise_id'
Specifies the exercise ID.

ent_set_exercise_id sets the exercise_id field of the entity. This may be called more than once (to change exercises, for example).

This can be called for Local vehicles only.

3.11 ent_activate

```
void ent_activate(vehicle_id)
    int32 vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID.

ent_activate allows appearance packets to be sent on the network for this vehicle.

This can be called for Local vehicles only.

3.12 ent_deactivate

```
void ent_deactivate(vehicle_id, reason)
    int32            vehicle_id;
    DeactivateReason reason;
```

'vehicle_id'
Specifies the vehicle ID.

ent_deactivate prohibits transmission of appearance packets for this vehicle. It will send a deactivate packet if the vehicle is not already inactive. The sent deactivate packet will be marked with the supplied **DeactivateReason**.

This can be called for Local vehicles only.

3.13 ent_active

```
int32 ent_active(vehicle_id)
    int32 vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID.

ent_active returns whether the vehicle is currently active (see ⟨undefined⟩ [ent`activate], page ⟨undefined⟩).

This can be called for Any vehicle.

3.14 ent_clear_thresh_stats

```
void ent_clear_thresh_stats()
```

ent_clear_thresh_stats resets all thresholding statistics to zero.

See ⟨undefined⟩ [ent`print`thresh`stats], page ⟨undefined⟩.

3.15 ent_print_thresh_stats

```
void ent_print_thresh_stats()
```

ent_print_thresh_stats prints RVA thresholding statistics.

3.16 ent_convert_location_to_dis

```
void ent_convert_location_to_dis(vehicle_id, internal_location, dis_location)
    int32       vehicle_id;
    float64     *internal_location;
    float64     *dis_location;
```

'**vehicle_id**'
Specifies the vehicle ID (or 0 if none available)

'**internal_location**'
Specifies a location in the internal format (TCC)

'**dis_location**'
Returns the same location in the DIS format (GCC or Atlas, depending upon how libentity was initialized)

ent_convert_location_to_dis converts the passed location from internal to the DIS format. This is provided as a convenience so libraries do not all have to hold on to the TCC.

3.17 ent_convert_velocity_to_dis

```
void ent_convert_velocity_to_dis(vehicle_id, internal_velocity,
                                 internal_location, dis_velocity)
    int32        vehicle_id;
    float64     *internal_velocity;
    float64     *internal_location;
    float32     *dis_velocity;
```

'**vehicle_id**'
Specifies the vehicle ID

'**internal_velocity**'
Specifies a velocity in the internal format (TCC)

'**internal_location**'
Specifies a location in the internal format (TCC) (pass a NULL pointer if none is known)

'**dis_velocity**'
Returns the same velocity in the DIS format (GCC or Atlas, depending upon how libentity was initialized)

ent_convert_velocity_to_dis converts the passed velocity from internal to the DIS format. This is provided as a convenience so libraries do not all have to hold on the transformation matrices. NOTE: the **dis_velocity** is returned as **float32 ***, since that is how it is stored in DIS packets.

3.18 ent_convert_location_from_dis

```
void ent_convert_location_from_dis(vehicle_id,
                                   dis_location, internal_location)
    int32       vehicle_id;
    float64    *dis_location;
    float64    *internal_location;
```

'vehicle_id'
Specifies the vehicle ID (or 0 if none available)

'dis_location'
Specifies a location in the DIS format (GCC or Atlas, depending upon how libentity was initialized)

'internal_location'
Returns the same location in the internal format (TCC)

ent_convert_location_from_dis converts the passed location from DIS to the internal format. This is provided as a convenience so libraries do not all have to hold on to the TCC.

3.19 ent_convert_velocity_from_dis

```
void ent_convert_velocity_from_dis(vehicle_id, dis_velocity,
                                   internal_velocity, internal_location)
    int32       vehicle_id;
    float32    *dis_velocity;
    float64    *internal_velocity;
    float64    *internal_location;
```

'vehicle_id'
Specifies the vehicle ID

'dis_velocity'
Specifies a velocity in the DIS format (GCC or Atlas, depending upon how libentity was initialized)

'internal_velocity'
Returns the same velocity in the internal format (TCC)

'internal_location'
Specifies a location in the internal format (TCC) (pass a NULL pointer if none known)

ent_convert_velocity_from_dis converts the passed velocity from DIS to the internal format. This is provided as a convenience so libraries do not all have to hold on the transformation matrices. NOTE: the **dis_velocity** is passed as `float32 *`, since that is how it is stored in DIS packets.

3.20 ent_format_location

```
char *ent_format_location(vehicle_id, x, y)
    int32   vehicle_id;
    float64 x, y;
```

'**vehicle_id**'
Specifies the vehicle ID.

'**x, y**' Specifies the location

ent_format_location converts the passed location to a character string suitable for transmission in a radio message. Currently this will be as a UTM string, but in the future the output format may become a parameter of the entity.

3.21 ent_set_vehicle_class

```
void ent_set_vehicle_class(vehicle_id, vehicle_class)
    int32        vehicle_id;
    VehicleClass vehicle_class;
```

'**vehicle_id**'
Specifies the vehicle ID.

'**vehicle_class**'
Specifies the new class (Static, Simple, Tank).

ent_set_vehicle_class is used to set the vehicle_class attribute of vehicle appearance. This attribute is not translated to network representations until absolutely necessary.

This can be called for Local vehicles only.

3.22 ent_set_force_id

```
void ent_set_force_id(vehicle_id, force_id)
    int32   vehicle_id;
    ForceID force_id;
```

'vehicle_id'
Specifies the vehicle ID.

'force_id'
Specifies the new force (distinguished, other, observer, target).

ent_set_force_id is used to set the force_id attribute of vehicle appearance. This attribute is not translated to network representations until absolutely necessary.

This can be called for Local vehicles only.

3.23 ent_set_guises

```
void ent_set_guises(vehicle_id, guises)
    int32          vehicle_id;
    VehicleGuises *guises;
```

'vehicle_id'
Specifies the vehicle ID.

'guises' Specifies the new guises.

ent_set_guises is used to set the guises attribute of vehicle appearance. This attribute is not translated to network representations until absolutely necessary.

This can be called for Local vehicles only.

3.24 ent_set_marking

```
void ent_set_marking(vehicle_id, marking)
    int32          vehicle_id;
    VehicleMarking *marking;
```

'vehicle_id'
Specifies the vehicle ID.

'marking' Specifies the new marking.

ent_set_marking is used to set the marking attribute of vehicle appearance. This attribute is not translated to network representations until absolutely necessary.

This can be called for Local vehicles only.

3.25 ent_set_position

```
void ent_set_position(vehicle_id, position)
    int32   vehicle_id;
    float64 position[3];
```

'vehicle_id'
Specifies the vehicle ID.

'position'
Specifies the new position.

ent_set_position is used to set the position attribute of vehicle appearance. This attribute is not translated to network representations until absolutely necessary.

This function also updates the position-based table component of the local vehicle.

This can be called for Local vehicles only.

3.26 ent_set_rotation

```
void ent_set_rotation(vehicle_id, rotation)
    int32   vehicle_id;
    float64 rotation[3][3];
```

'vehicle_id'
Specifies the vehicle ID.

'rotation'
Specifies the new rotation.

ent_set_rotation is used to set the rotation attribute of vehicle appearance. This attribute is not translated to network representations until absolutely necessary.

This can be called for Local vehicles only.

3.27 ent_set_orientation

```
void ent_set_orientation(vehicle_id, rotation, heading, pitch, roll)
    int32   vehicle_id;
    float64 rotation[3][3];
    float64 heading, pitch, roll;
```

'vehicle_id'
Specifies the vehicle ID.

'rotation'
Specifies the new rotation.

'heading'
'pitch'
'roll' Specifies the new orientation.

ent_set_orientation is used to set rotation and orientation attributes of vehicle appearance. This attribute is not translated to network representations until absolutely necessary.

It is assumed that the passed orientation angles and rotation matrix are equivalent. Note that although the rotation could be computed from the angles, or vice-versa, it is assumed that the caller has a good chance of knowing some sin, cos, atan, etc. which we would rather not recompute.

This can be called for Local vehicles only.

3.28 ent_set_appearance

```
void ent_set_appearance(vehicle_id, appearance)
    int32  vehicle_id;
    uint32 appearance;
```

'vehicle_id'
Specifies the vehicle ID.

'appearance'
Specifies the new appearance bits.

ent_set_appearance is used to set appearance attribute of vehicle appearance. This attribute is not translated to network representations until absolutely necessary.

This function performs **x = y**

This can be called for Local vehicles only.

3.29 ent_set_appearance_bits

```
void ent_set_appearance_bits(vehicle_id, appearance)
    int32  vehicle_id;
    uint32 appearance;
```

'**vehicle_id**'
Specifies the vehicle ID.

'**appearance**'
Specifies the bits to set.

ent_set_appearance_bits is used to set some bits in the appearance attribute of vehicle appearance. This attribute is not translated to network representations until absolutely necessary.

This function performs **x |= y**

This can be called for Local vehicles only.

3.30 ent_unset_appearance_bits

```
void ent_unset_appearance_bits(vehicle_id, appearance)
    int32  vehicle_id;
    uint32 appearance;
```

'**vehicle_id**'
Specifies the vehicle ID

'**appearance**'
Specifies the bits to clear.

ent_unset_appearance_bits is used to unset some bits of the appearance attribute of vehicle appearance. This attribute is not translated to network representations until absolutely necessary.

This function performs **x &= ~y**

This can be called for Local vehicles only.

3.31 ent_set_capabilities

```
void ent_set_capabilities(vehicle_id, capabilities)
    int32               vehicle_id;
    VehicleCapabilities *capabilities;
```

'**vehicle_id**'
Specifies the vehicle ID.

'**capabilities**'
Specifies the new capabilities.

ent_set_capabilities is used to set the capabilities attribute of vehicle appearance. This attribute is not translated to network representations until absolutely necessary.

This can be called for Local vehicles only.

3.32 ent_set_engine_speed

```
void ent_set_engine_speed(vehicle_id, speed)
    int32  vehicle_id;
    uint16 speed;
```

'**vehicle_id**'
Specifies the vehicle ID.

'**speed**' Specifies the new engine speed.

ent_set_engine_speed is used to set the engine_speed attribute of vehicle appearance. This attribute is not translated to network representations until absolutely necessary.

This can be called for Local vehicles only.

3.33 ent_set_velocity

```
void ent_set_velocity(vehicle_id, velocity)
    int32   vehicle_id;
    float64 velocity[3];
```

'vehicle_id'
Specifies the vehicle ID.

'velocity'
Specifies the new velocity.

ent_set_velocity is used to set the velocity attribute of vehicle appearance. This attribute is not translated to network representations until absolutely necessary.

This can be called for Local vehicles only.

3.34 ent_set_artic_value

```
void ent_set_artic_value(vehicle_id, artic_name, value)
    int32    vehicle_id;
    char     *artic_name;
    float64  value;
```

'vehicle_id'
Specifies the vehicle ID.

'artic_name'
Specifies the name of the articulation (a libreader symbol)

'value' Specifies the new value of the articulation

ent_set_artic_value is used to set the value of an articulation named **name**. The **value** is interpreted depending on the type of articulation being set. For instance, in the case of a turret (such as one called "**primary-turret**"), the value is interpreted as azimuth radians, with 0 being along the vehicle's **X** axis and rotating counterclockwise.

This can be called for Local vehicles only.

3.35 ent_set_artic_rate

```
void ent_set_artic_rate(vehicle_id, artic_name, value)
    int32    vehicle_id;
    char     *artic_name;
    float64  value;
```

'vehicle_id'
Specifies the vehicle ID.

'artic_name'
Specifies the name of the articulation (a libreader symbol)

'value' Specifies the new value of the articulation rate

ent_set_artic_rate is used to set the change value of an articulation named **name**. The **value** is interpreted depending on the type of articulation being set. For instance, in the case of a turret (such as one called "**primary-turret**"), the value is interpreted as azimuth radians per second, with 0 being along the vehicle's X axis and rotating counterclockwise.

This can be called for Local vehicles only.

3.36 ent_get_exercise_id

```
uint8 ent_get_exercise_id(vehicle_id)
    int32 vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID.

ent_get_exercise_id is used to get the exercise ID of a vehicle.

This can be called for Any vehicle.

3.37 ent_get_vehicle_class

```
VehicleClass ent_get_vehicle_class(vehicle_id)
    int32 vehicle_id;
```

'**vehicle_id**'

Specifies the vehicle ID.

ent_get_vehicle_class is used to get the vehicle_class attribute of vehicle appearance. When translations from network representation to internal representation are necessary, these will be done upon invocation of the get function, and saved for future calls.

This can be called for Any vehicle.

3.38 ent_get_force_id

```
ForceID ent_get_force_id(vehicle_id)
    int32 vehicle_id;
```

'**vehicle_id**'

Specifies the vehicle ID.

ent_get_force_id is used to get the force_id attribute of vehicle appearance. When translations from network representation to internal representation are necessary, these will be done upon invocation of the get function, and saved for future calls.

This can be called for Any vehicle.

3.39 ent_get_guises

```
void ent_get_guises(vehicle_id, guises)
    int32           vehicle_id;
    VehicleGuises   *guises;
```

'**vehicle_id**'

Specifies the vehicle ID.

'**guises**' Returns the guises

ent_get_guises is used to get the guises attribute of vehicle appearance. When translations from network representation to internal representation are necessary, these will be done upon invocation of the get function, and saved for future calls.

This can be called for Any vehicle.

3.40 ent_get_guise

```
ObjectType ent_get_guise(vehicle_id, viewing_force)
    int32   vehicle_id;
    ForceID viewing_force;
```

'vehicle_id'
Specifies the vehicle ID.

'viewing_force'
Specifies the force of the viewer.

ent_get_guise is used to get the guise attribute of vehicle appearance. When translations from network representation to internal representation are necessary, these will be done upon invocation of the get function, and saved for future calls.

Unlike **ent_get_guises** (see ⟨undefined⟩ [ent˙get˙guises], page ⟨undefined⟩), this function gets the appropriate guise relative to the passed viewing_force.

This can be called for Any vehicle.

3.41 ent_get_marking

```
void ent_get_marking(vehicle_id, marking)
    int32           vehicle_id;
    VehicleMarking *marking;
```

'vehicle_id'
Specifies the vehicle ID.

'marking' Returns the marking.

ent_get_marking is used to get the marking attribute of vehicle appearance. When translations from network representation to internal representation are necessary, these will be done upon invocation of the get function, and saved for future calls.

This can be called for Any vehicle.

3.42 ent_get_position

```
void ent_get_position(vehicle_id, position)
    int32   vehicle_id;
    float64 position[3];
```

'vehicle_id'
: Specifies the vehicle ID.

'position'
: Returns the position.

ent_get_position is used to get the position attribute of vehicle appearance. When translations from network representation to internal representation are necessary, these will be done upon invocation of the get function, and saved for future calls.

This can be called for Any vehicle.

3.43 ent_get_rotation

```
void ent_get_rotation(vehicle_id, rotation)
    int32   vehicle_id;
    float64 rotation[3][3];
```

'vehicle_id'
: Specifies the vehicle ID.

'rotation'
: Returns the rotation.

ent_get_rotation is used to get the rotation attribute of vehicle appearance. When translations from network representation to internal representation are necessary, these will be done upon invocation of the get function, and saved for future calls.

This can be called for Any vehicle.

3.44 ent_get_orientation

```
void ent_get_orientation(vehicle_id, heading, pitch, roll)
    int32   vehicle_id;
    float64 *heading;
    float64 *pitch;
    float64 *roll;
```

'vehicle_id'
Specifies the vehicle ID.

'heading' Returns the heading (0 == East, increasing counter-clockwise).

'pitch' Returns the pitch (0 == Level, increasing up).

'roll' Returns the roll (0 == Level, increasing counter-clockwise).

ent_get_orientation is used to get the orientation attribute of a vehicle. When translations from network representation to internal representation are necessary, these will be done upon invocation of the get function, and saved for future calls.

If the vehicle orientation was set with **ent_set_orientation**, this is computationally inexpensive. If, however the orientation was set with **ent_set_rotation**, many transcendental operations are required. Either way, the returned data will be valid.

Pass a NULL pointer for any unwanted values.

This can be called for Any vehicle.

3.45 ent_get_direction

```
void ent_get_direction(vehicle_id, direction)
    int32   vehicle_id;
    float64 direction[3];
```

'vehicle_id'
Specifies the vehicle ID.

'direction'
Returns the direction.

ent_get_direction is used to get the direction attribute of vehicle appearance. When translations from network representation to internal representation are necessary, these will be done upon invocation of the get function, and saved for future calls.

Note that direction is derived from rotation, but will be cheaper to compute than rotation under DIS.

This can be called for Any vehicle.

3.46 ent_get_appearance

```
uint32 ent_get_appearance(vehicle_id)
    int32 vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID.

ent_get_appearance is used to get the appearance attribute of vehicle appearance. When translations from network representation to internal representation are necessary, these will be done upon invocation of the get function, and saved for future calls.

This can be called for Any vehicle.

3.47 ent_get_capabilities

```
void ent_get_capabilities(vehicle_id, capabilities)
    int32               vehicle_id;
    VehicleCapabilities *capabilities;
```

'vehicle_id'
Specifies the vehicle ID.

'capabilities'
Returns the capabilities.

ent_get_capabilities is used to get the capabilities attribute of vehicle appearance. When translations from network representation to internal representation are necessary, these will be done upon invocation of the get function, and saved for future calls.

This can be called for Any vehicle.

3.48 ent_get_engine_speed

```
uint16 ent_get_engine_speed(vehicle_id)
    int32 vehicle_id;
```

'vehicle_id'
: Specifies the vehicle ID.

ent_get_engine_speed is used to get the engine_speed attribute of vehicle appearance. When translations from network representation to internal representation are necessary, these will be done upon invocation of the get function, and saved for future calls.

This can be called for Any vehicle.

3.49 ent_get_velocity

```
void ent_get_velocity(vehicle_id, velocity)
    int32   vehicle_id;
    float64 velocity[3];
```

'vehicle_id'
: Specifies the vehicle ID.

'velocity'
: Returns the velocity.

ent_get_velocity is used to get the velocity attribute of vehicle appearance. When translations from network representation to internal representation are necessary, these will be done upon invocation of the get function, and saved for future calls.

This can be called for Any vehicle.

3.50 ent_get_speed_squared

```
float64 ent_get_speed_squared(vehicle_id)
    int32 vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID.

ent_get_speed_squared is used to get the speed_squared attribute of vehicle appearance. When translations from network representation to internal representation are necessary, these will be done upon invocation of the get function, and saved for future calls.

Note that speed_squared is derived from velocity, but since the value will be cached within libentity, it is better to call this function than to call ent_get_velocity and compute speed squared manually.

This can be called for Any vehicle.

3.51 ent_get_speed

```
float64 ent_get_speed(vehicle_id)
    int32 vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID.

ent_get_speed is used to get the speed attribute of vehicle appearance. When translations from network representation to internal representation are necessary, these will be done upon invocation of the get function, and saved for future calls.

Note that speed is derived from speed_squared, but since the value will be cached within libentity, it is better to call this function than to call ent_get_velocity or ent_get_speed_squared and compute speed manually.

This can be called for Any vehicle.

3.52 ent_get_stationary

```
int32 ent_get_stationary(vehicle_id)
    int32 vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID.

ent_get_stationary is used to get the stationary attribute of vehicle appearance. When translations from network representation to internal representation are necessary, these will be done upon invocation of the get function, and saved for future calls.

This can be called for Any vehicle.

3.53 ent_get_artic_euler

```
void ent_get_artic_euler(vehicle_id, artic_name, yaw, pitch, roll)
    int32    vehicle_id;
    char    *artic_name;
    float64 *yaw;
    float64 *pitch;
    float64 *roll;
```

'vehicle_id'
Specifies the vehicle ID.

'artic_name'
Specifies the name of the articulation (a libreader symbol)

'yaw' Returns the yaw value (in radians) of the articulation

'pitch' Returns the pitch value (in radians) of the articulation

'roll' Returns the roll value (in radians) of the articulation

ent_get_artic_euler is used to get the euler angles for an articulation named **artic_name**. Any of the pointers for **yaw**, **pitch**, or **roll** can be **NULL**, in which case that component of the articulation value will not be returned.

This can be called for Any vehicle.

3.54 ent_get_artic_euler_rate

```
void ent_get_artic_euler_rate(vehicle_id, artic_name,
                              yaw_rate, pitch_rate, roll_rate)
    int32    vehicle_id;
```

```
        char    *artic_name;
        float64 *yaw_rate;
        float64 *pitch_rate;
        float64 *roll_rate;
```

'**vehicle_id**'
Specifies the vehicle ID.

'**artic_name**'
Specifies the name of the articulation (a libreader symbol)

'**yaw_rate**'
Returns the yaw rate (in radians per second) of the articulation

'**pitch_rate**'
Returns the pitch rate (in radians per second) of the articulation

'**roll_rate**'
Returns the roll rate (in radians per second) of the articulation

ent_get_artic_euler_rate is used to get the euler rates for an articulation named **artic_name**. Any of the pointers for **yaw_rate**, **pitch_rate**, or **roll_rate** can be **NULL**, in which case that component of the articulation rate will not be returned.

This can be called for Any vehicle.

3.55 ent_get_artic_pivot

```
    void ent_get_artic_pivot(vehicle_id, artic_name, position)
        int32    vehicle_id;
        char    *artic_name;
        float64  position[3];
```

'**vehicle_id**'
Specifies the vehicle ID.

'**artic_name**'
Specifies the name of the articulation (a libreader symbol)

'**position**'
Returns the position of the pivot point (in vehicle coordinates) of the articulation.

ent_get_artic_pivot is used to get the pivot point for an articulation named **artic_name**. For example, the pivot point for a gun is the *base* of the gun, and the pivot point for a turret is the location of the axis of rotation.

This can be called for Any vehicle.

3.56 ent_get_artic_position

```
void ent_get_artic_position(vehicle_id, artic_name, position)
    int32    vehicle_id;
    char    *artic_name;
    float64  position[3];
```

'vehicle_id'
Specifies the vehicle ID.

'artic_name'
Specifies the name of the articulation (a libreader symbol)

'position'
Returns the position of the end point (in vehicle coordinates) of the articulation.

ent_get_artic_position is used to get the position for an articulation named **artic_name**. For example, the position for a gun is the *tip* of the gun. The position for a turret is the location of the axis of rotation since a turret doesn't usually have a unique length.

This can be called for Any vehicle.

3.57 ent_get_artic_rotation

```
void ent_get_artic_rotation(vehicle_id, artic_name, rotation)
    int32    vehicle_id;
    char    *artic_name;
    float64  rotation[3][3];
```

'vehicle_id'
Specifies the vehicle ID.

'artic_name'
Specifies the name of the articulation (a libreader symbol)

'rotation'
Returns the rotation matrix for the articulation.

ent_get_artic_rotation is used to get the rotation matrix for an articulation named **artic_name**. This rotation matrix can be used to transform a position in articulation coordinates into vehicle coordinates. Note that to do the transformation completely, you will have to consider the articulation position (see ⟨undefined⟩ [ent˙get˙artic˙position], page ⟨undefined⟩).

This can be called for Any vehicle.

3.58 ent_get_turret_articulation

```
void ent_get_turret_articulation(vehicle_id, articulation)
    int32                      vehicle_id;
    DIS_ARTICULATION_PARAMETER *articulation;
```

'**vehicle_id**'

Specifies the vehicle ID.

'**articulation**'

Returns the DIS representation of the turret's orientation

ent_get_turret_articulation is used to get the DIS representation of a turret's orientation.

This can be called for Any vehicle.

3.59 ent_get_rotation_sp

```
void ent_get_rotation_sp(vehicle_id, rotation)
    int32   vehicle_id;
    float32 rotation[3][3];
```

'**vehicle_id**'

Specifies the vehicle ID.

'**rotation**'

Returns the rotation (in single precision).

ent_get_rotation_sp is used to get the rotation attribute of vehicle appearance (single precision version needed to build appearance packets). When translations from network representation to internal representation are necessary, these will be done upon invocation of the get function, and saved for future calls.

This can be called for Any vehicle.

3.60 ent_get_velocity_sp

```
void ent_get_velocity_sp(vehicle_id, velocity)
    int32   vehicle_id;
    float32 velocity[3];
```

'vehicle_id'

Specifies the vehicle ID.

'velocity'

Returns the velocity (in single precision).

ent_get_velocity_sp is used to get the velocity attribute of vehicle appearance (single precision version needed to build appearance packets). When translations from network representation to internal representation are necessary, these will be done upon invocation of the get function, and saved for future calls.

This can be called for Any vehicle.

3.61 ent_get_physdb

```
PHYSDB_DATA *ent_get_physdb(vehicle_id)
    int32   vehicle_id;
```

'vehicle_id'

Specifies the vehicle ID.

ent_get_physdb looks up the object type of the vehicle in the physdb database (see section 'physdb_key' in LibPhysDB Programmer's Manual). The returned data is a pointer to the physical information typical of an object with that object type (such as its dimensions). Because of the lazy evaluation and cacheing, calling this function will yield better performance that calling **otm_query** explicitly (see section 'otm_query' in LibOTMatch Programmer's Manual).

This can be called for Any vehicle.

3.62 ent_get_altitude_agl

```
float64 ent_get_altitude_agl(vehicle_id, ctdb)
    int32 vehicle_id;
    CTDB *ctdb;
```

'vehicle_id'
    Specifies the vehicle ID.

'ctdb' Pointer to the CTDB terrain database structure.

ent_get_altitude_agl is used to get the altitude AGL (Above Ground Level) of a vehicle.

Note that altitude_agl is derived from position, but since the value will be cached within libentity. it is better to call this function than to call ent_get_position and compute altitude AGL manually.

This can be called for Any vehicle.

3.63 ent_get_dis_guises

```
void ent_get_dis_guises(vehicle_id, regular, alternate)
    int32           vehicle_id;
    DIS_ENTITY_TYPE *regular;
    DIS_ENTITY_TYPE *alternate;
```

'vehicle_id'
    Specifies the vehicle ID.

'regular' Pointer to the regular DIS guise for the vehicle.

'alternate'
    Pointer to the alternate DIS guise for the vehicle.

ent_get_dis_guises is used to get the normal and alternate DIS guises of a vehicle.

This can be called for Any vehicle.

3.64 ent_get_dis_location

```
void ent_get_dis_location(vehicle_id, location)
    int32    vehicle_id;
    float64 *location;
```

'vehicle_id'
Specifies the vehicle ID.

'location'
Pointer to the DIS location for the vehicle.

ent_get_dis_location is used to get the DIS location (in a Z-down GCC coordinate system) of a vehicle.

This can be called for Any vehicle.

3.65 ent_get_dis_velocity

```
void ent_get_dis_velocity(vehicle_id, velocity)
    int32    vehicle_id;
    float32 *velocity;
```

'vehicle_id'
Specifies the vehicle ID.

'velocity'
Pointer to the DIS velocity for the vehicle.

ent_get_dis_velocity is used to get the DIS velocity (in a Z-down GCC coordinate system) of a vehicle.

This can be called for Any vehicle.

3.66 ent_get_dis_orientation

```
void ent_get_dis_orientation(vehicle_id, orientation)
    int32            vehicle_id;
    DIS_EULER_ANGLE *orientation;
```

'vehicle_id'

Specifies the vehicle ID.

'orientation'

Pointer to the DIS velocity for the vehicle.

ent_get_dis_orientation is used to get the DIS orientation of a vehicle.

This can be called for Any vehicle.

3.67 ent_get_dis_appearance

```
DIS_ENTITY_APPEARANCE ent_get_dis_appearance(vehicle_id)
    int32           vehicle_id;
```

'vehicle_id'

Specifies the vehicle ID.

ent_get_dis_appearance is used to get the DIS appearance bits of a vehicle.

This can be called for Any vehicle.

3.68 ent_get_dis_capabilities

```
void ent_get_dis_capabilities(vehicle_id, capabilities)
    int32                     vehicle_id;
    DIS_ENTITY_CAPABILITIES *capabilities;
```

'vehicle_id'

Specifies the vehicle ID.

'capabilities'

Pointer to the DIS capabilities for the vehicle.

ent_get_dis_capabilities is used to get the DIS capabilities bits of a vehicle.

This can be called for Any vehicle.

4 Access Keys

In addition to the get functions just described, libentity also provides libaccess keys with which many variables can be fetched at once. These keys, and the type of argument they expect are given below:

ent_exercise_id
    `uint8 *arg`

ent_vehicle_class
    `VehicleClass *arg`

ent_force_id
    `ForceID *arg`

ent_guises `VehicleGuises *arg`

ent_marking
    `VehicleMarking *arg`

ent_position
    `float64 arg[3]`

ent_rotation
    `float64 arg[3][3]`

ent_direction
    `float64 arg[3]`

ent_appearance
    `uint32 *arg`

ent_capabilities
    `VehicleCapabilities *arg`

ent_engine_speed
    `uint16 *arg`

ent_velocity
    `float64 arg[3]`

ent_speed_squared
    `float64 *arg`

ent_speed `float64 *arg`

ent_stationary
    `int32 *arg`

ent_rotation_sp
    `float32 arg[3][3]`

ent_velocity_sp

    float32 arg[3]

ent_physdb

    PHYSDB_DATA **arg

ent_dis_location

    DIS_WORLD_COORDINATES *arg

ent_dis_velocity

    DIS_VECTOR *arg

ent_dis_orientation

    DIS_EULER_ANGLE *arg

ent_dis_appearance

    DIS_ENTITY_APPEARANCE *arg

ent_dis_capabilities

    DIS_ENTITY_CAPABILITIES *arg

Libeoorder

Table of Contents

1 Overview

Libeonorder inplements an enabling task which checks an associated task authorization object. If the authorized bit is set, libeonorder's predicate function returns TRUE, otherwise it returns FALSE. The GUI code is responsible for setting the authorization bit. When the predicate function is invoked, it first checks to see if the task authorization object exists as a reference in the task's state object. If the authorization object does not yet exist, it is created.

2 Functions

The following sections describe each function provided by libeonorder, including the format and meaning of its arguments, and the meaning of its return values (if any).

2.1 eono_init

```
void eono_init()
```

eono_init initializes libeonorder. Call this before any other libeonorder function.

2.2 eono_class_init

```
void eono_class_init(parent_class)
    CLASS_PTR parent_class;
```

'**parent_class**'
Class of the parent (declared with **class_declare_class**)

eono_class_init creates a handle for attaching eonorder class information to vehicles. The **parent_class** will likely be **safobj_class**.

2.3 eono_create

```
void eono_create(vehicle_id, params, po_db)
    int vehicle_id;
    EONORDER_PARAMETRIC_DATA *params;
    PO_DATABASE *po_db;
```

'**vehicle_id**'
Specifies the vehicle ID

'**params**' Specifies initial parameter values

'**po_db**' Specifies the PO DATABASE used for the exercise

eono_create creates the eonorder class information for a vehicle and attaches it vehicle's block of libclass user data.

2.4 eono_destroy

```
void eono_destroy(vehicle_id)
    int vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

eono_destroy frees the eonorder class information for a vehicle. This should be called before freeing the class user data with **class_free_user_data**.

2.5 eono_create_task

```
PO_DB_ENTRY *eono_create_task(po_db)
    PO_DATABASE *po_db;
```

'po_db' Specifies the PO DATABASE for the exercise

eono_delete_task creates an enabling class with model **SM_EOnOrder**. When the task is created, no state references are created.

2.6 eono_delete_task

```
int32 eono_delete_task(po_db, eonorder_task)
    PO_DATABASE *po_db;
    PO_DB_ENTRY *eonorder_task;
```

'vehicle_id'
Specifies the vehicle ID

'eonorder_task'
Specifies the enabling task to be destroyed

eono_delete_task destroys the specified on-order enabling task, along with its' referenced authorization object (if any).

ADST-TR-W003268

Libetcm

Table of Contents

1 Overview

Libetcm implements an enabling task which detects when a unit is about to cross a control measure. The criteria which libetcm uses to detect this are context-dependent and include the previous taskframe, i.e. what the unit is currently doing, as well as the taskframe containing the enabling task, i.e. what the unit would do if libetcm's predicate returned a non-zero value. This contextual "matrix," as well as the simulation state of the unit, are used by the predicate function registered by libetcm with libtask. The contextual information is known to the task manager and is passed in the argument list to the predicate function of the enabling task when it is called:

```
PO_DB_ENTRY *current_opaque_task_frame;
PO_DB_ENTRY *next_task_frame;
```

In addition, the following information is passed to the predicate function when it is called in case the current executing task frame does not provide enough context for the predicate function to make its decision, e.g. the current frame is a transparent override:

```
int32        number_of_executing_tasks;
PO_DB_ENTRY *executing_tasks[];
```

This library is also responsible for setting the shared parameters of the appropriate task in the task frame pointed to by `PO_DB_ENTRY *next_task_frame`.

For example, suppose a unit is following a route and is about to cross a phase line, where the unit has been ordered to launch an assault. The control measure enabling task is responsible for calculating how far away from the phase line the unit must begin to prepare for the assault. Libetcm's predicate function returns a non-zero value when the unit reaches this location. This library is also responsible for setting the shared parameters of the prepare-to-assault task in its own frame so that the prepare-to-assault task will be able to prepare for the assault. This may invovle slowing the unit down, keeping to the projected path along its current route, and halting at the desired location in preparation for the assault.

The task state machine is written using the AAFSM format which is translated to C using the '`fsm2ch`' utility (see section 'Overview' in LibTask Programmer's Manual).

Libetcm depends on libuflwrte, libpo, libclass, libctdb, libaccess, libreader, and libparmgr.

1.1 Task Parameters

The format of the parametric data is as follows:

```
(SM_ETCM (waypt_error <distance meters>)
 )
```

The **waypt_error** parameter specifies the distance from a vertex of a control measure at which a vehicle will perceive that it has crossed the control measure.

1.2 Task Parameters

In enabling tasks for control measures, the parameter block of the task data structure is empty.

2 Functions

The following sections describe each function provided by libetcm, including the format and meaning of its arguments, and the meaning of its return values (if any).

2.1 etcm_init

```
void etcm_init()
```

etcm_init initializes libetcm. Call this before any other libetcm function.

2.2 etcm_class_init

```
void etcm_class_init(parent_class)
    CLASS_PTR parent_class;
```

'**parent_class**'
Class of the parent (declared with **class_declare_class**)

etcm_class_init creates a handle for attaching etcm class information to vehicles. The **parent_class** will likely be **safobj_class**.

2.3 etcm_create

```
void etcm_create(vehicle_id, params, po_db, ctdb)
    int vehicle_id;
    ETCM_PARAMETRIC_DATA *params;
    PO_DATABASE          *po_db;
    CTDB                 *ctdb;
```

'**vehicle_id**'
Specifies the vehicle ID

'**params**' Specifies initial parameter values

'**po_db**' Specifies the PO database where the task can be found '**ctdb**' Specifies the terrain database currently in use

etcm_create creates the etcm class information for a vehicle and attaches it vehicle's block of libclass user data.

2.4 etcm_destroy

```
void etcm_destroy(vehicle_id)
    int vehicle_id;
```

'**vehicle_id**'
Specifies the vehicle ID

etcm_destroy frees the etcm class information for a vehicle. This should be called before freeing the class user data with **class_free_user_data**.

2.5 etcm_init_task_state

```
void etcm_init_task_state(task, state)
    TaskClass *task;
    TaskStateClass *state;
```

'**task**' Specifies a pointer to the task class object to be initialized.

'**state**' Returns the initialized state

Given a new **SM_ETCM** task that is about to be created, **etcm_init_task_state** initializes the model size, and state variables.

ADST-TR-W003268

LibExecmat

Table of Contents

1 Overview

Libexecmat creates the execution matrix editor for units and subordinate units. The execution matrix is used to create and assign missions. A mission is made up of a series of task frames separated by control measures (a control measure can be NULL). Reaching a control measure advances the mission to the next task frame. The control measure is referred to as an ETCM (Enabling Task Control Measure) because reaching this control measure enables the mission to advance to the next task frame. A task frame can be made "On Order". A mission will stay in its current task frame if the next task frame is "On Order". Authorizing an "On Order" advances the mission to the authorized task frame.

Each displayed task frame is really made up of two task frames - the Preparatory task frame and the Actual task frame. Only the Actual task frame is displayed in the execution matrix. The task frame editor MUST return the two task frames linked together by the `previousMissionFrame` field. Each task frame has a primary task which is used to determine whether the task frame is finished. (If the primary task is done, the task frame is done.)

In addition to the ETCM enabling task, there is another enabling task called the In-Phase enabling task. The In-Phase enabling task determines whether the unit's peer units are in the same phase as it is. This is done by seeing if the peer units have completed their preparatory task frames for the same phase.

In each of the prep and actual task frames, there is a postfix logic stack which determines the conditions that must be met before this taskframe is executed. The logic stack algorithm for the preparatory task frame is:

```
Primary task of previous frame is done  OR
ETCM of phase is crossed OR
On Order of task frame has been given (if applicable)
```

If any of these conditions is met, the preparatory task frame is executed.

The logic stack algorithm for the actual task frame is:

```
Primary task of prepatory task is done AND
Other peer units are in-phase AND
On Order of actual task frame has been given (if applicable)
```

If all of these conditions are met, the actual task frame is executed. Note that there is no guarantee that the ETCM of the phase has been reached. This is done because there might be some user error where the ETCM could never be reached and the mission still needs to advance.

When the user clicks the "Done" button on the execution matrix editor, the task frames are linked into a mission for each unit. The mission is linked backwards using the **previousMissionFrame** field in the task frame. If the unit has not already been assigned a mission, this mission is assigned to the unit. Although the mission code is general enough to support a tree structure, the current missions are flat.

The execution matrix editor is not controlled by libeditor, so it has to keep track of its own state when it becomes active, suspended, etc.

2 Functions

The following sections describe each function provided by libexecmat, including the format and meaning of its arguments, and the meaning of its return values (if any).

2.1 execmat_init

```
void execmat_init()
```

execmat_init initializes libexecmat. Call this function before calling any other libexecmat functions.

2.2 execmat_init_gui

```
EXECMAT_GUI_PTR execmat_init_gui(data_path, reader_flags,
                                          gui, tactmap, tcc, map_erase_gc,
                                 sensitive, refresh_event, db, exit_fcn,exit_arg)
        char                    *data_path;
        int32                    reader_flags;
        SGUI_PTR                 gui;
        TACTMAP_PTR              tactmap;
        COORD_TCC_PTR            tcc;
        GC                       map_erase_gc;
        SNSTVE_WINDOW_PTR        sensitive;
        CALLBACK_EVENT_PTR       refresh_event;
        PO_DATABASE             *db;
        ASSIGN_EXIT_FUNCTION       exit_fcn;
        ADDRESS                    exit_arg;
```

'**data_path**'
Specifies the directory where data files are expected

'**reader_flags**'
Specifies flags to be passed to **reader_read** when reading data files

'**gui**' Specifies the SAF GUI

'**tactmap**' Specifies the tactical map

'**tcc**' Specifies the map coordinate system

'**map_erase_gc**'
Specifies the GC which can erase things from the tactical map

'db' Specifies the persistent object database

'exit_fcn, exit_arg'
Specifies a function to call when the assignment is completed

execmat_init_gui creates the execution matrix editor. The execution matrix editor consists of up to 6 phases of tasks for UNITORG_MAX_BREADTH units.

2.3 execmat_set_unit

```
void execmat_set_unit(emgui, unit_id, tasking_type)
    EXECMAT_GUI_PTR emgui;
    ObjectID        *unit_id;
    EXECMAT_TASKING_TYPE tasking_type;
```

'emgui' Specifies EXECMAT_GUI_PTR data structure

'unit_id' Specifies the unit id.

'tasking_type'
Specifies SUBORDINATE_TASKING or UNIT_TASKING

execmat_set_unit fills in the emgui data structure units with either the unit_id passed in or subordinate units of the unit_id passed in depending on the tasking_type. If the units (or subordinate units) are currently executing a mission, their po task frames are retrieved and filled into the emgui data structure as well as the pushbutton widgets of the matrix user interface. The names of the units are retrieved from the unit po and filled in the label widgets of the matrix user interface.

2.4 execmat_create

```
void execmat_state(gui, mode, state)
    EXECMAT_GUI_PTR gui;
    SGUI_MODE_PTR   mode;
    SGUI_MODE_STATE state;
```

'gui' Specifies the SAF GUI

'mode' Specifies the mode to use

'state' Specifies the new state

execmat_state sets the state of the execution matrix editor. This is equivalent in functionality to edt_state(), but is needed because the execution matrix editor is not controlled by libeditor.

LibFCS

Table of Contents

1 Overview

libfcs profides a fire-control system abstraction for SAF vehicles which have a large number of **libguns** components (see section 'Overview' in LibGUNS Programmer's Manual), such as aircraft. libfcs provides an interface where weapon launcher commands can be specified in terms of the munition which will be fired; libfcs uses selection algorithms to determine which launcher is most suitable to direct the **libguns** commands for a given munition. In the future, libfcs can be expanded to support selection algorithms which can perform such tasks as launching missiles in the proper order from an aircraft with multiple missile launcher stations in order to keep the aircraft as balanced as possible, or to avoid a launched missile from colliding with an un-launched missile.

The parameters of the fire control system are specified in the configuration file for the vehicle containing a libfcs fire control system as follows:

```
(SM_FireControl (components <name1> <name2> ...)
                )
```

The **name** of each component being controlled by the fire control system is specified. Each **name** must be a gun component of the vehicle.

2 Usage

The software library 'libfcs.a' should be built and installed in the directory '/common/lib/'. You will also need the header file 'libfcs.h' which should be installed in the directory '/common/include/libinc/'. If these files are not installed, you need to do a 'make' in the libfcs source directory. If these files are already built, you can skip the section on building libfcs.

2.1 Building Libfcs

The libfcs source files are found in the directory '/common/libsrc/libfcs'. 'RCS' format versions of the files can be found in '/nfs/common_src/libsrc/libfcs'.

If the directory 'common/libsrc/libfcs' does not exist on your machine, you should use the 'genbuild' command to update the common directory hierarchy.

To build and install the library, do the following:

```
# cd common/libsrc/libfcs
# co RCS/*,v
# make install
```

This should compile the library 'libfcs.a' and install it and the header file 'libfcs.h' in the standard directories. If any errors occur during compilation, you may need to adjust the source code or 'Makefile' for the platform on which you are compiling. libfcs should compile without errors on the following platforms:

- Mips
- SGI Indigo
- Sun Sparc

2.2 Linking with Libfcs

Libfcs can be linked into an application program with the following link time flags: 'ld [source .o files] -L/common/lib -lfcs'. If your compiler does not support '-L' syntax, you can use the archive explicitly: 'ld [source .o files] /common/lib/libfcs.a'.

Libfcs depends on libaccess libcomponents libreader libguns libparmgr libvtab and libclass.

3 Functions

The following sections describe each function provided by libfcs, including the format and meaning of its arguments, and the meaning of its return values (if any).

3.1 fcs_init

```
void fcs_init()
```

fcs_init initializes libfcs. Call this before any other libfcs functions.

3.2 fcs_class_init

```
void fcs_class_init(parent_class)
    CLASS_PTR parent_class;
```

'**parent_class**'
Specifies the parent class of entity (probably safobj_class).

fcs_class_init creates a handle for attaching FCS class information to vehicles. The **parent_class** is one created with **class_declare_class**.

3.3 fcs_create

```
void fcs_create(vehicle_id, parms)
    int32                vehicle_id;
    FCS_PARAMETRIC_DATA *parms;
```

'**vehicle_id**'
Specifies the vehicle_id of the vehicle to be created.

'**parms**' Specifies parametric data for the fire control system.

fcs_create creates the FCS class information for a vehicle and atttaches it to the vehicle's libclass user data.

3.4 fcs_destroy

```
void fcs_destroy(vehicle_id)
    int32                 vehicle_id;
```

'vehicle_id'
Specifies the vehicle_id of the vehicle to be created.

fcs_destroy frees the FCS class information for a vehicl‸

3.5 fcs_load

```
void fcs_load(vehicle_id, munition, store, quantity)
    int32  vehicle_id;
    uint32 munition;
    int32  store;
    int32  quantity;
```

'vehicle_id'
Specifies the vehicle ID

'munition'
Specifies the type of munition to load

'store' Specifies from what store to load the munition from

'quantity'
Specifies the quantity of munitions to load into the gun

fcs_load starts the loading procedure for a **quantity** number of **munitions** from the supplied **store** for an apropriate gun component. **fcs_load** will dynamically determine the appropriate component to direct the **GUNS_SET_LOAD_MUNITION** command
(see section 'GUNS_SET_LOAD_MUNITION' in LibGUNS Programmer's Manual) from the list of possible components specified as parametric data to **libfcs**.

3.6 fcs_quantities

```
void fcs_quantities(vehicle_id, munition, current_limit, absolute_limit)
    int32  vehicle_id;
    uint32 munition;
```

```
        int32  *current_limit;
        int32  *absolute_limit;
```

'vehicle_id'
Specifies the vehicle ID

'munition'
Specifies the type of munition to load

'current_limit'
Specifies the current limit on this munition type

'absolute_limit'
Specifies the absolute limit on this munition type

fcs_quantities returns (by reference) the current and absolute maxima of the number of rounds that the FCS can fire of the specified munition type on the given vehicle. These quantities are related to but not necessarily equal to supplies available and should be interpreted as how many rounds the FCS can fire given unlimited supplies. These may be based on limitations of the actual weapons or implementations of the code modeling the weapons.

3.7 fcs_ready

```
    void fcs_ready(vehicle_id, munition, ready, missile_id)
        int32  vehicle_id;
        uint32 munition;
        int32  *ready;
        int32  *missile_id;
```

'vehicle_id'
Specifies the vehicle ID

'munition'
Specifies the munition to check

'ready' Returns whether a gun containing the specified munition is ready for firing

'id' Returns vehicle ID of a ready missile

fcs_ready returns (by reference) whether an appropriate gun component is ready to fire the requested munition. For missile launchers, the vehicle ID of the ready missile is returned (by reference) as well. A gun is generally not ready if it is not loaded or is in the process of loading or unloading munitions.

fcs_ready will dynamically determine the appropriate component to direct the **GUNS_GET_READY_TO_FIRE** command (see section 'GUNS_GET_READY_TO_FIRE' in LibGUNS Programmer's Manual) from the list of possible components specified as parametric data to **libfcs**.

3.8 fcs_fire

```
void fcs_fire(vehicle_id, munition, quantity)
    int32  vehicle_id;
    uint32 munition;
    int32  quantity;
```

'vehicle_id'
Specifies the vehicle ID

'munition'
Specifies the munition

'quantity'
Specifies quantity of loaded munition to shoot

fcs_fire attempts to launch a weapon of the specified munition from one of the possible guns in the fire control system. The weapon is fired in whatever direction it is currently pointing.

fcs_fire will dynamically determine the appropriate component to direct the **GUNS_SET_FIRE** command (see section 'GUNS_SET_FIRE' in LibGUNS Programmer's Manual) from the list of possible components specified as parametric data to **libfcs**.

3.9 fcs_fire_at_target

```
void fcs_fire_at_target(vehicle_id, target_id, munition, quantity)
    int32  vehicle_id;
    int32  target_id;
    uint32 munition;
    int32  quantity;
```

'vehicle_id'
Specifies the vehicle ID

'target_id'
Specifies the target to fire at

'munition'

Specifies the munition

'quantity'

Specifies quantity of loaded munition to shoot

fcs_fire_at_target attempts to launch a weapon of the specified munition from one of the possible guns in the fire control system. The weapon is fired with the intent to hit the target, if possible. If the weapon is a guided munition (such as a missile), there may be other actions required during the flight of the munition to ensure that it will hit the target.

fcs_fire_at_target will dynamically determine the appropriate component to direct the GUNS_SET_FIRE_ command (see section 'GUNS_SET_FIRE_AT_TARGET' in LibGUNS Programmer's Manual) from the list of possible components specified as parametric data to libfcs. •

3.10 fcs_fire_at_location

```
void fcs_fire_at_location(vehicle_id, location, munition, quantity)
    int32   vehicle_id;
    float64 location[3];
    uint32  munition;
    int32   quantity;
```

'vehicle_id'

Specifies the vehicle ID

'location'

Specifies the location to fire at

'munition'

Specifies the munition

'quantity'

Specifies quantity of loaded munition to shoot

fcs_fire_at_location attempts to launch a weapon of the specified munition from one of the possible guns in the fire control system. The weapon is fired with the intent to hit the location, if possible. If the weapon is a guided munition (such as a missile), there may be other actions required during the flight of the munition to ensure that it will hit the location.

fcs_fire_at_location will dynamically determine the appropriate component to direct the GUNS_SET_FIRE_AT_LOCATION command (see section 'GUNS_SET_FIRE_AT_LOCATION' in Lib-

GUNS Programmer's Manual) from the list of possible components specified as parametric data to **libfcs**.

LibFormationDB

Table of Contents

1 Overview

LibFormationDB provides a database of named standard military formations which can be used for initial placement of units as well as for station keeping of units during movement. LibFormationDB uses a database encoded in libreader format (see section 'Overview' in LibReader Programmer's Manual) d to represent the placement of units in a formation. Note that in this context, the term '**units**' can refer to individual vehicles or to unit aggregates (such as sections, platoons, companies, etc.).

LibFormationDB provides two primary interface routines for accessing formation information. **formdb_expand** can be used to apply a formation and spacing to units for a given location and direction. Position information (where a unit should be to be in station) is returned, as well as desired scan-sectors for each unit. The routine can take current positions of the units into account in order to assign formations which minimize vehicle movement (such as reversing a formation or using an alternate assignment of units to formation stations). **formdb_create_routes** can be used to apply a specified formation and spacing to units which should be following a supplied route. Routes are generated for each unit which are similar to the input route and have the property that if each unit follows each route, the units will appear to be keeping station, at least at the route vertices. Note that this routine is not suitable for roadmarch, since roadpoints for followers would not be preserved.

1.1 Algorithms

To be placed in a formation, units are laid out one at a time in lines called rays. A given formation may contain one or more rays in which to place units. Any ray (except the first one specified) will refer to a previously specified ray to indicate a reference starting position for that ray. The first ray in a formation refers to no other ray.

Each ray in a formation is given a unit one at a time in ray-breadth order. If a ray exceeds some specified maximum number of units, that ray is skipped from receiving more units. The formation laydown algorithm uses data in the formation database file to decide the order by which units are assigned the formation, thus allowing the explicit positioning of specific units such as a unit-leader. Additional information about the placement algorithm is contained in the description of the LibFormationDB file format (see Section 1.2 [File Format], page 2).

The algorithm used to minimize vehicle movement in the assignment of vehicles to a formation template for **formdb_expand** is to assign vehicles in all formation orderings and to choose the one which minimizes crossings. Among those assignments with the same number of crossings, the one containing the minimum sum of of distances to be travelled by every unit is chosen.

The algorithm used to generate follower routes in **formdb_create_routes** is to use **formdb_expand** at the begining of the route to identify the starting position of each unit. **formdb_expand** is used at every waypoint which is at least the specified sampling distance apart. At those waypoints, the direction chosen is the mean of the incoming and outgoing directions. The formation ordering remains constant over an entire route. Route segments are inferred as straight line connections between formation expanded waypoints.

1.2 File Format

Formation information is stored in the formation database '**formdb.rdr**' as follows:

```
(
  (<formation_name>
    (<ray-specifier1>
     <ray-specifier2>
     ...)
    (<placement-number1> <placement-number2> <placement-number3> ...)
    <other placement lists>
    ...
   )
  ...
)
```

<formation_name> specifies the name of the formation.

<ray-specifier> specifies the incremental placement of vehicles along a ray as follows:

```
(<reference> <compass_offset> <Z_offset> <spacing_factor> <max>)
```

<reference> is the index of the ray that this ray keys off. The indices of the rays are zero-based. The first ray on which a vehicle will be placed will have a reference of **-1**.

<compass_offset> indicates the direction to place subsequent units from previous units already placed in a ray. For the first unit in a ray, the **<compass_offset>** specifies the offset from the first unit in the **referenced** ray. The **<compass_offset>** for the first vehicle in the first ray is not used. The values of **<compass_offset>** are specified by the following macro names, which are defined in '**formdb.rdr**':

- `FORMDB_FRONT`
- `FORMDB_BACK`
- `FORMDB_RIGHT`
- `FORMDB_LEFT`
- `FORMDB_FRONT_RIGHT`
- `FORMDB_FRONT_LEFT`
- `FORMDB_BACK_RIGHT`
- `FORMDB_BACK_LEFT`

The interpretation of the **<compass_offset>** directions are as in the following table:

```
-----------------------------------------------------------------
|                     |                   |                     |
|  FORMDB_FRONT_LEFT  |   FORMDB_FRONT    | FORMDB_FRONT_RIGHT  |
|                     |                   |                     |
|---------------------------------------------------------------|
|                     |                   |                     |
|                     |         |         |                     |
|                     |       |-|-|       |                     |
|          FORMDB_LEFT|       | | |       | FORMDB_RIGHT        |
|                     |       |   |       |                     |
|                     |       |---|       |                     |
|                     |                   |                     |
|                     | (previously placed|                     |
|                     |       unit)       |                     |
|---------------------------------------------------------------|
|                     |                   |                     |
|   FORMDB_BACK_LEFT  |    FORMDB_BACK    | FORMDB_BACK_RIGHT   |
|                     |                   |                     |
-----------------------------------------------------------------
```

<Z_offset> specifies whether or not the placement along the ray includes a Z component, for 3-dimensional formations. This is only useful for air-formations. The values for <Z_offset> can be:

- `FORMDB_LEVEL`
- `FORMDB_UP`
- `FORMDB_DOWN`

<spacing_factor> specifies a positive real multiplicative factor to use when applying a formation spacing to vehicles along a ray. Vehicles placed on a diagonal ray will have an additional implied multiplicative factor equal to the `sqrt(2)`. This implies that a formation with a uniform spacing factor for all rays will result in all units being placed on a virtual grid. For uniform spacing, the **<spacing_factor>** will typically be equal to 1.0.

<max> specifies the integer maximum number of vehicles to be placed on a ray. A value of -1 means an infinite number of vehicles may be placed on this ray.

<placement-numberN> specifies the assignment order in which units are placed into a formation. A unit's "job" (also referred to in ModSAF as its promotion index) is used to index this list to decide when that unit should be placed with respect to other units in the formation. For example, the ordering (2 3 1 4) specifies that the unit with a job of 1 should be placed second, the unit with the job of 2 should be placed third, the unit with the job of 3 should be placed first, and so on. This allows precise ordering of vehicles in the formation to produce tactically correct assignments. By convention, job 1 refers to a unit leader, such as a platoon leader, 2 refers to the leader's "wingman", 3 refers to the unit's second-in-command, such as a platoon sergeant, and 4 refers to the "wingman" of the second-in-command. Units with job numbers greater than the length of the list will be placed after all other units in job number order. Negative indices can be used to represent the placement of vehicles in last-to-first order. For example, the ordering (-1 -2 -3 -4) indicates that the unit with a job of 1 will be placed last.

Note that in ModSAF, "jobs", as reported in the **taskOrgIndex** and **functionalOrgIndex** fields of a UnitClass Persistent Object, are indexed starting at zero, as opposed to the "jobs" referred to in the libFormationDB data file, which are indexed starting at one (in order to match typical Platoon training manuals). Hence, the assignment for the unit 1 specified in '**formdb.rdr**' may refer to a ModSAF entity containing a **functionalOrgIndex** of value 0 in its corresponding UnitClass Persistent Object.

See section 'Unit Class' in LibPO Programmer's Manual.

Multiple placement lists may be specified in order to allow formation orderings that minimize crossing of vehicles or distance travelled.

As an example of complete formation specification, the following could be used to specify an echelon-right formation.

```
;; Echelon formations are like column with a slant
;;     2         2         1         1
;;      1         1         2         2
;;       4         3         3         4
;;        3         4         4         3
;;         .         .         .         .
;;    .    .    .    .
;;     .    .    .    .
(echelon-right
 ((-1  FORMDB_RIGHT_BACK FORMDB_LEVEL 1.0 -1))
 (;; Wingman in front
  (2 1 4 3)
  (2 1 3 4)
  ;; Leader in front
  (1 2 3 4)
  (1 2 4 3)

  ;; Reverse formations
  (3 4 1 2)
  (4 3 1 2)
  (4 3 2 1)
  (3 4 2 1)
 )
 )
```

1.3 Data Structures

Formation queries utilize the following data structure to encode unit information for both input and output:

```
typedef struct formdb_data
{
    /* Inputs */
    int32   job;
    float64 current_position[3];

    /* Outputs */
    float64 desired_position[3];
    float64 scan_ccw;
    float64 scan_cw;
} FORMDB_DATA;
```

The input fields of FORMDB_DATA are job and current_position, and they are used as input to the formationdb query routines. The remaining fields are output fields set by the query function, as follows:

desired_position is the location that this unit should should be at in order to be in formation.

scan_ccw and scan_cw define algorithmically generated scan sectors for the unit, represented as counter-clockwise and clockwise radians in vehicle coordinates. Zero is out the front of the vehicle and increases positively counter-clockwise. The scan limits are interpreted the same way as the slew limits are interpreted in libPhysDB (see section 'physdb_nearest_angle' in LibPhysDB Programmer's Manual). The scan sectors generated for vehicles in a formation are typical for that vehicles in that formation.

Routes for station keeping are represented via the ROUTE_POINTS data structure in 'stdroute.h'

Output routes generated by the formdb_create_routes routine will have point_id which reference the indices of the input route. Also, because of input route sampling, some input route point indices may never be referenced by a generated output route.

1.4 Examples

The test programs 'ftest.c' and 'rtest.c' demonstrate the use of LibFormationDB to generate the initial placement of units and the generation of station-keeping routes, respectively.

2 Functions

The following sections describe each function provided by libFormationDB, including the format and meaning of its arguments, and the meaning of its return values (if any).

2.1 formdb_init

```
int32 formdb_init(directory, flags, ctdb)
    char  *directory;
    uint32 flags;
    CTDB  *ctdb;
```

'directory'
: Specifies the directory where the formdb file is expected

'flags'
: Specifies reader options (see section 'reader_read' in LibReader Programmer's Manual)

'ctdb'
: Specifies a CTDB terrain database to use for checking formations against water obstacles.

formdb_init initializes libformdb, causing it to read its data file 'formdb.rdr' from the specified directory. The flags are as in reader_read. The return value is zero if the read succeeds, or one of the libreader return values: READER_READ_ERROR READER_FILE_NOT_FOUND.

Note that the libreader function reader_init must be called before this function.

2.2 formdb_expand

```
int32 formdb_expand(formation_name, spacing, direction, location,
                    optimize, n_units, unit_data, assignment_key)
    char        *formation_name;
    float64      spacing;
    float64      direction;
    float64      location[3];
    int32        optimize;
    int32        n_units;
    FORMDB_DATA unit_data[];
    int32       *assignment_key;
```

'**formation name**'
Specifies the name of the formation. This should be a libreader symbol.

'**spacing**' Specifies a desired spacing between units, in meters

'**direction**'
Specifies the desired direction of the units in the formation, in radians. 0 is interpreted as north, and directions increase positively counter-clockwise.

'**location**'
Specifies the desired location of the center of mass of the formation. If it is equal to [0.0, 0.0, 0.0], then the center of mass will remain unchanged.

'**optimize**'
Specifies whether to allow reordering of the desired formation to minimize unit crossing and unit movement.

'**n_units**' Specifies the number of units in the **unit_data** array.

'**unit_data**'
Specifies and returns data for the units. Desired vehicle placement and scan sector information will be returned in this data.

'**assignment_key**'
If **optimize** is **TRUE**, an index for the optimal assignment chosen for the formation will returned here. If **optimize** is **FALSE**, the assignment specified by **assignment_key** will be used. **assignment_key** can be **NULL**, in which case the first assignment specified in the '**formdb.rdr**' database will be used if **optimize** is **FALSE**.

formdb_expand calculates desired positions, station-keeping information and scan sector information for the units represented in the passed in **unit_data** to be in the specified **formation**, **spacing** and **direction**. Note that the input fields of the the **unit_data** must be filled out prior to this call.

assignment_key can be used to record an initial optimal assignment and to reuse that assignment in later calls to **formdb_expand**.

A return value of 0 means that the routine succeeded. A return value of -1 indicates failure. The only source of failure is an unrecognized formation name. In the case of failure, each unit's **desired_position** will be equal to its **current_position**, and nominal values will be supplied for scan sectors.

2.3 formdb_create_routes

```
int32 formdb_create_routes(formation_name, spacing, n_units, unit_data,
                           input_route, sampling_distance,
                           output_routes, input_following_unit,
                           assignment_key)
     char         *formation_name;
     float64       spacing;
     int32         n_units;
     FORMDB_DATA   unit_data[];
     ROUTE_POINTS *input_route;
     float64       sampling_distance;
     ROUTE_POINTS  output_routes[];
     int32        *input_following_unit;
     int32         assignment_key;
```

'formation name'
Specifies the name of the formation. This should be a libreader symbol.

'spacing' Specifies a desired spacing between units, in meters

'n_units' Specifies the number of units in the **unit_data** array.

'unit_data'
Specifies job and current location data for the units. The return fields scan_ccw and scan_cw will be set.

'input_route'
Specifies the intput route as an ROUTE_POINTS structure (see '**stdroute.h**')

'sampling_distance'
Specifies the minimum distance between points in the input route that will be used to create waypoints for the output routes.

'output_routes'
Returns the routes that all the units in the **unit_data** should follow to stay in formation. On input, this should be an array of uninitialized **ROUTE_POINTS**. On output, this array will contain data with each **ROUTE_POINTS** structure containing an initialized allocated array of **ROUTE_POINT**. This memory must be deallocated when no longer needed via **STDDEALLOC(output_route[i].points)**.

'input_following_unit'
Returns the index into the **unit_data** array of the unit which, due to the assignment of units in a formation, could follow the input route and remain in formation. If no such unit exists, **-1** will be returned.

'assignment_key'
Specifies the desired job position relationships (or to choose an optimal relationship, if -1)

formdb_create_routes applies the specified **formation** and **spacing** to the units in **unit_data**

and the **input_route** and generates **output_routes** for every unit. The output routes will be similar to the input_route except for:

1. route-point displacements to account for station-keeping offsets,
2. fewer points to smooth out input points that are too close together for followers to check-point against. Input points that are closer to each other than **sampling_distance** will be ignored in the generation of the output routes.

Every **ROUTE_POINT** in an output route will have a **point_id** referring to one of the original **point_id** indices in the **input_route**. Note that this routine is not suitable for generating road following routes, since waypoints for followers are not preserved.

Formations with an even spread and an odd number of units may contain a unit which could follow the input route, as opposed to the generated output route. This is reported in **input_following_unit**. Formations with an even number of vehicles typically generate output routes which straddle the input route and will have a **input_following_unit** of **-1**.

A return value **>=** 0 means that the routine succeeded, and indicates the assignment order which was chosen (see Section 2.2 [formdb`expand], page 7). A return value of **-1** indicates failure. The only sources of failure are an unrecognized formation name or an input route that is contains less than two waypoints. In the case of failure, the output routes will be identical to the input routes, and nominal values will be supplied for scan sectors.

Note that data allocated within the **output_routes** array must be freed when no longer in use.

2.4 formdb_generate_roadmarch_order

```
void formdb_generate_roadmarch_order(n_units, unit_data, order_array)
    int32         n_units;
    FORMDB_DATA   unit_data[];
    int32         order_array[];
```

'**n_units**' Specifies the number of units in the **unit_data** array.

'**unit_data**'

Specifies job and current location data for the units. The return fields of scan_ccw and scan_cw will be filled out.

'**order_array**'

Returns the order of march. Each element of this array is an index into the **unit_data** array.

Given **unit_data**, **formdb_generate_roadmarch_order** return the order on which the units should follow one another on the road. This is currently defined as the preferred column order. The order of march is defined by the order of indices returned in **order_array**.

2.5 formdb_occupy_area

```
void formdb_occupy_area(radius, location, n_units, unit_data, directions)
    float64     radius;
    float64     location[3];
    int32       n_units;
    FORMDB_DATA unit_data[];
    float64     directions[];
```

'radius' Specifies the desired radius of the area

'location'
Specifies the desired location of the center of mass of the formation. If it is equal to [0.0, 0.0, 0.0], then the center of mass will remain unchanged.

'n_units' Specifies the number of units in the **unit_data** array.

'unit_data'
Specifies and returns data for the units. Desired vehicle placement and scan sector information will be returned in this data.

'directions'
Returns the directions that each unit should face.

formdb_occupy_area returns desired positions for the units to be on a circle of a given **radius** at a **location** to provide for all around security.

2.6 formdb_herringbone

```
void formdb_herringbone(spacing, direction, location, n_units, unit_data)
    float64     spacing;
    float64     direction;
    float64     location[3];
    int32       n_units;
    FORMDB_DATA unit_data[];
```

'**spacing**' Specifies a desired spacing between units, in meters

'**direction**'
Specifies the desired direction of the units in the formation, in radians. 0 is interpreted as north, and directions increase positively counter-clockwise.

'**location**'
Specifies the desired location of the center of mass of the formation. If it is equal to [0.0, 0.0, 0.0], then the center of mass will remain unchanged.

'**n_units**' Specifies the number of units in the **unit_data** array.

'**unit_data**'
Specifies and returns data for the units. Desired vehicle placement and scan sector information will be returned in this data.

formdb_herringbone returns desired positions for the vehicles to get off of the road from column formation. For simplicity, this is currently modeled just as a staggered column. Because there is no road information in formdb, there is no guarantee that this actually gets the units off the road.

LibFwa

Table of Contents

1 Overview

Libfwa implements an instance of the hull class of components. It provides a low-fidelity model of fwa vehicle dynamics. Capabilities are modeled only to the second order (maximum velocity, maximum acceleration).

The parameters of a fwa vehicle are specified in its configuration file as follows:

```
(fwa (c_drag_super <float >)
     (c_drag_sub <float >)
     (vehicle_mass <float> kg)
     (thrust_min <float> N)
     (thrust_max <float> N)
     (lift_min <float> N)
     (lift_max <float> N)
     (side_min <float> N)
     (side_max <float> N)
     (induced_drag_factor <float> )
     (takeoff_alt <float m>)
     (speed_tau <float sec>)
     (fpa_tau <float sec>)
     (track_tau <float sec>)
     (roll_tau <float sec>)
     (aoa_tau <float sec>)
     (fuel_usage <float kg/m>)
     (airplane_drag_index <float >)
     (takeoff_speed <float mps2>)
     (thrust_map <char[32] filename )
     (hard_turn_rate <float portion of current maximum>)
     (standard_turn_rate <float degrees/second>)
     (easy_turn_rate <float degrees/second>)
     (hard_climb_rate <float portion of current maximum>)
     (standard_climb_rate <float portion of current maximum>)
     (easy_climb_rate <float portion of current maximum>)
     (fuel_usage  ;; All values floats, list in increasing order
                 (            <percent>     <percent>      ...)
                 (<altitude>  <liter/sec>   <liter/sec>    ...)
                 (<altitude>  <liter/sec>   <liter/sec>    ...)
                 (...         ...           ...            ...))
)
```

Applications interface to the fwa model primarily through the libhulls interface. The most common interface for controlling vehicle motion in the air is HULLS_SET_FLY_LEVEL. Libfwa will do the normalization only if it is necessary (for example, if the vehicle is already pointing the right way, no normalization is needed).

Libfwa supports only one instantiation per vehicle (i.e., a vehicle may not have more than one fwa hull).

The following equations describe the dynamics of the fixed wing hull.

```
air_density = initial_air_density * exp(-Z/HR)
initial_air_density = .0249
HR = 34,602.5 - .14604Z
Z is the altitude of the vehicle

thrust = (mass * gravity * sin(flight_path_angle) +
                mass * speed_rate_goal + drag)/cos(angle_of_attack)
track_force = mass * (speed)^2 * cos(flight_path_angle) *
                track_rate_goal
flight_path_angle_force = mass * speed *
                flight_path_angle_rate_goal     +
                mass * gravity * cos(flight_path_angle)
roll_angle = arctan(track_force/flight_path_angle_force)

normal force = sqrt(flight_path_angle_force^2 + track_force^2)
Coef_lift = 21,522.3
lift_goal = normal_force / (1.0 + (2.0 * thrust)/
                                air_density * Coef_lift * speed^2)
angle_of_attack = 2 * lift_goal /
        (air_density * speed^2 * Coef_lift)
lift = (air_density * speed^2 * Coef_lift * angle_of_attack)/2


coef_sub_or_super : coef_drag_subsonic when speed < speed of sound
                  : coef_drag_supersonic otherwise
coef_drag : drag coef of weapons + drag coef of airplane
drag = air_density * speed^2 * coef_sub_or_super * coef_drag +
        lift * angle_of_attack;


speed_rate = (thrust * cos(angle_of_attack) - drag) / mass * gravity
        * sin(flight_path_angle);
flight_path_angle_rate = (lift + thrust * sin(angle_of_attack) +
                cos(roll_angle) + side * sin(roll_angle))/speed
track_rate = (lift + thrust + sin(angle_of_attack)) * sin(roll_angle)
                / mass * speed * cos(flight_path_angle)
```

Applications interface to the fixed wing aircraft (fwa) model primarily through the libhulls interface. The libhulls library defines a common set of functions (and the semantics of those functions) which are invoked on instances of the hulls class (such as those instantiated by libtracked, libfwa, or libmissile).

It is possible to modify the fwa model by changing an exisiting hulls interface function or by adding a completely new function. To modify an existing hulls interface function requires the following actions:

1. If the change occurs only in the function body, a change to the function code in the libfwa library is all that is needed. If the change occurs to the function's argument list, change the function code in the libfwa library and change the hulls interface structure definition found in libhulls.h. Also to maintain the common hulls interface, change the code for the modified function in any other hull specific component library (such as libtracked and libmissile).
2. Recompile ModSAF.

To add an additional libfwa function to the current model requires the following actions:

1. Write the function as part of the libfwa library. The function is written in the code which manages the libfwa class information attached to each vehicle (fwa_class.c).
2. Add the function and its declaration to any of the other hull specific component libraries. This maintains the common hulls interface.
3. In the libfwa source code that handles libhull initialization processing, include a **function_number, function** entry identifying the new function for the **cmpnt_define_instance** function and every other hull instance library (libtracked, libmissile, etc.)
4. In libhulls.h, add an entry to identify the new macro and associate it with a function code number. This new addition means that the number of hulls functions must be incremented by one. The hulls interface structure definition that appears in libhulls.h must include a structure to define the new function's argument list.
5. Recompile ModSAF.

To replace this fwa model with a completely different one requires the following actions:

1. Decide on the get functions and set functions that would be required in the new model. Try to map these needed functions to the existing hulls interface. A function can map if its argument list can remain the same. Functions that can not map must be added to the hulls interface.
2. For those functions that can map to the existing hulls interface but whose code body you want to change, edit the code for the function in the libfwa source file that contains the code to manage the libfwa class information (such as fwa_class.c).
3. For those functions that can't map to the existing hulls interface, add an additional function to the hulls interface. The addition procedure was described above.
4. Recompile ModSAF.

If an interface function is no longer needed, it is possible but not required, to remove it. Deletion of an interface function is only allowed when that function is not needed in any of the specific component libraries,

The deletion process requires these steps:

1. Delete the function code from each specific component library.
2. In the generic component library, remove the "function_number, function" entry identifying the excess function in the "cmpnt_define_instance" function call. This function call is found in the library's initialization code segment. In the library's public header file, remove the entry for the excess macro and its associatiated function code number. Decrease the number of interface functions by one. Delete the structure that defines the excess function's argument list in the interface structure definition.
3. Recompile ModSAF.

2 Examples

To get the component number of my hull:

```
extern int32 my_hull;

if ((my_hull = cmpnt_locate(vehicle_id, reader_get_symbol("hull"))) ==
    CMPNT_NOT_FOUND)
  printf("Vehicle %d does not seem to have a hull\n", vehicle_id);
```

To then give a command to that hull:

```
if (my_hull != CMPNT_NOT_FOUND)
  HULLS_SET_DIRECTION_SPEED(vehicle_id, hull, dirvec, speed, 0.0, 0.0);
```

3 Functions

The following sections describe each function provided by libfwa, including the format and meaning of its arguments, and the meaning of its return values (if any).

3.1 fwa_init

```
void fwa_init()
```

fwa_init initializes libfwa. Call this before calling any other libfwa functions.

3.2 fwa_class_init

```
void fwa_class_init(parent_class)
    CLASS_PTR parent_class;
```

'**parent_class**'
Class of the parent (declared with **class_declare_class**).

fwa_class_init creates a handle for attaching fwa class information to vehicles. The parent_class is one created with class_declare_class.

3.3 fwa_tick

```
void fwa_tick(vehicle_id, ctdb)
    int32 vehicle_id;
    CTDB  *ctdb;
```

'**vehicle_id**'
Specifies the vehicle ID

'**ctdb**' Specifies the terrain database

fwa_tick ticks the fwa hull dynamics model.

3.4 fwa_collision

```
void fwa_collision(vehicle_id, position, coll_type,
                        other_id, other_mass, other_velocity)
    int32   vehicle_id;
    float64 position[3];
    uint32  coll_type;
    int32   other_id;
    float64 other_mass;
    float64 other_velocity[3];
```

'vehicle_id'
: Specifies the vehicle ID

'position'
: Specifies the position of impact in world coordinates

'coll_type'
: Specifies the type of collision these values are defined in the library libcollision.

'other_id'
: Specifies the vehicle ID of the other party (or 0 if terrain)

'other_mass'
: Specifies the mass of the other party

'other_velocity'
: Specifies the velocity of the other party

fwa_collision tells the fwa hull dynamics model that a collision occured. The coll_type should be one of the libcollision constants:

COLL_TREES
: Indicates crossing a treeline or canopy edge.

COLL_BUILDINGS
: Indicates crossing a building or other structure. If the other structure is represented on the network, the vehicle ID of that structure should be provided.

COLL_GROUND
: *Should not be checked for ground vehicles.*

COLL_PLATFORMS
: Indicates intersecting a platform (vehicle, DI, etc.).

COLL_MISSILES
: Indicates intersecting a missile (an entity on the network with a munition type.

fwa_collision sets the vehicles appearance to smoking and flaming. If the collision is with the ground, the vehicles appearance is also set to destroyed.

3.5 fwa_damage

```
    void fwa_damage(vehicle_id, damage)
        int32 vehicle_id;
        int32 damage;
```

'vehicle_id'
Specifies the vehicle ID

'damage' Specifies whether the fwa dynamics should simulate being damaged

fwa_damage tells the fwa hull dynamics model that it is damaged (or not) depending on the boolean value of the **damage** flag.

Libgenradio

Table of Contents

1 Overview

LibGenRadio provides rudimentary radio communications to ModSAF entities. It allows transmission of ASCII strings using the radio protocols of SIMNET and DIS, including issuance of transmitter and signal PDU's.

Initially, functionality is limited to vehicles sending brief messages, which are then received and displayed by the GUI.

1.1 Examples

```
/* currently in main.c */ grad_init(data_dir, READER_OVERRIDE, valve, 0);

/* currently in so_init.c */ grad_class_init(safobj_class);

/* currently in so_local.c */ grad_create(vehicle_id, (GENRADIO_PARAMETRIC_DATA *)parms);

/* currently done from tasks */ grad_send_text(vehicle_id, "Standing by");

/* currently done from so_local.c */ grad_destroy(vehicle_id);
```

2 Functions

The following sections describe each function provided by libgenradio, including the format and meaning of its arguments, and the meaning of its return values (if any).

2.1 grad_init

```
void grad_init(data_dir, flags, valve, protocol)
    char         *data_dir;
    uint32        flags;
    PV_VALVE_PTR  valve;
    int32         protocol;
```

'data_dir'
Path to data files.

'flags' Flags to pass to libreader.

'valve' Libpktvalve handle to use for network access.

'protocol'
Protocol version in effect for this run of ModSAF.

grad_init initializes LibGenRadio. Call this before any other libgenradio function.

2.2 grad_class_init

```
void grad_class_init(parent_class)
    CLASS_PTR parent_class;
```

'parent_class'
Class of the parent (declared with class_declare_class)

grad_class_init creates a handle for attaching genradio class information to vehicles. The parent_class will likely be safobj_class.

2.3 grad_create

```
void grad_create(vehicle_id, params)
    int vehicle_id;
    GENRADIO_PARAMETRIC_DATA *params;
```

'vehicle_id'
Specifies the vehicle ID

'params' Specifies initial parameter values

grad_create creates the genradio class information for a vehicle and attaches it vehicle's block of libclass user data.

There are currently no parametric data.

2.4 grad_destroy

```
void grad_destroy(vehicle_id)
    int vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

grad_destroy frees the genradio class information for a vehicle. This should be called before freeing the class user data with **class_free_user_data**.

2.5 grad_send_text

```
void grad_send_text(vehicle_id, string)
    int32 vehicle_id;
    char *string;
```

'vehicle_id'
Specifies the vehicle

'string' Points to the text to be sent over the radio.

grad_send_text initiates transmission of the specified text over the vehicle's radio. The length of the string sent is truncated to **GRAD_MAX_MSG_LEN**, which is defined in libgenradio.h. An internal copy of the text is maintained, so the memory indicated by **string** may safely be altered or deallocated any time after this function returns.

2.6 grad_subscribe

```
void grad_subscribe(vehicle_id, message_queue)
    int32       vehicle_id;
    QUEUE_QUEUE message_queue;
```

'**vehicle_id**'
  Identifies the vehicle containing the radio

'**ctdb**' Specifies the queue for incoming messages

grad_subscribe requests that incoming messages be enqueued on the passed queue. The messages are placed in the queue using a **GRAD_MESSAGE** structure:

```
typedef struct grad_message
{
    uint32 time_received;
    int32  sender_id;
    char   text[];
} GRAD_MESSAGE;
```

Only ModSAF generated ASCII text messages will be enqueued.

Libgenradio will take care of the deallocation of the queue when the vehicle is destroyed; however, it is the responsibility of the caller to flush any messages remaining on the queue.

2.7 grad_tick

```
void grad_tick(vehicle_id, ctdb)
    int32 vehicle_id;
    CTDB *ctdb;
```

'**vehicle_id**'
  Identifies the vehicle containing the radio to be ticked.

'ctdb' Specifies the terrain database

grad_tick checks radio state and issues PDU's as required.

ADST-TR-W003268

LibGenturret

Table of Contents

1 Overview

Libgenturret implements an instance of the turret class of components. It provides a low-fidelity model of generic turret dynamics and capabilities. Turrets that can not support 360 degree slewing (as reported in libphysdb) are supported. Turrets can be described as having either a continuous range of slew rates or set discrete rates.

The parameters of a generic turret are specified in the configuration file for the vehicle containing such a turret as follows

```
(generic-turret (physdb-name <name>)
(rates continuous <min-slew deg/sec> <max-slew deg/sec>))
    OR
(generic-turret (phsydb-name <name>)
                (rates discrete <rate1 slew deg/sec>
                                <rate2 slew deg/sec>
                                <rate3 slew deg/sec>
                                ;; rates must be monotonically increasing
                                ...))
```

The <name> must match the name of the turret as specified in the libphysdb database. The rates are in degrees per second. For continuously slewable turrets, the minimum slew rate is the slowest the turret can slew without stopping, and will frequently be 0. For discretely slewable turrets, the rates are the various speeds the turret can slew.

Applications interface to the generic turret model primary through the libturrets interface. The most efficient interface for controlling turret azimuth is TURRETS_SET_AZIMUTH.

Libgenturret supports up to 4 instantiations per vehicle (i.e., a vehicle can have up to 4 generic turrets).

The libturrets library defines a common set of functions (and the semantics of those functions) which are invoked on instances of the turrets class (such as those instantiated by libgenturret).

It is possible to modify the generic turret model by changing an exisiting turret interface function or by adding a completely new interface function. The process of modification of an existing libgenturret function is fairly simple when the change occurs only in the function body. In that case, the programmer would only need to change the function code in the libgenturret library. The process of modification of an existing libgenturret function is more complicated when the change occurs to the function's argument list. In that case, the programmer would need to change both the function code in the libgenturret library and the turrets interface structure definition found in libturrets.h. Currently libgenturret is the only turret specific component, but if there were more, the programmer would also need to change the code for the modified function in those libraries to maintain the common turrets interface. When all these changes have been made to the source code, ModSAF would need to be recompiled.

To add an additional interface function to the current model, a programmer would need to perform the following actions:

1. Write the function as part of the libgenturret library. The function is written in the code that manages the libgenturret class information attached to each vehicle (gtur_class.c).
2. Currently libgenturret is the only turret specific component, but if there were others, the programmer would need to add the function and its declaration to their libraries to maintain the common turrets interface.
3. In the libturrets source code that handles libturrets initialization processing, include a **`function_number, function`** entry identifying the new function for the **`cmpnt_define_instance`** function.
4. In libturrets.h, add an entry to identify the new macro and associate it with a function code number. Increment the number of turret interface functions by one. Include a structure to define the new function's argument list in the turrets interface structure definition.
5. Recompile ModSAF.

To replace this genturret model with a completely different one would require the following actions:

1. Decide on the interface functions that would be required in the new model. Try to map these needed functions to the existing turrets interface. A function can map if its argument list can remain the same. Functions that can not map must be added to the turrets interface.
2. For those functions that can map to the existing turrets interface but whose code body you want to change, edit the code for the function in the libgenturrets source file that contains the code to manage the libgenturret class information (gtur_class.c).
3. For those functions that can't map to the existing turrets interface, add an additional function to the turrets interface. The addition procedure was described above.
4. Recompile ModSAF.

Since libgenturret is the only specific turret library, it would be safe, but not required, to remove any model functions that are no longer needed. This deletion of functions would be more problemetic if there were multiple turret models. In that case it would be necessary to check that the function was not needed in one of the other specific component libraries. To remove an unnecessary libgenturret function from the current model, a programmer would need to perform the following actions:

1. Delete (or comment out) the function code from the libgenturret library (see gtur_class.c). If there were other specific turret component libraries, the function would be deleted from those libraries as well.
2. In the libturrets initialization code, remove the "function_number, function" entry identifying the excess function in the "cmpnt_define_instance" function call.
3. In libturrets.h, remove the entry that identifies the excess macro and its associatiated function code number. Delete the number of turret interface functions by one. Delete the structure that defines the excess function's argument list.
4. Recompile ModSAF.

2 Examples

To get the component number of a turret with a particular name (such as "primary-turret"):

```
int32 turret;

if ((turret = cmpnt_locate(vehicle_id, name)) ==
    CMPNT_NOT_FOUND)
  printf("Vehicle %d does not seem to have a turret called \"%s\".\n",
         vehicle_id,
         name);
```

To then give a command to that turret (the macro is defined by libturrets; it assembles a **TURRETS_INTERFACE** structure, and calls **cmpnt_invoke**):

```
if (turret != CMPNT_NOT_FOUND)
  TURRETS_SET_AZIMUTH(vehicle_id, turret, azimuth);
```

3 Functions

The following sections describe each function provided by libgenturret, including the format and meaning of its arguments, and the meaning of its return values (if any).

3.1 generic_turret_init

```
void generic_turret_init()
```

generic_turret_init initializes libgenturret. Call this before calling any other libgenturret functions.

3.2 generic_turret_class_init

```
void generic_turret_class_init(parent_class)
    CLASS_PTR parent_class;
```

'**parent_class**'

Class of the parent (declared with **class_declare_class**).

generic_turret_class_init creates a handle for attaching generic_turret class information to vehicles. The parent_class is one created with class_declare_class.

3.3 generic_turret_tick

```
void generic_turret_tick(vehicle_id)
    int32 vehicle_id;
```

'**vehicle_id**'

Specifies the vehicle ID

generic_turret_tick ticks the generic turret dynamics model.

3.4 generic_turret_damage

```
void generic_turret_damage(vehicle_id, damage)
    int32 vehicle_id;
    int32 damage;
```

'vehicle_id'
Specifies the vehicle ID

'damage' Specifies whether the generic turret should simulate being damaged

generic_turret_damage tells the generic turret model that it is damaged (or not) depending on the boolean value of the **damage** flag.

LibGraphics

Table of Contents

1 Overview

LibGraphics implements a C2 subclass for the display and editing of persistent objects. It handles the display of points, lines, text, and taskframes, and the editing of points, lines, and text. Each class of object has a corresponding sensitive class, which is used when the graphic is displayed. LibGraphics also allows other libraries to register their own sensitive classes which are used when the displaying text associated with objects of other classes (such as Units).

2 Usage

The software library '`libgraphics.a`' should be built and installed in the directory '`/common/lib/`'. You will also need the header file '`libgraphics.h`' which should be installed in the directory '`/common/include/libinc/`'. If these files are not installed, you need to do a '`make`' in the libgraphics source directory. If these files are already built, you can skip the section on building libgraphics.

2.1 Building Libgraphics

The libgraphics source files are found in the directory '`/common/libsrc/libgraphics`'. '`RCS`' format versions of the files can be found in '`/nfs/common_src/libsrc/libgraphics`'.

If the directory '`common/libsrc/libgraphics`' does not exist on your machine, you should use the '`genbuild`' command to update the common directory hierarchy.

To build and install the library, do the following:

```
# cd common/libsrc/libgraphics
# co RCS/*,v
# make install
```

This should compile the library '`libgraphics.a`' and install it and the header file '`libgraphics.h`' in the standard directories. If any errors occur during compilation, you may need to adjust the source code or '`Makefile`' for the platform on which you are compiling. libgraphics should compile without errors on the following platforms:

- Mips
- SGI Indigo
- Sun Sparc

2.2 Linking with Libgraphics

Libgraphics can be linked into an application program with the following link time flags: '`ld [source .o files] -L/common/lib -lgraphics [other libraries]`'. If your compiler does not support '-L' syntax, you can use the archive explicitly: '`ld [source .o files] /common/lib/libgraphics.a`'.

Libgraphics depends directly on the following libraries: libselect, libprivilege, libsafgui, libtactmap, libcoordinates, libsensitive, libeditor, libreader, libclass, and libpo.

3 Functions

The following sections describe each function provided by libgraphics, including the format and meaning of its arguments, and the meaning of its return values (if any).

3.1 grph_init

```
void grph_init()
```

grph_init initializes libgraphics. Call this function before calling any other libgraphics functions.

3.2 grph_create_editors

```
int32 grph_create_editors(data_path, reader_flags,
                          gui, tactmap, tcc, map_erase_gc,
                          sensitive, refresh_event, db, select)
    char                *data_path;
    int32                reader_flags;
    SGUI_PTR             gui;
    TACTMAP_PTR          tactmap;
    COORD_TCC_PTR        tcc;
    GC                   map_erase_gc;
    SNSTVE_WINDOW_PTR    sensitive;
    CALLBACK_EVENT_PTR   refresh_event;
    PO_DATABASE         *db;
    SELECT_TOOL_PTR      select;
```

'**data_path**'
Specifies the directory where data files are expected

'**reader_flags**'
Specifies flags to be passed to **reader_read** when reading data files

'**gui**' Specifies the SAF GUI

'**tactmap**' Specifies the tactical map

'**tcc**' Specifies the map coordinate system

'**map_erase_gc**'
Specifies the GC which can erase things from the tactical map

'sensitive'
Specifies the sensitive window for the tactical map

'refresh_event'
Specifies the event which fires when the map is refreshed

'db' Specifies the persistent object database

'select' Specifies the select tool

grph_create_editors creates the graphics editors. The data file ('graphics.rdr') is read either from '.' or the specified data path, depending upon the reader_flags. The reader_flags are as in reader_read. The return value is zero if the read succeeds, or one of the libreader return values: READER_READ_ERROR, READER_FILE_NOT_FOUND.

3.3 grph_register_association

```
void grph_register_association(gui, po_class, snstve_class)
    SGUI_PTR              gui;
    PersistentObjectClass po_class;
    SNSTVE_CLASS          *snstve_class;
```

'gui' Specifies the SAF GUI

'po_class'
Specifies the persistent object class (objectClassUnit, etc.)

'snstve_class'
Specifies the sensitive class

grph_register_association tells the graphics editors the sensitive class which should be used when displaying text associated with objects of the specified PO class. This should be called *after* all graphics editors have been created.

3.4 grph_edit_text

```
void grph_edit_text(gui, id)
    SGUI_PTR  gui;
    ObjectID *id;
```

'gui' Specifies the GUI

'id' Specifies the ID of the text to edit

grph_edit_text starts up the text editor for the passed text object. This function is provided to allow editing of the text associated with an object. Note that the caller is responsible for making sure the GUI is in a state where the text editor can be started (i.e., another OBJECT_MODE editor is not running).

3.5 grph_class_init

```
void grph_class_init(parent_class)
    CLASS_PTR parent_class;
```

'parent_class'
Specifies the parent class (probably c2obj_class)

grph_class_init creates a handle for attaching graphics class information to entries. The **parent_class** is one created with **class_declare_class**.

3.6 grph_create

```
void grph_create(entry)
    PO_DB_ENTRY *entry;
```

'entry' Specifies the graphic entry

grph_create creates the graphics class information for a entry and attaches it to the entry's libclass user data. This function simply returns if no graphics editors have been created (running without a GUI), or the entry is not of a class for which libgraphics is responsible.

3.7 grph_destroy

```
void grph_destroy(entry)
    PO_DB_ENTRY *entry;
```

'entry' Specifies the graphic entry

grph_destroy frees the graphics class information for a entry.

3.8 grph_changed

```
void grph_changed(entry)
    PO_DB_ENTRY *entry;
```

'**entry**' Specifies the graphic entry

grph_changed updates displayed graphic in response to a libpo **object_changed** event.

3.9 grph_overlay_changed

```
void grph_overlay_changed(entry)
    PO_DB_ENTRY *entry;
```

'**entry**' Specifies the graphic entry

grph_overlay_changed updates displayed graphic in response to a libpo **object_changed** event for the graphic's overlay.

LibGuns

Table of Contents

1 Overview

Guns is a SAF components class. The purpose of a components class is to define a common set of functions which are invoked on instances of that class, and the semantics of those functions. Other than defining these functional semantics, components classes don't actually *do* anything.

Access to gun functions is achieved through macros defined by libguns. These macros invoke ‘cmpnt_invoke’ with a code number which identifies the function to run. Libcomponents then runs this function for the particular gun mode via a jump table.

The table below shows how the gun component relationships have been currently implemented via the ModSAF library structure.

specific libraries	generic library	architectural library
libbalgun	libguns	libcomponents
libmlauncher	libguns	libcomponents

As mentioned above, libguns requires the services of libcomponents, an architectural library which provides a level of abstraction away from the specific gun component interfaces. When the ModSAF application gets set up to run, the libguns initialization process directs libcomponents to define a gun component class. This information enables libcomponents to define a structure to accommodate all of the gun instantiations a simulated object is allowed to have. The libguns initialization process also tells libcomponents the number of its defined gun interface functions. This enables a simulated object's user data to be allocated enough space to hold the address of each of the interface functions defined in libguns.

The parametric data of libcomponents identifies each component that needs to be modeled when a vehicle is simulated. For example, a component entry for a T72 tank might look like this: (see the file named USSR_T72M_params.rdr)

```
(SM_Components (hull         SM_TrackedHull)
               (turret       SM_GenericTurret)
               (machine-gun  [SM_BallisticGun | 0])
               (main-gun     [SM_BallisticGun | 1])
               (visual       SM_Visual))
```

A T72M simulated vehicle (which belongs to the safobj class) will have component sub-class data which tells the ModSAF software to maintain a structure that includes one libbalgun instantiation for the machine gun and one libbalgun instantiation for the main gun.

Since an application will interface to libbalgun or libmlauncher through libguns, a tank's main gun shooting control commands (which are performed by libbalgun) and an airplane's missile launcher commands (which are performed by libmlauncher) are both issued via the interface defined by libguns. A command to load ammunition is therefore the same whether the object is loading a main gun, machine gun, or missile launcher. What is different is the type or ammunition to load and that is passed as input to the gun function. Similarly, an application can obtain information about the state of any of its guns though the libguns interface. The table below shows the relationship between the specific and generic library for the guns component.

Instantiations of of the library:	Belong to generic component class:	Have a command interface defined in:
libbalgun	guns	libguns
libmlauncher	guns	libguns

The interface to libguns is defined in its public header file (libguns.h). This interface lets an application set gun controls or get gun information without knowing which specific gun model is being used. Applications interface to the ballistic gun model or missile launcher model primarily through the macros defined in libguns. These macros map to functions which are invoked on instances of the guns sub-class (such as the libmlauncher component instantiated for an airplane or the libbalgun component instantiated for a tank).

One interface for controlling a gun is the macro **GUNS_SET_UNLOAD_MUNITION** which maps to a function which starts the procedure of transfering a given number of loaded munitions to a storage bin. A possible definition for this macro is shown below.

```
#define GUNS_SET_UNLOAD_MUNITION(_vid, _cnum, _store, _quantity)
{
    GUNS_INTERFACE _gif;
    _gif.u.set_unload_munition.store    = _store;
    _gif.u.set_unload_munition.quantity = _quantity;
    cmpnt_invoke(GUNS_SET_UNLOAD_MUNITION_FCN, _vid, _cnum, (ADDRESS)&_gif);
}
```

The **GUNS_INTERFACE** structure defined in libguns.h is the structure which is passed to any gun function. This structure is a union of structures that each define an argument list for a gun interface function. An abbreviated example that assumes there are only two gun functions is shown below. Typically there will be many interface functions and therefore more structure definitions in the union. The macros hide this structure from the users of these functions.

```
typedef struct guns_interface
```

```
{
    union
    {
        struct guns_set_unload_munition
        {
            int32 store;
            int32 quantity;
        } set_unload_munition;
        struct guns_get_loaded_munition
        {
    uint32 munition;
            int32  quantity;
         } get_loaded_munition;
    } u;
} GUNS_INTERFACE;
```

Issuing a command to an objects's gun component is done by invoking one of the macros defined in libguns. These macros identify the specific component function which needs to be called. For example, invoking the **GUNS_SET_UNLOAD_MUNITION** macro will result in the calling of the specific component function named set_unload_munition. In the public header file of each generic library, macros are associated with a function code number so that a call to the libcomponents library (via the cmpnt_invoke function) will dispatch a call to the appropriate function. The specific component functions are defined and installed by the specific libraries (libbalgun and libmlauncher). In this case, both libraries install a function with the same name, "set_unload_munition" (there is no name conflict because each function is declared static). It is the specific function (either libbalgun's set_unload_munition or liblmauncher's set_unload_munition) which is called when the macro is invoked.

Invoking the macro results in two actions: (1) setting up of the interface structure and (2) passing of necessary information to libcomponent. The macro passes the vehicle id, component number, and function pointer index to libcomponent so that the appropriate library (libbalgun or libmlauncher) data can be accessed. The requested function can require input (such as a storage bin to unload the munition into and a quantity to transfer into the storage bin) and/or output (such as a setting) . Therefore, libcomponents must also be passed the address of the interface structure that holds this data.

2 Examples

To initialize libbalgun, an instance of the gun class which provides for up to BGUN_MAX_GUNS guns per entity:

```
int32 i;
char buf[256];

for (i = 0; i < BGUN_MAX_GUNS; i++)
{
    (void) sprintf(buf, "gun%d", i);
    bgun_user_data_handle[i] =
      class_reserve_user_data(parent_class, buf, bgun_print);
}

/* Tell libcomponents we are available. */
cmpnt_define_instance(SM_BallisticGun, BGUN_MAX_GUNS,
                      bgun_user_data_handle,
                      bgun_create, bgun_destroy,
                      GUNS_GET_MAGAZINE_SIZE_FCN, get_magazine_size,
                      GUNS_SET_LOAD_MUNITION_FCN, set_load_munition,
                      GUNS_SET_UNLOAD_MUNITION_FCN, set_unload_munition,
                      GUNS_GET_LOADED_MUNITION_FCN, get_loaded_munition,
                      GUNS_SET_LAUNCHER_POSITION_FCN,set_launcher_position,
                      GUNS_GET_LAUNCHER_POSITION_FCN,get_launcher_position,
                      GUNS_SET_ELEVATION_FCN, set_elevation,
                      GUNS_SET_TARGET_FCN, set_target,
                      GUNS_SET_LOCATION_FCN, set_location,
                      GUNS_GET_TARGET_IS_TRACKED_FCN,get_target_is_tracked,
                      GUNS_GET_LOCATION_IS_TRACKED_FCN,
                      get_location_is_tracked,
                      GUNS_GET_READY_TO_FIRE_FCN, get_ready_to_fire,
                      GUNS_GET_MUNITION_READY_FCN, get_munition_ready,
                      GUNS_SET_FIRE_FCN, set_fire,
                      GUNS_SET_FIRE_AT_TARGET_FCN, set_fire_at_target,
                      GUNS_SET_FIRE_AT_LOCATION_FCN, set_fire_at_location,
                      GUNS_GET_ALLOWED_MUNITIONS_FCN, get_allowed_munitions);
```

To get the component number of a gun with a particular name (such as "main-gun"):

```
int32 gun;

if ((gun = cmpnt_locate(vehicle_id, name)) ==
    CMPNT_NOT_FOUND)
  printf("Vehicle %d does not seem to have a gun called \"%s\".\n",
         vehicle_id,
         name);
```

To then give a command to that gun (the macro is defined by libguns; it assembles a **GUNS_INTERFACE** structure, and calls **cmpnt_invoke**):

```
if (gun != CMPNT_NOT_FOUND)
  GUNS_SET_ELEVATION(vehicle_id, gun, elevation);
```

3 Functions

The following sections describe each function provided by libguns, including the format and meaning of its arguments, and the meaning of its return values (if any).

3.1 guns_init

```
void guns_init();
```

guns_init initializes libguns. Call this function after cmpnt_init, and before any specific gun init functions.

3.2 GUNS_GET_MAGAZINE_SIZE

```
void  GUNS_GET_MAGAZINE_SIZE(_vid, _cnum, _quantity)
    int32  _vid;
    int32  _cnum;
    int32 *_quantity;
```

'_vid' Specifies the vehicle ID

'_cnum' Specifies the gun component number

'_quantity'
Returns the maximum magazine size of munitions loadable

GUNS_GET_MAGAZINE_SIZE returns (by reference) the maximum magazine of munitions that can be simultaneously loaded into the gun.

3.3 GUNS_SET_LOAD_MUNITION

```
void  GUNS_SET_LOAD_MUNITION(_vid, _cnum, _munition, _store, _quantity)
    int32  _vid;
    int32  _cnum;
    int32  _store;
    uint32 _munition;
    int32  _quantity;
```

'_vid' Specifies the vehicle ID

'_cnum' Specifies the gun component number

'_munition'
Specifies the type of munition to load

'_store' Specifies from what store to load the munition from

'_quantity'
Specifies the quantity of munitions to load into the gun

GUNS_SET_LOAD_MUNITION starts the loading procedure of a number of munitions from the supplied store. Illegal requests to load too many munitions will be clipped to legal amounts.

3.4 GUNS_SET_UNLOAD_MUNITION

```
void  GUNS_SET_UNLOAD_MUNITION(_vid, _cnum, _store, _quantity)
    int32  _vid;
    int32  _cnum;
    int32  _store;
    int32  _quantity;
```

'_vid' Specifies the vehicle ID

'_cnum' Specifies the gun component number

'_store' Specifies into what store to unload the munition into

'_quantity'
Specifies the quantity of munitions to unload from the gun

GUNS_SET_UNLOAD_MUNITION starts the unloading procedure of a number of already loaded munitions into the supplied store. Illegal requests to unload too many munitions will be clipped to legal amounts.

3.5 GUNS_GET_LOADED_MUNITION

```
void  GUNS_GET_LOADED_MUNITION(_vid, _cnum, _munition, _quantity)
    int32   _vid;
    int32   _cnum;
    uint32 *_munition;
    int32  *_quantity;
```

'_vid' Specifies the vehicle ID

'_cnum' Specifies the gun component number

'_munition'
Returns the loaded munition

'_quantity'
Returns the loaded quantity

GUNS_GET_LOADED_MUNITION returns (by reference) the quantity and munition loaded in the gun.

3.6 GUNS_SET_LAUNCHER_POSITION

```
void  GUNS_SET_LAUNCHER_POSITION(_vid, _cnum, _active)
    int32   _vid;
    int32   _cnum;
    int32   _active;
```

'_vid' Specifies the vehicle ID

'_cnum' Specifies the gun component number

'_active' Specifies whether to set the launcher active or not

GUNS_SET_LAUNCHER_POSITION starts process of putting gun in active or inactive position based on the boolean value of active. For things like a TOW missile launcher or a LOSAT mast launcher, this may take some time and/or result in appearance modifiers changing in the entity. A gun must made active before it is ready to fire.

3.7 GUNS_GET_LAUNCHER_POSITION

```
void  GUNS_GET_LAUNCHER_POSITION(_vid, _cnum, _active)
    int32   _vid;
    int32   _cnum;
    int32   *_active;
```

'_vid' Specifies the vehicle ID

'_cnum' Specifies the gun component number '_active' Returns current state of gun position

GUNS_GET_LAUNCHER_POSITION retrieves state of gun launcher. This returns TRUE if the gun has only one position or if the gun is in the firing position.

3.8 GUNS_SET_ELEVATION

```
void  GUNS_SET_ELEVATION(_vid, _cnum, _el)
    int32   _vid;
    int32   _cnum;
    int32   _el;
```

'_vid' Specifies the vehicle ID

'_cnum' Specifies the gun component number

'_el' Specifies the desired elevation

Certain guns (such as a tank main gun) can be elevated. For those guns, GUNS_SET_ELEVATION will select a desired elevation for that gun. If the gun does not support elevation, this will not do anything. If the requested elevation is not possible, the requested elevation will be clipped to a legal value.

Setting the elevation of the gun automatically takes the gun out of the target tracking mode (see Section 3.9 [GUNS'SET'TARGET], page 10) or location tracking mode (see Section 3.10 [GUNS'SET'LOCATION], page 11).

3.9 GUNS_SET_TARGET

```
void  GUNS_SET_TARGET(_vid, _cnum, _targetid)
    int32   _vid;
    int32   _cnum;
    int32   _targetid
```

'_vid' Specifies the vehicle ID

'_cnum' Specifies the gun component number

'_targetid'
Specifies the id of the target to track

GUNS_SET_TARGET puts the gun in an automatic mode where the gun will attempt, through use of the gun's turret component, to track on the targetid. Note that if the gun's turret is not slewable, this may not do anything.

Setting the elevation of the gun (see Section 3.8 [GUNS˙SET˙ELEVATION], page 10) automatically takes the gun out of this target tracking mode.

3.10 GUNS_SET_LOCATION

```
    void  GUNS_SET_LOCATION(_vid, _cnum, _location)
        int32   _vid;
        int32   _cnum;
        float64 _location[3];
```

'_vid' Specifies the vehicle ID

'_cnum' Specifies the gun component number

'_location'
Specifies the location to track

GUNS_SET_LOCATION puts the gun in an automatic mode where the gun will attempt, through use of the gun's turret component, to track on the location. Note that if the gun's turret is not slewable, this may not do anything.

3.11 GUNS_GET_TARGET_IS_TRACKED

```
    void  GUNS_GET_TARGET_IS_TRACKED(_vid, _cnum, _targetid, _result)
        int32   _vid;
        int32   _cnum;
        int32   _targetid;
        int32   *_result;
```

'_vid' Specifies the vehicle ID

'_cnum' Specifies the gun component number

'_targetid'
Specifies the id of the target to track

'_result' Returns whether the target is tracked

GUNS_GET_TARGET_IS_TRACKED returns (by reference) whether the vehicle **_targetid** is successfully tracked by the gun.

3.12 GUNS_GET_LOCATION_IS_TRACKED

```
void  GUNS_GET_LOCATION_IS_TRACKED(_vid, _cnum, _location, _result)
    int32   _vid;
    int32   _cnum;
    int32   _location;
    int32  *_result;
```

‘_vid’	Specifies the vehicle ID
‘_cnum’	Specifies the gun component number
‘_locat	
	Specifies the location to track
‘_result’	Returns whether the location is tracked

GUNS_GET_LOCATION_IS_TRACKED returns (by reference) whether the position **_location** is successfully tracked by the gun.

3.13 GUNS_GET_READY_TO_FIRE

```
void  GUNS_GET_READY_TO_FIRE(_vid, _cnum, _ready, _id)
    int32   _vid;
    int32   _cnum;
    int32  *_ready;
    int32  *_id;
```

‘_vid’	Specifies the vehicle ID
‘_cnum’	Specifies the gun component number
‘_ready’	Returns whether the gun is ready for firing
‘_id’	Returns vehicle ID of ready missile

GUNS_GET_READY_TO_FIRE returns (by reference) whether the gun is ready to fire. For missile launchers, the vehicle ID of the ready missile is returned (by reference) as well. The gun is generally not ready if it is not loaded or is in the process of loading or unloading munitions.

3.14 GUNS_SET_FIRE

```
void  GUNS_SET_FIRE(_vid, _cnum, _quantity)
    int32   _vid;
    int32   _cnum;
    int32   _quantity;
```

'_vid' Specifies the vehicle ID

'_cnum' Specifies the gun component number

'_quantity'
Specifies quantity of loaded munition to shoot

GUNS_SET_FIRE launches a weapon. The weapon is fired in whatever direction it is currently pointing. If the weapon is not loaded, this will do nothing. If _quantity specifies an amount greater than that currently loaded by the gun, the amount will be clipped down to the loaded amount.

3.15 GUNS_SET_FIRE_AT_TARGET

```
void  GUNS_SET_FIRE_AT_TARGET(_vid, _cnum, _quantity, _targetid)
    int32   _vid;
    int32   _cnum;
    int32   _quantity;
    int32   _targetid;
```

'_vid' Specifies the vehicle ID

'_cnum' Specifies the gun component number

'_quantity'
Specifies quantity of loaded munition to shoot

'_targetid'
Specifies the target to fire at

GUNS_SET_FIRE_AT_TARGET launches a weapon at _targetid. If the weapon is not tracked on the target, this will most likely miss the target. If the weapon is not loaded, this will do nothing. If _quantity specifies an amount greater than that currently loaded by the gun, the amount will be clipped down to the loaded amount.

3.16 GUNS_SET_FIRE_AT_LOCATION

```
void  GUNS_SET_FIRE_AT_LOCATION(_vid, _cnum, _quantity, _location)
    int32   _vid;
    int32   _cnum;
    int32   _quantity;
    float64 _location[3];
```

'_vid' Specifies the vehicle ID

'_cnum' Specifies the gun component number

'_quantity'
Specifies quantity of loaded munition to shoot

'_location'
Specifies the location to fire at

GUNS_SET_FIRE_AT_LOCATION launches a weapon at _location. If the weapon is not tracked on the location, this will most likely miss the location. If the weapon is not loaded, this will do nothing. If _quantity specifies an amount greater than that currently loaded by the gun, the amount will be clipped down to the loaded amount.

3.17 GUNS_GET_ALLOWED_MUNITIONS

```
void  GUNS_GET_ALLOWED_MUNITIONS(_vid, _cnum, _num, _munitions,
                                 _cur, _abs)
    int32            _vid;
    int32            _cnum;
    int32            *_num;
    GUNS_MUNITIONS   _munitions;
    GUNS_QUANTITIES  _cur;
    GUNS_QUANTITIES  _abs;
```

'_vid' Specifies the vehicle ID

'_cnum' Specifies the gun component number

'_num' Returns the length of the list (<= GUNS_MAX_MUNITIONS)

'_munitions'
Returns the list of allowed munitions

'_cur' Returns the list of current limits on munition quantities;

'_abs' Returns the list of absolute limits on munition quantities;

GUNS_GET_ALLOWED_MUNITIONS returns (by reference) a list of munitions which can be loaded in the gun. The data is returned in a **GUNS_MUNITIONS** data structure which is declared as follows:

```
typedef uint32 GUNS_MUNITIONS[GUNS_MAX_MUNITIONS];
```

In addition, two lists of quantities are returned (by reference.) One, is the list of quantities of munitions which the gun currently can fire. one is the list of maximum quantities of munitions the gun can fire. Both of these are based on internal limitations of the gun implementation and may *not* be equal to the supplies of the munitions available to the vehicle.

The lists of quantities are returned in a **GUNS_QUANTITIES** data structure which is declared as follows:

```
typedef int32 GUNS_QUANTITIES[GUNS_MAX_MUNITIONS];
```

The current value of **GUNS_MAX_MUNITIONS** is 4.

LibHM

Table of Contents

1 Overview

LibHM provides utility functions for managing *Height/Mach* (HM) diagrams. The library contains an editor '**hmedit**', which can be used to generate a thrust map, which is a generalization about engine capabilities which can be derived from an HM diagram.

The library contains the tool make_uu, which takes a new .map file, and uuencodes it so it can be stored and used by rcs. .uu versions of the .map file are converted by the makefile back to the .map format.

The library provides functions for reading these thrust maps, and accessing the information therein. It also provides utility functions for computing air density and true mach number.

1.1 Examples

The following code fragments from a version of libFWA, demonstrate usage:

```
/* Read the thrust map */
fwa->thrust_map = hm_get_thrust_map(params->thrust_map);

...
current_thrust_max = hm_thrust(fwa->thrust_map,
                               fwa->position[Z],
                               fwa->speed.actual);
```

2 Functions

The following sections describe each function provided by libhm, including the format and meaning of its arguments, and the meaning of its return values (if any).

2.1 hm_init

```
void hm_init(data_directory)
    char *data_directory;
```

'data_directory'

Specifies the directory where thrust maps are expected

hm_init initializes libhm. The passed data directory will be used when reading thrust maps, if the named files cannot be found in '.'.

2.2 hm_get_thrust_map

```
HM_THRUST_MAP_PTR hm_get_thrust_map(file_name)
    char *file_name;
```

'file_name'

Specifies the file in which the thrust map is stored

hm_get_thrust_map returns the thrust map stored in the name file. Note that the data is cached so that future references to the same file will return the same pointer. A NULL return value indicates an error occured.

2.3 hm_thrust

```
float64 hm_thrust(thrust_map, altitude, velocity)
    HM_THRUST_MAP_PTR thrust_map;
    float64           altitude;
    float64           velocity;
```

'thrust_map'
Specifies the thrust map (created by **hm_get_thrust_map**)

'altitude'
Specifies current altitude (meters)

'velocity'
Specifies current speed (meters/second)

hm_thrust looks up the *maximum* thrust available (in Newtons) given the passed altitude and velocity.

2.4 hm_power

```
float64 hm_power(thrust_map, altitude, velocity, mass, drag)
    HM_THRUST_MAP_PTR thrust_map;
    float64           altitude;
    float64           velocity;
    float64           mass;
    float64           drag;
```

'thrust_map'
Specifies the thrust map (created by **hm_get_thrust_map**)

'altitude'
Specifies current altitude (meters)

'velocity'
Specifies current speed (meters/second)

'mass' Specifies current vehicle mass (kilograms)

'drag' Specifies current drag (newtons)

hm_power looks up the available power, given the altitude and velocity (which are used to determine thrust), and mass and drag.

2.5 hm_air_density

```
float64 hm_air_density(altitude)
    float64 altitude;
```

'altitude'
Specifies an altitude (meters)

hm_air_density computes the air density at a passed altitude.

2.6 hm_air_density_rat_sq

```
float64 hm_air_density_rat_sq(altitude)
    float64 altitude;
```

'altitude'
Specifies an altitude (meters)

hm_air_density_rat_sq returns the square of the ratio of air density at an altitude to air density at sea level (needed in some equations). Equivalent to square(hm_air_density(altitude)/hm_air_density(0.0)), but much cheaper to compute.

2.7 hm_mach_velocity

```
float64 hm_mach_velocity(mach, altitude)
    float64 mach;
    float64 altitude;
```

'mach' Specifies a mach number

'altitude'
Specifies an altitude (meters)

hm_mach_velocity computes the velocity (meters/second) which corresponds to the passed mach number for the given altitude.

2.8 hm_velocity_mach

```
float64 hm_velocity_mach(velocity, altitude)
    float64 velocity;
    float64 altitude;
```

'velocity'

Specifies a velocity (meters/second)

'altitude'

Specifies an altitude (meters)

hm_velocity_mach computes the mach number which corresponds to the passed velocity for the given altitude.

ADST-TR-W003268

LibHulls

Table of Contents

1 Overview

Hulls is a SAF components class. The purpose of a components class is to define a common set of functions which are invoked on instances of that class, and the semantics of those functions. Other than defining these functional semantics, components classes don't actually *do* anything.

Access to hull functions is achieved through macros defined by libhulls. These macros invoke **cmpnt_invoke** with a code number which identifies the function to run. Libcomponents then runs this function for the particular hull mode via a jump table.

The table below shows how the hulls component relationships have been currently implemented via the ModSAF library structure.

specific libraries	generic library	architectural library
libtracked	libhulls	libcomponents
libfwa	libhulls	libcomponents
libmissile	libhulls	libcomponents
librwa	libhulls	libcomponents
libwheeled	libhulls	libcomponents

As mentioned above, libhulls requires the services of libcomponents, an architectural library which provides a level of abstraction away from the specific hulls component interfaces. When the ModSAF application gets set up to run, the libhulls initialization process directs libcomponents to define a hull component class. This information enables libcomponents to define a structure to accommodate all of the hull instantiations a simulated object is allowed to have. The libhulls initialization process also tells libcomponents the number of its defined hull interface functions. This enables a simulated object's user data to be allocated enough space to hold the address of each of the hull interface functions defined in libhulls.

The parametric data of libcomponents identifies each component that needs to be modeled when a vehicle is simulated. For example, a component entry for a T72 tank might look like this: (see the file named USSR_T72M_params.rdr)

```
(SM_Components (hull        SM_TrackedHull)
               (turret      SM_GenericTurret)
               (machine-gun [SM_BallisticGun | 0])
               (main-gun    [SM_BallisticGun | 1])
               (visual      SM_Visual))
```

A T72M simulated vehicle (which belongs to the safobj class) will have component sub-class data that tells the ModSAF software to maintain a structure that includes one libtracked instantiation.

Since an application will interface to libtracked, libfwa, librwa, libwheeled, or libmissile through libhulls, a tank's movement control commands (which are performed by libtracked) and an airplane's movement commands (which are performed by libfwa) are both issued via the interface defined by libhulls. A command to change the controls is therefore the same whether the hulls component belongs to a tank or an airplane. What is different are the actual values used to set the controls and those values are passed as input to the function. Similarly, an application can obtain information about the state of its hull though the libhulls interface. The table below shows the relationship between the specific and generic library for the hulls component.

Instantiations of	Belong to generic	Have a command
of the library:	component class:	interface defined in:
libtracked	hulls	libhulls
libfwa	hulls	libhulls
libmissile	hulls	libhulls
librwa	hulls	libhulls
libwheeled	hulls	libhulls

The interface to libhulls is defined in its public header file (libhulls.h). This interface lets an application set hull controls or get hull information without knowing which specific hull model is being used. Applications interface to the tracked, fwa, or missile model primarily through the macros defined in libhulls. These macros map to functions which are invoked on instances of the hulls sub-class (such as the libtracked component instantiated for a tank, the libfwa component instantiated for an airplane, or the libmissile component instantiated for a missile).

The libhulls interface is defined in libhulls.h, the public header file for libhulls. One interface for controlling vehicle motion is the **HULLS_SET_DIRECTION_SPEED** macro which maps to a function that sets a desired direction of travel and a speed. Each macro is associated with a function code number so that a call to the libcomponents library (via the cmpnt_invoke function) will dispatch a call to the appropriate library (such as libfwa, libmissile, or libtracked). The definition for this macro might appear as shown below.

```
#define HULLS_SET_DIRECTION_SPEED(Hvid, Hcnum, Hdir, Hsp, Hmtr, Hma)
{
    HULLS_INTERFACE _hif;
    _hif.u.set_direction_speed.direction = Hdir;
    _hif.u.set_direction_speed.speed = Hsp;
    _hif.u.set_direction_speed.max_turn_rate = Hmtr;
    _hif.u.set_direction_speed.max_accel = Hma;
    cmpnt_invoke(HULLS_SET_DIRECTION_SPEED_FCN, Hvid, Hcnum, (ADDRESS)&_hif);
}
```

The HULLS_INTERFACE structure defined in libhulls.h is the structure that is passed to any hull function. This structure is a union of structures that each define an argument list for a hull function. An abbreviated example that assumes there are only two hull functions is shown below. Typically there will be many functions and therefore more strucure definitions in the union. The macros hide this structure from the users of these functions.

```
typedef struct hulls_interface
{
    union
    {
        struct hulls_set_direction_speed
        {
            float64 *direction;
            float64  speed;
            float64  max_turn_rate;
            float64  max_accel;
        } set_direction_speed;
        struct hulls_set_velocity_gear
        {
            float64 *velocity;
            int32    gear;
            float64  max_turn_rate;
            float64  max_accel;
        } set_velocity_gear;
    } u;
} HULLS_INTERFACE;
```

Issuing a command to an objects's hulls component is done by invoking one of the macros defined in libhulls. These macros identify the specific component function which needs to be called. For example, invoking the HULLS_SET_DIRECTION_SPEED macro will result in the calling of the set_direction_speed specific component function. In the public header file of each generic library, macros are associated with a function code number so that a call to the libcomponents library (via the cmpnt_invoke function) will dispatch a call to the appropriate function. The specific component functions are defined and installed by the specific libraries (libtracked, libfwa and libmissile). In this case, all three libraries install a function with the same name, "set_direction_speed" (there is no name conflict because each function is declared static). It is the specific function (either libfwa's set_direction_speed, libtracked's set_direction_speed, or libmissile's set_direction_speed) that is called when the macro is invoked.

Invoking the macro results in two actions: (1) setting up of the interface structure and (2) passing of necessary information to libcomponent. The macro passes the vehicle id, component number, and function pointer index to libcomponent so that the appropriate library (such as libtracked, libfwa, or libmissile) data can be accessed. The requested function can require input (such as a direction and speed) and/or output (such as a setting) . Therefore, libcomponents must also be passed the address of the interface structure that holds this data.

In the code segement:

```
cmpnt_invoke(HULLS_SET_DIRECTION_SPEED_FCN, Hvid, Hcnum, (ADDRESS)&_hif);
```

HULLS_SET_DIRECTION_SPEED_FCN serves as the function pointer index, Hvid provides the vehicle id, Hcnum provides the component number, and &_hif provides the address for the function's argument lists.

2 Examples

To initialize libtracked, an instance of the hull class:

```
tracked_user_data_handle =
  class_reserve_user_data(parent_class, "tracked", tracked_print);

/* Tell libcomponents we are available. */
cmpnt_define_instance(SM_TrackedHull, 1, &tracked_user_data_handle,
                      tracked_create, tracked_destroy,
                      HULLS_SET_DIRECTION_SPEED_FCN, set_dir_speed,
                      HULLS_SET_VELOCITY_GEAR_FCN, set_vel_gear,
                      HULLS_SET_VELOCITY_DIRECTION_FCN, set_vel_dir,
                      HULLS_SET_VELOCITY_ORIENTATION_FCN, set_vel_ori,
                      HULLS_SET_POSITION_DIRECTION_FCN, set_pos_dir,
                      HULLS_SET_GOAL_CORRIDOR_FCN, set_goal_corr,
                      HULLS_SET_TARGET_ID_FCN, set_target_id,
                      HULLS_SET_TARGET_POSITION_FCN, set_target_position,
                      HULLS_GET_ETA_FCN, get_eta,
                      HULLS_GET_TAKEOFF_ALT_FCN, get_takeoff_alt,
                      HULLS_SET_TAKEOFF_FCN, set_takeoff,
                      HULLS_SET_LANDED_FCN, set_landed,
                      HULLS_SET_FLY_LEVEL_FCN,set_fly_level,
                      HULLS_GET_TURN_PERFORMANCE_FCN, get_turn_performance,
                      HULLS_GET_CLIMB_PERFORMANCE_FCN, get_climb_performance,
                      HULLS_GET_FUEL_NEEDED_FCN, get_fuel_needed,
                      HULLS_GET_MAX_RANGE_FCN, get_max_range,
                      HULLS_SET_EXTERNAL_CONTROL_FCN, set_external_control,
                      HULLS_GET_LIMITS_FCN, get_limits);
```

To get the component number of my hull:

```
extern int32 my_hull;

if ((my_hull = cmpnt_locate(vehicle_id, reader_get_symbol("hull"))) ==
    CMPNT_NOT_FOUND)
  printf("Vehicle %d does not seem to have a hull\n", vehicle_id);
```

To then give a command to that hull (the macro is defined by libhulls; it assembles a `HULLS_INTERFACE` structure, and calls `cmpnt_invoke`):

```
if (my_hull != CMPNT_NOT_FOUND)
  HULLS_SET_DIRECTION_SPEED(vehicle_id, hull, dirvec, speed, 0.0, 0.0);
```

3 Functions

The following sections describe each function provided by libhulls, including the format and meaning of its arguments, and the meaning of its return values (if any).

3.1 hulls_init

```
void hulls_init();
```

hulls_init initializes libhulls. Call this function after cmpnt_init, and before any specific hull init functions.

3.2 HULLS_SET_EXTERNAL_CONTROL

```
HULLS_SET_EXTERNAL_CONTROL(vehicle_id, component_number, velocity,
                           direction, position, roll_angle, max_turn_rates,
                           max_accel)
    int32   vehicle_id;
    int32   component_number;
    float64 velocity[3];
    float64 direction[3];
    float64 position[3];
    float64 roll_angle;
    float64 max_turn_rates[3];
    float64 max_accel;
```

'vehicle_id'
Specifies the vehicle ID

'component_number'
Specifies the hull component number

'velocity'
Specifies desired velocity (meters per second)

'direction'
Specifies desired direction

'position'
Specifies desired position

'max_turn_rate'
: Specifies the maximum desired turn rate (radians per second)

'max_accel'
: Specifies the maximum desired acceleration (meters per second squared)

HULLS_SET_EXTERNAL_CONTROL is a macro which sets a desired direction of travel and movement velocity, as well as an expected position. A negative speed indicates backward movement is desired. It is assumed that the hull should face down the Y component of the direction. Some hulls may not support backward movement, in which case they will reverse the desired direction. The maximum turn rate (in radians per second) and maximum acceleration (in meters per second per second) will default to maximum if specified as zero.

The direction vector need not be a unit vector (specific component models may often be able to avoid normalizing this vector at all, saving a square root).

3.3 HULLS_SET_DIRECTION_SPEED

```
    HULLS_SET_DIRECTION_SPEED(vehicle_id, component_number,
                              direction, speed, max_turn_rate, max_accel)
        int32   vehicle_id;
        int32   component_number;
        float64 direction[3];
        float64 speed;
        float64 max_turn_rate;
        float64 max_accel;
```

'vehicle_id'
: Specifies the vehicle ID

'component_number'
: Specifies the hull component number

'direction'
: Specifies desired direction

'speed' Specifies desired speed (meters per second)

'max_turn_rate'
: Specifies the maximum desired turn rate (radians per second)

'max_accel'
: Specifies the maximum desired acceleration (meters per second squared)

HULLS_SET_DIRECTION_SPEED is a macro which sets a desired direction of travel and a speed.

A negative speed indicates backward movement is desired. It is assumed that the hull should face down the Y component of the direction. Some hulls may not support backward movement, in which case they will reverse the desired direction. The maximum turn rate (in radians per second) and maximum acceleration (in meters per second per second) will default to maximum if specified as zero.

The direction vector need not be a unit vector (specific component models may often be able to avoid normalizing this vector at all, saving a square root).

3.4 HULLS_SET_VELOCITY_GEAR

```
HULLS_SET_VELOCITY_GEAR(vehicle_id, component_number,
                        velocity, gear, max_turn_rate, max_accel)
    int32   vehicle_id;
    int32   component_number;
    float64 velocity[3];
    int32   gear;
    float64 max_turn_rate;
    float64 max_accel;
```

'**vehicle_id**'
Specifies the vehicle ID

'**component_number**'
Specifies the hull component number

'**velocity**'
Specifies the desired velocity (meters per second)

'**gear**' Specifies the desired gear

'**max_turn_rate**'
Specifies the maximum desired turn rate (radians per second)

'**max_accel**'
Specifies the maximum desired acceleration (meters per second squared)

HULLS_SET_VELOCITY_GEAR is a macro which sets a desired velocity, and a direction of movement (forward/backward). It is assumed that the hull should face down the Y component of the direction. Some hulls may not support backward movement, in which case they will reverse the desired velocity. The maximum turn rate (in radians per second) and maximum acceleration (in meters per second per second) will default to maximum if specified as zero.

The **gear** should be one of:

`HULLS_GEAR_FORWARD`
Forward movement

`HULLS_GEAR_REVERSE`
Backward movement

3.5 HULLS_SET_VELOCITY_DIRECTION

```
HULLS_SET_VELOCITY_DIRECTION(vehicle_id, component_number,
                             velocity, direction, roll_angle,
                             max_turn_rates, max_accel)
    int32   vehicle_id;
    int32   component_number;
    float64 velocity[3];
    float64 direction[3];
    float64 roll_angle;
    float64 max_turn_rates[3];
    float64 max_accel;
```

'`vehicle_id`'
Specifies the vehicle ID

'`component_number`'
Specifies the hull component number

'`velocity`'
Specifies the desired velocity (meters per second)

'`direction`'
Specifies the desired direction

'`roll_angle`'
Specifies the desired roll angle (radians)

'`max_turn_rates`'
Specifies maximum desired turn rates (yaw, pitch, roll) in radians per second

'`max_accel`'
Specifies the maximum desired acceleration (in meters per second square)

`HULLS_SET_VELOCITY_DIRECTION` is a macro which sets a desired hull direction and movement velocity. Some hull models require that the direction and velocity vectors be colinear; these models will strive to achieve the velocity, and will select forward or backward movement, depending on the dot product of the two vectors. Some hull models also may ignore the roll_angle. The maximum turn rates are in the order `yaw`, `pitch`, `roll`. The maximum turn rates (in radians per second) and maximum acceleration (in meters per second per second) will default to maximum if specified as zero (or a NULL pointer).

3.6 HULLS_SET_VELOCITY_ORIENTATION

```
HULLS_SET_VELOCITY_ORIENTATION(vehicle_id, component_number,
                               velocity, orientation,
                               max_turn_rates, max_accel)
     int32   vehicle_id;
     int32   component_number;
     float64 velocity[3];
     float64 orientation[3];
     float64 max_turn_rates[3];
     float64 max_accel;
```

'vehicle_id'
Specifies the vehicle ID

'component_number'
Specifies the hull component number

'velocity'
Specifies the desired velocity

'orientation'
Specifies the desired orientation (yaw, pitch, roll) in radians

'max_turn_rates'
Specifies maximum desired turn rates (yaw, pitch, roll) in radians per second

'max_accel'
Specifies the maximum desired acceleration (in meters per second square)

HULLS_SET_VELOCITY_ORIENTATION is a macro which sets a desired hull orientation and movement velocity. The behavior is exactly as in HULLS_SET_VELOCITY_DIRECTION, except the direction and roll angle are specified as an angular triple (yaw, pitch, roll) in radians.

3.7 HULLS_SET_POSITION_DIRECTION

```
HULLS_SET_POSITION_DIRECTION(vehicle_id, component_number,
                             position, direction)
     int32   vehicle_id;
     int32   component_number;
```

```
        float64 position[3];
        float64 direction[3];
```

'vehicle_id'
Specifies the vehicle ID

'component_number'
Specifies the hull component number

'position'
Specifies the desired position

'direction'
Specifies the desired direction at that position

HULLS_SET_POSITION_DIRECTION is a macro which sets a desired hull position and direction. The hull model will achieve a position and direction as close as possible to these desires in a manner which is appropriate for the type of hull being modeled (for a tracked hull: move to position, then turn in place; for a wheeled hull: do a three point turn; etc.).

3.8 HULLS_SET_GOAL_CORRIDOR

```
    HULLS_SET_GOAL_CORRIDOR(vehicle_id, component_number,
                            approach_speed, position, direction,
                            corridor_width)
        int32   vehicle_id;
        int32   component_number;
        float64 approach_speed;
        float64 position[3];
        float64 direction[3];
        float64 corridor_width;
```

'vehicle_id'
Specifies the vehicle ID

'component_number'
Specifies the hull component number

'approach_speed'
Specifies the approach speed

'position'
Specifies the start of the corridor

'direction'
Specifies the direction of the corridor

'corridor_width'
Specifies the width of the corridor

HULLS_SET_GOAL_CORRIDOR is a macro which sets a desired position, a desired speed to approach that position and a corridor (expressed as a direction and a width) which is not to be exceeded at the time the hull crosses the position. This is used, for example, to approach a corner on a road without going so fast you will miss the corner.

3.9 HULLS_SET_TARGET_ID

```
HULLS_SET_TARGET_ID(vehicle_id, component_number, id)
    int32 vehicle_id;
    int32 component_number;
    int32 id;
```

'vehicle_id'
Specifies the vehicle ID

'component_number'
Specifies :he hull component number

'id' Specifies the ID of the target

HULLS_SET_TARGET_ID is a macro which sets the target which the hull should pursue (using whatever method is supported by that hull). Note that this is primarily used by missile hulls, but o:her hulls must support some version of the functionality.

3.10 HULLS_SET_TARGET_POSITION

```
HULLS_SET_TARGET_POSITION(vehicle_id, component_number, position)
    int32   vehicle_id;
    int32   component_number;
    float64 position[3];
```

'vehicle_id'
Specifies the vehicle ID

'component_number'
Specifies the hull component number

'position'

Specifies the target position

HULLS_SET_TARGET_POSITION is a macro which sets the position which the hull should pursue (using whatever method is supported by that hull). Note that this is primarily used by missile hulls, but other hulls must support some version of the functionality.

3.11 HULLS_GET_ETA

```
HULLS_GET_ETA(vehicle_id, component_number, position, eta)
    int32            vehicle_id;
    int32            component_number;
    float64          position[3];
    {float64|int32} *eta;
```

'vehicle_id'

Specifies the vehicle ID

'component_number'

Specifies the hull component number

'position'

Specifies the desired position

'eta' Returns the time it will take to get to that position

HULLS_GET_ETA is a macro which computes an estimated time of arrival at a point (in seconds), given current hull state, and operating parameters. Since this is not a function, the compiler will automatically cast the returned eta to whatever type is needed.

3.12 HULLS_GET_TAKEOFF_ALT

```
HULLS_GET_TAKEOFF_ALT(Hvid, Hcnum, Haltitude)
    int32            Hvid;
    int32            Hcnum;
    float64          Haltitude;
```

'Hvid' Specifies the vehicle ID

'Hcnum' Specifies the hull component number

'Haltitude'
Returns the altitude required for completion of take off

HULLS_GET_TAKEOFF_ALT returns (by reference) the altitude required for this hull to complete its take off.

3.13 HULLS_SET_TAKEOFF

```
HULLS_SET_TAKEOFF(Hvid, Hcnum, Haltitude)
    int32           Hvid;
    int32           Hcnum;
    float64         Haltitude;
```

'Hvid' Specifies the vehicle ID

'Hcnum' Specifies the hull component number

'Haltitude'
Specifies the target altitude

HULLS_SET_TAKEOFF sets the altitude for this hull to complete its take off.

3.14 HULLS_SET_LANDED

```
HULLS_SET_LANDED(Hvid, Hcnum)
    int32           Hvid;
    int32           Hcnum;
```

'Hvid' Specifies the vehicle ID

'Hcnum' Specifies the hull component number

HULLS_SET_LANDED places the hull on the ground.

3.15 HULLS_SET_FLY_LEVEL

```
HULLS_SET_FLY_LEVEL(vehicle_id, component_number, track, speed,
                    altitude, fpa, max_turn_rates, max_accel)
```

```
int32   vehicle_id;
int32   component_number;
float64 track[2];
float64 speed;
float64 altitude;
float64 fpa;
float64 max_turn_rates[3];
float64 max_accel;
```

'vehicle_id'
Specifies the vehicle ID

'component_number'
Specifies the hull component number

'track' Specifies the 2-D direction (X-Y as a normalized vector)

'speed' Specifies speed (meters per second)

'altitude'
Specifies altitude (meters)

'fpa' Specifies the flight path angle (radians)

'max_turn_rates'
Specifies maximum desired turn rates (track, pitch, roll) in radians per second

'max_accel'
Specifies the maximum desired acceleration (in meters per second square)

HULLS_SET_FLY_LEVEL is a macro which sets a desired hull direction (track) speed and altitude. The maximum turn rates are in the order track, pitch, roll. The maximum turn rates (in radians per second) and maximum acceleration (in meters per second per second) will default to maximum if specified as zero (or a NULL pointer).

Note that specifying a flight path angle (fpa) which is too large, could cause an airplane to stall.

3.16 HULLS_GET_TURN_PERFORMANCE

```
HULLS_GET_TURN_PERFORMANCE(vehicle_id, component_number,
                           max_turn, easy_turn, standard_turn, hard_turn)
    int32   vehicle_id;
    int32   component_number;
    float64 max_turn;
    float64 easy_turn;
    float64 standard_turn;
    float64 hard_turn;
```

'vehicle_id'
: Specifies the vehicle ID

'component_number'
: Specifies the hull component number

'max_turn'
: Specifies the maximum turn rate for the vehicle in radians per second.

'easy_turn'
: Specifies the turn rate for the vehicle in radians per second when trying to do an easy turn.

'standard_turn'
: Specifies the turn rate for the vehicle in radians per second when trying to do an standard turn.

'hard_turn'
: Specifies the turn rate for the vehicle in radians per second when trying to do an hard turn.

HULLS_GET_TURN_PERFORMANCE is a macro which queries the vehicle for current turn rates, which can be selected by a controller to set the vehicles desired turn rate.

3.17 HULLS_GET_CLIMB_PERFORMANCE

```
HULLS_GET_CLIMB_PERFORMANCE(vehicle_id, component_number,
                            max_climb, easy_climb, standard_climb, hard_climb)
    int32   vehicle_id;
    int32   component_number;
    float64 max_climb;
    float64 easy_climb;
    float64 standard_climb;
    float64 hard_climb;
```

'vehicle_id'
: Specifies the vehicle ID

'component_number'
: Specifies the hull component number

'max_climb'
: Specifies the maximum climb rate for the vehicle in radians per second.

'easy_climb'
: Specifies the climb rate for the vehicle in radians per second when trying to do an easy climb.

'standard_climb'
Specifies the climb rate for the vehicle in radians per second when trying to do an standard climb.

'hard_climb'
Specifies the climb rate for the vehicle in radians per second when trying to do an hard climb.

HULLS_GET_CLIMB_PERFORMANCE is a macro which queries the vehicle for current climb rates, which can be selected by a controller to set the vehicles desired climb rate.

3.18 HULLS_GET_FUEL_NEEDED

```
HULLS_GET_FUEL_NEEDED(vehicle_id, component_number, altitude, speed,
distance, fuel_needed)
    int32   vehicle_id;
    int32   component_number;
    float64 altitude;
    float64 speed;
    float64 distance;
    float64 fuel_needed;
```

'vehicle_id'
Specifies the vehicle ID

'component_number'
Specifies the hull component number

'altitude'
Specifies the altitude to return to point.

'speed' Specifies the speed to return to point.

'distance'
Specifies the distance to point.

'fuel_needed'
Specifies the fuel that is needed to get to the point

HULLS_GET_FUEL_NEEDED is a macro which determines how much fuel is needed to get to a point, given the distance to the point, and the desired altitude and speed to use to reach that point.

3.19 HULLS_GET_MAX_RANGE

```
HULLS_GET_MAX_RANGE(vehicle_id, component_number, max_range)
    int32   vehicle_id;
    int32   component_number;
    float64 max_range;
```

'vehicle_id'
Specifies the vehicle ID

'component_number'
Specifies the hull component number

'max_range'
Returns the fuel that is needed to get to the point

HULLS_GET_MAX_RANGE is a macro which determines the maximum range of the vehicle. For land vehicles, it ignores factors such as terrain. For fixed-wing aircraft, it uses a standard thrust rate and a nominal altitude of 18000 meters. In both of the above cases, HULLS_GET_MAX_RANGE bases its calculation on the actual amount of fuel available. In the case of a missile, this macro returns the "maximum effective range" as specified in the missile's parameter file. This is a static value and does not change after the missile is launched.

3.20 HULLS_GET_LIMITS

```
HULLS_GET_LIMITS(vehicle_id, component_number, position, direction,
                 max_speed, max_accel, max_decel, max_turn, min_radius,
                 max_sideways, max_up)
    int32    vehicle_id;
    int32    component_number;
    float64  position[3];
    float64  direction[3];
    float64 *max_speed;
    float64 *max_accel;
    float64 *max_decel;
    float64 *max_turn;
    float64 *min_radius;
    float64 *max_sideways;
    float64 *max_up;
```

'vehicle_id'
Specifies the vehicle ID

'component_number'
Specifies the hull component number

'position'
Specifies sample position (uses current if NULL is passed)

'direction'
Specifies sample directoin (uses current if NULL is passed)

'max_speed'
Returns maximum possible speed in meters/second

'max_accel'
Returns maximum possible acceleration in meters/second/second

'max_decel'
Returns maximum possible deceleration in meters/second/second

'max_turn'
Returns maximum possible turn rate in radians/second

'min_radius'
Returns minimum possible turn radius in meters

'max_sideways'
The maximum speed a vehicle can move sideways. Zero for vehicles that can't move sideways.

'max_up' The maximum speed a vehicle can climb in a hover. Zero for non hovering vehicles.

HULLS_GET_LIMITS is a macro which determines the maximum performance limits of a hull. The position and direction specify the state for which limits are desired. If NULL is passed, the current values of the entity will be used.