ADVANCED DISTRIBUTED SIMULATION TECHNOLOGY

ModSAF

PROGRAMMER'S REFERENCE

MANUAL

VOL. 3

(Libremote - Libuoccpos)

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

ADVANCED DISTRIBUTED SIMULATION TECHNOLOGY

ModSAF

PROGRAMMER'S REFERENCE

MANUAL

VOL. 3

(Libremote - Libuoccpos)

Ver 1.0 - 20 December 1993

CONTRACT NO. N61339-91-D-0001

D.O.: 0021

CDRL SEQUENCE NO. A001

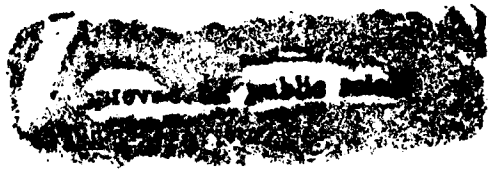
Prepared for:

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



Libremote

Table of Contents

1 Overview 1

2 Functions 5

2.1 remote_init 5

3 Events 7

3.1 remote_local_deactivated_event 7

1 Overview

Libremote is a small library that handles remote objects through the use of callback functions, which are a component of ModSAF's method of event handling. Using this method, libremote, the library that is interested in network packets belonging to remote entities, requests that the libcallback library tests for those packet receipt events (these tests are coded in libpktvalve) and then calls a registered libremote function when the event happens.

After ModSAF initialization is complete, the ModSAF software begins its normal processing which typically includes listening for packets on the network. When the libcallback events occur, the software generating the event (libpktvalve) passes libcallback the event handle and arguments; libcallback then calls the registered libremote callback function. With this method libcallback distributes event information from a low layer service provider (libpktvalve) to a higher layer library (libremote).

The use of libcallback, therefore, maintains the relationship between two libraries: (1) libpktvalve which detects vehicle appearance, deactivate, designator, and stealth packets; and (2) libremote which filters these packets, performs relevant actions (such as creating new remote vehicles via libvtab when appropriate), and then passes the packet data to approriate libraries (such as libentity, libdesignate, or libstealth).

Currently, libremote registers the following callback functions:

`received_vapp()`
: handles update or creation of a remote vehicle when a vehicle appearance packet is received

`received_deactivate()`
: handles removal of a remote vehicle when a deactivate packet is received

`received_designate()`
: handles update or creation of a designator when a designate packet is received

`received_stop_designate()`
: handles removal of a designator when a stop designate packet is received

`received_sapp()`
: handles the update or creation of a remote stealth when a stealth appearance packet is received

When the vehicle appearance packet for a remote vehicle (includes DI), missile, or structure (building or bridge) is received, the following processing takes place via the `received_vapp()`

function.

1. Translate the packet's network ID to a local computer vehicle id.
2. If the vehicle does not already exist, determine if this packet refers to an allowed type (VTAB_REMOTE_VEHIC VTAB_REMOTE_MISSILE, or VTAB_REMOTE_STRUCTURE). If it does then make a new remote vehicle by calling **safobj_create_remote(vehicle_id, sim_id, type, remote_tick_rate)** otherwise exit.
3. If this packet was for a newly created remote or for an already exisiting remote, pass the packet on to libentity for processing.

When the deactivate packet for a remote vehicle (includes DI), missile, or structure (building or bridge) is received, the following processing takes place via the **received_deactivate()** function.

1. Translate the packet's network ID to a local computer vehicle id.
2. If the vehicle does not already exist, exit.
3. If this packet was for an exisiting remote (not local) vehicle, pass the packet on to libentity for processing.

When the designate packet for a remote designator (includes laser beam), is received, the following processing takes place via the **received_designate()** function.

1. Translate the packet's network ID to a local computer vehicle id.
2. If the vehicle does not already exist, create it as a remote designator via a call to:

```
safobj_create_remote(vehicle_id, sim_id,
                     VTAB_REMOTE_DESIGNATOR, remote_tick_rate)
```

3. Pass the packet on through to libdesignate for processing via the function **dsg_packet_received(vehicle_id, packet)**.

When the stop designate packet for a remote designator (includes laser beam), is received, the following processing takes place via the **received_stop_designate()** function.

1. Translate the packet's network ID to a local computer vehicle id.
2. If the vehicle does not already exist, exit.
3. If the vehicle is classified as local rather than remote, exit.
4. Pass the packet on through to libdesignate for processing via the **dsg_packet_received(vehicle_id, packet)** function.

When the appearance packet for a remote stealth is received, the following processing takes

place via the **received_sapp()** function.

1. If this packet does not refer to the type **VTAB_REMOTE_STEALTH**, exit.
2. Translate the packet's network ID to a local computer vehicle id.
3. If the stealth does not already exist, then make a new remote vehicle by calling **safobj_create_remote(vehicle_id, sim_id, type, remote_tick_rate)**.
4. If this packet was for a newly created stealth or for an already exisiting stealth, pass the packet on to libstealth for processing via the function **stealth_appearance_received(vehicle_id, packet)**.

2 Functions

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

2.1 remote_init

```
void remote_init(packet_valve, types, tick_rate, exercise_id,
                 sim_addr, protocol)
    PV_VALVE_PTR        packet_valve;
    uint16              types;
    uint32              tick_rate;
    uint8               exercise_id;
    SimulationAddress   *sim_addr;
    int32               protocol;
```

'packet_valve'
: Specifies the packet valve used to receive packets for vehicles, missiles, structures, stealths, and designators.

'types'
: Specifies the types of remotes which should be created.

'tick_rate'
: Specifies the tick rate for remote vehicles.

'exercise_id'
: Specifies the exercise ID to listen to.

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

remote_init initializes libremote. This function installs packet handlers for remote vehicle appearance and deactivate packets. **types** specifies the types of vehicles which libremote should create (such as **VTAB_REMOTE_VEHICLE | VTAB_REMOTE_STRUCTURE**, which would prevent remote missiles from being registered). The tick rate specifies the rate at which remotes should be ticked for RVA purposes (note that RVA will not necessarily occur every tick; see libentity for more details). The exercise ID specifies the exercise which will be monitored for arriving vehicles (you can change this after init time via a call to **pv_change_exercise** on the simulationProtocolNumber, see section 'pv_change_exercise' in Libpktvalve Programmer's Manual).

When libremote creates a remote vehicle, it automatically includes the following sub-classes:

- entity
- pbtab

Call pv_init before this function. See section 'pv_init' in Libpktvalve Programmer's Manual.

3 Events

The following sections describe each event provided by libremote.

3.1 remote_local_deactivated_event

```
CALLBACK_EVENT_PTR remote_local_deactivated_event;
```

The **remote_local_deactivated_event** event fires when a local vehicle receives a deactivate packet (which under DIS 2.0.3, is an entity state packet with a magic bit set).

```
void handler(vehicle_id)
    int32 vehicle_id;
```

LibRouteMap

Table of Contents

1 Overview 1

2 Functions 3

2.1 routemap_create 3

2.2 routemap_preplan 3

1 Overview

LibRouteMap provides a convenient interface to the spatial planning algorithms in libMoveMap. The library creates a movemap at program startup which contains "large" terrain features for the whole database. Routines can preplan courses for units using this database.

The database is initialized to support a variety of unit width configuration spaces. When a call is made to plan a route through the database, the closest width which is greater than or equal to the desired width is used. Each additional width configurations adds to the storage and compute requirements.

2 Functions

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

2.1 routemap_create

```
ROUTEMAP_PTR routemap_create(ctdb, quad_p, num_widths, width, width...)
    CTDB      *ctdb;
    QUAD_DATA *quad_p;
    int32      num_widths;
    int32      width, width...;
```

'ctdb' Specifies the CTDB terrain database

'quad_p' Specifies the quadtree terrain database

'num_widths'
Specifies the number of unit widths to generate maps for

'width, width...'
Specifies unit widths, in increasing order

routemap_create creates a routemap for a terrain database. This routemap can be used to plan a route around large obstacles. The map is set up in advance to support planning for a variety of unit widths. The width used for a particular instance of planning will be the preplanned width which is no smaller than the passed unit width. The widths must be passed in increasing order.

2.2 routemap_preplan

```
void routemap_preplan(routemap, obstacle_mask, max_milliseconds,
                      goal, max_plan_deviation, unit_width, plan)
    ROUTEMAP_PTR  routemap;
    uint32        obstacle_mask;
    uint32        max_milliseconds;
    ROUTE_POINTS *goal;
    float64       max_plan_deviation;
    int32         unit_width;
    ROUTE_POINTS *plan;
```

'routemap'
Specifies the route map

'obstacle_mask'
Specifies the obstacles to avoid

'max_milliseconds'
Specifies the maximum time to spend planning the route

'goal' Specifies the goal path

'max_plan_deviation'
Specifies the maximum deviation allowed around each obstacle

'unit_width'
Specifies the width of the unit which will execute the plan

'plan' Returns the planned route

routemap_preplan performs obstacle avoidance around fixed obstacles which match the passed type mask, and returns the shortest course which can be found in the specified time. The returned plan is dynamically allocated using NS_ROUTE_ALLOCATE_POINTS (from 'stdroute.h'), and thus plan.points should be freed using STDDEALLOC. If no plan can be found within the allotted time, plan.num_pts with be 0, and plan.points will be NULL.

The obstacle_mask should be an inclusive OR of the following flags:

ROUTEMAP_RIVERS
River networks

ROUTEMAP_LAKES
Lakes

ROUTEMAP_BOULDERS
Boulder-covered areas which are specified in the libquad database

ROUTEMAP_CANOPIES
Tree canopies

Librwa

Table of Contents

1 Overview 1

1.1 Basic Modes 1
1.2 Basic Notation 1
1.3 Dynamics Model 2
1.3.1 Rotational Dynamics 2
1.3.2 Translational Dynamics 3
1.4 Control Model 3
1.4.1 Rotational Control 3
1.4.2 Translational Control 4
1.5 Limits and Clamping 8
1.6 Calibration and Stability 8

2 Parameters 11

3 Examples 13

4 Functions 15

4.1 rwa_init 15
4.2 rwa_class_init 15
4.3 rwa_destroy 15
4.4 rwa_collision 16
4.5 rwa_tick 16

1 Overview

Librwa implements an instance of the hull class of components. It provides a low fidelity model of rotary wing vehicle dynamics.

1.1 Basic Modes

The helicopter has two basic modes of flight, depending on speed. At low speed (SLOW mode), it uses the rotor only and can face in any direction. It maneuvers by tilting the rotor-vector. At high speed, it behaves somewhat like a fixed wing aircraft. It turns by flying a curving route, rather than rotating in place. It may either keep its jets horizontal (FAST-HORIZONTAL mode) or aligned with the velocity vector (FAST-ALIGNED mode).

1.2 Basic Notation

Vectors are represented by bold letters. Desired values have hats. Thus, $\boldsymbol{R}$ represents the rotor lift vector, $\hat{\boldsymbol{R}}$ represents the desired rotor lift vector, and R represents the scalar amount of lift.

When a vector is listed in terms of its components, it is enclosed in square brackets. Thus, $\boldsymbol{R} = [R_x, R_y, R_z]$

$\boldsymbol{R}$ The 3D vector indicating the direction and magnitude of the lift/thrust generated by the rotor. It points out through the drive shaft and is always perpendicular to $\boldsymbol{J}$.

$\boldsymbol{J}$ The 3D vector indicating the direction and magnitude of the thrust generated by the jets. It points out the nose of the helicopter and is always perpendicular to $\boldsymbol{R}$.

τ_x The time-constant for a generic quantity x, which can be R, track angle, FPA, etc.

λ Either the track angle (0=East, $\pi/2$=North) or the reciprocal of a time-constant, as will be specified in each context.

γ Flight path angle, or FPA (0 for horizontal, $\pi/2$ for straight up, $\pi/2$ for straight down).

ϕ Roll angle. 0 is vertical, $+\pi/2$ is 90 degrees to the right, $-\pi/2$ is 90 degrees to the left.

V The 3D velocity vector indicating the direction and speed.

V The scalar speed of the RWA

D The 3D vector indicating the direction and magnitude of the drag force. For notational convenience, it is taken to point along the velocity vector, not opposite to it.

g The 3D vector indicating the direction and magnitude of the gravitational acceleration. For notational convenience, it is taken to point up the Z axis, not opposite to it.

M The instantaneous mass of the RWA.

1.3 Dynamics Model

A purely vector representation was chosen to avoid singularities associated with trigonometric representations. Essential singularities (such as the direction of a 0-vector) persist, but those associated purely with the coordinate system (such as vertical) are eliminated. A single dynamics model is used for the entire flight regime, regardless of the speed or altitude. It simply sums the net forces of drag, gravity, rotor thrust, and jet thrust to compute the overall acceleration of the RWA's center of mass. A more detailed physics model of the orientational dynamics was explored in the initial formulation, based on the cyclic variation rotor angle of attack throughout its rotation and the thrust of the tail rotor. Careful examination revealed that, with an optimal controller, the overall dynamics of the controlled system would be a simply second order vector controller, and it was implemented as such.

1.3.1 Rotational Dynamics

The rotational dynamics are represented by the control of the jet and rotor. The physical dynamics and low level control are merged into a simple analytic form, capable of closed form solution. This is also the fastest control loop. Using a closed form control allows much longer time steps to be tolerated.

This section will describe the underlying physics, which justifies the merging of dynamics and control into a low-level model.

Gyroscopic dynamics are not modeled, as a reasonable controller would set its controls first to

eliminate gyroscopic effects, then impose the desired motion.

1.3.2 Translational Dynamics

```
V' =: V + dtA
P' =: P + dtV + (A * sqr(dt)) / 2
```

The drag function has linear and quadratic terms in velocity, and depends on air density, and is directed along the velocity vector.

1.4 Control Model

There are four basic control inputs: the 3D desired flight vector, and the desired azimuth along which the body should face. (Of course, in high-speed mode, the actual azimuth is along the flight vector.) This is in contrast to the FWA model for ModSAF, which has three basic control inputs, specifying the 3D desired flight vector. The additional RWA parameter comes from the decoupling of azimuth-faced from direction of flight at low speeds.

There are two basic levels of control: (a) rotating and scaling the rotor and jet thrust vectors, to meet net-acceleration goals, and (b) setting the net-acceleration to meet desired-velocity goals. They are analytically completely independent, and are implemented and described as such.

For testing, a third layer was added which sets desired-velocity to meet route-following goals.

1.4.1 Rotational Control

Both the rotor and the jet are modeled as second order critically damped vector controllers. They respond to the user's commands as if they were applied in continuous time through a zero order hold. Second order was chosen to avoid the instanteous speed changes associated with first order controllers; under-damping was rejected by the requirement to avoid oscillations; critical damping was chosen so that only one time-constant is needed.

To review the notation of a second order controller, assume we are controlling a vector quantity X

.

The current error term is:

```
epsilon = X - desired X
```

Because the control is exerted through a ZOH:

```
d epsilon/dt = dX / dt
```

The following algorithm updates the second order, critically damped controller.

```
C_1 = epsilon

lambda = 1 / tau

C_2 = d epsilon / dt + lambda epsilon

L = C_1  + dt C_2

d = e^(-lambda dt)

dX' / dt = (C_2 - lambda C_1) d

X' = dL + desired X
```

A vector controller having different time-constants for different components is quite feasible, but somewhat less compact to program. One involving cross-coupling between components would require the replacement of the scalar exponentials above with matrix exponentials, at greatly increased computational cost.

1.4.2 Translational Control

The FAST and SLOW modes differ greatly in how they do turns. In SLOW mode, there are two options. In the first turning option, the rotor vector is simply turned so as to accelerate the helicopter in the desired direction. Thus, if the helicopter is heading due East and is directed to go due West, it will turn its rotor vector backward to stop its eastward motion, hold it backward so as to accelerate to the westward, then straighten up the rotor vector so as to cruise along at the desired speed. The whole maneuver takes place with no North or South excursions at all, and there is no rolling. There is no change in the body orientation (i.e. in the jet vector J), as the vehicle can maintain any desired yaw (though roll and pitch are determined by the desired net acceleration). That is, it can maintain a 90 degree yaw in flying sideways, or a 180 degree yaw in flying backwards.

The second option of SLOW mode, and the only option in the two FAST modes, is to bank to turn as an airplane would do. Thus, the 180 degree reversal described above would be accomplished by following a curving path off to the North or South (depending on which direction was slightly less than 180 degrees), as an airplane would do.

The desired net acceleration in the first option of SLOW mode is given simply by the following expression. This net acceleration can only be achieved in SLOW mode.

```
desired A = (desired V - V) / tau V   (A)
```

Thus, changes in the X, Y, and Z velocity are treated symmetrically, except in that *HELO-LIMIT-FORCE-VECTOR* will clamp the force vector so as to not exceed the maximum lift currently generatable, preserving the Z component if possible. This is done so as to make terrain avoidance/following quite response in the very low-altitude regime where SLOW mode would by typically employed.

In the second SLOW option (only one in FAST mode) the track, FPA, and Z components are treated separately, in their response rates and the qualitative form.

First, the Z component of velocity is treated as a first order controller. In a classical first-order controller, one sets

```
desired x = x + tau dx/dt,
```

and attempts to match that constant rate of increase which would take one to the goal in time τ. As the FPA γ is determined by the Z component of velocity (divided by speed), the appropriate rate-constant to use is τ_γ. So the desired acceleration in the Z direction is as follows:

```
desired Az = (desired Az - Az) / tau gamma
```

Second, the (X,Y) component of velocity is treated as a kind of first order controller, with linear increase in magnitude (as with a one-dimensional first order controller), and constant rotation rate (which is novel, as far as I know). Because the (X,Y) direction determines track angle, the appropriate rate-constant for turning is τ_λ. Similarly, speed is adjusted using τ_S. Considering only the (X,Y) components of the vector, the desired behavior of steady rotation and steady scaling is modeled by the following equation.

```
V(t) = (a + bt)[cos(theta + t omega), sin(theta + t omega)]   (B)
```

Setting (B) to give V at $t = 0$ and $\dot{V}$ at time $t = \tau$ gives the desired parameters, and so determines the (X,Y) acceleration desired.

```
theta      = tan^-1(Vy, Vx)
theta      = tan^-1(desired Vy, desired Vx)
omega      = ANGLE-DIFF (desired theta, theta) / tau lambda
S          = sqrt(Vy^2 + Vx^2)
desired S  = sqrt(desired Vy^2 + desired Vx^2)
b          = (desired S - S) / tau S (C)
```

The function ANGLE-DIFF returns acute angle difference, so as to always turn through the smaller angle.

Differentiating (B) with respect to time gives the expression for the X and Y components of the desired acceleration, and the following overall expression:

```
desired Ay = -omega Vy + (b/S) Vx
desired Ax =  omega Vx + (b/S) Vy
desired Az =  desired Az - Az / tau gamma (D)
```

If there is no change in the (X,Y) speed, then $b = 0$ (D) gives an acceleration which is exactly perpendicular to the (X,Y) velocity, i.e. a pure rotation. But the rate of rotation declines at the angle left to traverse shrinks. Similarly, if there is no change in angle, then $\omega = 0$ and (D) gives an acceleration which is exactly along the (X,Y) velocity, i.e. a pure speed-up, with the rate of acceleration declining as the speed-gap shrinks.

It is worth noting that (C) and (D) are the key functions of a very nonlinear controller. For small angle or speed changes, it is essentially linear, while for large changes it is highly nonlinear but very well behaved.

It would be quite feasible to use (D) to determine the desired net acceleration in SLOW mode as well as FAST mode, if smooth turns were desired.

This is a fundamental improvement over the controller which this author produced for the ModSAF fixed wing aircraft dynamics. In the FWA controller, lateral motion (track and roll) were controlled completely independently from longitudinal ones (speed and FPA). This would produce noticable slowing in turns, loss of altitude, and so on. All these problems have now been fixed, as the controller above looks at the desired speed and rotation in one unified operation, so it correctly handles combined nonlinear maneuvers.

The desired net acceleration is then used in a force-balance equation to determine the desired $\hat{}R$ and $\hat{}J$ vectors. Thus, the rotor and helicopter forces must generate a net force which will cancel out the constant gravitational and nonlinear drag forces, then in addition impose the desired overall acceleration.

```
desired R + desired J = desired F = M desired A + Mg + D
```

How the three components of $\hat{}F$ are split up between $\hat{}R$ and $\hat{}J$ is what determines the two FAST modes.

In FAST-HORIZONTAL mode, the jet vector (and body orientation) is kept horizontal, with the rotor vector rotating in a plane perpendicular to it as the RWA rolls one way or another. The horizontal vector pointing along V is given by the following expression:

```
H = g cross (V cross g)
```

Then $\hat{}J$ is just that component of $\hat{}F$ parallel to H , while $\hat{}R$ is the component which is perpendicular. If $\hat{}J \cdot H < 0$, then the ideal motion would require reverse thrust, which is not possible, and the desired thrust vector is reset to a zero-vector.

In FAST-ALIGNED mode, $\hat{}J$ is just that component of $\hat{}F$ parallel to V , while $\hat{}R$ is the component which is perpendicular.

In the prototype LISP code, there is a global variable specifying whether or not inverted flight is allowed. For fast, tight terrain following, inversion is required (as how F15 flip when crossing ridge lines), as it might also be for aerial combat between RWA's. If inversion is not allowed, then the Z component of $\hat{}R$ is bounded from below by $mg/10$, so that it always points at least somewhat up, and the RWA never dives at more than -0.9 G's acceleration.

In both FAST modes, $\hat{}R$ is then immediately limited to what the rotor can currently generate, preserving the Z component if possible. Thus, it will make a slower turn if necessary to maintain altitude, and will only pull as many G's in the turn as it is physically capable of generating. The net effect in both modes is that the RWA will roll over to a specific bank angle, hold it steady through the turn, then roll out at the end.

Notice that the only way to rapidly slow down is to tilt the rotor backwards. Hence, if one tries to approach and stop at a point (hovering or landing) in either of the FAST modes, it will overshoot and start orbiting the point. There should be a means of doing an automatic mode change to avoid this problem, and to provide strong forward acceleration.

1.5 Limits and Clamping

Limit magnitude of R-hat and J-hat

Retreating blade stalls limit the RWA's forward velocity. This dynamic effect is not modeled, as the low-level controller would simply not request a velocity over that limit. This needs to be put in as a user-limit on speed.

Rate-limited update of R-hat and J-hat

Helo-bound-force-vector

Rho limits on lift

User limits on speed, turn rates, FPA. The maximum sustainable FPA was computed, as it is critical for FAST-ALIGNED mode (but not FAST-HORIZONTAL, which can do any FPA, including vertical rise/fall).

1.6 Calibration and Stability

The Comanche is reportedly able to do "snap turns" (i.e. a few seconds for 90 degrees) at up to 180 knots, or 92.6 m/s. [The limitation is supposedly retreating blade stall, seeming to imply that attempting to generate that much lift, with the low relative wind over the retreating blade, gets it right to the edge of a blade stall.] If it completes 90 degrees in 5 seconds, that requires a few G's of lateral acceleration and one to stay level, implying 3.1 G's total lift to complete the turn level, with no loss of speed.

As an example, the delivered LISP code assumed a 19,000 Kg mass, 80,000 Newtons of jet thrust, and 3.1 G's of rotor thrust. The drag coefficient is set at 16.49, so that the horizontal flight in FAST mode is drag-limited to 50% more than the above corner speed. The linear and quadratic components of drag are equal at 84.5 m/s.

Traditional linear control theory provides some guidance as to what will be stable combinations of parameters. However, the presence of rate limits, magnitude limits, and unpredictable step length all contribute to render a full nonlinear stability analysis quite difficult.

Some empirical testing and observation was done in addition to formal analysis.

Through all phases of development, the position of the RWA was randomly perturbed by up to 10cm along each axis at each tick, in an attempt to excite some unstable resonance. No problem was ever observed.

For testing purposes, a simple route-following procedure was used, in which the RWA aims at a point ahead of itself, a certain lead-distance down the route. Thus, on very long routes, it will rapidly pull onto the route, then follow it closely the rest of the way, rather than fly a very slowly converging straight line path toward the final end point. A higher level controller was written which sets the desired velocity $\hat{V}$ to be the position error divided by $\tau_p = 2.5$ seconds in SLOW mode and by 5.0 seconds in FAST mode. This will not be discussed further, as it was purely for a wrapper for testing.

With $\tau_R = 0.2$, $\tau_v = 1.0$, $\tau_p = 2.5$, and a maximum rotor lift of 3.1 G's, the SLOW and FAST modes were observed to be stable over the entire range of conditions described below. In SLOW mode, a formal analysis can be done to show that the eigenvalues are at $(-3.8312, -.5844 \pm 0.4249j)$. This corresponds to one mode with a time-constant of 0.261 seconds, and two decaying oscillatory ones with a time constant of 1.71 seconds and a period of 14.8 seconds. Thus, they decay into negligability before completing even a small fraction of a cycle. This was empirically confirmed by the lack of overshoot in turning, stopping in mid-air to hover, and landing within a few centimeters of a designated goal point.

If the max G's is limited to 2, then with the third-level control algorithm for testing, it starts overshooting in the Z direction because it can not generate enough force to pull out of rapid dives. Whether or not this occurs in the final ModSAF application depends on what algorithms are used to determine V , but it is an issue to be aware of. Clearly, looking only a short distance ahead will generally require larger forces, applied more rapidly, to acheive the desired result in the short time available - and limited force violates that assumption.

Time steps were randomly varied between 0.5 and 1.5 times the mean time step, and the mean time step was varied from 0.05 seconds to 0.75 seconds. This is broader than the requirements, as the maximum update rate ModSAF supports is 1/16 second, and 0.5 seconds is overload.

The effect of very short time ticks is make the dynamics and control behave in a more nearly ideal and continuous fashion. Short ticks do not cause sluggish response. In fact, longer ticks cause faster response in the translational modes. For example, if the time step is exactly tau V and equation (A) is used, then the velocity is matched exactly in one time step. As the ticks get shorter, a more realistic exponential decay is observed. If the ticks are slightly longer than τ_V , then a rapidly decaying oscillation is observed. When the ticks get out to $1.5\tau_V$, then oscillations are being halved in magnitude at each time step. In this calibration, $\tau_V = 1.0$, so none of these conditions will occur

until the system is well past its overload condition of 0.5 second time steps.

2 Parameters

Data file entry should read:

```
(SM_RWAHull
    (min_fpa                <float> degrees) ;; minimum dive angle
    (vel_response_time      <float> seconds) ;; rate of response when not
                                               rate limited
    (fpa_response_time      <float> seconds) ;; fpa response time in fast mode
    (track_response_time    <float> seconds) ;; respnse time in fast mode, when
                                               not g limited
    (thrust_response_time <float> seconds) ;; thrust response time in fast mode
    (jet_response_time      <float> seconds) ;; rate of change in jet thrust
    (rotor_response_time    <float> seconds) ;; rate of change of rotor thrust
    (rtr_ctrl_resp_time     <float> seconds) ;; time to change cntrl setting
    (jet_ctrl_resp_time     <float> seconds) ;; time to change cntrl setting
    (mass                   <float> Kg)      ;; vehicle mass
    (max_jet_thrust         <float> Newtons) ;; max jet thrust (assuming vehicle
                                                has a jet)
    (inversion              <int> True(1)/False(0)) ;; is inverison allowed?
    (has_jet                <int> True(1)/False(0)) ;; does this have a jet?
    (max_rotate_rate        <float> radians/sec) ;; maximum controlled rotating
                                                    rate
    (max_gs                 <float> gravitys) ;; max Gs the airframe can pull
    (vel_cruise             <float> metes/second) ;; cruise velocity
    (vel_max                <float> meters/second) ;; max foreward velocity
    (vel_max_back           <float> meters/second) ;; max velocity backwards
    (vel_max_up             <float> meters/second) ;; max velocity up
    (fuel_rate              <float> gallons/second) ;; gal per sec at 100% power
```

3 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);
```

4 Functions

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

4.1 rwa_init

```
void rwa_init()
```

rwa_init initializes librwa. Call this before any other librwa function.

4.2 rwa_class_init

```
void rwa_class_init(parent_class)
    CLASS_PTR parent_class;
```

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

rwa_class_init creates a handle for attaching rwa class information to vehicles. The **parent_class** will likely be **safobj_class**.

4.3 rwa_destroy

```
void rwa_damage(vehicle_id, damaged)
    int32 vehicle_id;
    int32 damaged;
```

'vehicle_id'
Specifies the vehicle ID

'damaged' This sets (TRUE) or unsets (FALSE) damage for the rwa component.

rwa_damage This sets the rwa component's current ability to function or not for the specified vehicle_id.

4.4 rwa_collision

```
void rwa_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'
: The vehicle who had the collision.

'position[3]'
: The position of the colision;

'coll_type'
: The type of collision

'other_id'
: The vehicle the collision was with (if any)

'other_mass'
: The mass of that other vehicle (if another vehicle)

'other_velocity[3]'
: The velocity of the other vehicle (if another vehicle)

rwa_collision Handles the RWA running into or being run into by something.

4.5 rwa_tick

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

'vehicle_id'
: Specifies the vehicle ID

'ctdb'
: Specifies the terrain database

rwa_tick ticks the RWA dynamics.

LibSAFGUI

Table of Contents

1 Overview 1

2 Usage 3

2.1 Building Libsafgui 3
2.2 Linking with Libsafgui 3
2.3 Examples 4

3 Functions 5

3.1 sgui_init 5
3.2 sgui_set_editor_toggle 5
3.3 sgui_shell_widget 5
3.4 sgui_realize 6
3.5 sgui_map_widget 6
3.6 sgui_editor_widget 6
3.7 sgui_map_scrollbars 7
3.8 sgui_add_menubar 7
3.9 sgui_manage_menubar 8
3.10 sgui_add_mode 8
3.11 sgui_set_mode 10
3.12 sgui_leave_mode 10
3.13 sgui_mode_control 11
3.14 sgui_push_help 11
3.15 sgui_pop_help 11
3.16 sgui_set_urgent_help 12
3.17 sgui_message_log 12
3.18 sgui_editor 12
3.19 sgui_add_message 13
3.20 sgui_clear_message_log 13
3.21 sgui_error 13
3.22 sgui_safe_focus 14

1 Overview

LibSAFGUI provides an environment in which functionality can be added to the SAF Graphical User Interface in a simple, modular fashion. It takes care of laying out all the major pieces (menubar, message log, editor, map, etc.), and provides functions to add new controls.

The library also takes care of *mode management*. The interface uses six different modes:

SGUI_SELECT_MODE
: Selection mode ("arrow" mode). Note that the software assumes only one such mode will be created.

SGUI_OBJECT_MODE
: Point, line, unit, etc. creation modes.

SGUI_TASKFRAME_MODE
: Task frame creation mode.

SGUI_ASSIGN_MODE
: Mission assignment mode

SGUI_TOOL_MODE
: Intervis, coordinate calculator, etc. modes.

SGUI_MAP_MODE
: Pan, zoom, etc. modes.

The system is always in one map mode, and activating a new map mode deactivates the previous one.

Transitions between the other modes are handled using the following algorithm:

user presses / current mode	SELECT	OBJECT	TASKFRAME	ASSIGN	TOOL
SELECT	-	PUSH	PUSH	PUSH	PUSH
OBJECT	POP	DISALLOWED	DISALLOWED	DISALLOWED	PUSH
TASKFRAME	POP	PUSH	DISALLOWED	DISALLOWED	PUSH
ASSIGN	POP	PUSH	PUSH	DISALLOWED	PUSH
TOOL	POP	POP/PUSH	POP/PUSH	POP/PUSH	POP/PUSH

In this table, **POP** indicates that the current mode is popped, and **PUSH** indicates that the pressed-on mode is pushed onto the top of the mode stack. **DISALLOWED** transitions are prevented by making the corresponding mode buttons in those modes insensitive.

To help the user keep track of his mode stack, help messages are pushed along with their corresponding modes. The user is shown the top three help messages.

A mode can also specify additional help messages which are shown when the mode is current.

The X resources are provided in the file safgui.xrdb which is installed in '**common/data**'. Any of these resources can be overridden by making changes to the user's private **.xresources** file on the X server machine.

2 Usage

The software library '**libsafgui.a**' should be built and installed in the directory '**/common/lib/**'. You will also need the header file '**libsafgui.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 libsafgui source directory. If these files are already built, you can skip the section on building libsafgui.

2.1 Building Libsafgui

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

If the directory '**common/libsrc/libsafgui**' 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/libsafgui
# co RCS/*,v
# make install
```

This should compile the library '**libsafgui.a**' and install it and the header file '**libsafgui.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. libsafgui should compile without errors on the following platforms:

- Mips
- SGI Indigo
- Sun Sparc

2.2 Linking with Libsafgui

Libsafgui can be linked into an application program with the following link time flags: '**ld [source .o files] -L/common/lib -lsafgui -lprivilege -lXm -lXt -X11**'. If your compiler does not support '**-L**' syntax, you can use the archive explicitly: '**ld [source .o files] /common/lib/libsafgui.a**

Libsafgui depends on libprivilege and Motif.

2.3 Examples

The test program 'test.c' in the libsafgui source directory creates an entire user interface. These are a few selected examples:

To create a quit button:

```
Widget w;
extern SGUI_PTR my_gui;
extern void quit();

w = sgui_add_menubar(my_gui, PRIV_SYSOP,
                     "File", "Quit", xmPushButtonWidgetClass);
XtAddCallback(w, XmNactivateCallback, quit);
```

To create the select mode:

```
SGUI_MODE_PTR mode;
extern SGUI_PTR my_gui;
extern void mode_callback();

/* Create the mode */
mode = sgui_add_mode(my_gui, SGUI_SELECT_MODE,
                     select_bits, select_width, select_height,
                     mode_callback, "Select", "Select an item to edit",
                     "selecting an item to edit");

/* Make this the default mode */
sgui_set_mode(mode);
```

To push a message onto the message log:

```
extern SGUI_PTR my_gui;

sgui_add_message(my_gui, "system", "This is a system message");
```

3 Functions

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

3.1 sgui_init

```
SGUI_PTR sgui_init(parent, title_string)
    Widget parent;
    char  *title_string;
```

'parent' Specifies parent widget (needn't be realized)

'title_string'
Specifies shell title (shown in window manager decorations)

sgui_init initializes a SAFSTATION GUI top level frame. This includes a menu bar, map area, mode buttons, etc. The returned handle is then passed to other libsafgui functions. The parent widget needn't be realized (the top level frame creates its own shell).

3.2 sgui_set_editor_toggle

```
void sgui_set_editor_toggle(gui, toggle)
    SGUI_PTR gui;
    Widget   toggle;
```

'gui' Specifies the graphical user interface

'toggle' Specifies the toggle button widget

sgui_set_editor_toggle gives the GUI a toggle button *widget* (not a gadget) which is to be kept in sync with the state of the editor (shown/hidden).

3.3 sgui_shell_widget

```
Widget sgui_shell_widget(gui)
    SGUI_PTR gui;
```

'gui' Specifies the graphical user interface

sgui_shell_widget returns the application shell used by the GUI.

3.4 sgui_realize

```
void sgui_realize(gui, big_cursor)
    SGUI_PTR gui;
    int32    big_cursor;
```

'gui' Specifies the graphical user interface

'big_cursor'
Specifies that an oversize cursor is desired

sgui_realize realizes the top level frame. Call this *after* creating the menu bar, the modes, etc. If the **big_cursor** flag is true, the library will register a very large (and easy-to-find) cursor with X for use on the GUI.

3.5 sgui_map_widget

```
Widget sgui_map_widget(gui)
    SGUI_PTR gui;
```

'gui' Specifies the graphical user interface

sgui_map_widget returns the map widget associated with the passed GUI. This can be passed to libsensitive, libtactmap, etc.

3.6 sgui_editor_widget

```
Widget sgui_editor_widget(gui)
    SGUI_PTR gui;
```

'gui' Specifies the graphical user interface

sgui_editor_widget returns the editor widget associated with the passed GUI. This can be passed to libeditor.

3.7 sgui_map_scrollbars

```
void sgui_map_scrollbars(gui, horizontal, vertical)
    SGUI_PTR gui;
    Widget  *horizontal;
    Widget  *vertical;
```

'gui' Specifies the graphical user interface

'horizontal'
Returns the horizontal map scroll bar

'vertical'
Returns the vertical map scroll bar

sgui_map_scrollbars returns the scroll bars attached to the map. These default to being unmanaged, and so must be managed by the application if they are to be used. It is up to another library (such as libPVD) to define callbacks for these scroll bars.

3.8 sgui_add_menubar

```
Widget sgui_add_menubar(gui, privilege, menu_name,
                        button_name, widget_class)
    SGUI_PTR    gui;
    PRIV_LEVEL  privilege;
    char       *menu_name;
    char       *button_name;
    WidgetClass widget_class;
```

'gui' Specifies the graphical user interface

'privilege'
Specifies the privilege level of the button

'menu_name'
Specifies the menu name (the button name in the menu bar)

'button_name'
Specifies the name of the button in the pulldown menu

'widget_class'
Specifies the class of the created widget (xmPushButtonWidgetClass, etc.)

sgui_add_menubar adds a button to the menubar structure. The menu_name identifies the label on the menu bar, and the button_name specifies the label within the pulldown. A widget of the passed class is created and returned so that activation callbacks, or other arguments can be set by the caller.

3.9 sgui_manage_menubar

```
void sgui_manage_menubar(gui, menu_name, shown)
    SGUI_PTR    gui;
    char       *menu_name;
    int32       shown;
```

'gui' Specifies the graphical user interface

'menu_name'
Specifies the menu name (the button name in the menu bar)

'shown' Specifies whether the cascade for the menu should be managed

sgui_manage_menubar changes the management of a cascade button on the menu bar. If shown==TRUE, the cascade will be managed; if shown==FALSE, the cascade will be unmanaged.

3.10 sgui_add_mode

```
SGUI_MODE_PTR sgui_add_mode(gui, class,
                            bitmap_data, bitmap_width, bitmap_height,
                            callback, callback_arg,
                            help_string, description)
    SGUI_PTR           gui;
    SGUI_MODE_CLASS    class;
    char              *bitmap_data;
    int32              bitmap_width;
    int32              bitmap_height;
    SGUI_MODE_CALLBACK callback;
    ADDRESS            callback_arg;
    char              *help_string;
    char              *description;
```

'gui' Specifies the graphical user interface

'class' Specifies the class of the mode

'bitmap_data'
'bitmap_width'
'bitmap_height'
Specify the bitmap which is placed on the mode button

'callback'
'callback_arg'
Specify the function to call when the mode is entered, exited, suspended or resumed

'help_string'
Specifies the default help string

'description'
Specifies a description of the what the user is doing in that mode

sgui_add_mode adds a mode button to the left side of the screen. The semantics of the mode are dependent upon its class. The callback is invoked when the mode is entered, exited, suspended, or resumed. The **description** should be something like "editing a point" or "doing area visibility calculations".

The mode classes are as follows:

SGUI_SELECT_MODE
Selection mode ("arrow" mode). Note that the software assumes only one such mode will be created.

SGUI_OBJECT_MODE
Point, line, unit, etc. creation modes.

SGUI_TASKFRAME_MODE
Task frame creation mode.

SGUI_ASSIGN_MODE
Mission assignment mode.

SGUI_TOOL_MODE
Intervis, coordinate calculator, etc. modes.

SGUI_MAP_MODE
Pan, zoom, etc. modes.

The callback should be declared as follows:

```
void callback(gui, mode, callback_arg, state)
```

```
SGUI_PTR        gui;
SGUI_MODE_PTR   mode;
ADDRESS         callback_arg;
SGUI_MODE_STATE state;
```

The **gui** and **callback_arg** are those passed to **sgui_add_mode**, and the **mode** is the one returned by that function. The **state** is one of the following:

SGUI_ACTIVE
: The mode is "current".

SGUI_SUSPENDED
: Another mode has been pushed on top of this mode.

SGUI_RESUMED
: The mode has been reactivated after being **SGUI_SUSPENDED**.

SGUI_INACTIVE
: The mode has been exited.

3.11 sgui_set_mode

```
void sgui_set_mode(mode)
    SGUI_MODE_PTR mode;
```

'mode' Specifies the mode (implies a GUI)

sgui_set_mode sets the passed mode (created with **sgui_add_mode**) to active. This should be used at initialization to set the default modes of the system. Once the system is running, this can be called to simulate user behavior; however, since some modes are not always available, this may not always succeed at run time.

3.12 sgui_leave_mode

```
void sgui_leave_mode(gui)
    SGUI_PTR gui;
```

'gui' Specifies the graphical user interface

sgui_leave_mode exits the current mode. This should be called by editor *Done* and *Abort* buttons. It is equivialent to calling **sgui_set_mode** with a mode of the **SELECT** class.

3.13 sgui_mode_control

```
void sgui_mode_control(mode, disable)
    SGUI_MODE_PTR mode;
    uint32        disable;
```

'mode' Specifies the mode

'disable' Specifies whether that mode should be disabled (regardless of the current mode or its semantics)

sgui_mode_control sets control flags for the passed mode. Currently this is just a **disable** flag, which causes the mode to be insensitive when set.

3.14 sgui_push_help

```
void sgui_push_help(mode, help_string)
    SGUI_MODE_PTR mode;
    char          *help_string;
```

'mode' Specifies the current mode

'help_string' Specifies the string to push

sgui_push_help pushes an additional help message for the passed mode.

3.15 sgui_pop_help

```
void sgui_pop_help(mode, pop_all)
    SGUI_MODE_PTR mode;
    int32         pop_all;
```

'mode' Specifies the current mode

'pop_all' Specifies whether to pop all, or just topmost

sgui_pop_help pops the additional mode help message (if any) for the passed mode. If pop_all is TRUE, all pushed messages will be popped; otherwise, only the topmost will be.

3.16 sgui_set_urgent_help

```
void sgui_set_urgent_help(mode, help_string)
    SGUI_MODE_PTR mode;
    char          *help_string;
```

'mode' Specifies the current mode

'help_string'
Specifies the string to show or NULL

sgui_set_urgent_help pushes an additional help message for the passed mode. The message is displayed above other help messages in a distracting manner. Pass NULL to remove the urgent message.

3.17 sgui_message_log

```
void sgui_message_log(gui, shown)
    SGUI_PTR gui;
    int32    shown;
```

'gui' Specifies the graphical user interface

'shown' Specifies whether to show the message log

sgui_message_log changes the status of the message log (shown=TRUE, hidden=FALSE).

3.18 sgui_editor

```
void sgui_editor(gui, shown)
    SGUI_PTR gui;
    int32    shown;
```

'gui' Specifies the graphical user interface

'shown' Specifies whether to show the editor

sgui_editor changes the status of the editor (shown=**TRUE**, hidden=**FALSE**).

3.19 sgui_add_message

```
void sgui_add_message(gui, kind, message)
    SGUI_PTR gui;
    char     *kind;
    char     *message;
```

'gui' Specifies the graphical user interface

'kind' Specifies the kind of message

'message' Specifies the message to log

sgui_add_message adds a **message** to the message log. The **kind** is used to name the output widget, so that resources (fonts, colors, etc.) can be used to customize output styles; it must be statically declared (such as a quoted string constant, or a libreader symbol).

3.20 sgui_clear_message_log

```
void sgui_clear_message_log(gui)
    SGUI_PTR gui;
```

'gui' Specifies the graphical user interface

sgui_clear_message_log empties the message log.

3.21 sgui_error

```
void sgui_error(gui, format, args...)
    SGUI_PTR gui;
    char     format[];
```

'gui' Specifies the graphical user interface

'format' Specifies the format of the error (like fprintf)

'args' Speicifies arguments to format

sgui_error reports an error on the GUI, in a manner similar to fprintf. The error is reported in three places simultaneously: **stderr**, the message log (kind == "**system**"), and in a popup. This should be only be called for errors which might be meaningful to a user.

3.22 sgui_safe_focus

```
void sgui_safe_focus(gui)
    SGUI_PTR gui;
```

'gui' Specifies the graphical user interface

sgui_safe_focus moves Motif's input focus to a "safe" place (the button for select mode). This can be used when an editor is exited to guarantee that focus does not remain with the inactive editor.

Libsafobj

Table of Contents

1 Overview 1

2 Algorithms 5

2.1 Local Safobj Creation 5
2.2 Local Safobj Updating 6
2.3 Remote Safobj Creation 6
2.4 Remote Safobj Updating 7

3 Global Variables 9

3.1 safobj_local_added_event 9
3.2 safobj_local_deleted_event 9

4 Functions 11

4.1 safobj_init 11
4.2 safobj_create_remote 11
4.3 safobj_create_local 12
4.4 safobj_create_local_with_id 12
4.5 safobj_destroy_local 13
4.6 safobj_destroy_remote 13
4.7 safobj_create_preview 14
4.8 safobj_mobility_kill 14
4.9 safobj_fire_kill 15
4.10 safobj_catastrophic_kill 15

1 Overview

Libsafobj provides the class superstructure in which all vehicle subclasses reside. There are two kinds of SAF objects, locals and remote. The subclasses which make up locals, and the default configurations of these subclasses for each type of vehicle are stored in a configuration file. Remotes are always made up of a fixed set of subclasses (entity, pbtab, and others).

Examples of ModSAF local vehicles include: airplanes, tanks, and missiles. Since the simulation requirements are less demanding for a missile, its class superstructure does not need to be as extensive as that needed for an airplane or tank.

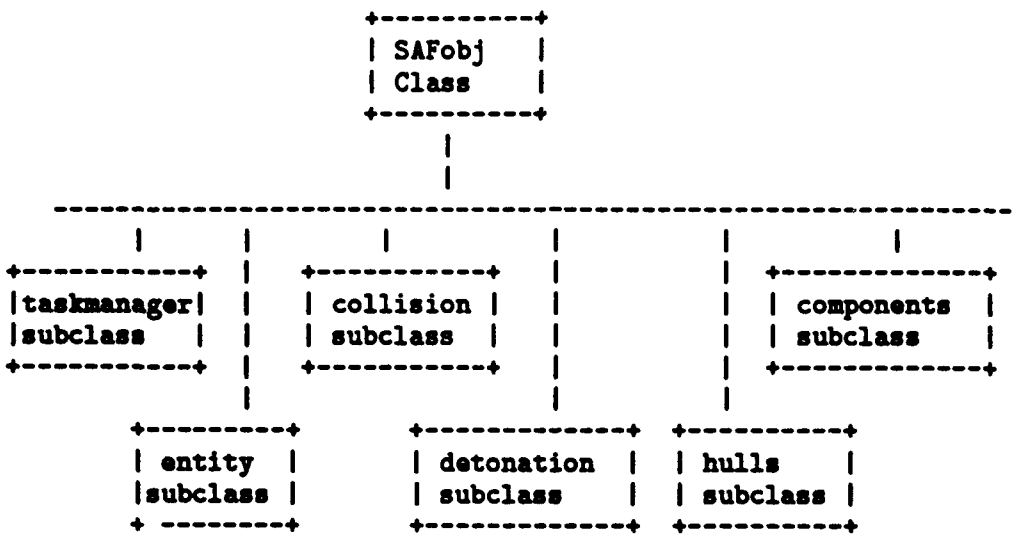
Assume that the configuration file that defines the model parameters for a missile had the following contents:

```
US_SPARROW_MODEL_PARAMETERS {
(SM_TaskManager)
(SM_Entity (length_threshold 10.0)
           (width_threshold 10.0)
           (height_threshold 10.0)
           (rotation_threshold 3.0)
           (turret_threshold 3.0)
           (gun_threshold 3.0)
           (vehicle_class vehicleClassSimple)
           (guises munition_US_Sparrow munition_US_Sparrow))
(SM_Collision (check buildings platforms)
              (announce buildings)
              (duration 5000)
              (feature_mass 10000.0)
              (fidelity high))
(SM_Detonation (check platforms)
               (announce buildings)
               (detonation_radius 20.0)
               (fidelity low))
(SM_Components (hull SM_MissileHull))
(SM_MissileHull (sensor_name apg-71)
                (sensor_on_board false)
                (pursuit_mode lead_pursuit)
                (range 22880.0)
                (launch_speed 500.0)
                (safe_time 1.0)
                (loal_time 2.0)
                (max_burn_time 60.0)
                (burn_max_turn 5.0)
                (coast_max_turn 5.0)
                (directionality 12.566370614359))
```

}

The class superstructure to correspond to the object defined above is shown below.

Class Superstructure When SAF Object (SAFobj) is Local Missile Vehicle



The sample configuration file indicates the modeling parameters and and subclasses (libraries) necessary for handling the simulation requirements for a Sparrow missile. The libentity, libcollision, libdetonation, and libtaskmgr libraries take care of making a simulated Sparrow missile a simulated network entity that can strike (collide with) a target, detonate, and perform tasks. In addition, a simulated Sparrow missile will have one simulated component (a hull).

The class superstructure for an airplane or tank includes many of the subclasses (such as task manager, collision, components) required for a missile. However, many more subclasses are required to meet the simulation requirements of an airplane or tank. Some of these additional simulation requirements include the following actions:

- calculate how it is damaged when hit by direct fire;
- keep track of the current level of its fuel and ammunition;
- have it "appear in" the position-based table (pbtab);
- have it "appear in" the vehicle table (vtab);
- and spot other vehicles.

Adding a subclass requires making the following changes to this library:

1. Modify **safobj_init** to initialize the subclass.
2. Modify **safobj_create_local_with_id** to create the subclass at the appropriate point (after those subclasses in lower layers, and before those in higher layers).
3. Modify **safobj_destroy_local** to also destroy the new subclass.
4. Modify **safobj_install_handlers** (a private function) to install handlers for all new events which the subclass needs (such as the receipt of a particular kind of packet).
5. Extend existing handlers or write new handlers for all events which the subclass needs, putting the invocation of the event at an appropriate point relative to other subclass handlers.

2 Algorithms

Libsafobj can create or update a SAF object (safobj). A safobj is considered local when it originates from this computer and remote when it originates elsewhere on the LAN.

2.1 Local Safobj Creation

The function **safobj_create_local** is used when making a local safobj such as a plane or tank vehicle. Input passed to this function serves as a configuration file key to find default values for all subclasses. An input unit entry identifies the unit that caused the creation of this object, it may be NULL. The steps performed by this function to create a local safobj are as follows:

1. Calls **vtab_vehicle_number** in libvtab to generate a vehicle table (vtab) id from the entity's network id.
2. Makes the local safobj by creating a block of user data to hold the pointers to the subclass information and then stores that pointer as the **user_data** of the libvtab vehicle.
3. Composes the safobj subclasses by calling a create function for all the subclasses that make up a local vehicle (all subclasses that are listed in the object's configuration file). For example, a call to **pbt_create** in libpbtab is made when the configuration file has a SM_PBTab entry. This call will result in the creation of the position-based table (pbt) class information and the attachment of that information to the vehicle's block of libclass user data. More information about a safobj subclass can be found in the ModSAF library documentation and in the ModSAF Programmmer's Guide (see section 'ModSAF Software Architecture' in ModSAF Programmer's Guide).
4. Calls **prev_create** in libpreview to create the preview class information and attach it to the vehicle's block of libclass user data. Libpreview handles the ModSAF Previewer, an optional part of the user interface which can provide a rudimentary 3D out-the-window view of terrain and 3D dynamic models for network entities.
5. Calls **pvd_create** in libpvd to create the plan view display (pvd) class information and attach it to the vehicle's block of libclass user data. Libpvd handles the ModSAF PVD, the tactical map shown on the user interface. The PVD reflects the updated positions of network entities.
6. Posts the periodic invocation of **local_tick** with libsched to schedule subclass ticking according to the **tick_rate** value.
7. Fires the event named **safobj_local_added_event** to say that there is one more locally created vehicle now. Applications that have attached a handler to this event, will have their registered callback functions invoked.
8. If this is a unit entry, then call **pm_monitor_unit** to monitor changes made to the unit.

9. Returns the vehicle ID of the new vehicle (or 0, meaning creation failed).

2.2 Local Safobj Updating

When a safobj is ticked, the function `local_tick` does the following:

1. Checks to make sure the input vehicle id is present in the vehicle table. Exits if that id is not found in the vehicle table.
2. Performs vehicle tick processing in the following order: input PDU's, dynamics, output PDU's, sensors, tasks, and user interface. PDU input processing is performed first since an input event such as a collision or impact could disable a vehicle component. Dynamics processing, which includes hull and turret movement and weapon firing, is performed by ticking the collection of hull, turret, and gun components that a particular vehicle has available. Collision and detonation detections are also part of dynamics processing. Once dynamics processing has finished, enough processing has occurred to warrant PDU output processing which includes the sending of vehicle appearance and designator packets. Sensor processing, which only occurs if the vehicle is still alive, handles the visual and radar sensory input processing. Task processing follows since tasks often require a list of detected vehicles generated from the sensor processing. The last step of a vehicle tick is the updating of the ModSAF user interface.

2.3 Remote Safobj Creation

A remote safobj can be created or updated when a vehicle appearance, designate, or stealth packet is received. When a remote entity is first received, an entry is made in the vehicle table with the appropriate type since the the per-vehicle subclass information and the processing requirements for a safobj differ according to its type.

The function **safobj_create_remote**, used when making a remotely created safobj, does the following:

1. Makes the remote safobj by creating a block of user data to hold the pointers to the subclass information and then stores that pointer as the **user_data** of the libvtab vehicle.
2. Composes the safobj subclasses by calling a create function for all the subclasses that make up the remote. The subclasses needed by a remote differ according to its type. When the type is a `VTAB_REMOTE_VEHICLE`, `VTAB_REMOTE_MISSILE`, `VTAB_REMOTE_STRUCTURE`, or `VTAB_REMOTE_DESIGNATOR`, `pbt_create` in libpbtab is called. When the type is a

VTAB_REMOTE_DESIGNATOR, dsg_create in libdesignate is called. When the type is VTAB_REMOTE_STEALTH, stealth_create in libstealth is called. When the type is a VTAB_REMOTE_VEHICLE, VTAB_REMOTE_MISSILE, or VTAB_REMOTE_STRUCTURE, ent_create in libentity is called. These functions build the subclass information and attach it to the vehicle's block of libclass user data.

3. Calls prev_create in libpreview to create the preview class information and attach it to the vehicle's block of libclass user data. Libpreview handles the ModSAF preview, an optional part of the user interface which provides a rudimentary 3D out-the-window view of terrain and 3D dynamic models for network entities.
4. Calls pvd_create in libpvd to create the plan view display (pvd) information and attach it to the vehicle's block of libclass user data. Libpvd handles the ModSAF PVD, the tactical map shown on the user interface. The PVD reflects the updated positions of network entities.
5. Posts the periodic invocation of remote_tick with libsched to schedule subclass ticking according to the remote_tick_rate value.

2.4 Remote Safobj Updating

When a remote safobj is ticked, the function remote_tick does the following:

1. Exits if the input vehicle id is not present in the vehicle table (vtab).
2. Invokes entity tick processing via a call to ent_tick in libentity. Exits if ent_tick times out the remote causing the vehicle to no longer be present in the vehicle table.
3. Invokes the designator tick processing via a call to dsg_tick in libdesignator. Exits if dsg_tick times out the remote causing the vehicle to no longer be present in the vehicle table.
4. Invokes the stealth tick processing via a call to stealth_tick. Exits if stealth_tick times out the remote causing the vehicle to no longer be present in the vehicle table.
5. Updates the user interface by a call to pvd_tick in libpvd.

3 Global Variables

The sections below describe the global variables by including a synopsis and a description.

3.1 safobj_local_added_event

```
extern CALLBACK_EVENT_PTR safobj_local_added_event;
```

safobj_local_added_event is a libcallback event which can be accessed after libsafobj is initialized. Applications may attach a **safobj_local_added_event_handler** to this event via (see section 'callback_register_handler' in LibCallback Programmer's Manual).

```
void safobj_local_added_event_handler(vehicle_id, user_data)
    int32   vehicle_id;
    ADDRESS user_data;
```

safobj_local_added_event_handler is called when a new local entity is simulated. **vehicle_id** will contain the id of the new vehicle being simulated.

3.2 safobj_local_deleted_event

```
extern CALLBACK_EVENT_PTR safobj_local_deleted_event;
```

safobj_local_deleted_event is a libcallback event which can be accessed after libsafobj is initialized. Applications may attach a **safobj_local_deleted_event_handler** to this event via (see section 'callback_register_handler' in LibCallback Programmer's Manual).

```
void safobj_local_deleted_event_handler(vehicle_id, user_data)
    int32   vehicle_id;
    ADDRESS user_data;
```

safobj_local_deleted_event_handler is called when a local entity is about to be destroyed. **vehicle_id** will contain the id of the vehicle which is about to be destroyed.

4 Functions

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

4.1 safobj_init

```
void safobj_init(ctdb, db, quad_data, ent_valve)
    CTDB         *ctdb;
    PO_DATABASE *db;
    QUAD_DATA    *quad_data;
    PV_VALVE_PTR ent_valve;
```

'ctdb' Specifies the terrain used in this exercise

'db' Specifies the persistent object database used in this exercise

'ent_valve'
Specifies the packet valve used in this exercise for creating local vehicles

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

4.2 safobj_create_remote

```
void safobj_create_remote(vehicle_id, simnet_id, vtab_type, tick_rate)
    int32      vehicle_id;
    VehicleID *simnet_id;
    uint16     vtab_type;
    int32      tick_rate;
```

'vehicle_id'
Specifies the libvtab ID assigned to the vehicle.

'simnet_id'
Specifies the vehicle's network VehicleID.

'vtab_type'
Specifies the vtab type of the vehicle.

'tick_rate'
Specifies the rate at which the vehicle should tick.

safobj_create_remote creates a remote vehicle with the passed parameter values. Automatically creates pbtab and entity subclasses, and posts **ent_tick** function on scheduler.

See section 'remote_init' in LibRemote Programmer's Guide.

4.3 safobj_create_local

```
int32 safobj_create_local(name_symbol, vtab_type, tick_rate, unit_entry)
    char        *name_symbol;
    uint16       vtab_type;
    int32        tick_rate;
    PO_DB_ENTRY *unit_entry;
```

'name_symbol'
Specifies the symbolic name of the type of vehicle to create.

'vtab_type'
Specifies the vtab type of the vehicle.

'tick_rate'
Specifies the rate at which the vehicle should tick.

'unit_entry'
Itendifies the unit which caused the creation of this object. It may be null.

safobj_create_local creates a local vehicle. The **name_symbol** is used as the configuration file key to find default values for all subclasses. The vehicle ID of the new vehicle is returned (or 0, meaning creation failed).

4.4 safobj_create_local_with_id

```
int32 safobj_create_local_with_id(name_symbol, vtab_type, tick_rate,
                                  simnet_id, unit_entry)
    char        *name_symbol;
    uint16       vtab_type;
    int32        tick_rate;
    VehicleID   *simnet_id;
    PO_DB_ENTRY *unit_entry;
```

'name_symbol'
Specifies the symbolic name of the type of vehicle to create.

'vtab_type'
Specifies the vtab type of the vehicle.

'tick_rate'
Specifies the rate at which the vehicle should tick.

'simnet_id'
Specifies the network id to be used for this vehicle. This is useful when creating a vehicle that used to be simulated by a remote simulator (this process is referred to as migration), or to create a vehicle on behalf of another simulator.

'unit_entry'
Itendifies the unit which caused the creation of this object. It may be null.

safobj_create_local_with_id creates a local vehicle. The **name_symbol** is used as the configuration file key to find default values for all subclasses. The vehicle ID of the new vehicle is returned (or 0, meaning creation failed).

4.5 safobj_destroy_local

```
void safobj_destroy_local(vehicle_id, is_migration)
    int32 vehicle_id;
    int32 is_migration;
```

'vehicle_id'
Specifies the vehicle to destroy.

'is_migration'
Specifies whether the vehicle is really being destroyed (**FALSE**), or just migrating to another host (**TRUE**).

safobj_destroy_local destroys a local vehicle and frees all the memory associated with that vehicle (including all subclasses).

4.6 safobj_destroy_remote

```
void safobj_destroy_remote(vehicle_id)
    int32 vehicle_id;
```

'vehicle_id'
Specifies the vehicle to destroy.

safobj_destroy_remote destroys a remote vehicle and frees all the memory associated with that vehicle (including all subclasses). Remotes are usually automatically destroyed when timed out by libpktvalve. Some applications may use this function during vehicle migration to destroy the remote version of an entity before simulating it as a local vehicle.

4.7 safobj_create_preview

```
int32 safobj_create_preview(previewer, preview_number, tick_rate)
    PREV_PTR previewer;
    int32 preview_number;
    uint32 tick_rate;
```

'previewer'
Specifies the previewer to insert into the vehicle table

'preview_number'
Specifies the number of the previewer (in case there are more than one previews).

'tick_rate'
Specifies the rate at which the previewer should tick.

safobj_create_preview creates an entry in the vehicle table for a **LOCAL_STEALTH** view. This allows the GUI to show the position of each preview on the map.

4.8 safobj_mobility_kill

```
void safobj_mobility_kill(vehicle_id)
    int32 vehicle_id;
```

'vehicle_id'
Specifies the local vehicle to mobility kill.

safobj_mobility_kill performs the processing neccesary to mobility kill a local vehicle.

4.9 safobj_fire_kill

```
void safobj_fire_kill(vehicle_id)
    int32 vehicle_id;
```

'vehicle_id'
Specifies the local vehicle to firepower kill.

safobj_fire_kill performs the processing neccesary to firepower kill a local vehicle.

4.10 safobj_catastrophic_kill

```
void safobj_catastrophic_kill(vehicle_id)
    int32 vehicle_id;
```

'vehicle_id'
Specifies the local vehicle to catastrophic kill.

safobj_catastrophic_kill performs the processing neccesary to catastrophic kill a local vehicle.

Table of Contents

1 Overview 1
1.1 Examples 1

2 Functions 3
2.1 safsam_init 3
2.2 safsam_init 3
2.3 safsam_class_init 3
2.4 safsam_create 4
2.5 safsam_destroy 4

1 Overview

TEMPLATE: Describe what this library does here.

1.1 Examples

TEMPLATE: Give examples here.

2 Functions

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

TEMPLATE: Adjust alignment of descriptions

TEMPLATE: Correct argument lists and descriptions of these functions.

2.1 safsam_init

```
void safsam_init(po_db, tcc)
    PO_DATABASE  *po_db;
    COORD_TCC_PTR tcc;
```

'po_db' Specifies the PO database

'tcc' Specifies the TCC

safsam_init initializes libsafsam. Call this before any other libsafsam function.

2.2 safsam_init

```
void safsam_global_tick()
```

safsam_global_tick ticks all the samuel methodology vehicles.

2.3 safsam_class_init

```
void safsam_class_init(parent_class)
    CLASS_PTR parent_class;
```

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

safsam_class_init creates a handle for attaching safsam class information to vehicles. The **parent_class** will likely be **safobj_class**.

2.4 safsam_create

```
void safsam_create(vehicle_id, params, unit_entry)
    int vehicle_id;
    SAFSAM_PARAMETRIC_DATA *params;
    PO_DB_ENTRY *unit_entry;
```

'vehicle_id'
Specifies the vehicle ID

'params' Specifies initial parameter values

'unit_entry'
Specifies the unit being created

safsam_create creates the safsam class information for a vehicle and attaches it vehicle's block of libclass user data.

2.5 safsam_destroy

```
void safsam_destroy(vehicle_id)
    int vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

safsam_destroy frees the safsam class information for a vehicle. This should be called before freeing the class user data with **class_free_user_data**.

Table of Contents

1 Overview 1

2 Examples 3

3 Functions 5

3.1 safsoar_init 5
3.2 safsoar_class_init 5
3.3 safsoar_create 5
3.4 safsoar_destroy 6

1 Overview

TEMPLATE: Describe what this library does here.

2 Examples

TEMPLATE: Give examples here.

3 Functions

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

TEMPLATE: Adjust alignment of descriptions

TEMPLATE: Correct argument lists and descriptions of these functions.

3.1 safsoar_init

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

'db' Specifies PO database

safsoar_init initializes libsafsoar. Call this before any other libsafsoar function.

3.2 safsoar_class_init

```
void safsoar_class_init(parent_class)
    CLASS_PTR parent_class;
```

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

safsoar_class_init creates a handle for attaching safsoar class information to vehicles. The **parent_class** will likely be **safobj_class**.

3.3 safsoar_create

```
void safsoar_create(vehicle_id, params, unit_entry)
    int                    vehicle_id;
    SAFSOAR_PARAMETRIC_DATA *params;
    PO_DB_ENTRY            *unit_entry;
```

'vehicle_id'
Specifies the vehicle ID

'params' Specifies initial parameter values

'unit_entry'
Specifies unit entry

safsoar_create creates the safsoar class information for a vehicle and attaches it vehicle's block of libclass user data.

3.4 safsoar_destroy

```
void safsoar_destroy(vehicle_id)
    int vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

safsoar_destroy frees the safsoar class information for a vehicle. This should be called before freeing the class user data with **class_free_user_data**.

LibSched

Table of Contents

1 Overview 1

2 Examples 3

3 Functions 5

3.1 sched_init 5
3.2 sched_invoke_functions_until 5
3.3 sched_invoke_functions_once 5
3.4 sched_invoke_realtime_functions_once 6
3.5 sched_invoke_simtime_functions_once 6
3.6 sched_deferred_fncl 6
3.7 sched_simtime_fncl 7
3.8 sched_periodic_fncl 8
3.9 sched_cancel_fncl 9
3.10 sched_cancel_fncl_group 9
3.11 sched_change_fncl_period 9
3.12 sched_perf_monitor_on 10
3.13 sched_perf_monitor_off 10
3.14 sched_set_critical_performance 10

1 Overview

Libsched is a non-preemptive, ring based scheduler which calls functions either periodically, or once after a specified delay. The application program defines the number of rings, and the period of each. Libsched will then invoke all scheduled functions in sequence, invoking a given function at a rate no *faster* than that specified for its ring.

Rings are not prioritized. Hence, as the system load increases, functions in faster rings may be called with the same frequency as functions in slower rings.

The scheduler is self monitoring, so that if system performance drops below an acceptable minimum, the application will be notified.

Although the ModSAF system uses the scheduler to run the entire system, it can be used in a slave mode as well. For example, a tank simulator could perform all synchronous tick processing, then start the scheduler with a deadline of just prior to the next frame. The scheduler will invoke periodic and deferred functions until that time is passed, at which point control returns to the simulator main loop.

There are also calls to have the scheduler invoke all functions that are pending and then return.

The scheduler bases its timing on libtime's `time_realtime_clock`. Periodic and deferred function calls are relative to this real time clock. Libsched also provides a facility to specify that a function be called at a specific time on the simulation clock (see Section 3.7 [sched_simtime_fncl], page 7).

2 Examples

The extensive test program '`test.c`' in the libsched source directory shows example usage of all libsched functions. Here are some highlights:

To call a function `thingy_tick(5, 3.4, "foo")` once every 67 milliseconds:

```
tick = sched_periodic_fncl(thingy_tick, SCHED_ASAP, 67, 0,
                           A_INT, 5,
                           A_DOUBLE, 3.4,
                           A_PTR, "foo",
                           A_END);
```

To call a function `stop_doing_that()` after a 5132 millisecond delay:

```
stop = sched_deferred_fncl(stop_doing_that, 5132, 0, A_END);
```

To cancel the call to `stop_doing_that()` before it occurs:

```
sched_cancel_fncl(stop);
```

To call a function `do_that_now()` in 2000 *simulated* milliseconds:

```
sched_simtime_fncl(do_that_now, time_last_simulation_clock+2000, 0,
                   A_END);
```

3 Functions

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

3.1 sched_init

```
void sched_init(nrings, period[0], ..., period[nrings-1])
    int32  nrings;
    uint32 period[0], ..., period[nrings-1];
```

'nrings' Specifies number of event rings periods.

'period[0], ..., period[nrings-1]'
Specify durations of each period in increasing order (in milliseconds).

`sched_init` initializes the scheduler. It specifies how many periodic event rings will be used and their periods. The private constant `MAX_RINGS` in '`libsched_local.h`' sets the upper limit (it can be increased to be arbitrary large if need be). The current value of this constant is 8.

3.2 sched_invoke_functions_until

```
void sched_invoke_functions_until(deadline)
    uint32 deadline;
```

'deadline'
Specifies when to stop invoking functions.

`sched_invoke_functions_until` runs the scheduler until the specified deadline. The time used is that from `time_realtime_clock` (see section 'time_realtime_clock' in LibTime Programmer's Manual). A deadline of `0xFFFFFFFF` will run indefinitely.

3.3 sched_invoke_functions_once

```
void sched_invoke_functions_once()
```

sched_invoke_functions_once forces the scheduler to invoke all pending functions then return. That is, all functions that have been scheduled to be invoked at or before the time **sched_invoke_functions_or** is called will be invoked. The time used is that from **time_realtime_clock** or **time_simulation_time**. No calls are made to **time_advance_simulation_clock** so simulated time is not advanced. (see section 'time_realtime_clock' in LibTime Programmer's Manual).

3.4 sched_invoke_realtime_functions_once

```
void sched_invoke_realtime_functions_once()
```

sched_invoke_realtime_functions_once forces the scheduler to invoke all pending realtime functions then return. That is, all functions that have been scheduled to be invoked at or before the realtime **sched_invoke_realtime_functions_once** is called will be invoked. The time used is that from **time_realtime_clock**. No calls are made to **time_advance_simulation_clock** so simulated time is not advanced. (see section 'time_realtime_clock' in LibTime Programmer's Manual).

3.5 sched_invoke_simtime_functions_once

```
void sched_invoke_simtime_functions_once()
```

sched_invoke_simtime_functions_once forces the scheduler to invoke all pending simulation time functions then return. That is, all functions that have been scheduled to be invoked at or before the simulation time **sched_invoke_simtime_functions_once** is called will be invoked. The time used is that from **time_simulation_time**. No calls are made to **time_advance_simulation_clock** so simulated time is not advanced. (see section 'time_realtime_clock' in LibTime Programmer's Manual).

3.6 sched_deferred_fncl

```
SCHED_FNCL_PTR sched_deferred_fncl(func, delay_msecs, group,
                                   argtype, arg,
                                   argtype, arg,
                                   ...
                                   A_END)
    void  (*func)();
    uint32  delay_msecs;
```

```
int32   group;
int32   argtype;
argtype arg;
```

'func' Specifies the function to call.

'delay_msecs'
Specifies the real time (in milliseconds) delay before calling this function.

'group' Specifies a group.

'argtype' Specifies the type of each argument. Chosen from the set: A_INT, A_DOUBLE, A_PTR, A_SHORT, A_CHAR, A_FLOAT (defined in common/include/global/stdext.h).

'arg' Argument to the function.

sched_deferred_fncl calls the specified function after the specified delay (as measured against libtime's *realtime* clock). The group (typically a vehicle ID) can be used later to remove the function (see Section 3.10 [sched'cancel'fncl'group], page 9).

A zero-duration delay is defined:

```
#define SCHED_ASAP 0
```

which can be used to invoke a function as soon as the current thread completes its execution.

Up to four arguments are permitted.

See Section 3.7 [sched'simtime'fncl], page 7.

3.7 sched_simtime_fncl

```
SCHED_FNCL_PTR sched_simtime_fncl(func, simulation_clock, group,
                                  argtype, arg,
                                  argtype, arg,
                                  ...
                                  A_END)
    void  (*func)();
    uint32  simulation_clock;
    int32   group;
    int32   argtype;
    argtype arg;
```

'func' Specifies the function to call.

'simulation_clock'
Specifies the simulation time when this function should be called (note this is *not* in real-time).

'group' Specifies a group.

'argtype' Specifies the type of each argument. Chosen from the set: A_INT, A_DOUBLE, A_PTR, A_SHORT, A_CHAR, A_FLOAT (defined in common/include/global/stdext.h).

'arg' Argument to the function.

sched_simtime_fncl calls the specified function at the specified simulation clock time (*not* a realtime clock value). The group (typically a vehicle ID) can be used later to remove the function (see Section 3.10 [sched_cancel_fncl_group], page 9).

See Section 3.6 [sched_deferred_fncl], page 6.

3.8 sched_periodic_fncl

```
SCHED_FNCL_PTR sched_periodic_fncl(func, initial_delay, period, group,
                                   argtype, arg,
                                   argtype, arg,
                                   ...
                                   A_END)
    void  (*func)();
    uint32  initial_delay;
    uint32  period;
    int32   group;
    int32   argtype;
    argtype arg;
```

'func' Specifies the function to call periodically.

'initial_delay'
Specifies the real time delay before calling this function the first time (in milliseconds).

'group' Specifies a group.

'argtype' Specifies the type of each argument. Chosen from the set: A_INT, A_DOUBLE, A_PTR, A_SHORT, A_CHAR, A_FLOAT (defined in common/include/global/stdext.h).

'arg' Argument to the function.

sched_periodic_fncl calls the specified function after the specified **initial_delay**, and every **period** milliseconds thereafter (as measured against libtime's *realtime* clock). The group (typically a vehicle ID) can be used later to remove the function (see Section 3.10 [sched_cancel_fncl_group], page 9).

Up to four arguments are permitted.

3.9 sched_cancel_fncl

```
void sched_cancel_fncl(fncl)
    SCHED_FNCL_PTR fncl;
```

'fncl' Specifies deferred or periodic function to cancel.

sched_cancel_fncl cancels the specified function (created with either **sched_periodic_fncl** or **sched_deferred_fncl**).

3.10 sched_cancel_fncl_group

```
void sched_cancel_fncl_group(group)
    int32 group;
```

'group' Specifies the group.

sched_cancel_fncl_group cancel all functions which were created with the specified group value. This is typically used to cancel all the functions associated with a particular vehicle.

3.11 sched_change_fncl_period

```
void sched_change_fncl_period(fncl, new_period_msecs)
    SCHED_FNCL_PTR fncl;
    int32          new_period_msecs;
```

'fncl' Specifies the periodic function to change.

'new_period'
Specifies the new period.

sched_change_fncl_period change the period of the passed function (created with **sched_periodic_fncl**) to the new period specified.

3.12 sched_perf_monitor_on

```
void sched_perf_monitor_on(period)
    uint32 period;
```

'period' Specifies interval between reports (in milliseconds).

sched_perf_monitor_on enables performance monitoring, and report results every **period** milliseconds.

3.13 sched_perf_monitor_off

```
void sched_perf_monitor_off()
```

sched_perf_monitor_off disable performance monitoring.

3.14 sched_set_critical_performance

```
void sched_set_critical_performance(period, ratio,
                                    under_threshold, for_how_long,
                                    stress_function, relief_function)
    uint32  period;
    float64 ratio;
    uint32  under_threshold;
    uint32  for_how_long;
    void  (*stress_function)();
    void  (*relief_function)();
```

'period, ratio, under_threshold, for_how_long'
Specify acceptable performance characteristics.

'stress_function'
Specifies function to call when performance becomes unacceptable.

'relief_function'
Specifies function to call when performance becomes acceptable.

sched_set_critical_performance sets the required performance characteristics. If **ratio** loops through the **period** ring fail to sustain a period of **under_threshold** for a duration of **for_how_long**, the **stress_function** will be called. If the system recovers, the **relief_function** will be called.

LibSelect

Table of Contents

1 Overview 1

2 Usage 3

2.1 Building Libselect 3
2.2 Linking with Libselect 3
2.3 Examples 4

3 Functions 5

3.1 select_init 5
3.2 select_create 5
3.3 select_add_editor 5
3.4 select_editor_done 6
3.5 select_start_editor 6
3.6 select_add_selectable 7
3.7 select_allow_selection 7

1 Overview

LibSelect provides a generic facility for making objects on the map selectable for editing. It also provides a menu for system level editors which don't belong as mode buttons on the interface.

Objects on the map are classified using libSensitive SNSTVE_CLASS structures. When the selection tool is started (or restarted), it simply runs through all the classes which have been registered, and makes them sensitive to mouse input. At this time, it also installs a gesture handler for each, which will be called if the user clicks on the object.

Selection is a SAF GUI mode, and as such, it is suspended if the GUI enters another mode (see section 'sgui_add_mode' in LibSAFGUI Programmer's Manual). When this happens, libSelect removes the gesture callbacks and sensitivity of all the classes which it is managing. The callbacks and sensitivity are restored when the selection tool is resumed.

LibSelect also monitors the system privilege level (see section 'Overview' in LibPrivilege Programmer's Manual), and allows selectable classes to act like privileged buttons.

Finally, libSelect provides support for seemingly *modeless* editors which run in the context of the selection mode. This is useful for editors which the user may desire to leave up whenever no other editor is active.

2 Usage

The software library '**libselect.a**' should be built and installed in the directory '**/common/lib/**'. You will also need the header file '**libselect.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 libselect source directory. If these files are already built, you can skip the section on building libselect.

2.1 Building Libselect

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

If the directory '**common/libsrc/libselect**' 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/libselect
# co RCS/*,v
# make install
```

This should compile the library '**libselect.a**' and install it and the header file '**libselect.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. libselect should compile without errors on the following platforms:

- Mips
- SGI Indigo
- Sun Sparc

2.2 Linking with Libselect

Libselect can be linked into an application program with the following link time flags: '**ld [source .o files] -L/common/lib -lselect [other libraries]**'. If your compiler does not sup-

port '-L' syntax, you can use the archive explicitly: '`ld [source .o files] /common/lib/libselect.a`'.

Libselect depends directly on the following libraries: libsafgui, libtactmap, libsensitive, libeditor, and libprivilege.

2.3 Examples

The following is from the libunits initialization routine. It demonstrates how to define a selectable class:

```
/* Make the sensitive class */
SNSTVE_INIT_CLASS(edtr->snstve_class);

/* Set the dragging threshold for selection */
edtr->snstve_class.drag_thresh = 50;

/* Add the selectable class.  Require BATTLEMASTER privilege to
 * select, allow dragging at select time, and install the function
 * units_selected as the gesture handler.
 */
select_add_selectable(select, &edtr->snstve_class, PRIV_BATTLEMASTER,
                      TRUE, units_selected, edtr);
```

3 Functions

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

3.1 select_init

```
void select_init()
```

select_init initializes libselect. Call this before any other libselect function.

3.2 select_create

```
SELECT_TOOL_PTR select_create(gui, tactmap, map_erase_gc, sensitive)
    SGUI_PTR           gui;
    TACTMAP_PTR        tactmap;
    GC                 map_erase_gc;
    SNSTVE_WINDOW_PTR  sensitive;
```

'gui' Specifies the SAF GUI

'tactmap' Specifies the tactical map

'map_erase_gc'
Specifies the GC which erases things from the map

'sensitive'
Specifies the sensitive window

select_create creates a selection tool. This should be called before creating other editors and tools for a GUI, so that the selection icon appears at the top of the list.

3.3 select_add_editor

```
void select_add_editor(select, editor, load_editor_fcn, load_editor_arg)
    SELECT_TOOL_PTR    select;
    EDT_EDITOR_PTR     editor;
    SELECT_LOAD_EDITOR load_editor_fcn;
```

```
ADDRESS             load_editor_arg;
```

'select' Specifies the selection tool

'editor' Specifies the editor

'load_editor_fcn, load_editor_arg'
Specifies a function to call to load the editor with current values load_editor_fcn and load_editor_arg can both be NULL.

select_add_editor adds an editor to the list of editors maintained by libselect. The library will call the passed load_editor_fcn (if non-NULL) after the editor is brought up to get initial values. This function can initialize the editor using edt_load (see section 'edt_load' in LibEditor Programmer's Manual).

The loading function should be prototyped as follows:

```
void load_editor(editor, user_arg)
    EDT_EDITOR_PTR editor;
    ADDRESS user_arg;
```

3.4 select_editor_done

```
void select_editor_done(select, editor)
    SELECT_TOOL_PTR select;
    EDT_EDITOR_PTR  editor;
```

'select' Specifies the select tool

'editor' Specifies the editor which has finished

select_editor_done informs libselect that one of its editors (passed to select_add_editor) has finished. Call this from the editor's exit_fcn (see section 'edt_create' in LibEditor Programmer's Manual).

3.5 select_start_editor

```
void select_start_editor(select, editor)
    SELECT_TOOL_PTR select;
    EDT_EDITOR_PTR  editor;
```

'select' Specifies the select tool

'editor' Specifies the editor to start

select_start_editor causes libselect to start the passed editor, as though the user had clicked the editor's button. WARNING: the load function is not invoked, so if the caller needs to load the editor with any values, that should be done immediately after this function returns.

3.6 select_add_selectable

```
void select_add_selectable(select, class, privilege,
                           dragable, handler_function, user_data)
    SELECT_TOOL_PTR  select;
    SNSTVE_CLASS    *class;
    PRIV_LEVEL       privilege;
    int32            dragable;
    CALLBACK_HANDLER handler_function;
    ADDRESS          user_data;
```

'select' Specifies the select tool

'class' Specifies the libsensitive class

'privilege'
Specifies the system privilege needed to select object of this class

'dragable'
Specifies whether the objects should be dragable when selected

'handler_function, user_data'
Specifies the function to install as the sensitive gesture callback (see section 'Class Definition' in LibSensitive Programmer's Manual)

select_add_selectable registers a sensitive class with libselect, along with arguments to **callback_register_handler** which will be installed when the select tool is enabled.

3.7 select_allow_selection

```
void select_allow_selection(select, do_allow)
    SELECT_TOOL_PTR  select;
    int32            do_allow;
```

'select' Specifies the select tool

'do_allow'

Specifies whether to allow selection or not

select_allow_selection enables handlers and sensitivity for all objects. This can be called from another mode to allow selection from within that mode. Note that there may be surprising consequences if objects on the map are sensitive for other reasons. Also, multiple calls to allow will register handlers multiple times, so be sure to disable selection once for each enabling.

LibSema

Table of Contents

1 Overview 1

2 Functions 3

2.1 sema_init 3
2.2 sema_create_rwlock 3
2.3 sema_destroy_rwlock 4
2.4 sema_rwlock_op 4
2.5 sema_dump 4
2.6 sema_dump 5

1 Overview

Libsema provides a simple abstraction for controlling access to shared resources between multiple processes on a single hardware platform. Libsema uses semaphores to implement a locking mechanism which allows multiple simultaneous readers or a single writer to access the resource being locked.

A common use for the libsema read/write locks is to control access to a block of shared memory being used for interprocess communications between processes on the same hardware platform.

Libsema can use System V semaphores (Sun SPARC and SGI) or IRIX REACT semaphores (SGI only) to implement the read/write locks. The choice of semaphore abstractions is provided as a flag passing to the sema_init().

2 Functions

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

2.1 sema_init

```
int32 sema_init(path, nreaders, semasys5)
    char *path;
    int32 nreaders;
    int32 semasys5;
```

'arena_path'
: For the IRIX REACT semaphores, "path" specifies a UNIX file system path to an existing file which is used to initialize the semaphore abstraction.

'nreaders'
: Specifies the maximum number simultaneous read locks which will be allowed on the read/write locks. That is, semaphores are initialize with a value of "nreaders" when they are created.

'semasys5'
: Specifies whether to use System V semaphores (semasys5 = 1) or IRIX REACT semaphores (semasys5 = 0). The default is to use IRIX REACT semaphores.

sema_init initialized the libsema library. When using IRIX REACT semaphores, the "path" argument must specify a UNIX file system path to an existing file. This file is used to initialize the "arena" from which IRIX REACT semaphores are allocated. When using System 5 semaphores, this argument is ignored. The "nreaders" parameter specifies the maximum number of simultaneous read locks which will be allowed on libsema read/write locks before a block will occur.

2.2 sema_create_rwlock

```
SEMA_RWLOCK sema_create_rwlock()
```

sema_create_rwlock creates new read/write lock with a value equal to the **nreaders** specified in the **sema_init** function. This lock is passed back to the caller as a return value.

2.3 sema_destroy_rwlock

```
int32 sema_destroy_rwlock(rwlock)
    SEMA_RWLOCK rwlock;
```

'rwlock' Specifies the read/write lock.

Destroys the specified read/write lock.

2.4 sema_rwlock_op

```
int32 sema_rwlock_op(rwlock, lock_op)
    SEMA_RWLOCK rwlock;
    SEMA_OPTYPE lock_op;
```

'rwlock' Specifies the read/write lock.

'lock_op' Specifies the operation to be performed on the lock. The set of operations includes:

SEMA_READ_LOCK_OPTYPE Lock for reading

SEMA_WRITE_LOCK_OPTYPE Lock for writing

SEMA_READ_UNLOCK_OPTYPE Unlock for reading

SEMA_WRITE_UNLOCK_OPTYPE Unlock for writing

sema_rwlock_op performs the specified lock operation on the specified read/write lock. The SEMA_READ_LOCK_OPTYPE decrements the value of the lock by 1. If the value goes negative, the calling process is blocked until the value of the lock goes positive again. Similarly, the SEMA_WRITE_LOCK_OPTYPE decrements the value of the lock by a number equal to the maximum number of readers allowed (see **sema_init**). If the value of the read/write lock goes negative the calling process is blocked. The SEMA_READ_UNLOCK_OPTYPE increments the value of the lock by 1. This has the potential of unblocking a process waiting on the lock. The SEMA_WRITE_UNLOCK_OPTYPE increments the value of the lock by a number equal to the maximum number of readers allowed. This has the potential of unblocking one or more processes waiting on the lock.

2.5 sema_dump

```
void sema_dump_rwlock(rwlock, string)
```

```
SEMA_RWLOCK rwlock;
char        *string;
```

'rwlock' Specifies the read/write lock.

'string' Specifies the procedure name

Prints the current value of the specified lock to "stdout".

2.6 sema_dump

```
char *sema_error()
```

Returns an error string for the last error in the libsema.

LibSensitive

Table of Contents

1 Overview 1

1.1 Class Definition 1
1.2 Object Definition 3

2 Usage 7

2.1 Building Libsensitive 7
2.2 Linking with Libsensitive 7
2.3 Examples 8

3 Functions 9

3.1 snstve_init 9
3.2 SNSTVE_INIT_CLASS 9
3.3 snstve_miss_callback 9
3.4 snstve_create 10
3.5 snstve_refresh 10
3.6 snstve_add_object 10
3.7 snstve_remove_object 11
3.8 snstve_change_object 11
3.9 snstve_clear_class 12
3.10 snstve_clear 12
3.11 snstve_set_class 13
3.12 snstve_set_class_buttons 13

1 Overview

LibSensitive provides a facility for managing mouse-sensitive objects in an X windows environment. It was designed to fulfill the following requirements:

- Allow sensitive objects to be rectangular, circular, or linear.
- Allow rapid modification of sensitive objects.
- Allow objects to be *hot* (to automatically highlight when the mouse pointer passes over them), without using a significant amount of CPU.
- Allow objects to respond to any combination of mouse buttons.
- Allow thousands of simultaneous sensitive objects without significantly impacting performance.
- Allow application-defined classification of objects to facilitate changes to sensitivity and button responses of many objects at once.
- Allow sensitive object to be *dragable* (mouse movement beyond a certain threshold drags the highlight).
- Report the amount of dragging relative to the initial mouse-press location, in addition to reporting the absolute screen location (this allows an object to be picked and dragged by any point, instead of only by its origin).

An application creates sensitive objects using two structures: **SNSTVE_CLASS** and **SNSTVE_OBJECT**. The class defines the sensitivity attributes for a set of objects; the object defines the shape of a single object in a class.

1.1 Class Definition

A sensitive class is defined by an application simply by allocating a structure and using it. Every object must belong to a class, but an application could put each object in its own class. The **SNSTVE_CLASS** structure is defined as follows:

```
typedef struct snstve_class
{
    unsigned sensitive   : 1;
    unsigned hot         : 1;
    unsigned dragable    : 1;
    unsigned drag_thresh : 6;
    unsigned buttons     : 5;

    CALLBACK_EVENT_PTR excited;
```

```
        CALLBACK_EVENT_PTR gesture;

    } SNSTVE_CLASS;
```

sensitive
: Specifies whether objects in the class should be considered active. It is faster to change the sensitivity attribute of a class than to frequently remove and add it from the table.

hot
: Specifies whether objects in the class should highlight when the mouse passes over them.

dragable
: Specifies whether the user can drag class objects around (represented by dragging the highlight box). Note that dragging an object *does not* change the object; it is the application's responsibility to act on the dragging.

drag_thresh
: When **dragable** is **TRUE**, this specifies the amount of mouse movement necessary before dragging occurs (this is to help avoid accidental dragging).

buttons
: Indicates which mouse buttons should trigger the **gestures** callback; to get X button n, set bit [1 << n]; as in X, button 0 (the low-order bit) will get any button.

excited
: If **hot** is **TRUE**, this specifies a function to call when the cursor passes into or out of an object in this class.

gesture
: Specifies a function to call when the mouse is pressed, dragged, or released over an object in this class. The drag and release phases will not be invoked unless the pressed phase was invoked for an object.

The excited and gesture callbacks should both be defined as follows:

```
void callback(window, action, call_data, callback_arg)
    SNSTVE_WINDOW_PTR window;
    SNSTVE_ACTION     action;
    SNSTVE_CALL_DATA *call_data;
    ADDRESS           callback_arg;
```

The **action** is one of the following:

SNSTVE_EXCITED
: The mouse entered a hot object.

SNSTVE_UNEXCITED
: The mouse left a hot object.

SNSTVE_PRESS
: The user clicked on a sensitive object.

SNSTVE_MOTION
: The user dragged a clicked-in object (regardless of whether the object's **dragable** field is set).

SNSTVE_RELEASE
: The user released a clicked-in object.

The **call_data** is defined as follows:

```
typedef struct snstve_call_data
{
    SNSTVE_CLASS *class;
    int32        instance;
    ADDRESS      user_data;
    int32        button;
    uint16       screen_x;
    uint16       screen_y;
    int16        xoff;
    int16        yoff;
} SNSTVE_CALL_DATA;
```

class
instance
user_data
: Application-defined identification of the affected object. The class will be NULL, and the instance will be -1 if the no object was indicated (such as a click on the screen which did not touch an object).

button Specifies which button was used (1-5), or 0 when action is **SNSTVE_EXCITED** or **SNSTVE_UNEXCITED**.

screen_x
screen_y Specifies the location of the action in screen coordinates.

xoff
yoff Specifies the location of the action relative to where the mouse press occured (this is always 0,0 when action is **SNSTVE_PRESS**).

LibSensitive provides a macro to facilitate use of the class structure: **SNSTVE_INIT_CLASS**. This macro zero's out all the fields of the structure, and creates the two libCallback events (see section 'callback_define_event' in LibCallback Programmer's Manual).

1.2 Object Definition

Object definition occurs through the use of a template structure. The application creates a

SNSTVE_OBJECT structure which represents the object, then passes it to libSensitive for processing. When libSensitive returns, the application may keep the structure or free it immediately: libSensitive keeps no pointers to the structure (except for the class pointer referred to within the structure). Each object is defined as follows:

```
typedef struct snstve_object
{
    SNSTVE_KIND kind;

    SNSTVE_CLASS *class;
    int32        instance;
    ADDRESS      user_data;

    union
    {
        struct
        {
            int16 x;
            int16 y;
            int16 width;
            int16 height;
        } rect;
        struct
        {
            int16 x;
            int16 y;
            int16 radius;
        } circle;
        struct
        {
            int16 x0;
            int16 y0;
            int16 x1;
            int16 y1;
            int16 width;
        } line;
    } attr;

    /* other fields which the application should consider private...
     */
} SNSTVE_OBJECT;
```

class Specifies the application-defined classification of the object (see (undefined) [Class Definition], page (undefined)).

instance Specifies the application-defined instance of the application-defined class (this is passed back to the application in callbacks). The combination of **class** and **instance** must uniquely identify an object.

user_data

Specifies an arbitrary piece of data which the application associates with the object. This is provided in addition to the instance field to facilitate libraries like libTactMap which need to place special meaning in the instance attribute, yet would like applications to be able to define objects.

kind Specifies the type of object. It should be one of the following:

SNSTVE_RECTANGLE

A rectangle starting at **attr.rect.x**, **attr.rect.y**, with width of **attr.rect.width** pixels and height of **attr.rect.height** pixels. All units are in screen coordinates, and negative numbers are acceptable.

SNSTVE_CIRCLE

A circle centered at **attr.circle.x**, **attr.circle.y**, with radius of **attr.circle.rad** pixels. All units are in screen coordinates, and negative numbers are acceptable.

SNSTVE_CIRCLE

A line from **attr.line.x0**, **attr.line.y0** to **attr.line.x0**, **attr.line.y0**, with width **attr.line.width**. All units are in screen coordinates, and negative numbers are acceptable.

2 Usage

The software library '**libsensitive.a**' should be built and installed in the directory '**/common/lib/**'. You will also need the header file '**libsensitive.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 libsensitive source directory. If these files are already built, you can skip the section on building libsensitive.

2.1 Building Libsensitive

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

If the directory '**common/libsrc/libsensitive**' 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/libsensitive
# co RCS/*,v
# make install
```

This should compile the library '**libsensitive.a**' and install it and the header file '**libsensitive.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. libsensitive should compile without errors on the following platforms:

- Mips
- SGI Indigo
- Sun Sparc

2.2 Linking with Libsensitive

Libsensitive can be linked into an application program with the following link time flags: '**ld [source .o files] -L/common/lib -lsensitive -lXt -lX11 -lm**'. If your compiler does not sup-

port '-L' syntax, you can use the archive explicitly: '`ld [source .o files] /common/lib/libsensitive.a`'.

Libsensitive depends on X windows.

2.3 Examples

The test program '`test.c`' in the libsensitive directory shows usage of most libsensitive functions. The program operates as follows:

- The program fills the window with lines, rectangles and circles. All of these objects are hot. Some of these objects (orange lines, green circles, and red rectangles) also have dragable highlights.
- The program clears the screen and generates a new screen full of objects every 30 seconds.
- The program makes non-dragable circles and rectangles insensitive 15 seconds after regeneration.
- When a mouse press or release event occurs on a sensitive object, or when any mouse event occurs on the screen, the parameters of that event are printed out.
- Rectangles respond to the left button, circles respond to the middle and right buttons, and lines respond to any button.
- Orange lines, green circles, and red rectangles can be dragged to a new location.
- Two circles move around the screen. The green one can be dragged to a new location.

3 Functions

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

3.1 snstve_init

```
void snstve_init()
```

snstve_init initializes libsensitive. Call this before any other libsensitive functions.

3.2 SNSTVE_INIT_CLASS

```
void SNSTVE_INIT_CLASS(class)
    SNSTVE_CLASS class;
```

'class' Specifies the class structure (not a pointer) to initialize.

SNSTVE_INIT_CLASS is a macro which initializes a class structure. Once objects h ve been created in a class, the fields should only be modified using the function **snstve_set_class** and **snstve_set_class_buttons**.

See (undefined) [Class Definition], page (undefined).
See (undefined) [snstve set class], page (undefined).
See (undefined) [snstve set class buttons], page (undefined).

3.3 snstve_miss_callback

```
CALLBACK_EVENT_PTR snstve_miss_callback(window)
    SNSTVE_WINDOW_PTR window;
```

'window' Specifies the senstitive window.

snstve_miss_callback returns the libcallback event handle which is fired when the mouse is pressed, dragged or released on the window, outside the bounds of any sensitive object.

3.4 snstve_create

```
SNSTVE_WINDOW_PTR snstve_create(widget, draw_gc, erase_gc)
    Widget widget;
    GC     draw_gc;
    GC     erase_gc;
```

'widget' Specifies the widget where input is received and highlights are drawn

'draw_gc' Specifies the GC for drawing highlights

'erase_gc'
Specifies the GC for erasing highlights

snstve_create creates a sensitive window. This is passed to all other libsensitive routines. The GCs are used to draw and erase highlight (hot) boxes. Note that the **erase** should either clear the color plane used by the **draw**, or it should be tiled with the underlying picture. Note that it *is* safe to create more than one sensitive window acting on a single X window, however it could be confusing to the user (since overlapping objects could be hot in both).

3.5 snstve_refresh

```
void snstve_refresh(window)
    SNSTVE_WINDOW_PTR window;
```

'window' Specifies the sensitive window

snstve_refresh redraws the current highlight, if any (call this after any operation which clears the screen).

3.6 snstve_add_object

```
void snstve_add_object(window, object)
    SNSTVE_WINDOW_PTR window;
    SNSTVE_OBJECT     *object;
```

'window' Specifies the sensitive window

'object' Specifies the object to add

snstve_add_object adds an object to the sensitive window. Note that the object structure is *copied* into the window; the pointer is not saved by libsensitive.

3.7 snstve_remove_object

```
int32 snstve_remove_object(window, x, y, class, instance)
    SNSTVE_WINDOW_PTR window;
    int16             x, y;
    SNSTVE_CLASS      *class;
    int32             instance;
```

'window' Specifies the sensitive window

'x, y' Specifies the location of the object

'class' Specifies the application-defined class of the object

'instance'
Specifies the application-defined instance of the object

snstve_remove_object finds and removes an object from the sensitive window. The passed location should be any point within the bounding box of the object. Note that **snstve_clear** and **snstve_clear_class** are much faster than removing a number of objects one at a time. The return value is 1 for success, 0 if the object could not be found.

See (undefined) [snstve clear], page (undefined).
See (undefined) [snstve clear class], page (undefined).

3.8 snstve_change_object

```
int32 snstve_change_object(window, x, y, object)
    SNSTVE_WINDOW_PTR window;
    int16 x, y;
    SNSTVE_OBJECT *object;
```

'window' Specifies the sensitive window

'x, y' Specifies the old location of the object

'object' Specifies the object variables

snstve_change_object finds and relocates an object in the sensitive window. The passed location should be any point within the current bounding box of the object (where it is *now*, not where it is going). Note that when a hot object is changed with this routine, the highlight may be retained if the object is still under the pointer after the change. This would not be true if the object were removed with **snstve_remove_object** and re-added with **snstve_add_object**. Also, an object which is currently hot will only stay hot if it is The return value is 1 for success, 0 if the object could not be found.

See (undefined) [snstve remove object], page (undefined).
See (undefined) [snstve add object], page (undefined).

3.9 snstve_clear_class

```
void snstve_clear_class(window, class)
    SNSTVE_WINDOW_PTR window;
    SNSTVE_CLASS      *class;
```

'window' Specifies the sensitive window

'class' Specifies the class to clear

snstve_clear_class clears the sensitive window of all objects of a given application-defined class (this is *much* faster than removing objects one at a time).

See (undefined) [snstve remove object], page (undefined).
See (undefined) [snstve clear], page (undefined).

3.10 snstve_clear

```
void snstve_clear(window)
    SNSTVE_WINDOW_PTR window;
```

'window' Specifies the sensitive window

snstve_clear clears the entire sensitive window (this is *much* faster than removing objects one at a time, or a class at a time).

See (undefined) [snstve remove object], page (undefined).
See (undefined) [snstve clear class], page (undefined).

3.11 snstve_set_class

```
void snstve_set_class(window, class, sensitive, hot, dragable)
    SNSTVE_WINDOW_PTR window;
    SNSTVE_CLASS     *class;
    uint32            sensitive;
    uint32            hot;
    uint32            dragable;
```

'window' Specifies the sensitive window

'class' Specifies the application-defined class

'sensitive'
Specifies whether the class should be sensitive

'hot' Specifies whether the class should be hot

'hot' Specifies whether the class should be dragable

snstve_set_class sets the sensitivity attributes for all objects of a given application-defined class.

3.12 snstve_set_class_buttons

```
void snstve_set_class_buttons(window, class, remove, add)
    SNSTVE_WINDOW_PTR window;
    SNSTVE_CLASS     *class;
    uint16            remove;
    uint16            add;
```

'window' Specifies the sensitive window

'class' Specifies the application-defined class

'remove' Specifies the buttons to remove

'add' Specifies the buttons to add

snstve_set_class_buttons sets the buttons values for all objects of a given application-defined class, using the operation:

```
new_buttons = (old_buttons & ~remove) | add
```

LibSensors

Table of Contents

1 Overview 1

2 Examples 5

3 Functions 7

3.1 sensors_init 7
3.2 SENSORS_SET_ABSOLUTE 7
3.3 SENSORS_SET_RELATIVE 8
3.4 SENSORS_GET_SENSED 9
3.5 SENSORS_GET_TARGET_VISIBILITY 10
3.6 SENSORS_GET_LOCATION_VISIBILITY 11
3.7 SENSORS_SET_MODE 12
3.8 SENSORS_GET_MODE 13
3.9 SENSORS_GET_CAPABILITIES 13
3.10 SENSORS_SET_STATE 14
3.11 SENSORS_GET_STATE 15
3.12 SENSORS_GET_TARGET_INFO 15

1 Overview

Sensors 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 sensors functions is achieved through macros defined by libsensors. These macros invoke cmpnt_invoke with a code number which identifies the function to run. Libcomponents then runs this function for the particular sensor model via a jump table.

The table below shows how the sensors component relationships have been currently implemented via the ModSAF library structure.

specific libraries	generic library	architectural library
libradar	libsensors	libcomponents
libvisual	libsensors	libcomponents

As mentioned above, libsensors requires the services of libcomponents, an architectural library which provides a level of abstraction away from the specific component interfaces. When the ModSAF application gets set up to run, the libsensors initialization process directs libcomponents to define a sensors component class. This information enables libcomponents to define a structure to accommodate all the sensor instantiations a simulated object is allowed to have. The libsensors initialization process also tells libcomponents the number of its defined sensor 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 libsensors.

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 libvisual instantiation.

Since an application will interface to libvisual or libradar through libsensors, a tank's sensor commands (which are performed by libvisual) and an airplane's sensor commands (which are performed by libradar) are both issued via the interface defined by libsensors. A command to change viewing controls is therefore the same whether the sensing component is a tank commander's sight or an airplane's radar. 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 any of its sensors though the libsensors interface. The table below shows the relationship between the specific and generic library for the sensors component.

Instantiations of of the library:	Belong to generic component class:	Have a command interface defined in:
libradar	sensors	libsensors
libvisual	sensors	libsensors

The interface to libsensors is defined in its public header file (libsensors.h). This interface lets an application set sensor controls or get sensor information without knowing which specific sensor model is being used. Applications interface to the radar model or visual model primarily through the macros defined in libsensors. These macros map to functions which are invoked on instances of the sensors sub-class (such as the libradar component instantiated for an airplane or a libvisual component instantiated for a tank).

One interface for controlling a sensor is the SENSORS_SET_ABSOLUTE macro which maps to a function that sets the direction of attention, magnification, and scan parameters for the sensor. A possible definition for this macro is shown below.

```
#define SENSORS_SET_ABSOLUTE(_v, _c, _a, _e, _m, _w, _h, _vn, _vx, _rx)
{
    SENSORS_INTERFACE _sif;
    _sif.u.set_absolute.azimuth = _a;
    _sif.u.set_absolute.elevation = _e;
    _sif.u.set_absolute.magnification = _m;
    _sif.u.set_absolute.half_width = _w;
    _sif.u.set_absolute.half_height = _h;
    _sif.u.set_absolute.vmin = _vn;
    _sif.u.set_absolute.vmax = _vx;
    _sif.u.set_absolute.range_max = _rx;
    cmpnt_invoke(SENSORS_SET_ABSOLUTE_FCN, _v, _c, &_sif);
}
```

The SENSORS_INTERFACE structure defined in libsensors.h is the structure which is passed to any sensors interface function. This structure is a union of structures that each define an argument

list for a sensors interface function. An abbreviated example that assumes there are only a few interface 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 sensors_interface
{
    union
    {
        struct sensors_set_controls
        {
            float64 azimuth;
            float64 elevation;
            float64 magnification;
            float64 half_width;
            float64 half_height;
            float64 vmin, vmax;
            float64 range_max;
        } set_absolute, set_relative;
        struct sensors_set_state
        {
           SENSORS_STATE state;
        } set_state, get_state;
    } u;
} SENSORS_INTERFACE;
```

Issuing a command to an objects's sensor component is done by invoking one of the macros defined in libsensors. These macros identify the specific component function which needs to be called. For example, invoking the SENSORS_SET_ABSOLUTE macro will result in the calling of the set_absolute 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 (libradar and libvisual). In this case, both libraries install a function with the same name, set_absolute (there is no name conflict because each function is declared static). It is the specific function (either libradar's set_absolute or libvisual's set_absolute) 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 (such as libradar or libvisual) data can be accessed. The requested function can require input (such as an azimuth and an elevation) 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 `cmpnt_invoke(SENSORS_SET_ABSOLUTE_FCN, _v, _c, &_sif);` code segment shown above, SENSORS_SET_ABSOLUTE_FCN serves as the function

pointer index, `_v` provides the vehicle id, `_c`provides the component number, and `&_sif` provides the address for the function's argument lists.

2 Examples

To get the component number of my commander's sight:

```
extern int32 cmdr_sight;

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

To center that sight (the macro is defined by libsensors; it assembles a SENSORS_INTERFACE structure, and calls cmpnt_invoke):

```
if (cmdr_sight != CMPNT_NOT_FOUND)
  SENSORS_SET_RELATIVE(vehicle_id, cmdr_sight, 0.5, 0.5, 1.0, 1.0,
    0.0, 1.0, 1.0, 0.0);
```

To get a list of vehicles detected by that sensor:

```
VTAB_LIST list;

if (cmdr_sight != CMPNT_NOT_FOUND)
  SENSORS_GET_SENSED(vehicle_id, cmdr_sight, &list);
```

3 Functions

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

3.1 sensors_init

```
void sensors_init(directory, reader_flags)
    char  *directory;
    uint32 reader_flags;
```

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

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

sensors_init initializes libsensors. Call this function after **cmpnt_init**, and before any specific sensor init functions. To simplify definition of other data files, libsensors reads a file of macro definitions from the passed **directory** using the passed **reader_flags**. These macros define (in a libreader format) some of the constants defined by libsensors.

3.2 SENSORS_SET_ABSOLUTE

```
void SENSORS_SET_ABSOLUTE(vehicle_id, component_number,
                          azimuth_radians, elevation_radians,
                          half_width_radians, half_height_radians,
                          magnification, vmin, vmax, range_max)
    int32   vehicle_id;
    int32   component_number;
    float64 azimuth_radians;
    float64 elevation_radians;
    float64 half_width_radians;
    float64 half_height_radians;
    float64 magnification;
    float64 vmin, vmax;
    float64 range_max;
```

'vehicle_id'
Specifies the vehicle ID

'component_number'
Specifies the sensor component number

'azimuth_radians'
Specifies the desired azimuth in radians (attachment coordinates)

'elevation_radians'
Specifies the desired elevation in radians (attachment coordinates)

'half_width_radians'
For sensors which have controllable scan volumes (such as radar) specifies half the scan volume width in radians (the scan volume will go from -half_width_radians to +half_width_radians).

'height_radians'
For sensors which have controllable scan volumes (such as radar) specifies half the scan volume height in radians (the scan volume will go from -half_height_radians to +half_height_radians).

'magnification'
For sensors which have controllable magnification, specifies the desired magnification

'vmin' For sensors which can detect vehicles by relative motion, specifies the minimum relative velocity of vehicles, in meters per second, to be detected

'vmax' For sensors which can detect vehicles by relative motion, specifies the maximum relative velocity of vehicles, in meters per second, to be detected

'range_max'
Specifies the maximum range-of-interest (as opposed to detection range) for the sensor

SENSORS_SET_ABSOLUTE sets the direction of attention, magnification, and scan parameters for the sensor. Not all parameters may be relevant for a particular sensor. Depending upon the physical or psychological detection model, the direction of the sensor may impact whether a particular target is sensed. The direction is in sensor coordinates (positive elevation is up, positive azimuth is left), and may be clipped if it exceeds the limits of the sensor's orientability.

3.3 SENSORS_SET_RELATIVE

```
void SENSORS_SET_RELATIVE(vehicle_id, component_number,
                          azimuth, elevation, width, height,
  magnification, vmin, vmax, range_max)
    int32   vehicle_id;
    int32   component_number;
    float64 azimuth;
    float64 elevation;
    float64 width, height;
```

```
        float64 magnification;
        float64 vmin, vmax;
        float64 range_max;
```

'vehicle_id'
Specifies the vehicle ID

'component_number'
Specifies the sensor component number

'azimuth' Specifies the desired azimuth

'elevation'
Specifies the desired elevation

'width' For sensors which have controllable scan volumes, specifies the scan volume width

'height' For sensors which have controllable scan volumes, specifies the scan volume height

'magnification'
For sensors which have controllable magnification, specifies the desired magnification

'vmin' For sensors which can detect vehicles by relative motion, specifies the minimum relative velocity of vehicles to be detected

'vmax' For sensors which can detect vehicles by relative motion, specifies the maximum relative velocity of vehicles to be detected

'range_max'
Specifies the maximum range-of-interest (as opposed to detection range)

SENSORS_SET_RELATIVE sets the direction of attention, magnification and scan volume for the sensor. Not all parameters may be relevant for a particular sensor. Depending upon the physical or psychological detection model, the direction of the sensor may impact whether a particular target is sensed. All values are in the range 0 to 1 (azimuth:counterclockwise-to-clockwise, elevation:down-to-up, magnification:minimum-to-maximum, width:minimum-to-maximum, height:minimum-to-maximum, vmin:minimum-to-maximum, vmax:minimum-to-maximum, range_max:minimum-to-maximum).

3.4 SENSORS_GET_SENSED

```
    void SENSORS_GET_SENSED(vehicle_id, component_number, list)
        int32       vehicle_id;
        int32       component_number;
        VTAB_LIST *list;
```

'vehicle_id'
Specifies the vehicle ID

'component_number'
Specifies the sensor component number

'list' Returns the list of sensed vehicles

SENSORS_GET_SENSED returns a list of sensed vehicles. For visual sensors, the user data attached to each vehicle in the list is the address of a SENSED_VARS structure. It is mode fully described in libvisual.

3.5 SENSORS_GET_TARGET_VISIBILITY

```
void SENSORS_GET_TARGET_VISIBILITY(vehicle_id, component_number,
                                   target_id, ctdb, result,
                                   intersecting_location,
                                   intersecting_vehicle)
    int32    vehicle_id;
    int32    component_number;
    int32    target_id;
    CTDB    *ctdb;
    float64 *result;
    float64  intersecting_location[3];
    int32   *intersecting_vehicle;
```

'vehicle_id'
Specifies the vehicle ID

'component_number'
Specifies the sensor component number

'target_id'
Specifies the target to check for visibility against

'ctdb' Specifies the terrain that might obstruct the target

'result' Returns the result of the visibility check, as a fraction between 0.0 and 1.0, inclusive

'intersecting_location'
Returns the location of the obstruction if the target is fully obstructed by a vehicle or an object on the terrain

'intersecting_vehicle'
Returns the id of a vehicle obstructing the target if the target is fully obstructed by a vehicle

SENSORS_GET_TARGET_VISIBILITY returns what fraction of a target is visible from a sensor. If a target is fully obscured (as indicated by a result of 0.0, intersection_location will contain the

location of the obstruction. If another vehicle is the cause of the obstruction, intersecting_vehicle will contain the id of the obstructing vehicle.

3.6 SENSORS_GET_LOCATION_VISIBILITY

```
void SENSORS_GET_LOCATION_VISIBILITY(vehicle_id, component_number,
                                     location, location_height,
                                     location_width, ctdb, result,
                                     intersecting_location,
                                     intersecting_vehicle)
    int32    vehicle_id;
    int32    component_number;
    float64  location[3],
    float64  location_height,
    float64  location_width,
    CTDB     *ctdb;
    float64 *result;
    float64  intersecting_location[3];
    int32   *intersecting_vehicle;
```

'vehicle_id'
: Specifies the vehicle ID

'component_number'
: Specifies the sensor component number

'location'
: Specifies the location to check for visibility against

'location_height'
: Specifies a height (in meters) above the location to be used in any intervisibility calculations

'location_width'
: Specifies a width (in meters) to be used in any intervisibility calculations

'ctdb'
: Specifies the terrain that might obstruct the target

'result'
: Returns the result of the visibility check, as a fraction between 0.0 and 1.0, inclusive

'intersecting_location'
: Returns the location of the obstruction if the target is fully obstructed by a vehicle or an object on the terrain

'intersecting_vehicle'
: Returns the id of a vehicle obstructing the target if the target is fully obstructed by a vehicle

SENSORS_GET_LOCATION_VISIBILITY returns what fraction of a location is visible from a sensor. If a location is fully obscured (as indicated by a **result** of 0.0, **intersecting_location** will contain the location of the obstruction. If another vehicle is the cause of the obstruction, **intersecting_vehicle** will contain the id of the obstructing vehicle. In order to properly model a location occupying space (as opposed to a point location), **location_height** and **location_width** are used.

3.7 SENSORS_SET_MODE

```
void SENSORS_SET_MODE(vehicle_id, component_number, mode, target_id)
    int32        vehicle_id;
    int32        component_number;
    SENSORS_MODE mode;
    int32        target_id;
```

'vehicle_id'
Specifies the vehicle ID

'component_number'
Specifies the sensor component number

'mode' Specifies the operating mode (defined by given sensor)

'target_id'
Designates a target, as required for certain sensor modes.

SENSORS_SET_MODE sets the operating mode of the sensor. This primarily affects the type and number of targets detected by the sensor. If given a mode it does not support, **SENSORS_SET_MODE** will choose a suitable replacement. Defined modes are as follows:

SENSORS_PULSE_SEARCH
Pulse search

SENSORS_PD_SEARCH
Pulse doppler search

SENSORS_TWS_MANUAL
Track-While-Scan, manual center

SENSORS_TWS_AUTO
Track-While-Scan, automatic center

SENSORS_PULSE_STT
Single-Target-Track, pulse

SENSORS_PD_STT
Single-Target-Track, pulse doppler

3.8 SENSORS_GET_MODE

```
void SENSORS_GET_MODE(vehicle_id, component_number, mode, target_id)
    int32          vehicle_id;
    int32          component_number;
    SENSORS_MODE *mode;
    int32         *target_id;
```

'vehicle_id'
Specifies the vehicle ID

'component_number'
Specifies the sensor component number

'mode' Specifies the operating mode (defined by given sensor)

'target_id'
Designates what target, if any, has been designated for this mode.

SENSORS_GET_MODE returns the current operating mode of the sensor, which will either be the default mode or whatever was last set in the last SENSORS_SET_MODE. Note that the target_id indicates what target was designated, and *not* whether that target is currently being detected by the sensor.

3.9 SENSORS_GET_CAPABILITIES

```
void SENSORS_GET_CAPABILITIES(vehicle_id, component_number,
      az_list, el_list, az_lct, el_lct,
      az_lo, az_hi, el_lo, el_hi)
    int32      vehicle_id;
    int32      component_number;
    float64    **az_list, **el_list;
    int32      *az_lct, *el_lct;
    float64    *az_lo, *az_hi, *el_lo, *el_hi;
```

'vehicle_id'
Specifies the vehicle ID

'component_number'
Specifies the sensor component number

'az_list' Returns a pointer to an array of legal azimuth half-width settings for this sensor. These half-widths are in radians and specify a valid scan volume which is plus-or-minus this amount around a center beam.

'el_list' Returns a pointer to an array of legal elevation half-height settings for this sensor. These half-heights are in radians and specify a valid scan volume which is plus-or-minus this amount around a center beam.

'az_lct, el_lct'
Returns the number of valid elements in az_list and el_list, respectively.

'az_lo, az_hi'
Returns the counter-clockwise and clockwise limits that the azimuth of the center beam of the scan volume can be steered to, in radians.

'el_lo, el_hi'
Returns the lower and upper limits that the elevation of the center beam of a scan volume can be steered to, in radians.

SENSORS_GET_CAPABILITIES returns (by reference) information about legal values for the sensor's scan volume in terms of half-widths and half-heights. Also returned is information about the orientability of the scan volume.

3.10 SENSORS_SET_STATE

```
void SENSORS_SET_STATE(vehicle_id, component_number,
     state)
    int32         vehicle_id;
    int32         component_number;
    SENSORS_STATE state;
```

'vehicle_id'
Specifies the vehicle ID

'component_number'
Specifies the sensor component number

'state' Specifies the new state of the sensor

SENSORS_SET_STATE sets the state of the sensor. This state is independent of the state of the vehicle on which the sensor is mounted. Defined states are:

SENSORS_ACTIVE
The sensor is enabled

SENSORS_INACTIVE
The sensor is disabled

3.11 SENSORS_GET_STATE

```
void SENSORS_GET_STATE(vehicle_id, component_number,
     state)
    int32          vehicle_id;
    int32          component_number;
    SENSORS_STATE *state;
```

'vehicle_id'
Specifies the vehicle ID

'component_number'
Specifies the sensor component number

'state' Returns the current state of the sensor

SENSORS_GET_STATE returns (by reference) the state of the sensor. This state is independent of the state of the vehicle on which the sensor is mounted. Defined states are:

SENSORS_ACTIVE
The sensor is enabled

SENSORS_INACTIVE
The sensor is disabled

3.12 SENSORS_GET_TARGET_INFO

```
void SENSORS_GET_TARGET_INFO(vehicle_id, component_number,
     target_id, angular_size, angle_to_edge)
    int32          vehicle_id;
    int32          component_number;
    int32          target_id;
    float64       *angular_size;
    float64       *angle_to_edge;
```

'vehicle_id'
: Specifies the vehicle ID

'component_number'
: Specifies the sensor component number

'target_id'
: Specifies the target to check

'angular_size'
: Returns the angular size of the target (in square radians)

'angle_to_edge'
: Returns the angle (in radians) between the target and the nearest edge of the viewport (or gimbal limit).

SENSORS_GET_TARGET_INFO returns (by reference) information about a specified target from the viewpoint of the sensor.

LibShmqueue

Table of Contents

1 Overview 1

2 Functions 3

2.1 shmq_init 3
2.2 shmq_compute_size 3
2.3 create_ring_buffer 3
2.4 shmq_destroy_ring_buffer 4
2.5 shmq_read_entries 4
2.6 shmq_write_entry 5
2.7 shmq_dump 5
2.8 shmq_error 6

1 Overview

Libshmqueue implements a simple mechanism for doing interprocess communication between multiple processes on the same hardware platform. Libshmqueue implements a ring-buffer of data entries (i.e. data packets) which can be written to and read from with a minimum blocking time. Multiple processes can read from and write to the table.

Libshmqueue uses (2 + <Nreaders>) semaphores, where Nreader is the number of readers.

To prevent a reader from catching up writers in a ring buffer, the address of the next write entry is kept in the ring buffer header and a writer semaphore is used to protect the address. A writer takes the writer semaphore to read the next write entry address, then increases it and releases the semaphore after writing. A reader takes the writer semaphore for reading to calculate the the last unread entry.

To prevent writers from writing new entries while readers are reading at the same locations in a ring buffer, an active reader table (an array of <Nreaders> elements) is used together with a semaphore for the table. In addition, <Nreaders> semaphores, the reader semaphore array, is used. A reader follows the following sequence during reading: takes the table semaphore for writing, searches the active reader table for an empty element, saves the index of the element, writes the address of the first read entry to the table, releases the table semaphore, takes the reader semaphore of the index for reading, reads the entries, release the reader semaphore, takes the table semaphore to erase the content of the element, releases the table semaphore. A writer follows the following sequence during writing: takes the table semaphore for reading, if the next write entry is not listed in the table writes the entry and releases the table semaphore, otherwise, releases the table semaphore, takes the reader semaphore of the conflict reader for writing, writes the entry, releases the reader semaphore.

2 Functions

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

2.1 shmq_init

```
void shmq_init()
```

shmq_init initializes the libshmq library. Currently this procedure is a no-op.

2.2 shmq_compute_size

```
int32 shmq_compute_size(nentries, entry_size)
    int32           nentries;
    int32           entry_size;
```

'nentries'
: The number of entries in the shared memory table.

'entry_size'
: The maximum number of bytes for each entry.

shmq_compute_size computes the size of the shared memory segment for the ring buffer.

2.3 create_ring_buffer

```
SHMQ shmq_create_ring_buffer(buffer, nentries, entry_size, nreaders)
   ADDRESS buffer;
   int32 nentries;
   int32 entry_size;
   int32 nreaders;
```

'buffer' The address of the shared memory segment

'nentries'
: The number of entries in the shared memory table.

'entry_size'
The maximum number of bytes for each entry.

'nreaders'
The number of readers.

shmq_create_ring_buffer creates a shared ring_buffer. It is assumed that the a System5 shared memory segment has been created by the caller. The shared memory segment has enough space for a ring_buffer of nentries where each entry is entry_size bytes large. The shared ring buffer includes a libsema semaphore to control access among multiple processes.

2.4 shmq_destroy_ring_buffer

```
int32 shmq_destroy_ring_buffer(shmq)
    SHMQ shmq;
```

'shmq' The address of a shared ring buffer.

shmq_destroy_ring_buffer deletes the liblock semaphores. The caller has to free the shared memory segment.

2.5 shmq_read_entries

```
int32 shmq_read_entries(shmq, entry_offset, num_entry, data_sink_fn, user_data)
    SHMQ  shmq;
    int32 *entry_offset;
    int32 num_entry;
    SHMQ_DATA_SINK_FN data_sink_fn;
    ADDRESS  user_data;
```

'shmq' The table to be read.

'entry_offset'
The offset of the first entry to be read (pass SHMQ_FIRST_ENTRY to start with). Returns the offset of the next entry to read

'num_entry'
The number of entries to be read from the ring_buffer. 1,2,...N: N > "number of total unread entries": all unread entries will be read. N <= 0: all unread entries will be read.

'data_sink_fn'
The function to copy the data from the shared ring_buffer to user data buffer. Should be a simple function, such as bcopy().

'user_data'
The address of the user data buffer.

shmq_read_entries supports reading through a shared memory ring_buffer <n> entries

2.6 shmq_write_entry

```
int32 shmq_write_entry(shmq, entry, entry_size)
    SHMQ    shmq;
    ADDRESS entry;
    int32   entry_size;
```

'shmq' The address of a shared ring buffer.

'entry' The address of the entry to be written.

'entry_size'
The size in bytes of entry.

shmq_write_entry writes a new entry into the ring buffer.

2.7 shmq_dump

```
void shmq_dump(shmq, string, dump_locks)
    SHMQ shmq;
    char *string;
    int32 dump_locks;
```

'shmq' The address of a shared ring buffer.

'string' Specifies the procedure name

'dump_locks'
Specifies whether locks should be dumped

Prints out a cryptic one-line summary about the current state of the buffer.

2.8 shmq_error

```
char * shmq_error()
```

Returns an error string for the last error in the libshmqueue.

LibShmif

Table of Contents

1 Overview 1

2 Functions 3

2.1 shmif_init 3
2.2 shmif_open_shmif 3
2.3 shmif_read_entries 4
2.4 shmif_write_entry 5
2.5 shmif_update 5
2.6 shmif_delete_shmif 6
2.7 shmif_dump 6
2.8 shmif_protocol_kind_to_datatype 6
2.9 shmif_program_error 7
2.10 shmif_program_message 7
2.11 shmif_error_string 7

1 Overview

Libshmif implements the shared memory interface for the Atlas SAF. Libshmif is designed specifically to support the IPC needs of the AGPG project. In particular, it is designed to support communications between the SAFSim. the ModSAF Control, and the DIS Interface process. It should NOT be considered a general purpose library.

Libshmif uses the services provided by libsema to provide read/write locks. It uses the services of libshmtbl and libshmqueue to provide data structures which are safe for shared memory.

The shared memory interface consists of a "control structure" which occupies a System 5 shared memory segment. The control structure maintains information about each of the "channels" which make up the shared memory interface. Each channel consists of one or more "partitions" which are used to communicate a particular protocol (i.e. the Persistant Object Protocol) or subset of a protocol (i.e. Entity State PDUs of the DIS Protocol). Channel partitions are based on libshmtbl and libshmqueue.

Each process participating in the shared memory interface is allocated its own channel. The number of partitions in the channel depends upon the number of protocols in which that process is interested.

2 Functions

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

2.1 shmif_init

```
int32 shmif_init(appl_type, nreaders)
    SHMIF_APPLTYPE  appl_type;
    int32           nreaders;
```

'appl_type'
: The application type of the process. Valid values include: SHMIF_DISIF_APPLTYPE, SHMIF_SAFSIM_APPLTYPE, SHMIF_MODSAF_CTL_APPLTYPE.

'nreaders'
: The maximum number of processes which will be reading from the shmif. Currently this number should be set to 10 (i.e. 4 SHMIF_SAFSIM_APPLTYPE, 4 SHMIF_MODSAF_CTL_APPL 1 SHMIF_DISIF_APPLTYPE, 1 one to grow on).

`shmif_init` initializes the libshmif library.

2.2 shmif_open_shmif

```
SHMIF_SHMIF shmif_open_shmif(ftok_path, ftok_char, nprotocols,
                             protocols, exercise, create)
    char            *ftok_path;
    char            ftok_char;
    int32           nprotocols;
    uint8           *protocols;
    int32           exercise;
    int32           create;
```

'ftok_path'
: Specifies the name which will be used to find the shared memory segment containing the control structure for this shared memory interface.

'ftok_char'
: Specifies the name which will be used to find the shared memory segment containing the control structure for this shared memory interface.

'nprotocols'
: Specifies the number of protocols which will be included in the shared memory interface channel for this process.

'protocols'
: Specifies the protocols which will be included in the shared memory interface channel for this process.

'exercise'
: The id of the exercise in which this process is participating. Used by the shmif_read_entries function as a way of filtering out data from unwanted processes. A value of SHMIF_EXERCISE_ID. can be used to deactivate this filtering.

  SAFSim processes are only involve in one exercise at a time. They will be reading and writing data belonging to that exercise only.

  The DIS Interface, on the other hand, is interested in the data belonging to all the active exercises. When reading, the DIS Interface reads from all SAFSim channels regardless of exercise id. Similarly, the DIS Interface writes all data to the same channel regardless of exercise id.

'create'
: Indicates that this process will act as a server creating the shared communications control structure used by the other processes. In addition, the server is responsible for freeing up communications channels left occupied by processes which exit abnormally.

shmif_open_shmif installs or creates a shared memory interface having the specified name. Installs/creates and locks the shared communications control structure. Allocates a communications channel including one shared table for each of the data types specified in the "data_types" argument. Registers the channel in the communications control structure.

2.3 shmif_read_entries

```
int32 shmif_read_entries(shmif, protocol, exercise, data_sink_fn, data_sink_userdata)
    SHMIF_SHMIF          shmif;
    uint8                protocol;
    int32                exercise;
    SHMIF_DATA_SINK_FN   data_sink_fn;
    ADDRESS              data_sink_userdata;
```

'shmif'
: Specifies the shared memory interface to be used.

'protocol'
: Specifies the protocol to be read.

'exercise'
: Specifies the id of the exercise to which the data must belong. SHMIF_EXERCISE_ID_ALL

can be used if data from all active exercises is desired. In general, the SAFSim's will be interested in a particular exercise, and the DIS Interface will be interested in all active exercises.

'data_sink_fn'
A function which takes a pointer to a void as an argument and which returns void. For example: void sample_data_sink_fn(dataptr, length, user_data)

'data_sink_userdata'
The user data to be passed back to the data_sink_fn in the user_data parameter.

shmif_read_entries checks all the active channels belonging to an process whose application type matches one of those specified in **appl_types** and whose exercise id matches the one specified in **exercise**. For each of these channels, **shmif_read_entries** returns any data of type **data_type** which has NOT already been returned in a previous call to **shmif_read_entriess**. It does so by making successive calls to the **data_sink_fn** function. Returns the total number of entries (i.e. calls to **data_sink_fn**).

2.4 shmif_write_entry

```
int32 shmif_write_entry(shmif, protocol, kind, data, length)
    SHMIF_SHMIF         shmif;
    uint8               protocol;
    uint8               kind;
    ADDRESS             data;
    int32               length;
```

'shmif' Specifies the shared memory interface to be used.

'protocol'
Specifies the protocol family of the data parameter.

'kind' Specifies the PDU kind of the data parameter.

'data' The data (PDU) to be written.

'length' The length in bytes of the data to be written.

shmif_write_entry Writes the data supplied to this process' channel.

2.5 shmif_update

```
int32 shmif_update(shmif, data_type)
    SHMIF_SHMIF         shmif;
    SHMIF_DATATYPE      data_type;
```

'shmif' Specifies the shared memory interface to be used.

'data_type'
Channel partitions containing this type of data will be updated.

shmif_update Updates the channel partitions containing the specified data.

2.6 shmif_delete_shmif

```
int32 shmif_delete_shmif(shmif)
    SHMIF_SHMIF         shmif;
```

'shmif' Specifies the shared memory interface to be used.

shmif_delete_shmif deletes the specified shared memory interface.

2.7 shmif_dump

```
int32 shmif_dump(shmif, string, dump_locks)
    SHMIF_SHMIF shmif;
    char        *string;
    int32       dump_locks;
```

'shmif' Specifies the shared memory interface to be dumped.

'string' Prefix string.

shmif_dump prints debugging information about the shared memory interface.

2.8 shmif_protocol_kind_to_datatype

```
SHMIF_DATATYPE shmif_protocol_kind_to_datatype(protocol, kind)
    uint8               protocol;
    uint8               kind;
```

'protocol'
Specifies the protocol family.

'entry' Specifies the PDU kind.

shmif_protocol_kind_to_datatype returns the shmif datatype which corresponds to the specified protocol and PDU kind.

2.9 shmif_program_error

```
void shmif_program_error()
```

shmif_program_error prints out an error message corresponding to the last error.

2.10 shmif_program_message

```
void shmif_program_message()
```

shmif_program_message prints out an error message corresponding to the last error.

2.11 shmif_error_string

```
char * shmif_error_string()
```

shmif_error_string eturns an error string for the last error in the' libshmif.

LibShmtbl

1 Overview

Libshmtbl implements a simple mechanism for doing interprocess communication between multiple processes on the same hardware platform. Libshmtbl implements a double-buffered array of data entries (i.e. data packets) which supports simultaneous access to the table for reading and writing. Multiple processes can read from the table. However, the abstraction assumes that ONLY ONE process will be writing to the table, so writing is not controlled. Readers are denied access only when writer updates the table by swapping the buffers.

2 Functions

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

2.1 shmtbl_init

```
void shmtbl_init()
```

shmtbl_init initializes the libshmtbl library. Currently this procedure is a no-op.

2.2 shmtbl_alloc_table

```
SHMTBL_TABLE shmtbl_alloc_table(key, nentries, entry_size, flag, shmid)
    key_t           key;
    int32           nentries;
    int32           entry_size;
    int32           flag;
    SHMTBL_ID      *shmid;
```

'key' The key to be passed to shmget to get the SHMTBL_ID.

'nentries'
The number of entries in the shared memory table.

'entry_size'
The maximum number of bytes for each entry.

'flag' Access permissions for the shared memory segment which the table occupies.

'shmid' An out parameter returning the id of the shared memory table. This value can be used as a parameter to **shmtbl_install_table**.

shmtbl_alloc_table creates a shared table using a shared memory segment and installs it in the address space of the local process. The shared memory table has enough space for 2 buffers of nentries where each entry is entry_size bytes large and a libsema read/write lock to control access among multiple processes. **shmtbl_alloc_table** returns two values. The first is the address of the table in the local address space of the calling process which is returned as the return value of the function. The second is a handle for the shared memory table which can be passed to remote processes so that they can install the shared memory table. This value is returned in **shmid** as an out parameter.

2.3 shmtbl_install_table

```
SHMTBL_TABLE shmtbl_install_table(shmid)
    SHMTBL_ID shmid;
```

'shmid' The id of the shared memory table.

shmtbl_install_table takes the id of a shared memory table and installs it in the local address space of the calling process.

2.4 shmtbl_uninstall_table

```
int32 shmtbl_uninstall_table(shmtbl)
    SHMTBL_TABLE shmtbl;
```

'shmtbl' The address of a shared memory table.

shmtbl_uninstall_table takes the address of a shared memory table and removes it from the local address space of the calling process.

2.5 shmtbl_free_table

```
int32 shmtbl_free_table(shmtbl)
    SHMTBL_TABLE shmtbl;
```

'shmtbl' The address of a shared memory table.

shmtbl_free_table removes the specified shared table from the local address space, deletes deletes it.

2.6 shmtbl_get_version

```
int32 shmtbl_get_version(shmtbl)
    SHMTBL_TABLE shmtbl;
```

'shmtbl' The address of a shared memory table.

shmtbl_get_version returns the version of the shared memory table. The version is incremented each time the table is updated by **shmtbl_update**.

2.7 shmtbl_read_entry

```
int32 shmtbl_read_entry(shmtbl, entry)
    SHMTBL_TABLE shmtbl;
    ADDRESS *entry;
```

'shmtbl' The address of a shared memory table.

'entry' This in an IN/OUT parameter (i.e. a parameter used by the calling procedure to passing values into **shmtbl_read_entry** and a parameter used by **shmtbl_read_entry** pass values back out to the calling procedure. There are two valid values which can be passed into **shmtbl_read_entry**; SHMTBL_START_ITERATION and the address of the entry returned in the last call to **shmtbl_read_entry**. There are two valid values which can be returned by **shmtbl_read_entry**; the address of the next entry in the table and SHMTBL_ABORT_ITERATION.

shmtbl_read_entry supports reading through a shared memory table an entry at a time. It operates similarly to the strtok function.

The first call to **shmtbl_read_entry** with ***entry** equal to SHMTBL_START_ITERATION will lock the table for READING, set ***entry** equal to the address of the first element in the table, and return the length of that element in bytes.

Subsequent calls to **shmtbl_read_entry** with ***entry** equal to the last value returned iterate through the table an element at a time. The final call to **shmtbl_read_entry** (where ***entry** contains the address of the last element in the table) unlocks the table for READING, sets ***entry** equal to SHMTBL_ABORT_ITERATION, and returns 0.

The following is a code example of how **shmtbl_read_entry** should be used:

```
{
    ADDRESS entry;
    int32   length;
```

```
        entry = (void *) SHMTBL_START_ITERATION;
        length = shmtbl_read_entry((SHMTBL_TABLE) shmaddr, &entry);
        while (entry != SHMTBL_ABORT_ITERATION)
        {
            process_data(entry);
            length = shmtbl_read_entry((SHMTBL_TABLE) shmaddr, &entry);
        }
    }
```

2.8 shmtbl_write_entry

```
int32 shmtbl_write_entry(shmtbl, entry, entry_size)
    SHMTBL_TABLE shmtbl;
    ADDRESS      entry;
    int32        entry_size;
```

'shmtbl' The address of a shared memory table.

'entry' The address of the entry to be written.

'entry_size'
The size in bytes of entry.

shmtbl_write_entry writes a new entry into the table.

2.9 shmtbl_update

```
int32 shmtbl_update(shmtbl)
    SHMTBL_TABLE shmtbl;
```

'shmtbl' The address of a shared memory table.

shmtbl_update locks the specified shared memory table for writing, swaps the read and write buffers, increments the version counter, and unlocks the table.

2.10 shmtbl_dump

```
void shmtbl_dump(shmtbl, string, dump_locks)
```

```
SHMTBL_TABLE shmtbl;
char         *string;
int32         dump_locks;
```

'shmtbl' The address of a shared memory table.

'string' Specifies the procedure name

'dump_locks'
Specifies whether locks should be dumped

Prints out a cryptic one-line summary about the current state of the table.

2.11 shmtbl_error

```
char * shmtbl_error()
```

Returns an error string for the last error in the libshmtbl.

Libsmokedit

Table of Contents

1 Overview 1
1.1 Examples 1

2 Functions 3
2.1 sme_init 3
2.2 sme_create_editor 3

1 Overview

Libsmokedit is a library which implements a smoke mission editor. The editor will be used to create, delete, change, and start smoke missions. It contains four editables. The first editable, SMOKE_MISSION, displays the name of the "current" mission, and has a button to push to create and delete smoke missions, and a menu of created missions. There is a Scale editable (in integer format) which specifies the number of grenades to launch. The next editable is a CHOOSE_ONE list of smoke types, of which we only have one for the time being (red phosphorus L8A1). The next editable is a Place editable where the user clicks on the map to designate the location at which to drop the smoke grenades. The last editable is a TOGGLE editable that says "Launched/Waiting" which the user will select to start the smoke mission. A dialog box will come up to ensure the user knows this will start the mission. The user will not be able to cancel a mission once it has started and he will not be able to re-use a mission. However, it will be easy to create duplicate missions because the next time he creates a mission it will default to the parameters of the previous mission.

One future enhancement will be to include a time editable to plan missions in advance. This should be done when HHour is implemented.

The smoke missions will be stored in a UnitClass PO, with type diffuseMatterMediumSmoke-Cloud. The number and type of smoke grenades will be stored in the munitions array. The name of the mission will be stored in the formationTemplate and the location in the location field. When the user starts the mission, the shouldBeSimulated field will be set to notify a back end to start simulating the smoke. Although not completely clean, this design was the easiest way to introduce smoke into the PO protocol in a short period of time. Another enhancement would be to add a new type of PO PDU for smoke missions.

1.1 Examples

Look in `test.c` for a sample program using libsmokedit. The file `smokedit.rdr` describes an editor containing a smoke mission editor.

2 Functions

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

2.1 sme_init

```
void sme_init(data_path, flags)
char  *data_path;
uint32 flags;
```

'data_path'
: Specifies the directory path in which to look for the reader file.

'flags'
: Specifies the flags to use in the reader_read call to open the reader files.

sme_init initializes libsmokedit. Call this before any other libsmokedit function.

2.2 sme_create_editor

```
int32 sme_create_editor(gui,sensitive, select, tactmap, tcc,
                        map_erase_gc, refresh_event, db, sim_addr)
    SGUI_PTR            gui;
    SNSTVE_WINDOW_PTR   sensitive;
    SELECT_TOOL_PTR     select;
    TACTMAP_PTR         tactmap;
    COORD_TCC_PTR       tcc;
    GC                  map_erase_gc;
    CALLBACK_EVENT_PTR  refresh_event;
    PO_DATABASE        *db;
    SimulationAddress  *sim_addr;
```

'gui'
: Specifies the SAF GUI

'sensitive'
: Specifies the sensitivity window

'select'
: Specifies the selection tool

'tactmap'
: Specifies the tactical map

'tcc' Specifies the coordinate system

'map_erase_gc'
Specifies the GC used to erase on the tactical map (same as that passed to tactmap_create)

'refresh_event'
Specifies the event which fires upon map refresh

'db' Specifies the PO database.

'sim_addr'
Specifies the simulation address of the local workstation.

smo_create_editor creates a smoke mission editor and makes it available via the select editor.

LibStatmon

Table of Contents

1 Overview 1

2 Usage 3

2.1 Building Libstatmon 3
2.2 Linking with Libstatmon 3
2.3 Examples 4

3 Functions 7

3.1 statmon_init 7
3.2 statmon_register 7
3.3 statmon_create 8
3.4 statmon_enable 9
3.5 statmon_disable 9

1 Overview

LibStatMon provides a user interface for monitoring unit status. It provides a facility whereby tasks can register functions to generate user-readable status messages based upon internal task state (see Section 3.2 [statmon'register], page 7).

At program startup, another editor (in ModSAF, the Unit Operations editor) initializes libStatMon. This does little more than set the stage for more operations at run time. Also at startup, each task which has state which is shared (via the PO database) can register a function which is capable of translating this binary representation to a human-readable string.

Later, when the program is up and running, the controlling editor passes libStatMon a unit ID for monitoring. LibStatMon installs callbacks for various libPO events, so that it can update the status display whenever the tasks change.

The status monitor creates a Motif RowColumn widget for each opaque task frame in the monitored unit's stack, and one for its background frame. Each task in the task frame (for which a status function has been registered) is given a Widget in this RowColumn. In the topmost stack frame and the background frame, these are toggle buttons, in other frames they are labels. Status messages are generated and displayed for running tasks for which the toggle button has been selected.

Although the toggle buttons appear to be per-task, they are actually selecting status on a per-SAF-Model basis. That is, once the Vehicle-Follow-Route task has been selected for a unit; that task will default to being selected whenever it is active in a task frame for a selected unit (until the user explicitly chooses not to see that status message). This information is saved on a per-status-monitor basis, so that if more than one status monitor were create in a single ModSAF system, each would act independently.

As a mission progresses, the number of tasks and task frames which a unit uses changes. However, libStatMon never destroys a widget. Instead, it manages and unmanages widgets as they are needed, and changes their labels to reflect specific task names. This should reduce fragmentation of X server memory (a common problem when the display is not stable).

2 Usage

The software library 'libstatmon.a' should be built and installed in the directory '/common/lib/'. You will also need the header file 'libstatmon.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 libstatmon source directory. If these files are already built, you can skip the section on building libstatmon.

2.1 Building Libstatmon

The libstatmon source files are found in the directory '/common/libsrc/libstatmon'. 'RCS' format versions of the files can be found in '/nfs/common_src/libsrc/libstatmon'.

If the directory 'common/libsrc/libstatmon' 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/libstatmon
# co RCS/*,v
# make install
```

This should compile the library 'libstatmon.a' and install it and the header file 'libstatmon.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. libstatmon should compile without errors on the following platforms:

- Mips

2.2 Linking with Libstatmon

Libstatmon can be linked into an application program with the following link time flags: 'ld [source .o files] -L/common/lib -lstatmon [other ModSAF libraries]'. If your compiler does not support '-L' syntax, you can use the archive explicitly: 'ld [source .o files] /common/lib/libstatmon.

Libstatmon depends on libpo, libcoordinates, and libsched.

2.3 Examples

The following code segments appeared in an early version of the vehicle-level target assessment task. The function status translates the contents of the task to a user-readable format. Note the use of edt_format to format the vehicle ID in a standard fashion. edt_format should be used whenever possible, to provide a consistent interface (see section 'edt_format' in LibEditor Programmer's Manual).

```
#include "libvass_local.h"
#include <libstatmon.h>
#include <libeditor.h>
#include <stdext.h>
#include <p_safmodels.h>

static void status(task, state, task_id, context, description)
    TaskClass       *task;
    TaskStateClass  *state;
    ObjectID        *task_id;
    STATMON_CONTEXT *context;
    char             description[];
{
    VASSESS_STATE *statevars = (VASSESS_STATE *)state->data;
    char *d;
    int32 i;

    switch (statevars->state)
    {
      default:
        sprintf(description, "Not running");
        break;
      case suspended:
        sprintf(description, "Suspended");
        break;
      case looking:
        sprintf(description, "Looking for a target");
        break;
      case reevaluating:
        edt_format(context->gui, description,
                   "Best target is $v, using $S (permission $S)",
                   &statevars->recommendation.target,
                   statevars->recommendation.weapon,
                   vass_permission_string(shared->
                                          state.recommendation.permission));
        break;
    }
}

void vass_init()
```

```
{
    uint32 before_tasks[1];

    before_tasks[0] = SM_VSpotter;

    vassess_init_fsm(SM_VAssess,
                     NULL,            /* predicate */
                     1, before_tasks, /* before */
                     0, NULL          /* after */
                     );

    vass_init_access();
    vass_init_params();

    statmon_register(SM_VAssess, "Target Assessment", status);
}
```

3 Functions

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

3.1 statmon_init

```
void statmon_init()
```

statmon_init initializes libstatmon. Call this before any other libstatmon functions.

3.2 statmon_register

```
void statmon_register(saf_model, name, status)
    uint32                  saf_model;
    char                   *name;
    STATMON_STATUS_FUNCTION status;
```

'saf_model'
: Specifies the SAF model implemented by the task (from 'p_safmodels.h')

'name'
: Specifies the user-readable name of the task

'status'
: Specifies the function to translate task state variables to a descriptive message

statmon_register registers a status function with libstatmon. This should be called by any task which has state which might be meaningful to the user. The name is copied into allocated memory.

The status function should be declared as follows:

```
static void status(task, state, task_id, context, description)
    TaskClass       *task;
    TaskStateClass  *state;
    ObjectID        *task_id;
    STATMON_CONTEXT *context,
    char             description[];
```

To facilitate useful messages, the Object ID of the task, the PO database, and the GUI (which can be used by libEditor to do type conversions) are all passed as well (see section 'edt_format' in LibEditor Programmer's Manual). The resulting string should be returned in description (a 10K character buffer). The string should not start with the task name (that is done automatically), nor should it end with a \n. It should be NULL terminated.

LibStatMon translates embedded newlines into vertical bars (|), for readability. However, you can force a newline in the output by using \\n within your string. For example:

```
edt_format(context->gui, description,
           "Best target is %v\\nUsing %S",
           &statevars->recommendation.target,
           statevars->recommendation.weapon);
```

The context structure is declared as follows:

```
typedef struct statmon_context
{
    ADDRESS       gui;
    PO_DATABASE  *db;
    COORD_TCC_PTR tcc;
} STATMON_CONTEXT;
```

In the future, this structure could expand to include other context information which may be needed by tasks to generate useful messages.

3.3 statmon_create

```
STATMON_GUI_PTR statmon_create(context, controls_rc, text_disply,
                               normal_resume_btn, reaction_resume_btn,
                               cancel_btn, reaction_btn)
    STATMON_CONTEXT *context;
    Widget           controls_rc;
    Widget           text_disply;
    Widget           normal_resume_btn;
    Widget           reaction_resume_btn;
    Widget           cancel_btn;
    Widget           reaction_btn;
```

'context' Specifies the system run-time context

'controls_rc'
Specifies the Motif RowColumn widget in which the controls are to be placed

'text_disply'
Specifies the Motif Text widget in which status is to be displayed

'normal_resume_btn'
'reaction_resume_btn'
Specify buttons which are to be sensitive when the unit is executing a resumable task frame. Only will be managed, depeding upon whether the unit is executing a reaction.

'cancel_btn'
Specifies a button which is to be sensitive when the unit is executing a cancelable override

'cancel_btn'
Specifies a button which is to be sensitive when the unit is executing a reaction

statmon_create creates a status monitor GUI. The resume_btn, if non-NULL, will be made sensitive whenever the currently monitored unit has a resumable mission (a task frame stack with depth > 1). The cancel_btn, if non-NULL, will be made sensitive whenever the currently monitored unit has a cancelable override (a non-preprogrammed transparent frame).

3.4 statmon_enable

```
void statmon_enable(statmon, unit_id)
    STATMON_GUI_PTR statmon;
    ObjectID        *unit_id;
```

'statmon' Specifies the status monitor

'unit_id' Specifies the unit to monitor

statmon_enable enables the status monitor, and assigns the unit to be monitored.

3.5 statmon_disable

```
void statmon_disable(statmon)
    STATMON_GUI_PTR statmon;
```

'statmon' Specifies the status monitor

statmon_disable disables the status monitor.

LibStealth

Table of Contents

1 Overview 1

1.1 Attachment Modes 1
1.2 Algorithms 2

2 Functions 5

2.1 stealth_init 5
2.2 stealth_class_init 5
2.3 stealth_create 5
2.4 stealth_destroy 6
2.5 stealth_tick 6
2.6 stealth_appearance_received 6
2.7 stealth_set_preview 7
2.8 stealth_update_preview 7
2.9 stealth_get_position 8
2.10 stealth_get_direction 8
2.11 stealth_get_attachment 9
2.12 stealth_attach 9
2.13 stealth_set_mode 9
2.14 stealth_teleport 10
2.15 stealth_get_preview_number 10
2.16 stealth_get_color 10

1 Overview

Libstealth provides libentity-like functionality for interacting with stealth views. In this model, stealths (both remote stealths, and local stealth-preview views) are placed in the vehicle table, and given a VTAB_REMOTE_STEALTH or VTAB_LOCAL_STEALTH vehicle type. Applications can interact with all stealths the same way, regardless of their type. In the future, this library may be extended to support different network stealth protocols as well.

The stealth (also referred to as the Flying Carpet) presents a three-dimensional view of the simulated battlefield. Data packets projected onto the appropriate network exercise allow the objects they represent to be viewed on the stealth. Much of the libstealth code deals with stealth protocol packets.

The haphazard nature of the current stealth protocol (as defined in file{common/src/protocol/p_stlth.drn}) limits the ability of this library to provide a simple interface. Commands to teleport, attach, and set the mode (tether, orbit, free fly, etc) of the stealth are all separate, despite their close relationships. Furthermore there is very little feedback from the stealth about its current operational status. Future stealth work should correct these problems, at which time the interface to this library will probably be improved.

1.1 Attachment Modes

Currently there are five ModSAF stealth attachment modes:

tether — The stealth is attached to a vehicle at a certain distance and bearing. The stealth moves with the vehicle, changing its speed and direction automatically as the vehicle does. The spaceball movements allow the stealth to free fly relative to the vehicle, not the terrain.

compass — The stealth is attached to a vehicle at a certain distance and bearing. The stealth moves with the vehicle, changing its speed automatically as the vehicle does. However, the stealth will not change its direction. It remains facing the same direction even if the vehicle turns. The spaceball movements allow the stealth to free fly around the vehicle.

orbit — The stealth is attached to a vehicle at its center of mass. The stealth always moves with the vehicle. The spaceball movements allow the stealth to spherically travel around the vehicle which remains in the center.

mimic — The stealth becomes imbedded with the target vehicle, inheriting all components of velocity and orientation. The spaceball movements allow the stealth to slew its view up and down and to yaw around the Z axis of the target vehicle to facilitate viewing in all possible directions.

free fly — The stealth moves with the dynamics of an airplane. The stealth is not attached to any vehicle or terrain location. The spaceball movements allow the stealth to free fly around the terrain.

1.2 Algorithms

When ModSAF receives a stealth appearance packet, the last appearance time for that stealth is updated so that the stealth's time out clock can be reset. If enough time elapses (10 seconds) without a new stealth appearance packet, the stealth will be removed from ModSAF's vehicle table. ModSAF extracts the stealth's position and direction from the stealth appearance packet. The ModSAF Station needs this libstealth user data when drawing the PVD arrow icon which indicates the current location and azimuth of the stealth.

When a stealth attached packet is received, libstealth examines the vehicle table for vehicles with type, VTAB_REMOTE_STEALTH, to locate a stealth with the same simulation address as the sender. This needs to be done since the vehicle ID of the stealth is not part of the stealth attached packet. If an appropriate vehicle_id can not be found, libstealth ignores the packet. Otherwise, the libstealth user data is marked with the vehicle ID that the packet identifies as the stealth's current attachment vehicle. The stealth attached packet is sent by the stealth whenever the stealth becomes attached to a new vehicle (no matter whether this was caused by a packet sent from ModSAF or a stealth user's input. It is also sent by the stealth whenever it ceases to be attached to a vehicle, for whatever reason (a new attach packet from ModSAF, entering independent velocity mode, attached vehicle vanishing, etc). In this case, the 'detached' field is set to TRUE.

A ModSAF user can request a stealth teleport. Since the local preview views do not support the stealth protocol, the method used for teleport differs between LOCAL and REMOTE stealths. For local previews, teleports are performed by invoking a callback function provided at program initialization; this callback subsequently invokes a function in libpreview to perform the teleport. For remote stealths, a Teleport PDU is sent via libpktvalve to the stealth to cause it to move to a new location and/or assume a new orientation.

A ModSAF user can request a stealth attachment to a vehicle. Since the local preview views do not support the stealth protocol, the method used for attaching differs between LOCAL and REMOTE stealths. For local previews, attaches are performed by invoking a callback function provided at

program initialization; this callback subsequently invokes a function in libpreview to perform the attachment. For remote stealths, a Visibility PDU is sent via libpktvalve to the stealth to send it the exercise id. Then a Teleport PDU is sent using the position of the target_id retrieved from libentity as the teleport location. Then an Attach PDU is sent to cause the stealth to become attached to the indicated vehicle.

A ModSAF user can request a change in stealth attachment mode. Since the local preview views do not support the stealth protocol, the method used for mode change differs between LOCAL and REMOTE stealths. For local previews, mode changes are performed by invoking a callback function provided at program initialization; this callback subsequently invokes a function in libpreview to perform the mode change. For remote stealths, a Metamorphose PDU is sent via libpktvalve to the stealth to cause it to assume a particular attach mode while attached to a vehicle.

2 Functions

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

2.1 stealth_init

```
void stealth_init(valve, protocol)
    PV_VALVE_PTR valve;
    int32        protocol;
```

'valve' Specifies the packet value

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

stealth_init initializes libstealth. Call this before calling any other libstealth functions.

2.2 stealth_class_init

```
void stealth_class_init(parent_class)
    CLASS_PTR parent_class;
```

'parent_class'
Identifies the parent class for per-stealth information (probably safobj_class).

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

2.3 stealth_create

```
void stealth_create(vehicle_id, deactivate_fcn)
    int32  vehicle_id;
    void (*deactivate_fcn)(/* int32 vehicle_id */);
```

'vehicle_id'
Specifies the vehicle ID

'deactivate_fcn'
Specifies the function to call if the stealth deactivates

stealth_create creates the stealth class information for a stealth and attaches it to the stealth's libclass user data.

2.4 stealth_destroy

```
void stealth_destroy(vehicle_id)
    int32 vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

stealth_destroy frees the stealth class information for a stealth.

2.5 stealth_tick

```
void stealth_tick(vehicle_id)
    int32 vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

stealth_tick checks the stealth for network time out.

2.6 stealth_appearance_received

```
void stealth_appearance_received(vehicle_id, packet)
    int32       vehicle_id;
    PV_PACKET *packet;
```

'vehicle_id'
Specifies the vehicle ID

'packet' Specifies the packet (from libpktvalve)

stealth_appearance_received informs the stealth of a received appearance packet.

2.7 stealth_set_preview

```
void stealth_set_preview(vehicle_id, preview_number, preview_user_data,
                         teleport, attach)
    int32   vehicle_id;
    int32   preview_number;
    ADDRESS preview_user_data;
    void  (*teleport)(/* user_data, position[3], angle */);
    void  (*attach)(/* user_data, target_id, mode */);
```

'vehicle_id'
Specifies the vehicle ID

'preview_number'
Specifies the serial number of the preview

'preview_user_data'
Specifies user data which uniquely identifies the preview

'teleport'
Specifies a function to call to teleport the preview

'attach' Specifies a function to call to attach a preview (the mode is one of the dynamics modes from 'p_stlth.h').

stealth_set_preview sets parameters for local stealth view.

2.8 stealth_update_preview

```
void stealth_update_preview(vehicle_id, position, direction, attachment)
    int32   vehicle_id;
    float64 position[3];
    float64 direction[3];
    int32   attachment;
```

'vehicle_id'
Specifies the vehicle ID

'position'
Specifies the preview's position

'direction'
Specifies the preview's direction

'attachment'
Specifies the vehicle to which the preview is currently attached (or 0)

stealth_update_preview updates changing data regarding a local stealth view.

2.9 stealth_get_position

```
void stealth_get_position(vehicle_id, position)
    int32   vehicle_id;
    float64 position[3];
```

'vehicle_id'
Specifies the vehicle ID

'position'
Returns the position

stealth_get_position gets the position of a stealth vehicle.

2.10 stealth_get_direction

```
float64 stealth_get_direction(vehicle_id)
    int32 vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

stealth_get_direction gets the direction of a stealth vehicle.

2.11 stealth_get_attachment

```
int32 stealth_get_attachment(vehicle_id)
    int32 vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

stealth_get_attachment gets the current attachment of a stealth vehicle (not guaranteed to be correct). Returns 0 if unknown or not attached.

2.12 stealth_attach

```
void stealth_attach(vehicle_id, target_id)
    int32 vehicle_id;
    int32 target_id;
```

'vehicle_id'
Specifies the vehicle ID

'target_id'
Specifies the target of the attachment

stealth_attach attaches a stealth to another vehicle.

2.13 stealth_set_mode

```
void stealth_set_mode(vehicle_id, mode)
    int32 vehicle_id;
    int32 mode;
```

'vehicle_id'
Specifies the vehicle ID

'mode' Specifies the mode

stealth_set_mode sets the mode of the stealth to one of the choices for the MetamorphoseVariant in 'p_stlth.h'.

2.14 stealth_teleport

```
void stealth_teleport(vehicle_id, location, azimuth)
    int32   vehicle_id;
    float64 location[3];
    float64 azimuth;
```

'vehicle_id'
Specifies the vehicle ID

'location'
Specifies the location

'azimuth' Specifies the azimuth

stealth_teleport teleports a stealth to a new location & azimuth.

2.15 stealth_get_preview_number

```
int32 stealth_get_preview_number(vehicle_id)
    int32 vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

stealth_get_preview_number gets the preview number associated with a local stealth view (returns 0 for remote stealths).

2.16 stealth_get_color

```
int32 stealth_get_color(vehicle_id)
    int32 vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

stealth_get_color gets the force ID associated with a stealth. Each stealth is assigned a different force ID when it is received to allow visual distinction of different stealth views on the PVD.

LibSupplies

Table of Contents

1 Overview 1

2 Usage 3

2.1 Building Libsupplies 3
2.2 Linking with Libsupplies 3
2.3 Examples 4

3 Functions 5

3.1 supp_init 5
3.2 supp_class_init 5
3.3 supp_create 5
3.4 supp_destroy 6
3.5 supp_set_quantities 6
3.6 supp_get_quantities 6
3.7 supp_set_all_full 7
3.8 supp_set_percent 7
3.9 supp_get_amount 8
3.10 supp_get_max_amount 8
3.11 supp_decrement 8
3.12 supp_set_qty 9

1 Overview

Libsupplies provides a simple facility for supply management within a vehicle simulation. The vehicle sub-class maintains a list of munitions, and the amounts of those munitions. A negative amount signifies that the amount is never decremented. A single type of munition may be stored in more than one place. Functions are provided to check the levels of a given supply, to set the supply levels explicitly, or to decrement the amount of a supply.

The initial levels are specified in the vehicle's configuration, as follows:

```
(supplies (<munition integer> <quantity float> <auto_resupply charptr>)
          (<munition integer> <quantity float> <auto_resupply charptr>)
          ...
 )
```

Note that the first appearance of a munition goes in store 0, the second in store 1, and so on. The meaning of store numbering depends upon the components. The auto_resupply variable can left out or set as unlimited.

2 Usage

The software library 'libsupplies.a' should be built and installed in the directory '/common/lib/'. You will also need the header file 'libsupplies.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 libsupplies source directory. If these files are already built, you can skip the section on building libsupplies.

2.1 Building Libsupplies

The libsupplies source files are found in the directory '/common/libsrc/libsupplies'. 'RCS' format versions of the files can be found in '/nfs/common_src/libsrc/libsupplies'.

If the directory 'common/libsrc/libsupplies' 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/libsupplies
# co RCS/*,v
# make install
```

This should compile the library 'libsupplies.a' and install it and the header file 'libsupplies.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. libsupplies should compile without errors on the following platforms:

- Mips
- SGI Indigo
- Sun Sparc

2.2 Linking with Libsupplies

Libsupplies can be linked into an application program with the following link time flags: 'ld [source .o files] -L/common/lib -lsupplies -lvtab -lclass -lrdrconst -lparmgr

`-lreader -ll`'. If your compiler does not support '-L' syntax, you can use the archive explicitly: '`ld [source .o files] /common/lib/libsupplies.a`'.

Libsupplies depends on libclass, libvtab, libparmgr, and librdrconst.

2.3 Examples

To check if there is fuel, after running out of gas:

```
if (supp_get_amount(vehicle_id, munition_Fuel, 0) != 0.0)
{
    tracked->state = TRACKED_STATE_HEALTHY;
    cmpnt_available(vehicle_id, SM_TrackedHull, TRUE);
}
```

To use some fuel:

```
if (supp_decrement(vehicle_id, munition_Fuel, 0,
                   dt * speed / tracked->params->fuel_usage_mpl) == 0.0)
{
    tracked->state = TRACKED_STATE_NOGAS;
    cmpnt_available(vehicle_id, SM_TrackedHull, FALSE);
}
```

3 Functions

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

3.1 supp_init

```
void supp_init()
```

supp_init initializes libsupplies. Call this before calling any other libsupplies functions.

3.2 supp_class_init

```
void supp_class_init(parent_class)
    CLASS_PTR parent_class;
```

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

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

3.3 supp_create

```
void supp_create(vehicle_id, parms)
    int32                    vehicle_id;
    SUPPLIES_PARAMETRIC_DATA *parms;
```

'vehicle_id'
Specifies the vehicle ID

'parms' Specifies the initial parameter values

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

3.4 supp_destroy

```
void supp_destroy(vehicle_id)
    int32 vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

supp_destroy frees the supplies class information for a vehicle.

3.5 supp_set_quantities

```
void supp_set_quantities(vehicle_id, n_quantities, quantities)
    int32              vehicle_id;
    int32              n_quantities;
    MunitionQuantity *quantities;
```

'vehicle_id'
Specifies the vehicle ID

'n_quantities'
Specifies the number of quantities available

'quantities'
Specifies a list of munition quantities

supp_set_quantities sets the quantities of each supply to that specified in the array of MunitionQuantity. The order of munitions is significant (they are assigned to stores in order).

3.6 supp_get_quantities

```
void supp_get_quantities(vehicle_id, n_quantities, quantities)
    int32              vehicle_id;
    int32              n_quantities;
    MunitionQuantity *quantities;
```

'vehicle_id'
Specifies the vehicle ID

'n_quantities'
Specifies the number of quantities desired

'quantities'
Returns a list of munition quantities

supp_get_quantities gets the quantities of each supply. n_quantities identifies the number of quantities expected (any unused spaces will be filled with zeros).

3.7 supp_set_all_full

```
void supp_set_all_full(vehicle_id)
    int32 vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

supp_set_all_full restores every quantity to the initial level specified in the parametric data.

3.8 supp_set_percent

```
void supp_set_percent(vehicle_id, munition, store, percent)
    int32   vehicle_id;
    uint32  munition;
    int32   store;
    float64 percent;
```

'vehicle_id'
Specifies the vehicle ID

'munition'
Specifies the munition type

'store' Specifies the store number

'percent' Specifies the percentage (100.0 == full)

supp_set_percent sets the specified munition (in the specified store) to the specified percent of the amount set in the parametric data.

3.9 supp_get_amount

```
float64 supp_get_amount(vehicle_id, munition, store)
    int32  vehicle_id;
    uint32 munition;
    int32  store;
```

'vehicle_id'
Specifies the vehicle ID

'munition'
Specifies the munition type

'store' Specifies the store number

supp_get_amount returns the amount of the specified munition (in the specified store) which is currently available. If the amount is negative, which signifies automatic resupply is set, the absolute value of the amount is returned.

3.10 supp_get_max_amount

```
float64 supp_get_max_amount(vehicle_id, munition, store)
    int32  vehicle_id;
    uint32 munition;
    int32  store;
```

'vehicle_id'
Specifies the vehicle ID

'munition'
Specifies the munition type

'store' Specifies the store number

supp_get_amount returns the amount of the specified munition (in the specified store) which would be available if the vehicle were fully loaded.

3.11 supp_decrement

```
float64 supp_decrement(vehicle_id, munition, store, quantity)
    int32   vehicle_id;
```

```
        uint32  munition;
        int32   store;
        float64 quantity;
```

'vehicle_id'
Specifies the vehicle ID

'munition'
Specifies the munition type

'store' Specifies the store number

'quantity'
Specifies the amount to use

supp_decrement decrements the specified munition (in the specified store) by the specified amount and returns the amount remaining. This happens only if the auto_resupply variable is NOT set. If the amount is negative, meaning the auto_resupply variable is set for the specified munition, supp_decrement returns the negative of the amount.

3.12 supp_set_qty

```
int32 supp_set_qty(vehicle_id, munition, store, quantity)
    int32   vehicle_id;
    uint32  munition;
    int32   store;
    float64 quantity;
```

'vehicle_id'
Specifies the vehicle ID

'munition'
Specifies the munition type

'store' Specifies the store number

'quantity'
Specifies the amount to use

supp_set_qty looks for the specified munition (in the specified store) and sets it to the specified quantity. Negative quantities are allowed, and will signify an unlimited supply of the specified amount. The following error values can be returned:

'SUPP_SET_VEH_NF'

the vehicle was not found.

'SUPP_SET_STR_NF'

the specified munition/store was not found.

'SUPP_SET_DONE'

The specified munition/store was set.

LibTactmap

Table of Contents

1 Overview 1

2 Examples 13

3 Functions 15

3.1 tactmap_init 15
3.2 tactmap_create 15
3.3 tactmap_get_color 16
3.4 tactmap_set_scale 16
3.5 tactmap_set_center 17
3.6 tactmap_screen_to_map 17
3.7 tactmap_map_to_screen 17
3.8 tactmap_pixel_offset 18
3.9 tactmap_locale 18
3.10 tactmap_world_view 19
3.11 tactmap_get_scale 19
3.12 tactmap_create_object 20
3.13 tactmap_destroy_object 20
3.14 tactmap_add_object 20
3.15 tactmap_remove_object 21
3.16 tactmap_update_object 21
3.17 tactmap_add_transient_object 22
3.18 tactmap_tick 22
3.19 tactmap_redrawing 23
3.20 tactmap_refresh_non_terrain 23
3.21 tactmap_quad_data 23
3.22 tactmap_bundle_changes 24

4 Resources 25

1 Overview

LibTactMap provides a flexible 2D map drawing facility. An application can create an arbitrary number of maps, and specify terrain and other features for display. Some terrain features and all other features can also be made sensitive (and optionally hot, and dragable) using libSensitive. The map drawing routines are optimized for speed, and map redraw can span multiple ticks to facilitate use in conjunction with real time simulation (within a single process).

The following are the terrain features which an application can display:

TACTMAP_HYPSO
: Hypsometric background

TACTMAP_WATER
: Rivers and lakes (includes any polygon defined by its soil type, including no-go and slow-go terrain).

TACTMAP_ROADS
: Roads

TACTMAP_TREES
: Trees, tree lines, and tree canopies

TACTMAP_BUILDINGS
: Buildings, power pilons, other structures

TACTMAP_PIPELINES
: Pipelines

TACTMAP_POLITICAL
: Political Boundaries

TACTMAP_RAILROADS
: Railroads

TACTMAP_POWERLINES
: Powerlines

TACTMAP_CONTOURS
: Contour lines

TACTMAP_TOWNS
: Towns

TACTMAP_GRIDS
: Grid lines

In addition, an application can create an arbitrary number of the following objects:

TACTMAP_LINEAR

Linear objects (routes, minefields, etc.); the application specifies the vertices (in meters), line style, width, and GC.

TACTMAP_PIXMAP

Pixmaps; the application creates a pixmap and gives it a location (in meters).

TACTMAP_PICTURE

Pictures; the application provides a definition using the picture language described below, GCs, and coordinate system transforms.

TACTMAP_DRAWN

Objects the application draws itself; the application provides a function which is called to draw the object whenever the map is updated.

TACTMAP_TEXT

Text; the application provides a the string, location, GC, and some options.

TACTMAP_BGR

BGR Icons; the application provides an object type, an optional definition using the BGR language, a GC, location (in meters), a call sign, and rotation.

TACTMAP_LINE_INTERVIS

Line intervisibility; the application provides a two points the viewers agl, the targets agl, the targets height and width, the tree_opacity, and any vehicles between the two points. The visibility will be colored in a line from the first point to the second point.

TACTMAP_AREA_INTERVIS

Area intervisibility; the application provides a 2-D center point, a radius, the viewers agl, the targets agl, the targets height and width, the tree_opacity. The visibility will be colored in an area circle defined by the center point and the radius.

Individual attributes for each type of object are defined using the following structure:

```
typedef union tactmap_object_dfn
{
    struct tactmap_hypso
    {
        TACTMAP_HYPSO_KIND kind;
    } hypso;
    struct tactmap_water
    {
        TACTMAP_SENSITIVE_OPTIONS sensitive;
    } water;
    struct tactmap_roads
    {
        TACTMAP_SENSITIVE_OPTIONS sensitive;
    } roads;
    struct tactmap_trees
```

```
        {
            TACTMAP_SENSITIVE_OPTIONS sensitive;
            uint32 mask;
        } trees;
        struct tactmap_buildings
        {
            TACTMAP_SENSITIVE_OPTIONS sensitive;
        } buildings;
        struct tactmap_pipelines
        {
            TACTMAP_SENSITIVE_OPTIONS sensitive;
        } pipelines;
        struct tactmap_political
        {
            TACTMAP_SENSITIVE_OPTIONS sensitive;
        } political;
        struct tactmap_railroads
        {
            TACTMAP_SENSITIVE_OPTIONS sensitive;
        } railroads;
        struct tactmap_powerlines
        {
            TACTMAP_SENSITIVE_OPTIONS sensitive;
        } powerlines;
        struct tactmap_contours
        {
            float64 minor_spacing;
            float64 major_spacing;
        } contours;
        struct tactmap_towns
        {
            TACTMAP_SENSITIVE_OPTIONS sensitive;
        } towns;
        struct tactmap_grids
        {
            int32 interval;
            uint32 label_mask;
        } grids;
        struct tactmap_linear
        {
            TACTMAP_SENSITIVE_OPTIONS sensitive;
            LineStyle style;
            uint16 dashed;
    uint16 thickness;
            uint32 forward_arrow;
            uint32 backward_arrow;
            uint32 width;
            GC gc;
            int32 n_vertices;
            float64 *vertices;
```

```
    } linear;
    struct tactmap_pixmap
    {
        TACTMAP_SENSITIVE_OPTIONS sensitive;
        float64 x, y;
        uint32 width;
        uint32 height;
        Pixmap pixmap;
    } pixmap;
    struct tactmap_picture
    {
        TACTMAP_SENSITIVE_OPTIONS sensitive;
        uint32 dashed;
        READER_UNION *dfn;
        int32 num_gcs;
        GC gcs[TACTMAP_MAX_GCS];
        float64 magnification;
char *string;
int32 highlight;
        int32 num_csystems;
        struct tactmap_csystem
        {
            int32   parent_system;
            float64 angle;
            float64 dimensions[2];
            float64 origin[2];
        } csystems[TACTMAP_MAX_CSYSTEMS];
    } picture;
    struct tactmap_drawn
    {
        TACTMAP_SENSITIVE_OPTIONS sensitive;
        TACTMAP_DRAW_CALLBACK draw;
        ADDRESS draw_arg;
    } drawn;
    struct tactmap_text
    {
        TACTMAP_SENSITIVE_OPTIONS sensitive;
        GC gc;
        XFontStruct *font_struct;
        float64 x, y;
        uint32 see_through;
        TextAlignment alignment;
        int16 horizontal_offset;
        int16 vertical_offset;
        char *string;
    } text;
    struct tactmap_bgr
    {
        TACTMAP_SENSITIVE_OPTIONS sensitive;
        float64       x, y;
```

```
            float64         rotation;
            int32           override;
            READER_UNION *dfn;
            ObjectType      object_type;
            int32           size;
            int32           line_thickness;
        uint32          dashed;
        char            call_sign[TACTMAP_MAX_CALL_SIGN];
        int32           highlight;
            GC              gc;
        } bgr;
        struct tactmap_line_intervis
        {
            float64 point1[2];
            float64 point2[2];
            float64 viewer_agl;
            float64 target_agl;
            float64 target_width;
            float64 target_height;
            float64 tree_opacity;
            int32 n_veh;
            CTDB_VEHICLE_LOCATION *veh;
            int32 ignore1;
            int32 show_percentages;
            int32 grid_interval;
        } line_intervis;
        struct tactmap_area_intervis
        {
            float64 center_point[2];
            float64 radius;
            float64 viewer_agl;
            float64 target_agl;
            float64 target_height;
            float64 target_width;
            float64 tree_opacity;
        } area_intervis;
        /* Generic reference to a sensitive class */
        TACTMAP_SENSITIVE_OPTIONS sensitive;
    } TACTMAP_OBJECT_DFN;
```

With only a few exceptions (TACTMAP_HYPSO, TACTMAP_CONTOURS, and TACTMAP_GRIDS), any feature can be made sensitive. The specific options are specified with the following structure:

```
typedef struct tactmap_sensitive_options
{
    SNSTVE_CLASS *class;

    uint16          instance; /* Ignored for terrain features */
    ADDRESS         user_data;
```

```
} TACTMAP_SENSITIVE_OPTIONS;
```

See libSensitive for the meaning of these attributes (see section 'Overview' in LibSensitive Programmer's Manual). The `class`, `instance`, and `user_data` are up to the application to define. Note that for terrain features, only the `class` and `user_data` will be passed back in the callback; the instance will be changed to the libQuad ID for that object.

The instance information given by the application is only 16 bits, whereas libSensitive provides 32. This is because libTactMap often needs to create many sensitive objects for a single user-defined object (such as a LINEAR object, which is made up of many segments and vertices). LibTactMap provide macros to help applications interpret the instance value received in the `SNSTVE_CALL_DATA` of a libSensitive `gesture` or `excited` callback:

`TACTMAP_INSTANCE_TO_USER_INSTANCE(call_data->instance)`
: Extracts the 16 bits of instance information provided by the user from the 32 bit number passed from libSensitive.

`TACTMAP_INSTANCE_TO_INDEX(call_data->instance)`
: Extracts the libQuad feature index from the 32 bit instance number passed from libSensitive.

`TACTMAP_IS_VERTEX(call_data->instance)`
: If the `TACTMAP_CLASS` of the object is `TACTMAP_LINEAR`, this reveals whether the user clicked on a vertex of the line.

`TACTMAP_IS_SEGMENT(call_data->instance)`
: If the `TACTMAP_CLASS` of the object is `TACTMAP_LINEAR`, this reveals whether the user clicked on a segment of the line. It is the logical NOT of `TACTMAP_IS_VERTEX`.

`TACTMAP_INDEX(call_data->instance)`
: If the `TACTMAP_CLASS` of the object is `TACTMAP_LINEAR`, this reveals the index of the vertex (numbered from 0, as specified in the `vertices` field of the object definition) or segment (numbered from 0 — segment n connects vertex n to vertex n+1) which was selected.

Other fields of the object definition have the following meanings:

`hypso.kind`
: One of the following values:

  `TACTMAP_DITHER_HYPSO`
  : Dithered altitude map (higher elevations are more dense).

  `TACTMAP_COLOR_HYPSO`

Color altitude map (altitude ranges are represented by color bands; higher altitudes within a band are brighter).

trees.mask

A bitwise OR of the following choices:

TACTMAP_TREE_CANOPIES

Tree canopies (drawn as a hatched polygon)

TACTMAP_TREE_LINES

Tree lines

TACTMAP_TREE_INDIVIDUAL

Individual trees (drawn as little X's)

contours.minor_spacing

contours.major_spacing

A contour line is generated every **minor_spacing** meters in altitude. Those which are (roughly) evenly divisible by **major_spacing** are emphasized.

grids.interval

Interval between grid lines in meters. This is generally a multiple of 10.

grids.label_mask

Specifies which sides of the map to label, using bits defined by libCoordinates (bitwise OR the selection into a mask):

- **COORD_LABEL_LEFT**
- **COORD_LABEL_RIGHT**
- **COORD_LABEL_TOP**
- **COORD_LABEL_BOTTOM**

linear.style

Line style defined by the persistent object protocol ('**common/include/protocol/p_po.h**'):

LSplain Plain line

LSfrontA, LSfrontB

Fronts (semicircles facing left (A) or right (B))

LSminefield, LSminefieldAT, LSminefieldAP

Generic, anti-tank, and anti-personnel minefields (a bounding box with various kinds of circles inside)

LSberm Berm (a collection of intersecting semicircles)

LSATDitchA, LSATDitchB

Anti-tank ditches (a line with triangle-shaped teeth facing left (A) or right (B))

LSfortification

Fortification (looks something like a square wave)

`LSwire` Wire (a line with little X's on it)

`linear.dashed`
Indicates if lines should be drawn using a dashed GC.

`linear.thickness`
Indicates the pixel thickness of lines

`linear.forward_arrow`
`linear.backward_arrow`
Indicates whether arrows should be placed on the ends of the line.

`linear.width`
Width of the line (only used for minefield variants)

`linear.gc`
The GC to use for drawing (some attributes, such as line width, will be modified). The most important attribute set by the application is foreground color.

`linear.n_vertices`
Specifies how many vertices are used to define the line.

`linear.vertices`
Pointer to x y x y... format list of vertices. The contents of this array are copied, so it is safe to point this to a local variable when calling `tactmap_create_object()`. The size of this array should be (`linear.n_vertices*2*sizeof(float64)`).

`pixmap.x, pixmap.y`
The location of the pixmap (in map coordinates — meters).

`pixmap.width, pixmap.height`
The size of the pixmap (in screen coordinates — pixels).

`pixmap.pixmap`
The pixmap to copy to the screen (assumes pixmap origin 0,0 for the copy).

`picture.dashed`
Indicates if lines should be drawn using a dashed GC.

`picture.dfn`
The definition of the picture, which is described with the following language:

```
((<interior GC index>          ;; -1 to omit interior
  <border GC index>            ;; -1 to omit border
  <coordinate system index>
  <instruction>
  <instruction>
  ...
  )
 ...
)
```

`<instruction>` is one of the following:

```
(block  x0 y0 x1 y1 x2 y2 ...)
(line   width x0 y0 x1 y1 ...)
(disc   center_x center_y radius [x | y | m])
(circle width center_x center_y radius [x | y | m])
(label  center_x center_y)
```

x and y values are always relative to the dimensions passed in for the coordinate system; width values are in meters; and radius values can be relative to either dimension (x or y) or in meters (m).

picture.num_gcs

The number of GCs provided for drawing this picture.

picture.gcs

The GCs used to draw the picture.

picture.magnification

Picture magnification (1.0 == actual size, 2.0 == twice as big, etc.). Use 0.0 to treat the units of definition as pixels rather than meters.

picture.string

The string to use for labels.

picture.highlight

Flag indicating if the picture is highlighted (non-zero), and if so, index of the gc to use.

picture.num_csystems

The number of coordinate systems provided for drawing this picture (must be >= 1).

picture.csystems

List of coordinate systems. The first is relative to world coordinates; the rest are relative to the first. A coordinate system is made up of the following pieces:

parent_system

The index of the parent coordinate system. Use this one's index to be parented by world. Note that a system *may not* reference a parent with a higher index than its own.

angle The rotation of the coordinate system. 0 is down the Y axis of the reference system, and the angle (in radians) increases in a counter-clockwise direction.

dimensions

The scaling factor to apply to the X (width) and Y (length) dimensions (allows definition of generic pictures which are stretched to portray an individual vehicle).

origin The center of the coordinate system, relative to the reference system.

drawn.draw

Function to call to draw this object.

drawn.draw_arg
Single argument to pass to the drawn.draw function.

text.gc The GC to use for drawing the text. The most important attributes set by the application are foreground color, background color and font.

text.font_struct
The font metrics for the font used in text.gc. If NULL is passed, libTactMap will look up the XFontStruct on it own, however, this requires a query of the X server, and could thus hurt performance. If the application will not be changing the font frequently, it is better if this is specified.

An application can get the XFontStruct on its own using
XQueryFont(display, GContextFromGC(gc)). Note that this function allocates memory, so if an application changes the font used for a text object, it should free the old structure with
XFreeFontInfo(NULL, font_struct, 1).

text.x, text.y
The location of the text reference point (in map coordinates — meters).

text.see_through
Whether the text should be see-through. If FALSE a block around the text will be filled with the background color of the text.gc before drawing, as with XDrawImageString.

text.alignment
Text alignment defined by the persistent object protocol ('common/include/protocol/p_po.h'). The anchor point of displayed text is set via this argument.

text.horizontal_offset, text.vertical_offset
Pixel distance offsets which are applied after the text location is converted from meters to screen coordinates. Positive values are right and up.

text.string
The text to display. A NULL string is acceptable. Embedded newlines are handled correctly.

bgr.x, bgr.y
The location of the icon (in map coordinates — meters).

bgr.rotation
The rotation of the icon. 0 is North, and the angle (in radians) increases in a counterclockwise direction.

bgr.override
If TRUE, use the supplied bgr.dfn to draw the icon. Otherwise, drawn the icon based on an algorithmic interpretation of the object_type.

bgr.dfn An optional collection of character strings to be parsed by libBGR to draw the icon. This field is only used if bgr.override is FALSE; This definition is in the format as

described by libBGRDB (see section 'Overview' in LibBGRDB Programmer's Manual).

bgr.object_type
This specifies the object_type of the object represented by this icon. This field is only used if **bgr.override** is FALSE.

bgr.size This specifes the approximate BGR size of the icon in pixels. LibBGR uses this to calculate an actual size for the icon.

bgr.line_thickness
This specifies the thickness of drawn lines in the icon. Note that the thicker the lines, the longer libBGR takes to draw something. An odd numbered thickness (typically 3) works best.

bgr.dashed
Indicates if the icon should be drawn dashed.

bgr.call_sign
An optional null-terminated string to be drawn as part of the icon.

bgr.highlight
Flag indicating if the icon is highlighted.

bgr.gc The GC to use for drawing the icon. The most important attributes set by the application are foreground color, background color and font.

line_intervis.point1, line_intervis.point2
Specifies the end points of the intervisility line.

line_intervis.viewer_agl, line_intervis.target_agl
Specifies the viewer and target positions agl ("above ground level").

line_intervis.target_width, line_intervis.target_height,
Specifies the height and width of the target for visibility calculations.

line_intervis.tree_opacity
Specifies reduction of visibility resulting from trees. This number is a percentage between 0 and 1.

line_intervis.n_veh
Specifies number of vehicles passed in the vehicle structure.

line_intervis.veh
Specifies the vehicles to use in the intervisibility calculation.

line_intervis.ignore1
Specifies a vehicle to ignore. This is used when performing a vehicle to vehicle intervisibility calculation to ignore the vehicle we are looking at.

line_intervis.show_percentages
Specifies whether we display should show percentages. Used to determine whether a point to point or vehicle visibility is being calculated.

grid_interval
: Specifies the current grid interval on the screen. Used to determine how to draw the tick marks on the visibility line.

area_intervis.center_point
: Specifies the center point for the area intervisibility.

area_intervis.radius
: Specifies the radius of the area intervisibility

area_intervis.viewer_agl, area_intervis.target_agl
: Specifies the viewer and target positions agl ("above ground level").

area_intervis.target_width, area_intervis.target_height,
: Specifies the height and width of the target for visibility calculations.

area_intervis.tree_opacity
: Specifies reduction of visibility resulting from trees. This number is a percentage between 0 and 1.

The drawn.draw callback should be defined as follows:

```
void draw_callback(widget, drawable, draw_arg, boundary)
    Widget      widget;
    Pixmap      drawable;
    ADDRESS     draw_arg;
    XRectangle *boundary;
```

The function should draw onto the window XtWindow(widget), and return the boundary of what was drawn in boundary. It is the application's responsibility to determine if the object is visible (hint: use tactmap_map_to_screen). An object which is not drawn (probably because it is off the screen) should return a boundary with width and height of 0.

2 Examples

A complete example can be found in the test program '`test.c`' in the libtactmap source directory. A couple brief examples are given here.

Assuming the tactical map has been created, display roads, and make them hot:

```
extern TACTMAP_PTR my_tactmap;
extern SNSTVE_CLASS road_class;
TACTMAP_OBJECT_DFN dfn;
TACTMAP_OBJECT_PTR roads;

SNSTVE_INIT_CLASS(road_class);
road_class.sensitive = 1;
road_class.hot = 1;
bzero(&dfn, sizeof(dfn));
dfn.roads.sensitive.class = &road_class;
roads = tactmap_create_object(my_tactmap, TACTMAP_ROADS, &dfn);

tactmap_add_object(my_tactmap, roads);
```

To translate the location of a mouse click to map coordinates:

```
extern TACTMAP_PTR my_tactmap;
int32 screen_x, screen_y;
float64 map_x, map_y;

screen_x = xevent->xbutton.x;
screen_y = xevent->xbutton.y;
tactmap_screen_to_map(my_tactmap, screen_x, screen_y, &map_x, &map_y);
```

3 Functions

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

3.1 tactmap_init

```
void tactmap_init()
```

tmap_init initializes libtactmap.

Call this function before any other libtactmap functions.

3.2 tactmap_create

```
TACTMAP_PTR tactmap_create(widget, sensitive, sensitive_erase_gc,
                           refresh, tcc, ctdb, quad_data)
    Widget                    widget;
    SNSTVE_WINDOW_PTR         sensitive;
    GC                        sensitive_erase_gc;
    TACTMAP_REFRESH_CALLBACK  refresh;
    COORD_TCC_PTR             tcc;
    CTDB                     *ctdb;
    QUAD_DATA                *quad_data;
```

'widget' Specifies the widget for drawing

'sensitive'
Specifies the sensitive window

'sensitive_erase_gc'
Specifies the GC used by libsensitive for erasing

'refresh' Specifies the function to call at beginning and end of map refresh

'tcc' Specifies the coordinate system of the terrain database

'ctdb' Specifies the CTDB format terrain database

'quad_data'
Specifies the libQuad format terrain database

tactmap_create creates a tactical map. The map starts with no features displayed. The widget specifies the drawing area (libtactmap posts its own resize, expose, etc., handlers). The sensitive window should be created with a call to **snstve_create()** (see section 'snstve_create' in LibSensitive Programmer's Manual).

The **refresh** callback, if non-NULL should be declared as follows:

```
void refresh_callback(tactmap, refreshing)
    TACTMAP_PTR tactmap;
    int32       refreshing;  /* TRUE at beginning, FALSE at end */
```

3.3 tactmap_get_color

```
Pixel tactmap_get_color(tactmap, class)
    TACTMAP_PTR          tactmap;
    TACTMAP_OBJECT_CLASS class;
```

'tactmap' Specifies the tactical map

'class' Specifies the terrain feature class

tactmap_get_color returns the pixel value of a color used when drawing objects of the passed class. Pass -1 to get the background color.

3.4 tactmap_set_scale

```
void tactmap_set_scale(tactmap, scale)
    TACTMAP_PTR tactmap;
    float64     scale;
```

'tactmap' Specifies the tactical map

'scale' Specifies the map scale

tactmap_set_scale sets the map scale to 1:**scale**.

3.5 tactmap_set_center

```
void tactmap_set_center(tactmap, x, y)
    TACTMAP_PTR tactmap;
    int32       x;
    int32       y;
```

'tactmap' Specifies the tactical map

'x, y' Specify the new center (in meters)

tactmap_set_center sets the map center.

3.6 tactmap_screen_to_map

```
void tactmap_screen_to_map(tactmap, screen_x, screen_y, map_x, map_y)
    TACTMAP_PTR tactmap;
    int32       screen_x;
    int32       screen_y;
    float64     *map_x;
    float64     *map_y;
```

'tactmap' Specifies the tactical map

'screen_x, screen_y'
Specifies a point in screen coordinates (pixels)

'map_x, map_y'
Returns the corresponding poing in map coordinates (meters)

tactmap_screen_to_map converts a screen location to a map location.

3.7 tactmap_map_to_screen

```
int32 tactmap_map_to_screen(tactmap, map_x, map_y, screen_x, screen_y)
    TACTMAP_PTR tactmap;
    float64     map_x;
    float64     map_y;
    int32       *screen_x;
    int32       *screen_y;
```

'tactmap' Specifies the tactical map

'map_x, map_y'
Specifies a point in map coordinates (meters)

'screen_x, screen_y'
Returns the corresponding point in screen coordinates (pixels)

tactmap_map_to_screen converts a map location to a screen location. The return value indicates whether the point is visible (TRUE if visible).

3.8 tactmap_pixel_offset

```
void tactmap_pixel_offset(tactmap, screen_dx, screen_dy, map_dx, map_dy)
    TACTMAP_PTR tactmap;
    int*2       screen_dx;
    int*2       screen_dy;
    float64     *map_dx;
    float64     *map_dy;
```

'tactmap' Specifies the tactical map

'screen_dx, screen_dy'
Specifies an offset in screen coordinates (pixels)

'map_dx, map_dy'
Returns the corresponding offset in map coordinates (meters)

tactmap_pixel_offset converts an offset in screen coordinates to the equivalent offset in meters.

3.9 tactmap_locale

```
void tactmap_locale(tactmap, center_x, center_y, width, height)
    TACTMAP_PTR tactmap;
    float64     *center_x;
    float64     *center_y;
    float64     *width;
    float64     *height;
```

'tactmap' Specifies the tactical map

'center_x, center_y'
Returns the coordinates of the map center (meters)

'width' Returns the map width (meters)

'height' Returns the map height (meters)

tactmap_locale returns center and size of map in meters.

3.10 tactmap_world_view

```
void tactmap_world_view(tactmap, world_view_on)
    TACTMAP_PTR tactmap;
    int32       world_view_on;
```

'tactmap' Specifies the tactical map

'world_view_on'
Specifies whether world view mode should be enabled

tactmap_world_view puts the tactical map in a mode where the grid lines for the entire database are shown, and a box representing the viewport is laid on top.

3.11 tactmap_get_scale

```
float64 tactmap_get_scale(tactmap, width, height)
    TACTMAP_PTR tactmap;
    int32       width;
    int32       height;
```

'tactmap' Specifies the tactical map

'width' Specifies the requested sub-width (pixels)

'height' Specifies the requested sub-height (pixels)

tactmap_get_scale determines the map scale which would encapsulate a box of the passed size (in pixels).

3.12 tactmap_create_object

```
TACTMAP_OBJECT_PTR tactmap_create_object(tactmap, class, dfn)
    TACTMAP_PTR          tactmap;
    TACTMAP_OBJECT_CLASS class;
    TACTMAP_OBJECT_DFN   *dfn;
```

'tactmap' Specifies the tactical map

'class' Specifies the object class

'dfn' Specifies the initial definition of the object to be created

tactmap_create_object creates an object which can be placed on the tactical map.

See Section 3.13 [tactmap destroy object], page 20.
See Section 3.14 [tactmap add object], page 20.
See Section 3.15 [tactmap remove object], page 21.
See Section 3.16 [tactmap update object], page 21.

3.13 tactmap_destroy_object

```
void tactmap_destroy_object(tactmap, obj)
    TACTMAP_PTR        tactmap;
    TACTMAP_OBJECT_PTR obj;
```

'tactmap' Specifies the tactical map

'object' Specifies the object to add

tactmap_destroy_object destroys an object created with tactmap_create_object and frees any memory associated with that object.

See Section 3.12 [tactmap create object], page 20.
See Section 3.14 [tactmap add object], page 20.
See Section 3.15 [tactmap remove object], page 21.
See Section 3.16 [tactmap update object], page 21.

3.14 tactmap_add_object

```
void tactmap_add_object(tactmap, object)
    TACTMAP_PTR        tactmap;
    TACTMAP_OBJECT_PTR object;
```

'tactmap' Specifies the tactical map

'object' Specifies the object to add

tactmap_add_object adds an object (created with tactmap_create_object) to the map display list.

See Section 3.12 [tactmap'create'object], page 20.
See Section 3.13 [tactmap'destroy'object], page 20.
See Section 3.15 [tactmap'remove'object], page 21.
See Section 3.16 [tactmap'update'object], page 21.

3.15 tactmap_remove_object

```
void tactmap_remove_object(tactmap, object)
    TACTMAP_PTR        tactmap;
    TACTMAP_OBJECT_PTR object;
```

'tactmap' Specifies the tactical map

'object' Specifies the object to remove

tactmap_remove_object removes an object from the map display list.

See Section 3.12 [tactmap'create'object], page 20.
See Section 3.13 [tactmap'destroy'object], page 20.
See Section 3.14 [tactmap'add'object], page 20.
See Section 3.16 [tactmap'update'object], page 21.

3.16 tactmap_update_object

```
void tactmap_update_object(tactmap, object, dfn)
    TACTMAP_PTR        tactmap;
    TACTMAP_OBJECT_PTR object;
    TACTMAP_OBJECT_DFN *dfn;
```

'tactmap' Specifies the tactical map

'object' Specifies the object to update

'dfn' Specifies the new object definition

tactmap_update_object changes the attributes of an existing object.

See Section 3.12 [tactmap create object], page 20.
See Section 3.13 [tactmap destroy object], page 20.
See Section 3.14 [tactmap add object], page 20.
See Section 3.15 [tactmap remove object], page 21.

3.17 tactmap_add_transient_object

```
void tactmap_add_transient_object(tactmap, object)
    TACTMAP_PTR        tactmap;
    TACTMAP_OBJECT_PTR object;
```

'tactmap' Specifies the tactical map

'object' Specifies the object to add

tactmap_add_transient_object adds an object (created with tactmap_create_object) to the map display list. This is identical in operation to tactmap_add_object, except the object will be displayed on top of the pixmap where other objects are placed. Transient objects must be refreshed frequently, since popups and window drawing may erase them.

3.18 tactmap_tick

```
void tactmap_tick(tactmap, max_time_slice)
    TACTMAP_PTR tactmap;
    uint32      max_time_slice;
```

'tactmap' Specifies the tactical map

'max_time_slice'
Specifies the maximum run time this tick

tactmap_tick updates the map image. Call this function periodically to redraw the map (if

no redraw is needed, the function will just return). `max_time_slice` specifies the amount of time which is acceptable to spend redrawing this tick. If drawing operations exceed this time allotment, the refresh will not be completed until the next tick or later.

3.19 tactmap_redrawing

```
int32 tactmap_redrawing(tactmap)
    TACTMAP_PTR tactmap;
```

'tactmap' Specifies the tactical map

`tactmap_redrawing` returns `TRUE` if the map is in the process of refreshing itself. Drawing on a map which is in this state will probably disappear on the next call to `tactmap_tick()`.

See Section 3.2 [Argument refresh], page 15.

3.20 tactmap_refresh_non_terrain

```
void tactmap_refresh_non_terrain(tactmap)
    TACTMAP_PTR tactmap;
```

'tactmap' Specifies the tactical map

`tactmap_refresh_non_terrain` runs through all non-terrain objects and redraws them. This may be called after editing operations to correct accidental erasures.

3.21 tactmap_quad_data

```
QUAD_DATA *tactmap_quad_data(tactmap)
    TACTMAP_PTR tactmap;
```

'tactmap' Specifies the tactical map

`tactmap_quad_data` returns the libQuad quad data structure used by a tactical map.

3.22 tactmap_bundle_changes

```
void tactmap_bundle_changes(tactmap)
    TACTMAP_PTR tactmap;
```

'tactmap' Specifies the tactical map

tactmap_bundle_changes sets do_refresh to delay redrawing. This is used in the cases where there are many changes made to the tactmap and the redrawing should wait until the tactmap is ticked again.

4 Resources

Various attributes of the map display can be customized on a per-X-server basis through the use of resources. These are typically set in a user's '.xresources' file. For example, to draw blue (instead of green) trees, the entry in the '.xresources' file would read:

```
*TacticalMap.tree: blue
```

The full list of resource names, default values, and descriptions follows:

kground (default: Gray75)
Map background color

hypsoDither (default: Black)
Color used for drawing hypsometric dither patterns

water (default: steelBlue)
Unfordable water color

fordable (default: SkyBlue)
Fordable water color

noGo (default: FireBrick)
No-go terrain color (only available on a few databases)

slowGo (default: Gold3)
Slow-go terrain color (only available on a few databases)

road (default: Red1)
Road color

tree (default: DarkGreen)
Tree, tree line, and tree canopy color

building (default: Gray25)
Building (and other structure) color

pipeline (default: Cyan4)
Pipeline color

political (default: Orange3)
Color of political boundaries

railroad (default: Black)
Color used for railroads

powerline (default: Gray25)
Color used for powerlines

contour (default: Sienna4)
Color used for contour lines (minor are drawn thin, major are drawn thick)

town (default: Black)
Color used to draw town names

grid (default: Black)
Color used to draw grid lines and grid labels

compass (default: #710006)
Color used to draw North annotation

townFont (default: XtDefaultFont)
Font used for town names

gridFont (default: XtDefaultFont)
Font used for grid labels

hypsoMap (default: see below)
String specifying the altitude color bands. The format is as follows:

```
[<altitude> <dark color> <bright color>]*
```

The altitude specifies the lowest altitude of a color band. Altitudes within a band are interpolated between the dark and bright versions of the color. The default is the string:

```
"0 #b6c05c #ccd767 200 #ecec84 #f9f98b 500 #d4b068 #efc575
1000 #b87979 #d58c8c 2000 #934548 #b35558 3000 #a99197 #c0a5ab
4000 #bbacc8 #d8c7e7 5000 #cdabb2 #e3bdc5 6000 Gray80 White"
```

LibTask

Table of Contents

1 Overview 1

2 AAFSM Code Generator 3

2.1 Opening Comment 3
2.2 Name and Structure Declarations 4
2.3 Sub-task Declarations 4
2.4 Sub-frame Declarations 6
2.5 Criteria Declarations 8
2.6 Event Declarations 9
2.7 Special Commands 10
2.8 Start Body 12
2.9 State Definitions 13
2.10 End Body 14
2.11 Suspend Body 14

3 Functions 17

3.1 task_init 17
3.2 task_register 17
3.3 task_get_state 18
3.4 task_spawn_subtask 18
3.5 task_subtask_parameters 20
3.6 task_update_subtask 20
3.7 task_suspend_subtask 20
3.8 task_resume_subtask 21
3.9 task_delete_subtask 21
3.10 task_stop_subframe 22
3.11 task_get_db 22
3.12 task_get_unit 22
3.13 task_complete 23
3.14 task_pop_ownframe 23
3.15 task_class_init 23
3.16 task_create 24
3.17 task_destroy 24
3.18 task_exec 24
3.19 task_predicate 25
3.20 task_stop 25
3.21 task_suspend 26

3.22 task_stop_if_suspended 26
3.23 task_params 26
3.24 task_migrated 27
3.25 task_get_sequence 27

1 Overview

Libtask is a subclass which manages the association between Task class objects in the PO database, and the vehicle subclasses which execute those tasks. Each task is identified with a SAF model number (defined in 'p_safmodels.h'). When each task library is initialized it tells libtask of its presence and passes information so that libtask can execute the subclass when necessary.

Libtask also provides utility functions which are used by tasks, and by the *augmented asynchronous finite state machine* (AAFSM) code generator (see Chapter 2 [AAFSM Code Generator], page 3).

Libtask expects a subset of the following functions from each task:

Changed parameters
: This function is called for an executing task, just prior to its `tick` if its parameters changed in the PO database since its last tick. This is provided so that tasks do not need to check their parameters every tick or install object changed handlers. Changes to other objects which are referenced by the task's parameters, also trigger this event. This event will also be run if a task's parameters are overridden via a transparent task frame.

Predicate
: This function is called for *enabling tasks* to determine if their conditions have been met. For example, an enabling task which monitors crossing lines, will return `TRUE` if the line has been crossed, `FALSE` otherwise. Tasks which are not simply enabling tasks should pass a function which indicates if they are in their `ended` state. This way, a subsequent task frame may refer to an executing task as its enabling task, to trigger a transition when the task completes.

Start
: This function is called to start the task execution. The start function for each task should prepare the task for its first tick, which will be called immediately after the start. This may be called in the following cases:

  - When a task is first executed, or executed again after being stopped. In this case the `state` of the task will be `not_running` or `ended`.
  - When a task is resumed, after being Suspended. In this case, the `state` of the task will be `suspended`.
  - When a vehicle migrates from another hardware platform while the task is running. In this case the `state` of the task could be any valid state.

Tick
: This function is called to tick the task finite state machine.

Stop This function is called to prematurely stop execution of a task. Note that the end function may be called for a task which is already in its ended state.

Suspend This function is called when a task is suspended (either because an opaque frame was pushed on top of its frame, the **suspended** flag was set in the task, or it was removed temporarily from its frame by its creator).

The preprocessor program '**fsm2ch**' converts an AAFSM source file into a C program which automatically defines these functions, and includes them in a call to **task_register**.

2 AAFSM Code Generator

The AAFSM code generator is simply an 'awk' script which converts a finite state machine ('.fsm') source file into C code. The principle is similar to that used in 'yacc' and 'lex', where a simple language is used to describe the structure of the desired program, and fragments of C source code are used to specify the details.

Within a '.fsm' file, the finite state machine is distinguished by the use of the grave character (`). The following sections explain the different sections of the state machine definition.

2.1 Opening Comment

The opening comment is, of course, optional. It reminds readers of the software of the variables which are available within the state machine code fragments. It generally goes as follows:

```
/* Note that the variables:
 *   int32                 vehicle_id;
 *   PO_DB_ENTRY           *task_entry;
 *   PO_DB_ENTRY           *unit_entry;
 *   TaskStateClass        *state_object;
 *   <TASK>_PARAMETERS     *parameters;
 *   <TASK>_STATE          *state;
 *   <TASK>_VARS           *private;
 * are always available.  Also note that changes to the state data
 * are automatically transmitted on the network.
 */
```

The name of the **parameters**, **state**, and **private** structures, will differ between machines.

Also, it is customary to mention the events which are defined, and what they mean. The following events are required:

```
/* Events:      tick - state machine tick
 *              params - change in parameters
 */
```

2.2 Name and Structure Declarations

The state machine itself starts with the opening grave, and the following items:

name Specifies the name of the task. This name will be prepended to various function names which are generated automatically, such as **_init_fsm**, **_start**, **_end**, **_suspend**, and **_ended**. It is also used when accessing the private variables, by appending **_user_data_handle** (see section 'class_get_user_data' in LibClass Programmer's Manual). Finally, this **name** is used to name the enumerated type for machine state by appending **_states**.

PARAMETERS Specifies the structure used to store parameters of the task. The **data** portion of the **TaskClass** object which corresponds to this task is assumed to contain this structure.

STATE Specifies the structure used to store the task state. The **data** portion of the **TaskStateClass** object which is used by the task is assumed to contain this structure. This structure must contain a variable called **state** which is of type **enum <name>_state**.

VARS Specifies the structure used to store private data for the task. This is the data which is attached to the vehicle executing the task via its libclass user data handle.

For example, the bingo fuel task uses the following declarations:

```
`
ubingofuel UBINGOFUEL_PARAMETERS UBINGOFUEL_STATE UBINGOFUEL_VARS
```

The unit follow route task uses:

```
`
uflwrte UFLWRTE_PARAMETERS UFLWRTE_STATE UFLWRTE_VARS
```

2.3 Sub-task Declarations

Next are the *optional* sub-task declarations. Subtasks declarations describe the tasks which this task spawns, and provide a code fragment to fill in the spawned task's parameters. Each declaration contains the following parts:

SUBTASK: Indicates to the code generator that a subtask is being defined.

Name Provides a user-readable name which will be used to reference this task from now on. Each name must be unique.

member For each task type spawned, there should be an array in the state structure to hold the references to these tasks. This specifies which member of the structure is the array which holds instatiations of this task.

SAF Model Specifies the SAF model number of the task being spawned.

PARAMETERS
Specifies the structure of the parameters which are used by the spawned task.

reference count
Specifies how many of the reference slots are used by the spawned task.

state initialization function
Specifies a function provided by the spawned task's library to initialize the state variables of the spawned task.

BCKGRND If the subtask declaration line ends in the keyword **BCKGRND**, the task will be considered to already exist in the background, causing **SPAWN** to search for the task instead of creating it.

code fragment
A body of C code which initializes or updates the parameters of the spawned task from this task's parameters, state, or other information.

In addition to the usual variables, the following additional variables may be referenced within the code body:

<SUBTASK>_PARAMETERS *subtask
Parameters of the spawned task, which should be filled out by the code body

int32 is_update
Flag which is set when the code body is running to update an existing task, due to a **params** event in the parent task.

END_SUBTASK
Indicates the end of the subtask definition.

Note that the entire declaration up to the code body must be on one line, however for readability, some of the declarations have been split into two lines (the first ending with '\') in the examples below.

For example, the following was used by a version of unit-level route following to declare a vehicle-level route following subtask, and a vehicle-level orbit subtask:

```
SUBTASK: Vehicle_Follow_Route vflwrte_task SM_VFlwRte \
```

```
VFLWRTE_PARAMETERS 1 vflwrte_init_task_state
{
    subtask->route = private->route_id;
    subtask->speed = parameters->speed;
    subtask->altitude = parameters->altitude;
    subtask->move_type = parameters->move_type;
}
END_SUBTASK

SUBTASK: Vehicle_Orbit vorbit_task SM_VOrbit \
VORBIT_PARAMETERS 0 vorbit_init_task_state
{
    float64 my_pos[3];

    if (!is_update) /* Don't change after initially set */
    {
        ent_get_position(vehicle_id, my_pos);
        subtask->center_pt[X] = my_pos[X];
        subtask->center_pt[Y] = my_pos[Y];
    }

    subtask->radius = private->params->orbit_radius;
    subtask->speed = private->params->orbit_speed;
    subtask->altitude = parameters->altitude;
}
END_SUBTASK
```

2.4 Sub-frame Declarations

Those tasks which create other task frames may do so using a sub-frame declaration. These declarations have the following format:

Name Provides a user-readable name which will be used to reference this task frame from now on. Each name must be unique.

member For each task frame spawned, there should be an array in the state structure to hold the references to these task frames. This specifies which member of the structure is the array which holds instatiations of this frame.

instruction

Indicates the instruction which should be used to install the task frame. It must be one of the following:

TIIPopNone

Push this frame onto the stack

TIIPopNonOpaque

Pop all non-opaque frames down to the first opaque frame, then push this frame

TIIPopOpaque

Pop all frames down to and including the first opaque frame, then push this frame

optional flags

Finally, optional flag values may be used:

opaque Indicates that the task frame should be opaque (if omitted, the created frames will be transparent).

destroy Indicates that the task frame should be automatically destroyed when it completes.

code fragment

A body of C code which assembles the task frame.

In addition to the usual variables, the following additional variables may be referenced within the code body:

ObjectID *unit_id

Specifies the unit which will be executing the task frame

int32 which

Specifies which instantiation of the task frame is occurring

The body may include defined subtasks using the directive:

INCLUDE: <Name>

The **<Name>** specifies the name of the task to be included.

Note that the entire declaration up to the code body must be on one line, however for readability, some of the declarations have been split into two lines (the first ending with '\') in the examples below.

For example, the following were used in a version of the unit bingo-fuel task to define a return-to-base task frame.

```
SUBTASK: Unit_Return_To_Base urtb_task SM_URTB \
URTB_PARAMETERS 1 urtb_init_task_state
{
    subtask->base = parameters->base;
    subtask->speed = parameters->speed;
    subtask->altitude = parameters->altitude;
}
END_SUBTASK

SUBFRAME: Return_To_Base rtb_taskframe TIIPopOpaque opaque destroy
```

```
INCLUDE: Unit_Return_To_Base
END_SUBFRAME
```

2.5 Criteria Declarations

Each task may declare criteria under which it will want to control actuators. This information is passed on to libTaskPri, where it is merged with information from other tasks to determine which task controls each actuator during any tick (see section 'taskpri_register_for_group' in LibTaskPri Programmer's Manual).

For each declared output criteria, a task implicitly defines a variable within the context of the tick, params, and any other events. This variable is set to indicate whether the task has been selected to provide output, and is called *group*_enabled.

For example, if the prioritization of the weapons and sensors actuators is given by the SM_TaskPriority parameters:

```
(SM_TaskPriority (tasks
                 ("sensors" SM_VTargeter
                            SM_VSearch
                            )
                 ("weapons" SM_VTargeter
                            SM_VSearch
                            )
                 ...
```

this means that the tasks VTargeter and VSearch both might want to provide output to these actuators, and VTargeter has higher priority in both cases.

The FSM definition for VSearch might look like this:

```
(
vsrch VSEARCH_PARAMETERS VSEARCH_STATE VSEARCH_VARS

CRITERIA: sensors
  TASKPRI_RUNNING
END_CRITERIA

CRITERIA: weapons
  TASKPRI_AND(TASKPRI_RUNNING, TASKPRI_SELECTED(SM_VSearch, "sensors"))
END_CRITERIA
```

```
`
                tick()
`
                        params()
`
...
```

The first criteria says, "I want to control sensors whenever I am running." The second one says, "I want to control weapons whenever I am running, and the task SM_VSearch has control of the sensors."

As a result of these two declarations, there will be two variables defined in the context of the state machine:

```
int32 sensors_enabled;
int32 weapons_enabled;
```

Prior to issuing commands to an actuator, libvsearch must check to make sure the corresponding flag is TRUE.

2.6 Event Declarations

The next portion of the state machine definition is the declaration of events. At a minimum, the events tick and params must be defined. Each event starts with an opening grave, then has a name, and an argument list (which may be empty). For example, all tasks will start with the standard events:

```
`
                tick()
`
                        params()
```

If a task needs to define its own events, those should follow. They may take additional arguments. For example, suppose a task has a need for an event called my_event which takes an argument called reason which is of type EVENT_REASON, and an argument called time, which is of type uint32:

```
`
                                my_event(reason, time)
                                    EVENT_REASON reason;
```

```
                    uint32      time;
```

This event can be invoked from utility functions defined later in the '.fsm' file as follows:

```
TASK_OPEN_INVOCATION(my_event, vehicle_id, db, task_id)
    reason, time
TASK_CLOSE_INVOCATION
```

vehicle_id specifies the vehicle which is executing the task; db is a pointer to the PO_DATABASE; and task_id is a pointer to the ObjectID of the task for which the event occurred.

These macros expand to a sequence of calls which find the task and state PO objects, call the user-defined event, and update the task state on the network, if it changed.

2.7 Special Commands

The remaining sections of the state machine definition may make use of several special commands:

Transitions

Transitions to other states are indicated as follows:

```
^newstate; comments
```

newstate is the state to which a transition should occur. Anything after the ';' is assumed to be a comment and is skipped by the code generator. Comments should be included at each transition to describe the reason for transitioning (these comments may be used by automata-drawing programs).

Note that a transition implies a return from the event.

Spawning Tasks or Frames

Subtasks and Subframes which were defined earlier in the machine may be spawned using the command:

```
SPAWN <who> <Name> [<which>]
```

who is a pointer to the ObjectID of the unit which is to execute the task or task frame. The special value SELF may be used to run the task in the current unit. Name is the name of the task or task frame to spawn. The optional which is a small integer indicating which instantiation of the task or task frame is being spawned (it should be less than the maximum specified in the task or task frame declaration).

Stopping Tasks

Tasks which were SPAWNed can be stopped using the command:

```
STOP <Name> [<which>]
```

This happens automatically when the current task is suspended.

Resuming Tasks

Tasks which have been STOPped can be restarted using the command:

```
RESUME <Name> [<which>]
```

Note that it is also acceptable to reSPAWN a task which was suspended, to start its execution in the context of a different unit.

Deleting Tasks

Tasks which were SPAWNed can be deleted using the command:

```
DELETE <Name> [<which>]
```

This happens automatically when the current task ends.

Stopping Task Frames

Task frames which were SPAWNed can be stopped using the command:

```
STOP <Name> [<which>]
```

Checking if a Task is Finished

Tasks which have been spawned can be tested for completion as follows:

```
if (
    FINISHED <Name> [<which>]
    )
```

The FINISHED call translates to a single function call, so it may be combined with other logical operators in the conditional. For example:

```
if (
    FINISHED Go_Across_Bridge 0
    &&
    FINISHED Go_Across_Bridge 1
    )
  SPAWN unit_3 Go_Across_Bridge 2
```

Self-Destructing Frames

Some reactive tasks create task frames which self-destruct (the spawned frame contains a task which pops the frame, and the frame was created with the destroy option). In these cases, the task will need to clear out any references it has kept to the task frame after spawning. The following command can be used by a task to clear these references.

```
CLEAR <Name> [<which>]
```

For example, a version of the commit task would wait for a target which meets the commit criteria, then react by pushing an opaque frame which performed an intercept. The task had no reason to save the references to the spawned frame, so it cleared them right away (to avoid having references to a destroyed object, later).

```
'
evaluating
'                   tick
                    if (tgt_meets_commit_criteria(vehicle_id, private))
                    {
                        set_vassess_fire_at_will(vehicle_id, private,
                                                 parameters);
                        SPAWN SELF Air_To_Air_Intercept
                        CLEAR Air_To_Air_Intercept
                    }
'                       params
                        set_vassess_params(vehicle_id, private,
                                           parameters);
'
```

2.8 Start Body

Next comes a code fragment which starts the task (see Chapter 1 [Overview], page 1). The start body should perform any bookkeeping (such as initializing private variables), and should ensure that the task transitions into an acceptable state.

In the case of resuming a task, any subtasks which were created by this task will have been suspended when this task was suspended. Thus, it is important for the start body to either RESUME or reSPAWN the tasks in the start body. In general, it is better to reSPAWN the task, since the composition of a unit or the parameters of the task may have changed while the task was suspended.

For example, this is the start body of a version of the unit-level follow route task:

```
'
START
'
                save_route_id(private, task_entry);
                switch (state->state)
                {
                  case takingoff:
                  case following_route:
                  case at_end_of_route:
                        break;
                  case ended:
                  default:
                        if (vflwrte_able_to_followroute(vehicle_id,
                                                        private->ctdb))
                        {
```

```
                    SPAWN SELF Vehicle_Follow_Route
                    SPAWN SELF Vehicle_Search
                    if (parameters->move_type == VFLWRTE_MOVE_CONTOUR)
                      SPAWN SELF Vehicle_Fly_Contour
                    ^following_route; In the air
                }
                else
                {
                    SPAWN SELF Vehicle_Take_Off
                    ^takingoff; Need to take off
                }
            }
'
```

First, the machine saves the route ID in its private variables, so that it will be able to distinguish route changes from other parameter changes.

Next, it looks at the initial state. If the machine is already in a running state, such as after a migration occurs, the software does not need to do anything special (in this case). If the machine is not running, the software spawns some tasks and transitions to its first state, depending upon the current conditions.

2.9 State Definitions

Next, the state machine describes each state. Each description is organized as follows:

```
'
<state name>
'       tick
        code fragment
'               params
                code fragment
'                       <other event>
                        code fragment
'                               <other event>
                                code fragment
...
```

Each code fragment is run when in the specified state during the corresponding event.

2.10 End Body

The end body is a code fragment which runs whenever the task ends. A task can end with a transition to END:

```
~END
```

or it can end because it was in a frame which was popped. In either case, the end body executes. Tasks in reactive frames can cause the frame in which they reside to be popped when they end (the self-destruct behavior described above) using the command:

```
POP_OWN_TASKFRAME
```

in their end body.

The general form of the end body is:

```
'
END
'
        code fragment
```

The generated software automatically destroys any spawned tasks (but not spawned task frames) when a task is ended.

2.11 Suspend Body

The *optional* suspend body is a code fragment which runs whenever the task is suspended. A task is suspended when an opaque frame is pushed on top of its frame, or when it is temporarily removed from a background frame.

The general form of the suspend body is:

```
'
SUSPEND
'
        code fragment
```

The generated software automatically suspends any spawned tasks (but not spawned task frames) when a task is suspended.

3 Functions

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

3.1 task_init

```
void task_init()
```

`task_init` initializes libtask. Call this before calling any other libtask functions or any specific task initializations.

3.2 task_register

```
void task_register(saf_model_number, params, predicate,
                   start, tick, end, suspend, state_size,
                   n_before, before, n_after, after)
    uint32   saf_model_number;
    void   (*params)();
    int32  (*predicate)();
    void   (*start)();
    void   (*tick)();
    void   (*end)();
    void   (*suspend)();
    uint32   state_size;
    int32    n_before;
    uint32   before[];
    int32    n_after;
    uint32   after[];
```

'saf_model_number'
: Specifies the SAF model implemented by the task (from 'p_safmodels.h')

'params'
: Specifies the function to call when task parameters change

'predicate'
: Specifies the function to call for enabling tasks

'start'
: Specifies the function to call to start the task

'tick'
: Specifies the function to call each tick

'end'
: Specifies the function to call if the task must be stopped

'suspend' Specifies the function to call if the task is suspended (start will be invoked before the next tick)

'state_size'
Specifies the size of the task state data

'n_before'
Specifies how many task are in the before list

'before[]'
Specifies the list of tasks which must be executed before this one

'n_after' Specifies how many task are in the after list

'after[]' Specifies the list of tasks which must be executed after this one

task_register registers a task with libtask so that it may be executed later. Note that any of the functions may be passed as NULL. The predicate function for tasks which are not just enabling tasks should return whether they are in the ended state. This way, a subsequent task frame may refer to an executing task as its enabling task, to trigger a transition when the task completes.

Note that specifying before and after tasks *does not* guarantee that those tasks will be executing — it merely ensures that if those tasks are executing, they're execution will happen at the right time relative to this task.

3.3 task_get_state

```
ADDRESS task_get_state(db, task_id)
    PO_DATABASE *db;
    ObjectID    *task_id;
```

'db' Specifies the PO database

'task_id' Specifies the task's PO ObjectID

task_get_state finds state variables associated with a task (performs a sequence of po_get_object calls, and error checking).

3.4 task_spawn_subtask

```
void task_spawn_subtask(task_entry, unit_or_frame_id, spawn_task,
                        spawn_init, spawn_id)
```

```
PO_DB_ENTRY     *task_entry;
ObjectID        *unit_or_frame_id;
TaskClass       *spawn_task;
void           (*spawn_init)();
ObjectID        *spawn_id;
```

'task_entry'
: Specifies the task which is spawning a new task

'unit_or_frame_id'
: Specifies either the unit which is to execute the task (in its background frame), or the frame into which the task should be spawned

'spawn_task'
: Specifies the body of the task which will be spawned

'spawn_init'
: Specifies a function to call to initialize the task's state

'spawn_id'
: Specifies/Returns the Object ID of the spawned task

`task_spawn_subtask` starts a task in the specified frame (or if the `unit_or_frame_id` is that of a unit, then in that unit's background frame). If `spawn_id` already contains the Object ID of an existing task, that task will be reused. Otherwise a new task will be created and its ID will be returned in `spawn_id`.

The caller should fill in the `model`, `refcount`, `size`, and `data` parts of the task. The `spawn_init` function will be called to fill in the `refcount`, `size`, and `data` of the created (or reused) `state` object. This function should be prototyped as follows:

```
void spawn_init(task, state)
    TaskClass      *task;
    TaskStateClass *state;
```

For example, the init function used by the vehicle-level route following task, might look like this:

```
/*ARGSUSED*/
void vflwrte_init_task_state(task, state)
    TaskClass      *task;
    TaskStateClass *state;
{
    state->size = sizeof(VFLWRTE_STATE);
    state->refcount = 0;
    bzero(state->data, sizeof(VFLWRTE_STATE));
}
```

3.5 task_subtask_parameters

```
int32 task_subtask_parameters(task_entry, spawn_id, params)
    PO_DB_ENTRY     *task_entry;
    ObjectID        *spawn_id;
    ADDRESS         *params;
```

'task_entry'
Specifies the task which spawned the subtask

'spawn_id'
Specifies the Object ID of the spawned subtask

'params' Returns the parameter values of the subtask

task_subtask_parameters gets the current parameters used by a spawned task. This is used by the '**fsm2ch**' code generator to provide initial values for parameters when a subtask is updated.

3.6 task_update_subtask

```
void task_update_subtask(task_entry, spawn_task, spawn_id)
    PO_DB_ENTRY     *task_entry;
    TaskClass       *spawn_task;
    ObjectID        *spawn_id;
```

'task_entry'
Specifies the task which spawned the subtask

'spawn_task'
Specifies the new task parameters

'spawn_id'
Specifies the Object ID of the spawned subtask

task_update_subtask updates the parameters of a spawned task.

3.7 task_suspend_subtask

```
void task_suspend_subtask(task_entry, spawn_id)
    PO_DB_ENTRY *task_entry;
    ObjectID    *spawn_id;
```

'task_entry'
Specifies the task which spawned the subtask

'spawn_id'
Specifies the Object ID of the spawned subtask

task_suspend_subtask suspends execution of a spawned task by setting its **suspended** flag.

3.8 task_resume_subtask

```
void task_resume_subtask(task_entry, spawn_id)
    PO_DB_ENTRY *task_entry;
    ObjectID    *spawn_id;
```

'task_entry'
Specifies the task which spawned the subtask

'spawn_id'
Specifies the Object ID of the spawned subtask

task_resume_subtask resumes execution of a spawned task by clearing its **suspended** flag.

3.9 task_delete_subtask

```
void task_delete_subtask(task_entry, spawn_id)
    PO_DB_ENTRY *task_entry;
    ObjectID    *spawn_id;
```

'task_entry'
Specifies the task which spawned the subtask

'spawn_id'
Specifies the Object ID of the spawned subtask

task_delete_subtask deletes a task spawned with **task_spawn_subtask**, and zero's out the passed **spawn_id**.

3.10 task_stop_subframe

```
void task_stop_subframe(task_entry, spawn_id)
    PO_DB_ENTRY *task_entry;
    ObjectID    *spawn_id;
```

'task_entry'
Specifies the task which spawned the subframe

'spawn_id'
Specifies the Object ID of the spawned subframe

task_stop_subframe stops a spawned frame by clearing out its unit field.

3.11 task_get_db

```
PO_DATABASE *task_get_db(task_entry)
    PO_DB_ENTRY *task_entry;
```

'task_entry'
Specifies the task

task_get_db gets the PO database in which a task resides (for use by 'fsm2ch' code generator).

3.12 task_get_unit

```
PO_DB_ENTRY *task_get_unit(task_entry)
    PO_DB_ENTRY *task_entry;
```

'task_entry'
Specifies the task

task_get_unit gets the unit which is executing a task (for use by 'fsm2ch' code generator). Note that this function will not work for enabling tasks, since multiple units can be executing a single enabling task (NULL will be returned).

3.13 task_complete

```
int32 task_complete(task_entry, spawn_id)
    PO_DB_ENTRY *task_entry;
    ObjectID    *spawn_id;
```

'task_entry'
Specifies the task which spawned the subtask

'spawn_id'
Specifies the Object ID of the task being queried

task_complete calls the registered **predicate** function for a spawned subtask, and returns the result.

3.14 task_pop_ownframe

```
void task_pop_ownframe(task_entry)
    PO_DB_ENTRY *task_entry;
```

'task_entry'
Specifies a task in the frame

task_pop_ownframe unassigns the task frame in which the passed task resides.

3.15 task_class_init

```
void task_class_init(parent_class)
    CLASS_PTR parent_class;
```

'parent_class'
Class of the parent (declared with **class_declare_class**, such as C2obj class)

task_class_init creates a handle for attaching task class information to entries. The **parent_class** is one created with **class_declare_class**.

3.16 task_create

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

'entry' Specifies the task's entry in the PO database

'db' Specifies the PO database

task_create creates the task class information for a entry and attaches it to the entry's libclass user data.

3.17 task_destroy

```
void task_destroy(entry)
    PO_DB_ENTRY *entry;
```

'entry' Specifies the task entry

task_destroy frees the task class information for a entry.

3.18 task_exec

```
void task_exec(entry, vehicle_id)
    PO_DB_ENTRY *entry;
    int32       vehicle_id;
```

'entry' Specifies the task entry

'vehicle_id'
Specifies the ID of the vehicle which is to execute the task

task_exec executes the task associated with the passed entry in the context of the passed vehicle. Note that it is not safe to execute the same task entry on more than one vehicle, since their state may diverge.

3.19 task_predicate

```
int32 task_predicate(entry, vehicle_id, current_opaque_task_frame,
                     next_task_frame, num_executing_tasks, executing_tasks)
    PO_DB_ENTRY *entry;
    int32        vehicle_id;
    PO_DB_ENTRY *current_opaque_task_frame;
    PO_DB_ENTRY *next_task_frame;
    int32        num_executing_tasks;
    PO_DB_ENTRY  *executing_tasks[];
```

'entry' Specifies the task entry

'vehicle_id'
Specifies the ID of the vehicle which is to execute the predicate

'current_opaque_task_frame'
Specifies the exectuing opaque task frame in current context

'next_task_frame'
Specifies the frame containing the enabling task

'num_exectuing_tasks'
Specifies the number of executing tasks in current context

'exectuing_tasks'
Specifies the list of exectuing tasks in current context

task_predicate runs the predicate function associated with the task in the context of the passed vehicle, and returns its result. If no predicate exists for the task, 0 will be returned.

3.20 task_stop

```
void task_stop(entry, vehicle_id)
    PO_DB_ENTRY *entry;
    int32 vehicle_id;
```

'entry' Specifies the task entry

'vehicle_id'
Specifies the ID of the vehicle which is running the task

task_stop stops the execution of the task in the context of the passed vehicle. This typically is done when a task frame is popped.

3.21 task_suspend

```
void task_suspend(entry, vehicle_id)
    PO_DB_ENTRY *entry;
    int32 vehicle_id;
```

'entry' Specifies the task entry

'vehicle_id'
Specifies the ID of the vehicle which is running the task

task_suspend suspends the execution of the task in the context of the passed vehicle. This typically is done when a task frame pushed on top of another frame. The next time task_exec is called for this task/vehicle, the task will be restarted.

3.22 task_stop_if_suspended

```
void task_stop_if_suspended(entry)
    PO_DB_ENTRY *entry;
```

'entry' Specifies the task

task_stop_if_suspended sets a flag in the task indicating that if the task management software concludes that the task has been suspended, then the task should be stopped (see tmgr_tick.c:stop_taskfram for details).

3.23 task_params

```
void task_params(entry)
    PO_DB_ENTRY *entry;
```

'entry' Specifies the task

task_params triggers a params event for this task prior to the next tick, and marks the task as RUNNING. This is used by the task management software when task parameters are overridden in a transparent frame.

3.24 task_migrated

```
void task_migrated(entry)
    PO_DB_ENTRY *entry;
```

'entry' Specifies the task entry

task_migrated clears local task execution state. This is typically done when a vehicle migrates to another simulation host.

3.25 task_get_sequence

```
int32 task_get_sequence(entry)
    PO_DB_ENTRY *entry;
```

'entry' Specifies the task entry

task_get_sequence returns the sequence number associated with the passed task. The sequence numbers of tasks imply an order of execution, with lower numbers executing before higher ones.

LibTaskedit

Table of Contents

1 Overview 1

2 Usage 3

2.1 Building Libtaskedit 3
2.2 Linking with Libtaskedit 3
2.3 Examples 4

3 Functions 7

3.1 taskedit_init 7
3.2 taskedit_init_gui 7
3.3 taskedit_register 8
3.4 taskedit_add_enabling 9
3.5 taskedit_reinit 9
3.6 taskedit_get_editor 10
3.7 taskedit_override 10
3.8 taskedit_start_editor 11
3.9 taskedit_start_etask_editor 11

4 Adding Help 13

1 Overview

LibTaskEdit implements the SAF task frame and task editors. Individual tasks which can be assigned from the user interface register their unique editors with libTaskEdit, which then automatically incorporates them into the user interface. Definition of a task editor requires three things:

- Creating a file which defines a libEditor editor which corresponds to the public PARAMETERS part of the task shared data.
- Registering the name of this data file, as well as other information about the task with libTaskEdit.
- Defining the default initialization to be used in each task frame which is to contain the task.

See (undefined) [Examples], page (undefined).

2 Usage

The software library 'libtaskedit.a' should be built and installed in the directory '/common/lib/'. You will also need the header file 'libtaskedit.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 libtaskedit source directory. If these files are already built, you can skip the section on building libtaskedit.

2.1 Building Libtaskedit

The libtaskedit source files are found in the directory '/common/libsrc/libtaskedit'. 'RCS' format versions of the files can be found in '/nfs/common_src/libsrc/libtaskedit'.

If the directory 'common/libsrc/libtaskedit' 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/libtaskedit
# co RCS/*,v
# make install
```

This should compile the library 'libtaskedit.a' and install it and the header file 'libtaskedit.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. libtaskedit should compile without errors on the following platforms:

- Mips
- SGI Indigo
- Sun Sparc

2.2 Linking with Libtaskedit

Libtaskedit can be linked into an application program with the following link time flags: 'ld [source .o files] -L/common/lib -ltaskedit [other libraries]'. If your compiler does not

support '-L' syntax, you can use the archive explicitly: '`ld [source .o files] /common/lib/libtaskedit.a`'.

Libtaskedit depends directly on the following libraries: libsafgui, libtactmap, libcoordinates, libsensitive, libcallback, libpo, libeditor, libselect, libreader, libgraphics, and libprivilege.

2.3 Examples

The following are code fragments from an early version of the task libubingofuel (Unit Bingo Fuel), which demonstrate how to define an editable task.

The parameters are defined in the standard fashion:

```
typedef struct ubfuel_parameters
{
    ObjectID            base;
    uint16              _padding;
    float64             speed;
    float64             altitude;
} UBINGOFUEL_PARAMETERS;
```

The data file ('`ubfuel.rdr`') defines the editor for this data structure:

```
((name   "Bingo Fuel")
 (struct (base int16 3)
 (padding 16)
 (speed float64)
 (altitude float64)
 )
 (editor ("Refuel Point" OBJECT base NOCANCEL
 objectClassPoint objectClassText)
 ("Speed" SPEED speed)
 ("Altitude" ALTITUDE altitude)
 )
 (initial (base FORCE)
  (speed CONSTANT 250.0)
  (altitude CONSTANT 3000.0)
  )
 (render REVERT)
 )
```

Note that for tasks which are not enabling tasks, most of the `initial` parameters will never be

used, since they will come in the task frame definition.

Also note, that the **render** list for the task should read **(render REVERT)** in all cases.

The intialization methods are declared again for each task frame (note that initializations which do not differ between task frames need not be repeated):

```
(...
 ("Sweep"
  ...
  (SM_UBingoFuel BOTH
 (base FORCE "You must specify a refuel point")
 (speed CONSTANT 164.622)
 (altitude CONSTANT 6096.0)
 )
  ...
 )
 ...
)
```

Finally, to register the task with libTaskEdit, the library calls the following from its global **<task>_init** function:

```
void uflwrte_init()
{
    ...

    taskedit_register(SM_UBingoFuel, "ubfuel.rdr",
                      sizeof(UBINGOFUEL_PARAMETERS),
                      ubfuel_init_task_state);
}
```

The initialization function for libubfuel is defined as follows:

```
void ubfuel_init_task_state(task, state)
    TaskClass      *task;
    TaskStateClass *state;
{
    state->size = sizeof(UBINGOFUEL_STATE);
    state->refcount = UBINGOFUEL_STATE_REFERENCES;
    bzero(state->data, sizeof(UBINGOFUEL_STATE));
}
```

The **task** is provided in case task parameters are needed in order to initialize the state (generally

not the case). The refcount should be set to the number of `ObjectID` references which appear in the state structure.

3 Functions

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

3.1 taskedit_init

```
void taskedit_init()
```

`taskedit_init` initializes libtaskedit. Call this before any other libtaskedit function.

3.2 taskedit_init_gui

```
int32 taskedit_init_gui(taskframes_file, data_path, reader_flags,
                        dialog_parent,
                        gui, tactmap, tcc, map_erase_gc,
                        sensitive, select, refresh_event, db)
    char               *taskframes_file;
    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;
    SELECT_TOOL_PTR     select;
    CALLBACK_EVENT_PTR  refresh_event;
    PO_DATABASE        *db;
```

'taskframes_file'
: Specifies the name of the file with taskframe definitions

'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 shell which 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

'select' Specifies the selection tool

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

'db' Specifies the persistent object database

taskedit_init_gui creates the task frame editor. Individual task editors are also created at this time, so be sure to initialize the task libraries before calling this function (so they can register their editors). The **data_path** and **reader_flags** are used when reading the individual task editor definitions. Note that this registers an association with libgraphics, so it must be called *after* **grph_create_editors**. A non-zero return value indicates an error occurred.

3.3 taskedit_register

```
void taskedit_register(saf_model, data_file, sizeof_struct, init_fcn)
    uint32                 saf_model;
    char                  *data_file;
    uint32                 sizeof_struct;
    TASKEDIT_INIT_FUNCTION init_fcn;
```

'saf_model'
Specifies the SAF model of the task

'data_file'
Specifies the name of the data file where the task's editor is defined

'sizeof_struct'
Specifies the size of the task parameter data structure

'init_fcn'
Specifies the function to call to initialize task state

taskedit_register registers a task editor with libtaskedit. The caller provides the name of a data file, which libtaskedit will read when (if) the GUI is initialized. The **sizeof_struct** specifies the size of the public parameters of the task (**<task>_PARAMETERS**).

The **init_fcn** is responsible for initializing the state of the task. The function must set the **size**, **refcount**, **references**, and **data** portions of the state. The **task** will already be filled in by libtaskedit, and should not be modified; it is supplied only to provide context for initialization.

Note that the **init_fcn** may be called more than once (such as when a mission is reused), so the **refcount** field of the task should be set to an explicit value (number of parameter references plus number of state references), rather than being incremented by the number of state variables. For example,

```
statevars->refcount = UFLWRTE_STATE_REFERENCES;
```

is an appropriate expression, but

```
statevars->refcount += UFLWRTE_STATE_REFERENCES;
```

is not.

The **init_fcn** should be prototyped as follows:

```
void init_function(task, state)
    TaskClass      *task;
    TaskStateClass *state;
```

3.4 taskedit_add_enabling

```
void taskedit_add_enabling(saf_model)
    uint32 saf_model;
```

'saf_model'
: Specifies the SAF model of the enabling task

taskedit_add_enabling notifies libtaskedit that a task should be treated as an enabling task. The editor registered with this task in a previous call to **taskedit_register** will be added to this list of editors available in when the task editor is run in enabling task mode.

3.5 taskedit_reinit

```
void taskedit_reinit(db, taskframe)
    PO_DATABASE *db;
    PO_DB_ENTRY *taskframe;
```

'db' Specifies the persistent object database

'taskframe'
Specifies the task frame to reinitialize

taskedit_reinit calls the registered initialization function for all the tasks in a task frame. This is provided to allow reuse of missions (the tasks in a mission start out initialized, but once they have been used, they may not be in an acceptable state for reuse).

3.6 taskedit_get_editor

```
TASKEDIT_GUI_PTR taskedit_get_editor(gui)
    SGUI_PTR gui;
```

'gui' Specifies the SAF GUI

taskedit_get_editor finds a taskframe editor created for the passed GUI. Returns NULL if none exists.

3.7 taskedit_override

```
void taskedit_override(editor, unit_id)
    TASKEDIT_GUI_PTR editor;
    ObjectID         *unit_id;
```

'editor' Specifies the task frame editor

'unit_id' Specifies the unit whos tasks are to be modified

taskedit_override invokes the task editor in override mode. The editor allows modification of those tasks which exist in current frames being executed by the passed unit. Any modifications are placed in a transparent frame. The unit is monitored for changes in its task frames, which might cause the set of modifiable tasks to change.

3.8 taskedit_start_editor

```
void taskedit_start_editor(editor, destroyWhenDone, unit_type,
                           frame_id)
    TASKEDIT_GUI_PTR editor;
    uint32           destroyWhenDone;
    ObjectType       unit_type;
    ObjectID        *frame_id;
```

'editor' Specifies the task frame editor

'destroyWhenDone'
Specifies the value of the created frames' destroyWhenDone field

'unit_type'
Specifies the type of unit (if non-zero, only allowed task frames for that unit type will be sensitive)

'frame_id'
Specifies the address into which the frame should be placed

taskedit_start_editor invokes the task editor in normal mode. When the editor is exited, the ObjectID of the created frame will be placed into the address passed as frame_id. Thus, this must be non-volatile memory space.

3.9 taskedit_start_etask_editor

```
void taskedit_start_etask_editor(editor, task_id)
    TASKEDIT_GUI_PTR editor;
    ObjectID        *task_id;
```

'editor' Specifies the task frame editor

'task_id' Specifies the address into which the task should be placed

taskedit_start_etask_editor invokes the task editor in enabling-task mode. When the editor is exited, the ObjectID of the created task will be placed into the address passed as task_id. Thus, this must be non-volatile memory space.

4 Adding Help

It may be the case that some of the fields which make up an editor will need explanation at run time. Whereas forced choices require that a special help message be provided, most editable fields will just get the default help message provided by libeditor.

In some cases, this is acceptable. For example, the altitude parameter of a follow route mission is self explanatory. However, in cases where additional help is needed, there is a facility to add this help through the X resource database (see section 'X Resource Definitions' in LibEditor Programmer's Manual).

To add context-specific help, a task library should do the following:

1. Create an X resource file ('<taskname>.xrdb') in the task library with the special help in it. The resource lines should read:

   .SAFGUI..Editor.*.*Task Name*.*.*Field Name*.*.Help: *Help String*

   For example, to add a description to the Radar Orientation Azimuth field of the Targeting task editor, the resource might appear as follows:

   ```
   *.SAFGUI.*.Editor.*.Targeting.*.Radar Orientation Azimiuth.*.Help: \
   Specifies the desired azimuth of the center of the radar \
   volume (0 is interpreted as down the Y-axis of the vehicle \
   and is positive counter-clockwise).
   ```

2. Add the '.xrdb' file to the READERS list in the library 'Makefile'.
3. Add a line to the top level resource file 'src/ModSAF/ModSAF' to include this new resource file.

LibTaskframe

Table of Contents

1 Overview 1

2 Usage 3

2.1 Building Libtaskframe 3
2.2 Linking with Libtaskframe 3

3 Functions 5

3.1 taskfr_init 5
3.2 taskfr_class_init 5
3.3 taskfr_create 5
3.4 taskfr_destroy 6
3.5 taskfr_changed 6
3.6 taskfr_get_next 6
3.7 taskfr_check_enabling 6
3.8 taskfr_get_tasks 7
3.9 taskfr_subsequent_task 7
3.10 taskfr_primary_task 8
3.11 taskfr_task_search 8
3.12 taskfr_get_background_task 8
3.13 taskfr_set_background_task 9
3.14 taskfr_update_background_task 10
3.15 taskfr_get_label 10
3.16 taskfr_delete_all 11

4 Events 13

4.1 taskfr_assignment_callback 13

1 Overview

Libtaskframe provides simulation support for task frames. It provides functions which may be called to determine what task frames come after a given frame (in the context of a mission), what tasks belong to a task frame, and whether the enabling tasks in a task frame indicate it should be triggered.

The call to test enabling tasks (`taskfr_check_enabling`) uses the following algorithm:

1. Run the predicate function for each enabling task, to get its True/False value.
2. Get the first item off the logic stack.
3. Act on the current item:

   Task number
   : Push the value True/False value for that task from step 1 onto the evaluation stack.

   NOT operator
   : Complement the value on the top of the evaluation stack.

   AND, OR operators
   : Pop the top off the evaluation stack and logically combine it with the value on the new top of the evaluation stack.

   STOP operator
   : Return the value on the top of the evaluation stack.
4. Get the next value off the logic stack and go to step 3.

The check-enabling algorithm returns False if the stack underflows.

The two query functions (to get subsequent frames, and to get tasks) operate by querying the persistent object database. Although it is possible for the task frame to keep this information up to date by monitoring libPO events, it was implemented with queries for simplicity. This implementation should be monitored, and should be changed if the cost of the queries is too high.

2 Usage

The software library 'libtaskframe.a' should be built and installed in the directory '/common/lib/'. You will also need the header file 'libtaskframe.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 libtaskframe source directory. If these files are already built, you can skip the section on building libtaskframe.

2.1 Building Libtaskframe

The libtaskframe source files are found in the directory '/common/libsrc/libtaskframe'. 'RCS' format versions of the files can be found in '/nfs/common_src/libsrc/libtaskframe'.

If the directory 'common/libsrc/libtaskframe' 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/libtaskframe
# co RCS/*,v
# make install
```

This should compile the library 'libtaskframe.a' and install it and the header file 'libtaskframe.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. libtaskframe should compile without errors on the following platforms:

- Mips
- SGI Indigo
- Sun Sparc

2.2 Linking with Libtaskframe

Libtaskframe can be linked into an application program with the following link time flags: 'ld [source .o files] -L/common/lib -ltaskframe [other libraries]'. If your compiler does not

support '-L' syntax, you can use the archive explicitly: '`ld [source .o files] /common/lib/libtaskframe.a`'.

Libtaskframe depends on libclass, libpo, and libtask.

3 Functions

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

3.1 taskfr_init

```
void taskfr_init()
```

taskfr_init initializes libtaskframe. Call this before calling any other libtaskframe functions or any specific taskframe initializations.

3.2 taskfr_class_init

```
void taskfr_class_init(parent_class)
    CLASS_PTR parent_class;
```

'parent_class'
Specifies the parent class (such as C2obj class)

taskfr_class_init creates a handle for attaching taskframe class information to entries. The **parent_class** is one created with **class_declare_class**.

3.3 taskfr_create

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

'entry' Specifies the task frame's entry in the PO database
'db' Specifies the PO database

taskfr_create creates the taskframe class information for a entry and attaches it to the entry's libclass user data. This function will not attach any data if the PO object class is not **objectClassTaskFrame**.

3.4 taskfr_destroy

```
void taskfr_destroy(entry)
    PO_DB_ENTRY *entry;
```

'entry' Specifies the task frame's entry in the PO database

taskfr_destroy frees the taskframe class information for a entry.

3.5 taskfr_changed

```
void taskfr_changed(entry)
    PO_DB_ENTRY *entry;
```

'entry' Specifies the task frame's entry in the PO database

taskfr_changed updates the taskframe class information in response to a libpo object_changed event.

3.6 taskfr_get_next

```
int32 taskfr_get_next(entry, subsequent)
    PO_DB_ENTRY *entry;
    PO_DB_ENTRY *subsequent[];
```

'entry' Specifies the task frame

'subsequent'
Returns a list of subsequent frames

taskfr_get_next returns a list of task frames which come after this one. The return value indicates how many taskframes were returned.

3.7 taskfr_check_enabling

```
int32 taskfr_check_enabling(entry, vehicle_id,
                            current_opaque_task_frame,
                            num_executing_tasks, executing_tasks)
    PO_DB_ENTRY *entry;
    int32        vehicle_id;
    PO_DB_ENTRY *current_opaque_task_frame;
    int32        num_executing_tasks;
    PO_DB_ENTRY *executing_tasks[];
```

'entry' Specifies the task frame

'vehicle_id'
Specifies the vehicle execution context

'current_opaque_task_frame'
Specifies the exectuing opaque task frame in current context

'num_exectuing_tasks'
Specifies the number of executing tasks in current context

'exectuing_tasks'
Specifies the list of exectuing tasks in current context

taskfr_check_enabling returns a TRUE or FALSE value indicating whether the enabling tasks of the passed task frame indicate that it should be pushed.

3.8 taskfr_get_tasks

```
int32 taskfr_get_tasks(entry, tasks)
    PO_DB_ENTRY *entry;
    PO_DB_ENTRY *tasks[];
```

'entry' Specifies the task frame

'tasks' Returns a list of tasks in the task frame

taskfr_get_tasks returns a list of tasks in the passed frame. The return value indicates how many tasks were returned.

3.9 taskfr_subsequent_task

```
PO_DB_ENTRY *taskfr_subsequent_task(entry, model)
```

```
PO_DB_ENTRY *entry;
uint32      model;
```

'entry' Specifies the task frame

'model' Specifies the model number of the desired task

taskfr_subsequent_task finds the task with the specified model in one of the frames subsequent to the passed frame.

3.10 taskfr_primary_task

```
PO_DB_ENTRY *taskfr_primary_task(entry)
    PO_DB_ENTRY *entry;
```

'entry' Specifies the task frame

taskfr_primary_task finds the primary task of a frame (if no primary task can be found, returns NULL).

3.11 taskfr_task_search

```
PO_DB_ENTRY *taskfr_task_search(entry, model)
    PO_DB_ENTRY *entry;
    uint32      model;
```

'entry' Specifies the task frame

'model' Specifies the model number of the desired task

taskfr_task_search finds the task with the specified model in the passed frame. This is much more efficient than finding the task with po_query_for_current_objects.

3.12 taskfr_get_background_task

```
PO_DB_ENTRY *taskfr_get_background_task(db, unit_entry, task_model)
    PO_DATABASE *db;
```

```
PO_DB_ENTRY *unit_entry;
uint32      task_model;
```

'db' Specifies the PO database

'unit_entry'
Specifies the unit which is to execute the task

'task_model'
Specifies the SAF model number of the task

taskfr_get_background_task searches the background frame of the passed unit for a task with the specified model.

3.13 taskfr_set_background_task

```
PO_DB_ENTRY *taskfr_set_background_task(db, unit_entry, task_model,
                                        param_size, initial_params,
                                        state_size, initial_state)
    PO_DATABASE *db;
    PO_DB_ENTRY *unit_entry;
    uint32      task_model;
    uint32      param_size;
    ADDRESS     initial_params;
    uint32      state_size;
    ADDRESS     initial_state;
```

'db' Specifies the PO database

'unit_entry'
Specifies the unit which is to execute the task

'task_model'
Specifies the SAF model number of the task

'param_size'
Specifies the size of the task parameters

'initial_params'
Specifies initial values for the task parameters (or NULL)

'state_size'
Specifies the size of the task state

'initial_state'
Specifies initial values for the task state (or NULL)

taskfr_set_background_task ensures that a task of the specified **model** appears in the specified unit's background frame. This can be used at initialization to add default behavior (such as task arbitration) to individual-level units. If a new task must be created, it's parameters and state are initialized with the passed values (**NULL** pointers indicate that the corresponding data should be zeroed). The return value is the existing, or created, task.

Note that if a task of the corresponding model is found (such as would happen after a vehicle migration), the task is not changed.

3.14 taskfr_update_background_task

```
void taskfr_update_background_task(db, unit_entry, task_model,
                                   new_parameters)
    PO_DATABASE *db;
    PO_DB_ENTRY *unit_entry;
    uint32       task_model;
    ADDRESS      new_parameters;
```

'db' Specifies the PO database

'unit_entry'
Specifies the unit executing the background task

'task_model'
Specifies the SAF model of the background task

'new_parameters'
Specifies the new parameters desired

taskfr_update_background_task attempts to find a task of the specified **model** in the specified unit's background frame, and change its parameters to the new values.

3.15 taskfr_get_label

```
char *taskfr_get_label(db, entry)
    PO_DATABASE *db;
    PO_DB_ENTRY *entry;
```

'db' Specifies the PO database

'entry' Specifies the task frame

taskfr_get_label finds a label associated with a task frame. It attempts to find the label associated with the first object referenced by the primary task of the frame. If none can be found, the empty string "" is returned.

3.16 taskfr_delete_all

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

'db' Specifies the PO database

'entry' Specifies the task frame

taskfr_delete_all deletes all the tasks, their states and the taskframe itself.

4 Events

The following sections describe each event provided by libtaskframe.

4.1 taskfr_assignment_callback

```
CALLBACK_EVENT_PTR taskfr_assignment_callback;
```

The **taskfr_assignment_callback** event fires in response to the unit field of a task frame being modified. Note that this callback fires outside the the PO **object_changed** thread, so changes to the PO database in handlers are perfectly safe.

The handler should be prototyped as follows:

```
void handler(db, taskframe)
    PO_DATABASE *db;
    PO_DB_ENTRY *taskframe;
```

LibTaskMgr

Table of Contents

1 Overview 1

2 Usage 3

2.1 Building Libtaskmgr 3
2.2 Linking with Libtaskmgr 3

3 Functions 5

3.1 taskmgr_init 5
3.2 taskmgr_register_reentrant_task 5
3.3 taskmgr_class_init 5
3.4 taskmgr_create 6
3.5 taskmgr_destroy 7
3.6 taskmgr_tick 7
3.7 taskmgr_task_done 7
3.8 taskmgr_get_task 8

1 Overview

The task manager is responsible for running all the tasks in the current frames for a SAF vehicle. The algorithm used is as follows:

1. Get a list of roles which this vehicle is playing (such as company commander, platoon leader, etc.). One of these roles is always that of the vehicle itself.
2. For each of those roles, determine if any of the current task frames have been unassigned. Remove those task frames from the stack.
3. For each of those roles, determine if the enabling tasks of any subsequent frames indicate that those frames should be executed. If so, start one new frame. The frames immediately after the topmost frame on the task frame stack, and those immediately after the topmost *opaque* frame on the stack are tested. If more than one frame is enabled simultaneously, the user is notified, and one is chosen arbitrarily with opaque frames chosen over transparent frames.
4. For each role, traverse the current frames (including the background frame and all frames on the task frame stack down to the first opaque frame) and collect a list of tasks to execute. If a task of a certain class is encountered more than once, only the first occurence will be kept. *Currently no classification is done, and only exact SAF model redundancy will be detected.*
5. Sort the execution list such that the **before** and **after** restraints of each task are not violated (see section 'task_register' in LibTask Programmer's Manual).
6. Compare the resulting list of tasks, and determine if any tasks which were executed last tick are not in the new list. Of those, suspend those which are still running.
7. Execute the tasks in order.

2 Usage

The software library '`libtaskmgr.a`' should be built and installed in the directory '`/common/lib/`'. You will also need the header file '`libtaskmgr.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 libtaskmgr source directory. If these files are already built, you can skip the section on building libtaskmgr.

2.1 Building Libtaskmgr

The libtaskmgr source files are found in the directory '`/common/libsrc/libtaskmgr`'. '`RCS`' format versions of the files can be found in '`/nfs/common_src/libsrc/libtaskmgr`'.

If the directory '`common/libsrc/libtaskmgr`' 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/libtaskmgr
# co RCS/*,v
# make install
```

This should compile the library '`libtaskmgr.a`' and install it and the header file '`libtaskmgr.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. libtaskmgr should compile without errors on the following platforms:

- Mips
- SGI Indigo
- Sun Sparc

2.2 Linking with Libtaskmgr

Libtaskmgr can be linked into an application program with the following link time flags: '`ld [source .o files] -L/common/lib -ltaskmgr [other libraries]`'. If your compiler does not

support '-L' syntax, you can use the archive explicitly: '`ld [source .o files] /common/lib/libtaskmgr.a`'.

Libtaskmgr depends on libclass, libparmgr, libpo, librdrconst, libreader, libtask, libtaskframe, and libvtab.

3 Functions

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

3.1 taskmgr_init

```
void taskmgr_init()
```

taskmgr_init initializes libtaskmgr. Call this before any other libtaskmgr function.

3.2 taskmgr_register_reentrant_task

```
void taskmgr_register_reentrant_task(saf_model)
    uint32 saf_model;
```

'saf_model'
Specifies the SAF model number of the task

taskmgr_register_reentrant_task informs the taskmgr that the specified task was written such that it can be executed multiple times within the same vehicle. This is a requirement for persistent unit-level background tasks, since they may be executing on multiple levels of the hierarchy simultaneously.

In this context, reentrant implies that, like an enabling task, the task has no per-instantiation private state variable. Private variables which provide general system context (such as a PO_DATABASE) are acceptable, but local copies of information held in the shared TaskState structure are not.

3.3 taskmgr_class_init

```
void taskmgr_class_init(parent_class, acceptance_callback)
    CLASS_PTR                parent_class;
    TASKMGR_ACCEPTANCE_TEST acceptance_callback;
```

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

'acceptance_callback'
Specifies function to call to determine if a unit accepts an assigned order

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

The acceptance_callback should be declared as follows:

```
int32 acceptance_test(vehicle_id, taskframe)
    int32        vehicle_id;
    PO_DB_ENTRY *taskframe;
```

A return value of TRUE indicates that the unit should perform the assigned taskframe; FALSE indicates that the unit should ignore the assigned taskframe. This allows an application to model radio message loss, insubordination, etc. Note that this is called in the thread of a LibPO object changed callback (where the object which changed is the passed task frame). Thus to later accept a frame which was originally rejected, an application should modify the taskframe to trigger a new invokation of this callback.

A NULL acceptance_callback indicates that all orders should be accepted.

3.4 taskmgr_create

```
void taskmgr_create(vehicle_id, params, db)
    int32                    vehicle_id;
    TASKMGR_PARAMETRIC_DATA *params;
    PO_DATABASE             *db;
```

'vehicle_id'
Specifies the vehicle ID

'params' Specifies initial parameter values

'db' Specifies the persistent object database in which tasks and task frames can be found

taskmgr_create creates the taskmgr class information for a vehicle and attaches it to the vehicle's block of libclass user data.

3.5 taskmgr_destroy

```
void taskmgr_destroy(vehicle_id)
    int32 vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

taskmgr_destroy frees the taskmgr class information for a vehicle. This should be called before freeing the class user data with class_free_user_data.

3.6 taskmgr_tick

```
void taskmgr_tick(vehicle_id)
    int32 vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

taskmgr_tick executes the algorithm described in the overview of this document (see Chapter 1 [Overview], page 1).

3.7 taskmgr_task_done

```
PO_DB_ENTRY *taskmgr_task_done(vehicle_id, model)
    int32 vehicle_id;
    int32 model;
```

'vehicle_id'
Specifies the vehicle ID

'model' Model ID of a task to query for.

taskmgr_task_done checks list of currently executed tasks and returns a pointer to the task if the model has already been executed during this iteration of the taskmanager.

3.8 taskmgr_get_task

```
PO_DB_ENTRY *taskmgr_get_task(vehicle_id, model)
    int32 vehicle_id;
    int32 model;
```

'vehicle_id'
: Specifies the vehicle ID

'model'
: Model ID of a task to query for.

taskmgr_get_task searches the list of tasks currently being executed by this vehicle for one matching the passed model, and returns it. If non-NULL, the returned object is guaranteed to be of objectClassTask.

LibTaskPri

Table of Contents

1 Overview 1

2 Logic Meta-Language 3

2.1 Examples 4

3 Functions 5

3.1 taskpri_init 5
3.2 taskpri_register_for_group 5
3.3 taskpri_class_init 6
3.4 taskpri_create 6
3.5 taskpri_destroy 6
3.6 taskpri_select 7
3.7 taskpri_enabled 7

1 Overview

LibTaskPri provides a task management service to support mutual arbitration (see section 'Arbitration' in ModSAF Programmer's Guide). Tasks register the criteria under which they need to control a critical resource (such as output to an acutator like the hull), and each tick these criteria are resolved against a set of priorities to determine which task gets that resource for that tick.

Criteria for deciding when the resource should be controlled can be very simple (e.g., whenever the task is running) or quite complicated, involving tests of task state, the state of other tasks, etc. One way to deal with this sort of complexity is to use callback functions to make these determinations. However, in many cases this is more than is really needed, and would certainly lead to a lot of very similar code repeated throughout the system.

Instead, libTaskPri defines a simple meta-language in which the logical combinations which make up the criteria are put together in a data structure which is parsed at run time. C macros are provided to give the definition of this structure the feel of a programming language.

The AAFSM code generator (see section 'AAFSM Code Generator' in LibTask Programmer's Manual) is aware of libTaskPri, and provides convenient notation for interfacing to this library. Thus, many tasks which use libTaskPri never explicitly call libTaskPri functions (they are called by generate code, instead).

Each SAF object has an instance of the `TaskPriority` subclass, which has parameters in the following format:

```
(SM_TaskPriority (tasks
                  ("<group name>" <SAF Model> <SAF Model> <SAF Model>...)
                  ("<group name>" <SAF Model> <SAF Model> <SAF Model>...)
                  ...
                  )
 )
```

The **group name** is a character string identifying the resource which needs arbitration. This name is passed to libTaskPri in the registration and predicate functions, to identify the resource at run time.

The **SAF Models** listed express the priority of tasks which may control the resource. The order of these numbers implies a prioritization, with earlier tasks taking higher priority.

When the program is running, the tasks selection process can be viewed in real time by enabling `taskpri` debugging.

2 Logic Meta-Language

The logical expression which is used to determine whether a task wants to control a resource is specified using a data structure, organized as a post-fix (RPN) list of instructions. However, to improve readability and to reduce errors, as set of C macros can be used to construct this data structure more readably.

The macros are as follows:

TASKPRI_END
: Specifies the end of a list of instructions. This should always be the last thing specified.

TASKPRI_AND(expr, expr)
: TRUE if both expressions evaluate to TRUE.

TASKPRI_AND3(expr, expr, expr)
TASKPRI_AND4(expr, expr, expr, expr)
TASKPRI_AND5(expr, expr, expr, expr, expr)
TASKPRI_AND6(expr, expr, expr, expr, expr, expr)
: TRUE if all their expressions evaluate to TRUE.

TASKPRI_OR(expr, expr)
: TRUE if either expression evaluates to TRUE.

TASKPRI_OR3(expr, expr, expr)
TASKPRI_OR4(expr, expr, expr, expr)
TASKPRI_OR5(expr, expr, expr, expr, expr)
TASKPRI_OR6(expr, expr, expr, expr, expr, expr)
: TRUE if any of their expressions evaluate to TRUE.

TASKPRI_NOT(expr)
: TRUE if the expression is FALSE.

TASKPRI_RUNNING
: TRUE when the task is in any "running" state (all but the FSM-generated not_started, endied, suspended).

TASKPRI_IN_STATE(state)
: TRUE when the task is in the specified state.

TASKPRI_SELECTED(task model, group)
: TRUE when the other specified task has been selected to provide output for the specified group.

TASKPRI_FUNCTION(function, argument)
: The return value of the invocation

```
function(vehicle_id, argument, task_entry, state_entry)
    int32        vehicle_id;
    int32        argument;
    PO_DB_ENTRY *task_entry;
    PO_DB_ENTRY *state_entry;
```

See the example (see Section 2.1 [Examples], page 4) for sample usage.

2.1 Examples

Task provides output if in any running state except **idle**, unless in the **needy** state, in which case only if task **SM_Helper** is controlling the "**helper**" group, and the function **ready** with argument 3 returns **TRUE**:

```
(running && !(state == idle) &&
 (!(state == needy) ||
  ((SM_Helper is selected for "helper") && ready(3))))
```

Is expressed as:

```
TASKPRI_LOGIC stack[] = {
TASKPRI_AND3(TASKPRI_RUNNING,
             TASKPRI_NOT(TASKPRI_IN_STATE(idle)),
             (TASKPRI_OR(TASKPRI_NOT(TASKPRI_IN_STATE(needy)),
                         TASKPRI_AND(TASKPRI_SELECTED(SM_Helper,
                                                      "helper"),
                                     TASKPRI_FUNCTION(ready, 3)
                                     )
                         )
              )
            )
TASKPRI_END };

...

taskpri_register_for_group(SM_mytask,
                           TASKPRI_OFFSET(MYTASK_STATE),
                           reader_get_symbol("my-actuator"),
                           stack);
```

3 Functions

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

3.1 taskpri_init

```
void taskpri_init()
```

taskpri_init initializes libtaskpri. Call this before any other libtaskpri function.

3.2 taskpri_register_for_group

```
void taskpri_register_for_group(model, state_offset, group_symbol, criteria)
    uint32          model;
    uint32          state_offset;
    char           *group_symbol;
    TASKPRI_LOGIC  *criteria;
```

'model' Specifies the model number of the task

'state_offset'
Specifies the address of the state variable state within the task state data structure

'group_symbol'
Specifies the name of the group which is to be controlled, as a libreader symbol (see section 'reader_get_symbol' in LibReader Programmer's Manual)

'criteria'
Specifies the execution criteria (see Chapter 2 [Logic Meta-Language], page 3)

taskpri_register_for_group registers the method by which a task will indicate whether it wants to run a group. The state_offset can be conveniently passed as

```
TASKPRI_OFFSET(<TASKNAME>_STATE)
```

The criteria are presented as a logical collection of primitives in post-fix order. The logic is assumed to be stored in non-volatile (global, static, or dynamically allocated) memory.

3.3 taskpri_class_init

```
void taskpri_class_init(parent_class)
    CLASS_PTR parent_class;
```

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

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

3.4 taskpri_create

```
void taskpri_create(vehicle_id, params, db)
    int32                    vehicle_id;
    TASKPRI_PARAMETRIC_DATA *params;
    PO_DATABASE             *db;
```

'vehicle_id'
Specifies the vehicle ID

'params' Specifies initial parameter values

'db' Specifies the PO database

taskpri_create creates the taskpri class information for a vehicle and attaches it vehicle's block of libclass user data.

3.5 taskpri_destroy

```
void taskpri_destroy(vehicle_id)
    int32 vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

taskpri_destroy frees the taskpri class information for a vehicle. This should be called before freeing the class user data with class_free_user_data.

3.6 taskpri_select

```
void taskpri_select(vehicle_id, n_execs, exec_list)
    int32        vehicle_id;
    int32        n_execs;
    PO_DB_ENTRY *exec_list[];
```

'vehicle_id'
Specifies the vehicle ID

'n_execs' Specifies the number of tasks which will be executed

'exec_list'
Specifies the tasks which will be executed

taskpri_select selects which of the tasks which are going to be run will be allowed to provide output this tick. This is run as part of the task management algorithm.

3.7 taskpri_enabled

```
int32 taskpri_enabled(vehicle_id, group_symbol, task)
    int32        vehicle_id;
    char        *group_symbol;
    PO_DB_ENTRY *task;
```

'vehicle_id'
Specifies the vehicle ID

'group_symbol'
Specifies the name of the group which is to be controlled, as a libreader symbol (see section 'reader_get_symbol' in LibReader Programmer's Manual)

'task' Specifies the task which wants to provide output

taskpri_enabled returns whether the passed task has been selected to provide output this tick for the specified group.

LibTdbtool

Table of Contents

1 Overview 1

2 Functions 3

2.1 tdbtool_init 3

2.2 tdbtool_create_editor 3

1 Overview

Libtdbtool is a tool for calculating intervisibility between points, in a terrain area, in a vehicle area, and between vehicles. It uses libctdb to find the elevation and determine the visibility at each point. The tool also provides a cross section of the terrain. There are four types of intervisibility that can be performed: terrain point-to-point, terrain area, vehicle point-to-point and vehicle area. For terrain point-to-point, the cross section of that line is also calculated, as well as the distance between the starting and ending points, and the coordinates of the start and end points. The line that is drawn on the screen for terrain point-to-point shows green for fully visibile, green-to-black dithered for partially visible and black for blocked. The line also contains tick marks at the same interval as the current grid on the screen. Libtdbtool uses two tactmap objects to display in intervisiblity on the screen. See libtactmap for more information.

2 Functions

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

2.1 tdbtool_init

```
void tdbtool_init()
```

tdbtool_init initializes libtdbtool. Call this function before calling any other libtdbtool functions.

2.2 tdbtool_create_editor

```
int32 tdbtool_create_editor(data_path, reader_flags,
                            gui, tactmap, tcc, ctdb, map_erase_gc,
                            sensitive, refresh_event, db, pvd,
                            refresh_control_callback, refresh_control_arg)
    char                *data_path;
    int32               reader_flags;
    SGUI_PTR            gui;
    TACTMAP_PTR         tactmap;
    COORD_TCC_PTR       tcc;
    CTDB                *ctdb;
    GC                  map_erase_gc;
    SNSTVE_WINDOW_PTR   sensitive;
    CALLBACK_EVENT_PTR  refresh_event;
    PO_DATABASE         *db;
    PVD_GUI_PTR         pvd;
    void                (*refresh_control_callback)();
    ADDRESS             refresh_control_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

'ctdb' Specifies the terrain database

'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

'pvd' Specifies the PVD

'refresh_control_callback'
Specifies the routine to call when the terrain tools is started to turn off refreshing the non-terrain of the tactmap

'refresh_control_arg'
Specifies the refresh terrain event to be turned on and off.

tdbtool_create_editor creates the tdbtool editor. The data file ('**tdbtool.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**. When the tdbtool editor is invoked, the non terrain refresh in the tactmap is turned off. It is turned back on when the tool is exited.

LibTime

Table of Contents

1 Overview 1

2 Examples 3

3 Functions 5

3.1 time_init 5
3.2 time_advance_simulation_clock 5
3.3 time_realtime_clock 5
3.4 time_last_simulation_clock 6
3.5 time_last_simulation_dis_clock 6
3.6 time_last_realtime_clock 6
3.7 time_set_simulation_rate 6
3.8 time_simulation_to_realtime 7
3.9 time_parse_date 7
3.10 time_parse_time 7
3.11 time_tdg_from_seconds 8
3.12 time_tdg_from_date_time 8
3.13 time_tdg_from_unix 8

4 Macros 11

4.1 TIME_MS_TO_DIS_FACTOR 11
4.2 TIME_MS_TO_DIS 11
4.3 TIME_DIS_TO_MS 11

1 Overview

Libtime supports two millisecond resolution clocks: a real time clock which is driven by the system, and a simulation clock which moves at a constant positive factor * real time. A simulation clock in DIS time units is also provided.

The library also provides functions for use with the Persistent Object protocol TimeDateGroup structure, as well as conversion macros to convert between DIS and millisecond time units.

2 Examples

To get the value of the real time clock:

```
uint32 t;
t = time_realtime_clock();
```

To get a recent value of the real time clock, without the expense of reading the system clock:

```
uint32 t;
t = time_last_realtime_clock;
```

Given a known unix time (seconds since 1970), generate the appropriate TimeDateGroup:

```
#include <p_po.h>
TimeDateGroup tdg;
time_tdg_from_seconds(unix_seconds, &tdg);
```

3 Functions

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

3.1 time_init

```
void time_init(max_forward_step)
    uint32 max_forward_step;
```

'max_forward_step'
Specifies maximum reasonable time increment between calls.

time_init initializes libtime. Resets clocks. The **max_forward_step**, if non-zero, specifies the most real time which may pass between successive calls to **time_advance_simulation_clock** or **time_realtime_clock**. For example, passing a fairly large value (such as 1000 ms) will automatically cause time to stop advancing when single stepping with a debugger.

3.2 time_advance_simulation_clock

```
uint32 time_advance_simulation_clock()
```

time_advance_simulation_clock returns the time of the simulation clock in milliseconds.

3.3 time_realtime_clock

```
uint32 time_realtime_clock()
```

time_realtime_clock returns the time of the real time clock in milliseconds.

3.4 time_last_simulation_clock

```
uint32 time_last_simulation_clock
```

time_last_simulation_clock retains the last value returned by **time_advance_simulation_clock**.

3.5 time_last_simulation_dis_clock

```
uint32 time_last_simulation_dis_clock
```

time_last_simulation_dis_clock retains the last value returned by **time_advance_simulation_clock**, expressed in DIS time units. In DIS, the most significant 31 bits of a 32-bit time field contains the time in units of the following duration: 1 hour = 2^31-1 time units. This time implements DIS relative timestamping, and the least significant bit is always 0.

3.6 time_last_realtime_clock

```
uint32 time_last_realtime_clock
```

time_last_realtime_clock retains the last value returned by **time_realtime_clock**.

3.7 time_set_simulation_rate

```
void time_set_simulation_rate(positive_multiplier, limit)
    float64 positive_multiplier;
    uint32  limit;
```

'positive_multiplier'
: Specifies the rate of simulation relative to real time.

'limit'
: Specifies a simulated time to reach before it is time to go back to the rate 1.0 x real time.

time_set_simulation_rate sets the relationship between the simulation clock and real time. If a non-zero value is given as the limit, the simulation rate will switch back to 1.0 when that simulated time is reached.

3.8 time_simulation_to_realtime

```
uint32 time_simulation_to_realtime(simulation_clock)
    uint32 simulation_clock;
```

'simulation_clock'
Specifies a simulation clock time.

time_simulation_to_realtime determines the real time which corresponds to the passed **simulation_clock** time, given the current simulation rate (of course, if the simulation rate changes before the passed **simulation_clock** time is reached, this answer will be incorrect).

3.9 time_parse_date

```
uint32 time_parse_date(date_string)
    char *date_string;
```

'date_string'
Specifies "YYMMDD" string to parse.

time_parse_date given a string of the form "YYMMDD", returns an integer with the value YYMMDD. Returns 0 if the date is not of the correct format.

3.10 time_parse_time

```
uint32 time_parse_time(time_string)
    char *time_string;
```

'time_string'
Specifies "HHMM" string to parse.

Given a string of the form "HHMM", time_parse_time returns an integer with the value HHMM. It returns -1 if the time is not of the correct format.

3.11 time_tdg_from_seconds

```
void time_tdg_from_seconds(seconds, tdg)
    uint32          seconds;
    TimeDateGroup *tdg;
```

'seconds' Specifies a unix time.

'tdg' Returns a protocol time-date-group structure.

Given a "unix" time (seconds since 1970), time_tdg_from_seconds fills in the passed TimeDateGroup structure.

3.12 time_tdg_from_date_time

```
void time_tdg_from_date_time(date, time, tdg)
    uint32          date;
    uint32          time;
    TimeDateGroup *tdg;
```

'date' Specifies a date in YYMMDD format.

'time' Specifies a time in HHMM format.

'tdg' Returns a protocol time-date-group structure.

Given a date in the form YYMMDD and a time in the form HHMM, time_tdg_from_date_time fills in the passed TimeDateGroup structure.

3.13 time_tdg_from_unix

```
void time_tdg_from_unix(tdg)
    TimeDateGroup *tdg;
```

'tdg' Returns a protocol time-date-group structure filled in with current unix time.

time_tdg_from_unix fills in the passed TimeDateGroup structure, based upon the result of a call to unix **time**.

4 Macros

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

4.1 TIME_MS_TO_DIS_FACTOR

```
TIME_MS_TO_DIS_FACTOR
```

TIME_MS_TO_DIS_FACTOR is defined as the number of DIS time units in a millisecond. It is equal to (2^31 - 1) / 60 * 60 * 1000.

4.2 TIME_MS_TO_DIS

```
TIME_MS_TO_DIS(t)
```

't' Specifies a time in milliseconds

TIME_MS_TO_DIS returns the time t, passed in as milliseconds, as DIS time units.

4.3 TIME_DIS_TO_MS

```
TIME_DIS_TO_MS(t)
```

't' Specifies a time in DIS time units

TIME_DIS_TO_MS returns the time t, passed in as DIS time units, as milliseconds.

Table of Contents

1 Overview 1

2 Examples 5

3 Functions 7

3.1 tracked_init 7
3.2 tracked_class_init 7
3.3 tracked_tick 7
3.4 tracked_collision 8
3.5 tracked_damage 9

1 Overview

Libtracked implements an instance of the hull class of components. It provides a low-fidelity model of tracked 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 tracked vehicle are specified in its configuration file as follows:

```
(tracked (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 tracked 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. Libtracked will do the normalization only if it is necessary (for example, if the vehicle is already pointing the right way, no normalization is needed).

Libtracked supports only one instantiation per vehicle (i.e., a vehicle may not have more than one tracked 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 tracked model by changing an exisiting hulls interface function or by adding a completely new function.

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

1. If the change occurs only in the function body, only change the function code in the libtracked library. If the change occurs to the function's argument list, change the function code in both the libtracked 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 libtracked function to the current model would require the following actions:

1. Write the function as part of the libtracked library. The function is written in the code which manages the libtracked class information attached to each vehicle (trk_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 libtracked 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 tracked 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 libtracked source file that contains the code to manage the libtracked class information (trk_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 libtracked, including the format and meaning of its arguments, and the meaning of its return values (if any).

3.1 tracked_init

```
void tracked_init()
```

tracked_init initializes libtracked. Call this before calling any other libtracked functions.

3.2 tracked_class_init

```
void tracked_class_init(parent_class)
    CLASS_PTR parent_class;
```

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

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

3.3 tracked_tick

```
void tracked_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.

tracked_tick ticks the tracked hull dynamics model.

3.4 tracked_collision

```
void tracked_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

tracked_collision tells the tracked 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 tracked_damage

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

'vehicle_id'
: Specifies the vehicle ID

'damage'
: Specifies whether the tracked dynamics should simulate being damaged

tracked_damage tells the tracked hull dynamics model that it is damaged (or not) depending on the boolean value of the **damage** flag.

LibTurrets

Table of Contents

1 Overview 1

2 Examples 5

3 Functions 7

3.1 turrets_init 7
3.2 TURRETS_SET_AZIMUTH 7
3.3 TURRETS_SET_AZIMUTH_SPEED_MODE 7
3.4 TURRETS_SET_AZIMUTH_CURVE_MODE 8

1 Overview

Turrets 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 turret functions is achieved through macros defined by libturrets. These macros invoke 'cmpnt_invoke' with a code number which identifies the function to run. Libcomponents then runs this function for the particular turret mode via a jump table.

The table below shows how the turret component relationships have been currently implemented via the ModSAF library structure.

specific libraries	generic library	architectural library
libgenturret	libturrets	libcomponents

As mentioned above, libturrets requires the services of libcomponents, an architectural library which provides a level of abstraction away from the specific turret component interfaces. When the ModSAF application gets set up to run, the libturrets initialization process directs libcomponents to define a turrets component class. This information enables libcomponents to define a structure to accommodate all of the turret instantiations a simulated object is allowed to have. The libturrets initialization process also tells libcomponents the number of its defined turret interface functions. This enables a simulated object's user data to be allocated enough space to hold the address of each of the interface interface functions defined in libturrets.

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 libgenturret instantiation.

Since an application will interface to libgenturret through libturrets, a tank's turret control commands (which are performed by libgenturret) are issued via the interface defined by libturrets. Similarly, an application can obtain information about the state of its turret though the libturrets interface. The table below shows the relationship between the specific and generic library for the turrets component.

Instantiations of of the library:	Belong to generic component class:	Have a command interface defined in:
libgenturret	turrets	libturrets

The interface to libturrets is defined in its public header file (libturret.h). Applications interface to the generic turret model primarily through the macros defined in libturrets. These macros map to functions which are invoked on instances of the turrets sub-class (such as the libgenturret component instantiated for a tank).

One interface for controlling a turret is the TURRETS_SET_AZIMUTH macro which maps to a function that sets the desired azimuth (in vehicle coordinates) of the turret, and instructs the turret to move to that azimuth in an optimal way for target tracking. A possible definition for this macro is shown below.

```
#define TURRETS_SET_AZIMUTH(_vid, _cnum, _azim)
{
    TURRETS_INTERFACE _tif;
    _tif.u.set_azimuth.azimuth = _azim;
    cmpnt_invoke(TURRETS_SET_AZIMUTH_FCN, _vid, _cnum, (ADDRESS)&_tif);
}
```

The TURRETS_INTERFACE structure defined in libturret.h is the structure which is passed to any turrets interface function. This structure is a union of structures that each define an argument list for a turrets interface function. An abbreviated example that assumes there are only two interface functions is shown below. Typically there will be more interface functions and therefore more structure definitions in the union. The macros hide this structure from the users of these functions.

```
typedef struct turrets_interface
{
    union
    {
        struct turrets_set_azimuth
        {
            float64 azimuth;
```

```
        } set_azimuth;
         struct turrets_set_azimuth_speed_mode
        {
            float64                 azimuth;
            float64                 speed;
            TURRETS_SLEW_MODE_TYPE mode;
        } set_azimuth_speed_mode;
    } u;
} TURRETS_INTERFACE;
```

Issuing a command to an objects's turret component is done by invoking one of the macros defined in libturrets. These macros identify the specific component function which needs to be called. For example, invoking the TURRETS_SET_AZIMUTH macro will result in the calling of the set_azimuth 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 library (libgenturret). It is the specific function (libgenturret's set_azimuth) 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 (libgenturret) data can be accessed. The requested function can require input (such as an azimuth and a 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 `cmpnt_invoke(TURRETS_SET_AZIMUTH_FCN, _vid, _cnum, (ADDRESS)&_tif);` code segment shown above, TURRETS_SET_AZIMUTH_FCN serves as the function pointer index, `_vid` provides the vehicle id, `_cnum` provides the component number, and `&_tif` provides the address for the function's argument list.

2 Examples

To initialize libgenturret, an instance of the turret class which provides for up to GENERIC_TURRET_MAX_TURRETS turrets per entity:

```
int32 i;
char buf[256];

for (i = 0; i < GENERIC_TURRET_MAX_TURRETS; i++)
{
    (void) sprintf(buf, "turret%d", i);
    generic_turret_user_data_handle[i] =
      class_reserve_user_data(parent_class, buf, generic_turret_print);
}

/* Tell libcomponents we are available. */
cmpnt_define_instance(SM_GenericTurret, GENERIC_TURRET_MAX_TURRETS,
                      generic_turret_user_data_handle,
                      generic_turret_create, generic_turret_destroy,
                      TURRETS_SET_AZIMUTH_FCN, set_azimuth,
                      TURRETS_SET_AZIMUTH_SPEED_MODE_FCN,
                        set_azimuth_speed_mode,
                      TURRETS_SET_AZIMUTH_CURVE_MODE_FCN,
                        set_azimuth_curve_mode);
```

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 libturrets, including the format and meaning of its arguments, and the meaning of its return values (if any).

3.1 turrets_init

```
void turrets_init()
```

turrets_init initializes libturrets. Call this function after **cmpnt_init**, and before any specific turret init functions.

3.2 TURRETS_SET_AZIMUTH

```
void  TURRETS_SET_AZIMUTH(_vid, _cnum, _azim)
    int32       _vid;
    int32       _cnum;
    float64     _azim;
```

'_vid' Specifies the vehicle ID

'_cnum' Specifies the turret component number

'_azim' Specifies the desired azimuth in math radians. 0 is interpreted as pointing out the Y-axis of what the turret is attached to, and rotation is positive counter-clockwise.

TURRETS_SET_AZIMUTH sets the desired azimuth (in vehicle coordinates) of the turret, and instructs the turret to move to that azimuth in an optimal way for target tracking. A turret will typically only be able to move to this azimuth in two ticks or more (this is because DIS has entity articulations with speeds, so at least one tick must be consumed updating the desired slew rate in the entity appearance PDU). To use this to track a moving target, best results will be obtained by specifying a projected azimuth for two-ticks into the future (this is equivalent to "leading" the gun).

3.3 TURRETS_SET_AZIMUTH_SPEED_MODE

```
void TURRETS_SET_AZIMUTH_SPEED_MODE(_vid, _cnum, _azim, _sp, _mode)
    int32                 _vid;
    int32                 _cnum;
    float64               _azim;
    float64               _sp;
    TURRETS_SLEW_MODE_TYPE _mode;
```

'_vid' Specifies the vehicle ID

'_cnum' Specifies the turret component number

'_azim' Specifies the desired azimuth in math radians. 0 is interpreted as pointing out the Y-axis of what the turret is attached to and is positive counter-clockwise.

'_sp' Specifies a speed in radians-per-second

'_mode' Specifies the direction of travel for the turret.

TURRETS_SET_AZIMUTH_SPEED_MODE sets the desired azimuth (in vehicle coordinates) of the turret, the maximum speed (in radians per second) to get to that azimuth, and the way to slew the turret to that azimuth.

The _mode can have the following values:

TURRETS_SLEW_MINIMUM
: Move the turret in the direction that will get to the desired azimuth fastest.

TURRETS_SLEW_MAXIMUM
: Move the turret in the direction that will get to the desired azimuth slowest. If the turret is not fully rotatable, this will default to TURRETS_SLEW_MINIMUM.

TURRETS_SLEW_CLOCKWISE
: Move the turret clockwise to the desired azimuth. If the turret is not fully rotatable, this will default to TURRETS_SLEW_MINIMUM.

TURRETS_SLEW_COUNTERCLOCKWISE
: Move the turret counterclockwise to the desired azimuth. If the turret is not fully rotatable, this will default to TURRETS_SLEW_MINIMUM.

3.4 TURRETS_SET_AZIMUTH_CURVE_MODE

```
void TURRETS_SET_AZIMUTH_CURVE_MODE(_vid, _cnum, _azim, _nent, _ents, _mode)
    int32                 _vid;
    int32                 _cnum;
    float64               _azim;
    int32                 _nent;
```

```
TURRETS_SLEW_CURVE_ENTRY _ents[];
TURRETS_SLEW_MODE        _mode;
```

'_vid' Specifies the vehicle ID

'_cnum' Specifies the turret component number

'_azim' Specifies the desired azimuth in math radians. 0 is interpreted as pointing out the Y-axis of what the turret is attached to and is positive counter-clockwise.

'_nent' Specifies number of entries in the slew curve.

'_ents' Specifies an array of slew curve entries.

'_mode' Specifies the direction of travel for the turret.

TURRETS_SET_AZIMUTH_CURVE_MODE sets the desired azimuth (in vehicle coordinates) of the turret, an array of speeds (in radians per second) to get to that azimuth, and the way to slew the turret to that azimuth. The array of speeds is called a slew curve. A slew curve is a way of describing the slewing process towards an azimuth. The curve is defined by up to TURRETS_MAX_SLEW_CURVE_ENTRIES of TURRETS_SLEW_CURVE_ENTRIES. An entry is specified as follows:

```
typedef struct turrets_slew_curve_entry
{
    float64 range;
    float64 rate;
} TURRETS_SLEW_CURVE_ENTRY;
```

Each entry specifies the slew rate to use if the turret is greater than the specified range away from the desired azimuth. A curve is described by an array of entries, and the range field must be monotonically decreasing in such an array. When choosing a slew rate from an array of TURRETS_SLEW_CURVE_ENTRIES, a turret will choose the slew rate corresponding to the largest range that is less than the range of the current azimuth to the desired azimuth. The chosen rate may be adjusted in a turret dependent manner if the turret only supports discrete slew rates.

The _mode can have the following values:

TURRETS_SLEW_MINIMUM
: Move the turret in the direction that will get to the desired azimuth fastest.

TURRETS_SLEW_MAXIMUM
: Move the turret in the direction that will get to the desired azimuth slowest. If the turret is not fully rotatable, this will default to TURRETS_SLEW_MINIMUM.

TURRETS_SLEW_CLOCKWISE
: Move the turret clockwise to the desired azimuth. If the turret is not fully rotatable,

this will default to **TURRETS_SLEW_MINIMUM**.

TURRETS_SLEW_COUNTERCLOCKWISE

Move the turret counterclockwise to the desired azimuth. If the turret is not fully rotatable, this will default to **TURRETS_SLEW_MINIMUM**.

Libuactcontact

Table of Contents

1 Overview 1
1.1 Task Parameters 2
1.2 Parametric Data 4

2 Functions 5
2.1 uactcontact_init 5
2.2 uactcontact_class_init 5
2.3 uactcontact_create 5
2.4 uactcontact_destroy 6
2.5 uactcontact_init_task_state 6

3 Algorithms 7

1 Overview

LibUActContact is a unit level reactive task that monitors enemy activity and reacts to contact. SM_UActionOnContact constantly checks if the unit is under fire or enemy vehicles are spotted. If either of those requirements are met, SM_UActionOnContact reacts by executing an appropriate response. The reactions that are currently supported are:

'Contact Drill'
: Transparent Frame with a SM_UTargeter task.

'Action Drill'
: Opaque Frame with a SM_UAssault task.

'Occupy Position'
: Opaque Frame with a SM_UPrepOcpyPos and a SM_UTargeter task.

'No Action'
: Goes back to monitoring for enemy contact

The reaction executed depends on the parameter values that were set by the operator in the User Interface.

If a Contact Drill is executed, a transparent taskframe with a SM_UTargeter task (see section 'Libutargeter' in LibUTargeter Programmer's Manual) is pushed. The unit continues to execute the primary task while shooting at the enemy.

If an Action Drill is executed, an opaque taskframe with a SM_UAssault task (see section 'Libuassault' in LibUAssault Programmer's Manual) task is pushed. The SM_UActionOnContact task creates an assault objective at the computed enemy location. This objective is passed to the assault and the SM_UAssault is executed.

If a Occupy Position is executed, an opaque taskframe with a SM_UPrepOcpyPos task (see section 'Libupoccpos' in LibUPrepOcpyPos Programmer's Manual) and a SM_UTargeter task. The SM_UActionOnContact task creates an objective facing the enemy and chooses logical target reference points with the engagement area trp at the enemy location. These parameters are passed to the SM_UPrepOcpyPos task.

If the SM_UActionOnContact reacts to enemy activity by doing nothing it goes back to monitoring for enemy activity.

The SM_UActionOnContact never ends until the taskframes that it resides in is destroyed, but it

does go back to the monitoring state when action is no longer required (meaning there are no more enemies in sight). The SM_UActionOnContact is the *sponsoringTask* for the reactive taskframes listed above. This means the SM_UActionOnContact remains active even when the original frame was suspended. The reaction can then be easily monitored and stopped when necessary. Also, if the situation has changed enough to cause a different type of reaction, SM_UActionOnContact will stop the current reaction and start the appropriate reactive taskframe.

Other reactive taskframes can be added by searching for the string NEW_REACTION and adding the appropriate information.

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).

1.1 Task Parameters

When a SM_UActionOnContact task is created or modified, parameters in the parameter block of the task data structure are referenced. The parameters are described by the following structure:

```
typedef enum uactcontact_action
{
    UACTCONTACT_ACTION_OCCUPY_POSITION,
    UACTCONTACT_ACTION_ACTION_DRILL,
    UACTCONTACT_ACTION_CONTACT_DRILL,
    UACTCONTACT_ACTION_NONE
} UACTCONTACT_ACTION;

typedef struct uactcontact_parameters
{
    float64                 engagement_range;
    float64                 under_fire_enemy_threshold;
    UACTCONTACT_ACTION      under_fire_small_enemy_action;
    UACTCONTACT_ACTION      under_fire_large_enemy_action;
    float64                 not_under_fire_enemy_threshold;
    UACTCONTACT_ACTION      not_under_fire_small_enemy_action;
    UACTCONTACT_ACTION      not_under_fire_large_enemy_action;
    VTARGETER_FIRE_TECHNIQUE fire_technique;
    VASSESS_FIRE_TYPE       fire_type;
} UACTCONTACT_PARAMETERS;
```

'engagement_range'

Specifies the range at which the unit will engage with the enemy

'under_fire_enemy_threshold'

Specifies the threshold of vehicles to decide whether the enemy activity is considered large or small while under fire. While under fire, if there are more vehicles than the threshold, the action taken is specified by the under_fire_large_enemy_action parameter. While under fire, if there are less vehicles than the threshold, the action taken is specified by the under_fire_small_enemy_action parameter. A typical value is 3 vehicles.

'under_fire_small_enemy_action'

Specifies the action to take if the number of vehicles is less than the under_fire_enemy_threshold while under fire. The choices are an Assault, a Contact Drill (UTargeter), an Occupy Position, and no action.

'under_fire_large_enemy_action'

Specifies the action to take if the number of vehicles is more than the under_fire_enemy_threshol while under fire. The choices are are an Assault, a Contact Drill (UTargeter), an Occupy Position, and no action.

'not_under_fire_enemy_threshold'

Specifies the threshold of vehicles to decide whether the enemy activity is considered large or small while not under fire. While not under fire, if there are more vehicles than the threshold, the action taken is specified by the not_under_fire_large_enemy_action parameter. While not under fire, if there are less vehicles than the threshold, the action taken is specified by the not_under_fire_small_enemy_action parameter. A typical value is 3 vehicles.

'not_under_fire_small_enemy_action'

Specifies the action to take if the number of vehicles is less than the not_under_fire_enemy_thresl while not under fire. The choices are an Assault, a Contact Drill (UTargeter), an Occupy Position, and no action.

'not_under_fire_large_enemy_action'

Specifies the action to take if the number of vehicles is more than the not_under_fire_enemy_thre while not under fire. The choices are an Assault, a Contact Drill (UTargeter), an Occupy Position, and no action.

'fire_technique'

Specifies the type of firing method to use. The choices are simultaneous and alternating.

'fire_type'

Specifies the method of firing at the enemy. The three types are "Distributed Fire", "Volley Fire", and "None".

1.2 Parametric Data

The parametric data for `SM_UActionOnContact` is as follows:

'speed' Specifies the speed at which to conduct a reactive assault.

'monitor_period'
Specifies the time to wait before monitoring the enemy activity. This is usually about 5000 milliseconds.

'under_fire_time'
Specifies the time after receiving fire to consider the unit still under fire. If nobody in the unit has been fired upon for this time, the unit is considered not under fire. A typical value is 10000 milliseconds.

'defensible_position_dist'
Specifies the distance in front of the unit leader that the defensible position will be set up in a reactive occupy position.

'defensible_position_width'
Specifies the width of the defensible position for a reactive occupy position.

'trp_sector_depth'
Specifies the depth at which to locate the trps for a reactive occupy position. The depth is the distance out from the platoon leader in the direction of the enemy.

'trp_sector_width'
Specifies the width of the trp locations. The width is the distance between the two trps.

'retreat_speed'
Specifies the speed at which to conduct a withdraw. This speed is generally very fast. Currently withdraw is not supported as a reaction.

2 Functions

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

2.1 uactcontact_init

```
void uactcontact_init()
```

uactcontact_init initializes libuactcontact. Call this before any other libuactcontact function.

2.2 uactcontact_class_init

```
void uactcontact_class_init(parent_class)
    CLASS_PTR parent_class;
```

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

uactcontact_class_init creates a handle for attaching uactcontact class information to vehicles. The **parent_class** will likely be **safobj_class**.

2.3 uactcontact_create

```
void uactcontact_create(vehicle_id, params, po_db, ctdb)
    int32                        vehicle_id;
    UACTCONTACT_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

uactcontact_create creates the uactcontact class information for a vehicle and attaches it vehicle's block of libclass user data.

2.4 uactcontact_destroy

```
void uactcontact_destroy(vehicle_id)
    int vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

uactcontact_destroy frees the uactcontact class information for a vehicle. This should be called before freeing the class user data with class_free_user_data.

2.5 uactcontact_init_task_state

```
void uactcontact_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_UActionOnContact task that is about to be created, uactcontact_init_task_state initializes the model size, and state variables.

3 Algorithms

There are four reactions that can take place as a result of enemy contact. One of three (Not counting "No Action") reactive taskframes (Contact Drill, Assault, and Occupy Position) can get pushed when enemy contact is detected. Once this reactive taskframe gets pushed, the `SM_UActionOnContact` that pushed it becomes the *sponsoringTask*. This means the `SM_UActionOnContact` remains active and can monitor and stop the reactive taskframe.

The `SM_UActionOnContact` will stop the reactive taskframe if the number of enemy vehicles has changed to be out of the range that caused that specific reactive taskframe to be pushed.

For example:

```
range: 0 - 3          reactive t/f pushed:  Contact Drill
range: 3 - infinity   reactive t/f pushed:  Occupy Position

When the enemy is first detected only two vehicles are seen; a Contact
Drill is then pushed.  The unit will continue to execute a contact
drill until the number of enemy vehicles has reached 4.  At that time,
the contact drill will be stopped and the actions on contact will go
back to monitoring state and then immediately spawn an Occupy
Position.

The unit will continue to execute the Occupy Position until the number
of enemy vehicles drops below 2 vehicles.  It will then stop the
occupy position and start a contact drill.
```

As seen from the example, hysteresis is built in to prevent the constant toggling between occupy position and contact drill when the number of enemy vehicles hovers around the threshold (3 vehicles).

Also, if both reactive taskframes were specified to be an occupy position, the actions on contact will continue to run the same taskframe when crossing the threshold. This is done to eliminate unnecessary starting and stopping.

The reactive taskframe that gets pushed by `SM_UActionOnContact` can be overridden by selecting *RESUME SUSPENDED MISSION* during a reaction. This will stop the reaction and continue the mission. Also, the `SM_UActionOnContact` task will not respond to enemy activity until a new situation (more or less vehicles causing a transition to a different range) occurs.

Libuassualt

Table of Contents

1 Overview 1
1.1 Task Parameters 1

2 Functions 3
2.1 uassault_init 3
2.2 uassault_class_init 3
2.3 uassault_create 3
2.4 uassault_destroy 4
2.5 uassault_init_task_state 4

1 Overview

Libuassault implements a unit level task which assaults an objective. There are two ways a **SM_UAssault** can occur. First, a **SM_UAssault** can be planned. When it is planned, the assault objective is entered through the user interface. The other way a **SM_UAssault** can occur is through a reaction to enemy activity. In this case, the assault objective is passed in as the enemy location.

Once the assault objective is known, the route to the objective must be found. If the user entered the route through the user interface, that route is chosen. If not, a route is generated which links the current position to the assault objective.

At this point, the **SM_UAssault** task directs the subordinates to follow the route to the objective using the **SM_UTravel** task (see section 'Libutravel' in LibUTraveling Programmer's Manual). The unit is also shooting at the enemy while traveling toward the objective.

Once the unit reaches the objective or the starting unit strength has dropped below a certain percentage, the unit will conduct a **SM_UPrepOcpyPos** task (see section 'Libupoccpos' in LibUpoccpos Programmer's Manual). The **SM_UAssault** computes a battle position based on the direction from the original position to the assault objective. This computed battle position presumably puts the unit with it's back to the orignal position and front to the enemy location. This battle position is passed to the **SM_UPrepOcpyPos** task which finds defensible positions. Once these positions are secured, the **SM_UAssault** executes a **SM_UOcpyPos** task (see section 'Libuoccpos' in Libuoccpos Programmer's Manual). The **SM_UAssault** task stays in the occupy position state until told otherwise.

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).

1.1 Task Parameters

When a **SM_UAssault** task is created or modified, parameters in the parameter block of the task data structure are referenced. The parameters are described by the following structure:

```
typedef enum uassault_reason
{
    UASSAULT_REASON_PLANNED,
    UASSAULT_REASON_REACTIVE
} UASSAULT_REASON;
```

```
typedef struct uassault_parameters
{
    ObjectID            objective;
    ObjectID            route;
    uint32              padding1;
    float64             speed;
    float64             stop_assault_percent;
    float64             engagement_range;
    UASSAULT_REASON     uassault_reason;
    VASSESS_FIRE_TYPE   fire_type;
} UASSAULT_PARAMETERS;
```

'objective'

Specifies a persistent object which defines the assault objective. This object can be a point object, line object, or a text object.

'route' Specifies a persistent object which defines the optional route to the objective. The route is a line object.

'speed' Specifies the speed at which the unit will be moving during the assault.

'stop_assault_percent'

Specifies the percentage of the unit that must be killed in order to stop the assault.

'engagement_range'

Specifies the range at which the unit will engage with the enemy

'uassault_reason'

Specifies whether the assault was a reaction to enemy activity or a planned task.

'fire_type'

Specifies the method of firing at the enemy. The three types are "Distributed Fire", "Volley Fire", and "None".

2 Functions

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

2.1 uassault_init

```
void uassault_init()
```

uassault_init initializes libuassault. Call this before any other libuassault function.

2.2 uassault_class_init

```
void uassault_class_init(parent_class)
    CLASS_PTR parent_class;
```

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

uassault_class_init creates a handle for attaching uassault class information to vehicles. The **parent_class** will likely be **safobj_class**.

2.3 uassault_create

```
void uassault_create(vehicle_id, params, po_db, ctdb)
    int32                    vehicle_id;
    UASSAULT_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

uassault_create creates the uassault class information for a vehicle and attaches it vehicle's block of libclass user data.

2.4 uassault_destroy

```
void uassault_destroy(vehicle_id)
    int vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

uassault_destroy frees the uassault class information for a vehicle. This should be called before freeing the class user data with **class_free_user_data**.

2.5 uassault_init_task_state

```
void uassault_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_UAssault** task that is about to be created, **uassault_init_task_state** initializes the model size, and state variables.

LibUATAInt

Table of Contents

1 Overview 1
1.1 Task Parameters 1

2 Functions 3
2.1 uataint_init 3
2.2 uataint_class_init 3
2.3 uataint_create 3
2.4 uataint_destroy 4
2.5 uataint_init_task_state 4

1 Overview

Libuataint implements a unit-level task which controls a group (currently only one) of vehicles performing an air-to-air intercept against an enemy. 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).

Libuataint depends on libvtakeoff, libvataint, libpo, libvtab, libclass, libctdb, libaccess, libreader, and libparmgr.

1.1 Task Parameters

When a **SM_UATAInt** task is created or modified, parameters in the parameter block of the task data structure are referenced. The parameters are represented in the task data structure as follows:

```
typedef struct uataint_parameters
{
    VehicleID                target_id;
    uint16                   padding1;
    float64                  target_bearing;  /* radians */
    float64                  target_range;    /* meters */
    VATAINT_FIRE_PERMISSION  fire_permission;
    int32                    weapon_count;
    VATAINT_WEAPONS_ENABLED  weapons_enabled[VATAINT_MAX_WEAPONS];
    int32                    crank;
    VATAINT_DISENGAGE_METHOD disengage_method;
    int32                    padding2;
    float64                  beam_range;
} UATAINT_PARAMETERS;
```

'**target_id**'
: Specifies the id of the vehicle to intercept.

'**target_bearing**'
: Specifies the bearing to the target in radians. This parameter is only set if the target_id is not known.

'**target_range**'
: Specifies the range to the target in meters. This parameter is only set if the target_id is not known.

'**fire_permission**'

Specifies the fire permission (VATAINT_HOLD_FIRE, VATAINT_FIRE_AT_WILL) to be used during the intercept.

'weapon_count'

Specifies the number of weapons in the weapons_enabled list.

'weapons_enabled'

Specifies the weapons which the aircraft is allowed to shoot during the intercept.

'crank' Specifies whether to perform a crank maneuver after each shot taken during the intercept.

'disengage_method'

Specifies how the aircraft should disengage from the target it is intercepting if it does not destroy it. This can take the values VATAINT_INTERNAL, VATAINT_MERGE, and VATAINT_BUGOUT. If it is set to VATAINT_INTERNAL, the disengage method used will be based upon air-to-air intercept tactics taking into account the range to the target and whether a radar-guided missile is in flight. If it is set to VATAINT_MERGE, the aircraft will always go to the merge to disengage. If it is set to VATAINT_BUGOUT, the aircraft will always bugout (no later than 12 nm from the target) to disengage.

'beam_range'

Specifies the range in meters at which the aircraft should turn into the enemy target's radar beam.

2 Functions

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

2.1 uataint_init

```
void uataint_init()
```

uataint_init initializes libuataint. Call this before any other libuataint function.

2.2 uataint_class_init

```
void uataint_class_init(parent_class)
    CLASS_PTR parent_class;
```

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

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

2.3 uataint_create

```
void uataint_create(vehicle_id, params, po_db, ctdb)
    int32                    vehicle_id;
    UATAINT_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

uataint_create creates the uataint class information for a vehicle and attaches it vehicle's block of libclass user data.

2.4 uataint_destroy

```
void uataint_destroy(vehicle_id)
    int vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

uataint_destroy frees the uataint class information for a vehicle. This should be called before freeing the class user data with class_free_user_data.

2.5 uataint_init_task_state

```
void uataint_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_UATAint task that is about to be created, uataint_init_task_state initializes the model size, and state variables.

Libuatgrndtrgt

Table of Contents

1 Overview 1
1.1 Examples 1

2 Functions 3
2.1 uatgtg_init 3
2.2 uatgtg_class_init 3
2.3 uatgtg_create 3
2.4 uatgtg_destroy 4

1 Overview

TEMPLATE: Describe what this library does here.

1.1 Examples

TEMPLATE: Give examples here.

2 Functions

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

TEMPLATE: Adjust alignment of descriptions

TEMPLATE: Correct argument lists and descriptions of these functions.

2.1 uatgtg_init

```
void uatgtg_init()
```

uatgtg_init initializes libuatgrndtrgt. Call this before any other libuatgrndtrgt function.

2.2 uatgtg_class_init

```
void uatgtg_class_init(parent_class)
    CLASS_PTR parent_class;
```

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

uatgtg_class_init creates a handle for attaching uatgrndtrgt class information to vehicles. The **parent_class** will likely be **safobj_class**.

2.3 uatgtg_create

```
void uatgtg_create(vehicle_id, params)
    int vehicle_id;
    UATGRNDTRGT_PARAMETRIC_DATA *params;
```

'vehicle_id'
: Specifies the vehicle ID

'**params**' Specifies initial parameter values

uatgtg_create creates the uatgrndtrgt class information for a vehicle and attaches it vehicle's block of libclass user data.

2.4 uatgtg_destroy

```
void uatgtg_destroy(vehicle_id)
    int vehicle_id;
```

'**vehicle_id**'
Specifies the vehicle ID

uatgtg_destroy frees the uatgrndtrgt class information for a vehicle. This should be called before freeing the class user data with **class_free_user_data**.

LibUBingoFuel

Table of Contents

1 Overview 1
1.1 Task Parameters 1

2 Functions 3
2.1 ubfuel_init 3
2.2 ubfuel_class_init 3
2.3 ubfuel_create 3
2.4 ubfuel_destroy 4
2.5 ubfuel_init_task_state 4

1 Overview

Libubingofuel implements a unit-level task which controls a group (currently only one) of vehicles detecting that they have reached the bingo fuel level and flying to and landing at a refueling point. 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).

Libubingofuel depends on libvtab, libclass, libpo, libctdb, liburtb, libaccess, libreader, and libparmgr.

1.1 Task Parameters

When a **SM_Ubingofuel** task is created or modified, parameters in the parameter block of the task data structure are referenced. The parameters are represented in the task data structure as follows:

```
typedef struct ubingofuel_parameters
{
    ObjectID            base;
    uint16              _padding;
    float64             speed;
    float64             altitude;
} UBINGOFUEL_PARAMETERS;
```

'**base;**' Specifies the bingo refuel base for the plane to go to.

'**speed;**' Specifies the speed for the plane to go when going to the bingo refuel point.

'**altitude;**'
Specifies the altitude for the plane to be at when going to the bingo refuel point.

2 Functions

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

2.1 ubfuel_init

```
void ubfuel_init()
```

ubfuel_init initializes libubingofuel. Call this before any other libubingofuel function.

2.2 ubfuel_class_init

```
void ubfuel_class_init(parent_class)
    CLASS_PTR parent_class;
```

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

ubfuel_class_init creates a handle for attaching ubingofuel class information to vehicles. The **parent_class** will likely be **safobj_class**.

2.3 ubfuel_create

```
void ubfuel_create(vehicle_id, params, po_db, ctdb)
    int32                    vehicle_id;
    UBINGOFUEL_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

ubfuel_create creates the ubingofuel class information for a vehicle and attaches it vehicle's block of libclass user data.

2.4 ubfuel_destroy

```
void ubfuel_destroy(vehicle_id)
    int vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

ubfuel_destroy frees the ubingofuel class information for a vehicle. This should be called before freeing the class user data with class_free_user_data.

2.5 ubfuel_init_task_state

```
void ubfuel_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_UBingoFuel task that is about to be created, ubfuel_init_task_state initializes the model size, and state variables.

LibUCAP

Table of Contents

1 Overview 1
1.1 Task Parameters 1

2 Functions 3
2.1 ucap_init 3
2.2 ucap_class_init 3
2.3 ucap_create 3
2.4 ucap_destroy 4
2.5 ucap_init_task_state 4

1 Overview

Libucap implements a unit-level task which controls a group (currently only one) of vehicles performing a Combat Air Patrol (CAP). Its parameters include a location at which to perform the CAP, direction to orient the CAP, length of CAP legs, speed (for both the inbound and outbound legs), altitude, and radar control settings. 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).

Libucap depends on libvtakeoff, libvcap, libvsearch, libcomponents, libpo, libvtab, libclass, libctdb, libaccess, libstatmon, libeditor, libtaskedit, libreader, libparmgr, libhulls, and libtask.

1.1 Task Parameters

When a **SM_UCap** task is created or modified, parameters in the parameter block of the task data structure are referenced. The parameters are described by the following structure:

```
typedef enum ucap_vc_type
{
    UCAP_VC_PLUS_MINUS_1600,
    UCAP_VC_PLUS_1600,
    UCAP_VC_MINUS_1600
} UCAP_VC_TYPE;


typedef struct ucap_parameters
{
    ObjectID            position;
    uint16              padding1;
    float64             orientation;
    float64             length_of_legs;
    float64             inbound_leg_speed;
    float64             outbound_leg_speed;
    float64             altitude;
    int32               mode;
    UCAP_VC_TYPE        vc;
    float64             half_height;
    float64             half_width;
    float64             range;
    int32               use_location;
    int32               padding2;
    float64             interest_location[2];
    float64             interest_azimuth;
```

```
        float64             interest_elevation;
    } UCAP_PARAMETERS;
```

'position'
: Specifies a persistent object which defines the position of the cap. This object can be a point or a text object.

'orientation'
: Specifies the orientation of the CAPom the position specified in the previous parameter.

'length_of_legs'
: The length of the leg of the CAP

'inbound_leg_speed'
: The speed of the aircraft on the inbound leg of the CAP.

'outbound_leg_speed'
: The speed of the aircraft on the outbound leg of the CAP.

'altitude'
: The altitude to perform the CAP at.

'mode'
: Radar Search mode, either track while scan or manual or auto.

'vc'
: UCAP radar vc type unem

'half_height'
: This value represents half of the height of the desired radar scan volume.

'half_width'
: This value represents half of the width of the desired radar scan volume.

'range'
: This value represents the range of the desired radar scan volume.

'use_location'
: Represents the orientation type to use for the radar, either a location or azimuth/elevation.

'interest_location[2]'
: If use_location is set to location, the location to use for the radar scan volume center. This is an X Y location (array), not an object.

'interest_azimuth'
: If use_location is set to az/el, the azimuth to orient the radar to.

'interest_elevation'
: If use_location is set to az/el, the elevation to orient the radar to.

2 Functions

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

2.1 ucap_init

```
void ucap_init()
```

ucap_init initializes libucap. Call this before any other libucap function.

2.2 ucap_class_init

```
void ucap_class_init(parent_class)
    CLASS_PTR parent_class;
```

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

ucap_class_init creates a handle for attaching ucap class information to vehicles. The **parent_class** will likely be **safobj_class**.

2.3 ucap_create

```
void ucap_create(vehicle_id, params, po_db, ctdb)
    int                    vehicle_id;
    UCAP_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

ucap_create creates the ucap class information for a vehicle and attaches it vehicle's block of libclass user data.

2.4 ucap_destroy

```
void ucap_destroy(vehicle_id)
    int vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

ucap_destroy frees the ucap class information for a vehicle. This should be called before freeing the class user data with class_free_user_data.

2.5 ucap_init_task_state

```
void ucap_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_UCAP task that is about to be created, ucap_init_task_state initializes the model size, and state variables.

LibUCommit

Table of Contents

1 Overview 1
1.1 Task Parameters 1

2 Functions 3
2.1 ucommit_init 3
2.2 ucommit_class_init 3
2.3 ucommit_create 3
2.4 ucommit_destroy 4
2.5 ucommit_init_task_state 4

1 Overview

Libucommit implements a unit-level task which detects when a group (currently only one) of vehicles have detected a radar target which meets the commit criteria. It creates an intercept task frame and pushes it onto the unit's stack, causing the unit to perform an air-to-air intercept on the target. 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).

1.1 Task Parameters

When a **SM_UCommit** task is created or modified, parameters in the parameter block of the task data structure are referenced. The parameters are represented in the task data structure as follows:

```
typedef struct ucommit_parameters
{
    float64                  tgt_range_threshold;
    float64                  tgt_aspect_threshold;
    float64                  tgt_speed_threshold;
    VATAINT_FIRE_PERMISSION  fire_permission;
    int32                    weapon_count;
    VATAINT_WEAPONS_ENABLED  weapons_enabled[VATAINT_MAX_WEAPONS];
    uint32                   crank;
    VATAINT_DISENGAGE_METHOD disengage_method;
    float64                  beam_range;
} ucommit_PARAMETERS;
```

All of the first three thresholds must be met before the **commit** task will be initiated.

'**tgt_range_threshold**'
: Specifies the maximum range between vehicle and target that would trigger this task.

'**tgt_aspect_threshold**'
: Specifies the maximum target aspect angle (left or right) that would cause this task to trigger.

'**tgt_speed_threshold**'
: Specifies the minimum speed that the target must be flying in order to trigger this task.

The following parameters are all used as inputs to the air to air intercept task.

'fire_permission'
Specifies the permission to pass to the air to air intercept task. The choices are VATAINT_HOLD_FIRE and VATAINT_FIRE_AT_WILL.

'weapon_count'
Specifies the number of weapons in the weapons enabled list.

'weapons_enabled'
Specifies a list of the weapons that an aircraft is allowed to shoot during an air to air intercept task.

'crank' Specifies whether to perform a crank manuever after each shot taken during an air to air intercept task.

'disengage_method'
Specifies how the aircraft should disengage from a target during an intercept if the target was not destroyed. The possible values for this variable are VATAINT_INTERNAL, VATAINT_MERGE, and VATAINT_BUGOUT.

'beam_range'
Specifies the range in meters at which the aircraft should turn into the enemy target's radar beam.

2 Functions

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

2.1 ucommit_init

```
void ucommit_init()
```

ucommit_init initializes libucommit. Call this before any other libucommit function.

2.2 ucommit_class_init

```
void ucommit_class_init(parent_class)
    CLASS_PTR parent_class;
```

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

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

2.3 ucommit_create

```
void ucommit_create(vehicle_id, params, po_db)
    int32                    vehicle_id;
    UCOMMIT_PARAMETRIC_DATA *params;
    PO_DATABASE             *po_db;
```

'vehicle_id'
Specifies the vehicle ID

'params' Specifies initial parameter values

'po_db' Specifies the PO database where the task can be found

ucommit_create creates the ucommit class information for a vehicle and attaches it vehicle's block of libclass user data.

2.4 ucommit_destroy

```
void ucommit_destroy(vehicle_id)
    int vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

ucommit_destroy frees the ucommit class information for a vehicle. This should be called before freeing the class user data with **class_free_user_data**.

2.5 ucommit_init_task_state

```
void ucommit_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_UCommit** task that is about to be created, **ucommit_init_task_state** initializes the model size, and state variables.

Libudsmnt

Table of Contents

1 Overview 1
1.1 Task Parameters 1

2 Functions 3
2.1 udsmnt_init 3
2.2 udsmnt_class_init 3
2.3 udsmnt_create 3
2.4 udsmnt_destroy 4
2.5 udsmnt_init_task_state 4

1 Overview

Libudsmnt implements a unit-level task which dismounts a group of subordinate DIs from a specified IFV.

When the IFV is selected to dismount its DIs, the initial action is to determine if that IFV is carrying any DIs. If there are DIs to dismount, a halt task issued to that IFV.

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).

1.1 Task Parameters

When a **SM_UDsmnt** task is created or modified, parameters in the parameter block of the task data structure are referenced. The parameters are represented in the task data structure as follows:

```
typedef struct udsmnt_parameters
{
    int32 prep_var;
} UDSMNT_PARAMETERS;
```

'**prep_var**'

Specifies whether the task is a placeholder preparation for another task frame (and therefore should end immediately). If **prep_var** is **FALSE**, the task will never end.

2 Functions

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

2.1 udsmnt_init

```
void udsmnt_init()
```

udsmnt_init initializes libudsmnt. Call this before any other libudsmnt function.

2.2 udsmnt_class_init

```
void udsmnt_class_init(parent_class)
    CLASS_PTR parent_class;
```

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

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

2.3 udsmnt_create

```
void udsmnt_create(vehicle_id, params, po_db, ctdb)
    int32                   vehicle_id;
    UDSMNT_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

udsmnt_create creates the udismount class information for a vehicle and attaches it vehicle's block of libclass user data.

2.4 udsmnt_destroy

```
void udsmnt_destroy(vehicle_id)
    int vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

udsmnt_destroy frees the udismount class information for a vehicle. This should be called before freeing the class user data with **class_free_user_data**.

2.5 udsmnt_init_task_state

```
void udsmnt_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_UDsmnt** task that is about to be created, **udsmnt_init_task_state** initializes the model size, and state variables.

Libuenemy

Table of Contents

1 Overview 1

2 Functions 3

2.1 uenemy_init 3
2.2 uenemy_class_init 3
2.3 uenemy_create 3
2.4 uenemy_destroy 4
2.5 init_task_state 4
2.6 uenemy_get_spotted_from_state 4
2.7 uenemy_get_number_seen 5
2.8 uenemy_get_closest_vehicle_position 5
2.9 uenemy_get_center_of_mass 6
2.10 uenemy_reaction_init 7
2.11 uenemy_reaction_done 8

1 Overview

Libuenemy implements a unit level background task which maintains a list of the spotted enemy vehicles that all the vehicles of the unit can see. This is done by looking at the lists of each of the vehicle level **SM_VEnemy** tasks and merging the lists.

Libuenemy also provides a method for detecting the following:

minefields
incoming artillery
enemy superiority
high casualties
air raids

These variables are put in the state and are accessed by registering functions with the **uenemy_reaction_ini** function. The caller of this function provides a function for each of the above situations. For example, there will be a incoming_artillery_function that will be called when the boolean variable **artillery** is set.

Since these variables are state variables, libuenemy will monitor the state changes using callbacks and call the appropriate functions when the state changes.

2 Functions

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

2.1 uenemy_init

```
void uenemy_init()
```

uenemy_init initializes libuenemy. Call this before any other libuenemy function.

2.2 uenemy_class_init

```
void uenemy_class_init(parent_class)
    CLASS_PTR parent_class;
```

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

uenemy_class_init creates a handle for attaching uenemy class information to vehicles. The **parent_class** will likely be **safobj_class**.

2.3 uenemy_create

```
void uenemy_create(vehicle_id, params, po_db, unit_entry)
    int                      vehicle_id;
    UENEMY_PARAMETRIC_DATA  *params;
    PO_DATABASE             *po_db;
    PO_DB_ENTRY             *unit_entry;
```

'vehicle_id'
Specifies the vehicle ID

'params' Specifies initial parameter values

'po_db' Specifies the po database for use for the vehicle information.

'unit_entry'
Specifies the persistent object database entry for the unit that this uenemy is created for.

uenemy_create creates the uenemy class information for a vehicle and attaches it vehicle's block of libclass user data.

2.4 uenemy_destroy

```
void uenemy_destroy(vehicle_id, is_migration)
    int32 vehicle_id;
    int32 is_migration;
```

'vehicle_id'
Specifies the vehicle ID

'is_migration'
Specifies that the destroy is due to migration

uenemy_destroy frees the uenemy class information for a vehicle. This should be called before freeing the class user data with class_free_user_data.

2.5 init_task_state

```
void uenemy_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_UEnemy task that is about to be created, uenemy_init_task_state initializes the model size, and state variables.

2.6 uenemy_get_spotted_from_state

```
int32 uenemy_get_spotted_from_state(db, task, live, list)
    PO_DATABASE       *db;
    TaskClass         *task;
    UENEMY_FILTER_LD   live;
    VTAB_LIST          list;
```

'db' The current persistent object database.

'task' This unit's uenemy task.

'live' An enum of defined type UENEMY_FILTER_LD (live/dead), which is one of the following: UENEMY_FILTER_LD_LIVE - list live vehicles only UENEMY_FILTER_LD_DEAD - list dead vehicles only UENEMY_FILTER_LD_ANY - don't filter for live or dead status

'list' The VTAB_LIST for the seen vehicle's to be added to. NOTE: this must be an already existing list.

Fill out an existing VTAB list with the all vehicles the unit currently can see that pass the filter settings.

2.7 uenemy_get_number_seen

```
int32 uenemy_get_number_seen(db, task, live)
    PO_DATABASE       *db;
    TaskClass         *task;
    UENEMY_FILTER_LD   live;
```

'db' The persistent object database being used.

'task' This unit's uenemy task.

'live' An enum of defined type UENEMY_FILTER_LD (live/dead), which is one of the following: UENEMY_FILTER_LD_LIVE - list live vehicles only UENEMY_FILTER_LD_DEAD - list dead vehicles only UENEMY_FILTER_LD_ANY - don't filter for live or dead status

uenemy_get_number_seen returns the number of vehicles that fit the parameters specified in the function call. It uses the enums detailed in the parameter definitions above to specify what you want counted.

2.8 uenemy_get_closest_vehicle_position

```
void uenemy_get_closest_vehicle_position(db, task, live, my_pos, his_pos)
    PO_DATABASE       *db;
    TaskClass         *task;
    UENEMY_FILTER_LD  live;
    float64           *my_pos;
    float64           *his_pos;
```

'db' The persistent object database being used.

'task' This unit's uenemy task.

'live' An enum of defined type UENEMY_FILTER_LD (live/dead), which is one of the following: UENEMY_FILTER_LD_LIVE - list live vehicles only UENEMY_FILTER_LD_DEAD - list dead vehicles only UENEMY_FILTER_LD_ANY - don't filter for live or dead status

'my_pos' A pointer to an two dimensional array containing this unit's representative position, or the position that the closest vehicle to is desired.

'his_pos' A pointer to a two dimensional array into which the location of the appropriate vehicle location is placed. If no appropriate vehicle is found, his_pos is filled with zeros.

This function returns the position of the closest vehicle that fits the parameters specified in the function call. See either the texinfo or the header file for the enums available for use for specification to this function.

2.9 uenemy_get_center_of_mass

```
int32 uenemy_get_center_of_mass(db, task, live, com_pos)
    PO_DATABASE       *db;
    TaskClass         *task;
    UENEMY_FILTER_LD  live;
    float64           *com_pos;
```

'db' The persistent object database being used.

'task' This unit's uenemy task.

'live' An enum of defined type UENEMY_FILTER_LD (live/dead), which is one of the following: UENEMY_FILTER_LD_LIVE - list live vehicles only UENEMY_FILTER_LD_DEAD - list dead vehicles only UENEMY_FILTER_LD_ANY - don't filter for live or dead status

'com_pos' A pointer to a two dimensional array into which the location of the appropriate vehicle location is placed. If no appropriate vehicle is found, his_pos is filled with zeros.

This function returns the position of the avg center of mass of all the vehicles that the unit currently knows about that pass the filter settings passed in in the parameters to the function call. It's return value will be the number of sighted vehicles that passed the filter.

2.10 uenemy_reaction_init

```
UENEMY_REACTION_CB_PARMS_PTR uenemy_reaction_init(po_db, unit_entry,
                                                  mine_function,arty_function,
                                                  high_loss_function,
                                                  super_enemy_function,
                                                  air_raid_function, user_data)
    PO_DATABASE       *po_db;
    PO_DB_ENTRY       *unit_entry;
    void             (*mine_function)();
    void             (*arty_function)();
    void             (*high_loss_function)();
    void             (*super_enemy_function)();
    void             (*air_raid_function)();
    ADDRESS            user_data;
```

'po_db' The persistent object database being used.

'unit_entry'
The persistent object database entry for the unit initializing to use the reaction flags.

'mine_function'
The function passed in that will be called when the minefield boolean variable in the state is set. This function will be called every time the state object changes and the minefield boolean variable is set.

'arty_function'
The function passed in that will be called when the artillery boolean variable in the state is set. This function will be called every time the state object changes and the artillery boolean variable is set.

'high_loss_function'
The function passed in that will be called when the high_losses boolean variable in the state is set. This function will be called every time the state object changes and the high_losses boolean variable is set.

'super_enemy_function'
The function passed in that will be called when the enemy_superiority boolean variable in the state is set. This function will be called every time the state object changes and the enemy_superiority boolean variable is set.

'arty_function'
: The function passed in that will be called when the air_raid boolean variable in the state is set. This function will be called every time the state object changes and the air_raid boolean variable is set.

'user_data'
: The data that is passed to each one of the above functions when called.

This function returns a pointer to a structure that contains most of the above information, UENEMY_REACTION_CB_PARMS_PTR. The purpose of the return pointer is so that this pointer can be passed to the uenemy_reaction_done function.

This function will register to a object_changed_event. Every time the uenemy state object changes, the reaction boolean state variables are checked. For every boolean variable set, the corresponding function is called and passed the user_data.

2.11 uenemy_reaction_done

```
void uenemy_reaction_done(params_ptr, po_db)
    UENEMY_REACTION_CB_PARMS_PTR  params_ptr;
    PO_DATABASE                   *po_db;
```

'ptr'
: This ptr of type UENEMY_REACTION_CB_PARMS_PTR is passed to this function. This pointer points to the structure that needs to be deallocated in this function. This pointer is also the return value for the uenemy_reaction_init.

'po_db'
: The persistent object database being used.

The purpose of this function is to unregister the object changed event handler and to deallocate the memory allocated in the uenemy_reaction_init function.

LibUFlwRte

Table of Contents

1 Overview 1
1.1 Task Parameters 1

2 Functions 3
2.1 uflwrte_init 3
2.2 uflwrte_class_init 3
2.3 uflwrte_create 3
2.4 uflwrte_destroy 4
2.5 uflwrte_init_task_state 4

1 Overview

Libuflwrte implements a unit-level task which controls a group (currently only one) of vehicles following a route. 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).

Libuflwrte depends on libvtakeoff, libvflwrte, libvorbit, libpo, libvtab, libclass, libctdb, libaccess, libreader, and libparmgr.

1.1 Task Parameters

When a **SM_UFlwRte** task is created or modified, parameters in the parameter block of the task data structure are referenced. The parameters are represented in the task data structure as follows:

```
typedef struct uflwrte_parameters
{
    ObjectID            route;
    int16               padding1;
    float64             speed;
    float64             altitude;
    VFLWRTE_MOVE_TYPE   move_type;
    int32               padding2;
    int32               mode;
    UCAP_VC_TYPE        vc;
    float64             half_height;
    float64             half_width;
    float64             range;
    int32               use_location;
    int32               padding3;
    float64             interest_location[2];
    float64             interest_azimuth;
    float64             interest_elevation;
} UFLWRTE_PARAMETERS;
```

'route' Route is a persistent object which defines a route to be followed. This object can be a point object, line object, or a text object.

'speed' Specifies the speed for the route.

'altitude'
Specifies the altitude for the route.

'move_type'
Specifies the movement type which can be one of the following values:

'VFLWRTE_MOVE_LOW_LEVEL'
Follow at a constant altitude, only increasing altitude to go over an obstacle.

'VFLWRTE_MOVE_CONTOUR'
Follow the contour of the earth.

'VFLWRTE_MOVE_NOE'
Nap of Earth - maintain constant altitude agl, veering around obstacles.

'mode' Radar Search mode, either track while scan or manual or auto.

'vc' UCAP radar vc type unem

'half_height'
This value represents half of the height of the desired radar scan volume.

'half_width'
This value represents half of the width of the desired radar scan volume.

'range' This value represents the range of the desired radar scan volume.

'use_location'
Represents the orientation type to use for the radar, either a location or azimuth/elevation.

'interest_location[2]'
If use_location is set to location, the location to use for the radar scan volume center. This is an X Y location (array), not an object.

'interest_azimuth'
If use_location is set to az/el, the azimuth to orient the radar to.

'interest_elevation'
If use_location is set to az/el, the elevation to orient the radar to.

2 Functions

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

2.1 uflwrte_init

```
void uflwrte_init()
```

uflwrte_init initializes libuflwrte. Call this before any other libuflwrte function.

2.2 uflwrte_class_init

```
void uflwrte_class_init(parent_class)
    CLASS_PTR parent_class;
```

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

uflwrte_class_init creates a handle for attaching uflwrte class information to vehicles. The **parent_class** will likely be **safobj_class**.

2.3 uflwrte_create

```
void uflwrte_create(vehicle_id, params, po_db, ctdb)
    int32       vehicle_id;
    UFLWRTE_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

uflwrte_create creates the uflwrte class information for a vehicle and attaches it vehicle's block of libclass user data.

2.4 uflwrte_destroy

```
void uflwrte_destroy(vehicle_id)
    int vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

uflwrte_destroy frees the uflwrte class information for a vehicle. This should be called before freeing the class user data with class_free_user_data.

2.5 uflwrte_init_task_state

```
void uflwrte_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_UFlwRte task that is about to be created, uflwrte_init_task_state initializes the model size, and state variables.

Libumount

Table of Contents

1 Overview 1
1.1 Task Parameters 1

2 Functions 3
2.1 umount_init 3
2.2 umount_class_init 3
2.3 umount_create 3
2.4 umount_destroy 4
2.5 umount_init_task_state 4

1 Overview

Libumount implements a unit-level task which mounts a group of subordinate DIs onto the closest friendly IFV.

When the subordinate DIs are issued a mount task, the initial action is to issue a halt task to those DIs. A search for the closest friendly IFV within a specified range follows. If an IFV is found, a halt task is issued to the IFV and the DIs begin moving towards that IFV and mounts them upon reaching them.

The state machine will wait for all subordinates DIs to get to an IFV and then go to a ended state.

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).

1.1 Task Parameters

When a SM_UMount task is created or modified, parameters in the parameter block of the task data structure are referenced. The parameters are represented in the task data structure as follows:

```
typedef struct umount_parameters
{
    int32 prep_var;
} UMOUNT_PARAMETERS;
```

'prep_var'

Specifies whether the task is a placeholder preparation for another task frame (and therefore should end immediately). If prep_var is FALSE, the task will never end.

2 Functions

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

2.1 umount_init

```
void umount_init()
```

umount_init initializes libumount. Call this before any other libumount function.

2.2 umount_class_init

```
void umount_class_init(parent_class)
    CLASS_PTR parent_class;
```

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

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

2.3 umount_create

```
void umount_create(vehicle_id, params, po_db, ctdb)
    int32                    vehicle_id;
    UMOUNT_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

umount_create creates the umount class information for a vehicle and attaches it vehicle's block of libclass user data.

2.4 umount_destroy

```
void umount_destroy(vehicle_id)
    int32 vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

umount_destroy frees the umount class information for a vehicle. This should be called before freeing the class user data with class_free_user_data.

2.5 umount_init_task_state

```
void umount_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_UMount task that is about to be created, umount_init_task_state initializes the model size, and state variables.

LibUnitOrg

Table of Contents

1 Overview 1

2 Functions 3

2.1 unitorg_init 3
2.2 unitorg_class_init 3
2.3 unitorg_create 3
2.4 unitorg_destroy 4
2.5 unitorg_changed 4
2.6 unitorg_get_context 4
2.7 unitorg_get_roles 5
2.8 unitorg_get_responsible_unit 6
2.9 unitorg_get_capable_vehicles 6
2.10 unitorg_initialize_SAF_capabilities 7
2.11 unitorg_update_SAF_capabilities 7

3 Global Variables 9

3.1 unitorg_task_organized_changed_event 9
3.2 unitorg_function_organized_changed_event 9
3.3 UNITORG_MAX_BREADTH 9

1 Overview

LibUnitOrg manages unit organization information within the context of the SAF simulation. It tracks both the task organized and functionally organized superior and subordinate relationships of units, in order to provide this information without the need for frequent persistent object database queries.

LibUnitOrg also tracks changes to units which are made by outside sources (such as the GUI), and updates the simulation accordingly.

2 Functions

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

2.1 unitorg_init

```
void unitorg_init()
```

unitorg_init initializes libunitorg. Call this before any other libunitorg function.

2.2 unitorg_class_init

```
void unitorg_class_init(parent_class)
    CLASS_PTR parent_class;
```

'parent_class'
Specifies the parent class (probably c2obj_class)

unitorg_class_init creates a handle for attaching unitorg class information to entries. The parent_class is one created with class_declare_class.

2.3 unitorg_create

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

'entry' Specifies the unit entry

'db' Specifies the PO database

unitorg_create creates the unitorg class information for a entry and attaches it to the entry's libclass user data. If the passed entry is not objectClassUnit, this routine will simply return.

2.4 unitorg_destroy

```
void unitorg_destroy(entry)
    PO_DB_ENTRY *entry;
```

'entry' Specifies the unit entry

unitorg_destroy frees the unitorg class information for a entry.

2.5 unitorg_changed

```
void unitorg_changed(entry)
    PO_DB_ENTRY *entry;
```

'entry' Specifies the unit entry

unitorg_changed updates unit in response to a libpo object_changed event.

2.6 unitorg_get_context

```
void unitorg_get_context(entry, task_organized,
                         superior,
                         peers, n_peers, max_peers,
                         subordinates, n_subordinates, max_subordinates)
    PO_DB_ENTRY  *entry;
    int32         task_organized;
    PO_DB_ENTRY **superior;
    PO_DB_ENTRY  *peers[];
    int32        *n_peers;
    int32         max_peers;
    PO_DB_ENTRY  *subordinates[];
    int32        *n_subordinates;
    int32         max_subordinates;
```

'entry' Specifies the unit entry

'task_organized'
Specifies whether to get the task organized or functionally organized context, True or False.

'superior'
Specifies where to return the superior unit

'peers' Specifies an array to return peer units

'n_peers' Specifies number of peers returned

'max_peers'
Specifies maximun number of peers to return

'subordinates'
Specifies an array to return subordinate units

'n_subordinates'
Specifies number of subordinates returned

'max_subordinates'
Specifies maximun number of subordinates to return

unitorg_get_context returns the superior, peers and subordinates of a unit. Any of superior, peers, or subordinates may be NULL, in which case no units for that category will be returned. If peers or subordinates are non-null, then those arrays must be large enough to hold max_peers and max_subordinates, respectively.

If task_organized is TRUE, then information about the task organized hierarchy will be returned. Otherwise, information about the functionally organized hierarchy will be used.

This routine is much more efficient at retrieving information about a unit hierarchy than doing a po_query_for_current_objects, (see section 'po_query_for_current_objects' in LibPO Programmer's Manual) since the hierarchy information is maintained and updated incrementally by libUnitOrg.

2.7 unitorg_get_roles

```
int32 unitorg_get_roles(entry, max_roles, roles)
    PO_DB_ENTRY  *entry;
    int32         max_roles;
    PO_DB_ENTRY  *roles[];
```

'entry' Specifies the unit entry

'max_roles'
Specifies the length of roles array used to return role information

'roles' Returns all the roles that the entry is acting as, including itself

unitorg_get_roles returns (by reference) all the unit entries that a unit is acting as, including itself. This information is derived from the task-organized unit hierarchy. A unit will be acting in the role of the unit's superior if it is the subordinate of that superior with the lowest job number (promotion index). This relationship can be recursive, in that the superior may also be acting in the role of the superior's superior, and so on. The number of roles that the entry is acting as (including itself) is returned by this routine.

This routine is much more efficient at retrieving information about a unit hierarchy than doing a **po_query_for_current_objects**, (see section 'po_query_for_current_objects' in LibPO Programmer's Manual) since the hierarchy information is maintained and updated incrementally by libUnitOrg.

2.8 unitorg_get_responsible_unit

```
PO_DB_ENTRY *unitorg_get_responsible_unit(entry)
    PO_DB_ENTRY *entry;
```

'entry' Specifies the unit entry

unitorg_get_responsible_unit returns the entry, lowest in the unit hierarchy, that is responsible for the passed in **entry**. For example, if the input is a platoon object, the return value will be the vehicle acting as the leader of the platoon.

2.9 unitorg_get_capable_vehicles

```
int32 unitorg_get_capable_vehicles(entry, result, max, capabilities_mask)
    PO_DB_ENTRY *entry;
    PO_DB_ENTRY *result[];
    int32       max;
    uint32      capabilities_mask;
```

'entry' Specifies the unit entry

'result' Specifies an array to store the result in.

'max' Specifies the size of the result array.

'capabilities_mask'
Specifies a mask of capabilities to search for when accumulating vehicles.

unitorg_get_capable_vehicles returns the non-unit entries subordinate to the passed in **entry** into the **result** array. This function recursively descends the fuctional hierarchy to find all the non-unit subordinates. The number of non-unit subordinates found is returned. **capabilities_mask** contains the capabilites of the vehicles to be included in the query. The mask will contain values such as **SAFCapabilityMobility**, as encoded in the libPO protocol file 'p_po.h'. A mask of 0 means to include all vehicles, regardless of capability. If passed an entry corresponding to a vehicle, that vehicle may be returned, if it has the required capability.

2.10 unitorg_initialize_SAF_capabilities

```
void unitorg_initialize_SAF_capabilities(entry, capabilities)
    PO_DB_ENTRY *entry;
    uint32       capabilities;
```

'entry' Specifies the unit entry

'capabilities'
Specifies the capabilities of the unit, as specified by a mask of SAFCapabilities values defined by the libPO unitClass object (see section 'Unit Class' in LibPO Programmer's Manual).

unitorg_initialize SAF_capabilities sets an entry's SAF capabilities. All this function does is to modify the entry's PO.

2.11 unitorg_update_SAF_capabilities

```
void unitorg_update_SAF_capabilities(entry, capabilities)
    PO_DB_ENTRY *entry;
    uint32       capabilities;
```

'entry' Specifies the unit entry

'capabilities'
Specifies the capabilities of the unit, as specified by a mask of SAFCapabilities values defined by the libPO unitClass object (see section 'Unit Class' in LibPO Programmer's Manual).

unitorg_set SAF_capabilities modifies an entry's SAF capabilities. This is only valid for entries corresponding to vehicles, as superior units inherit capabilities from their subordinate units.

This routine may have no effect when a vehicle is just created; `unitorg_initialize_SAF_capabilities` should be used in that case.

3 Global Variables

The sections below describe the global variables by including a synopsis and a description. The global variables are callback events which will be fired when changes to the unit hierarchy are made. Each event can be handled by attaching a unitorg_changed_event_handler, as described below:

```
void unitorg_changed_event_handler(object, user_data)
    PO_DB_ENTRY *object;
    ADDRESS      user_data;
```

unitorg_changed_event_handler is called when the organization hierarchy of a unit is changed. object will contain the entry of the unit which has a changed superior.

3.1 unitorg_task_organized_changed_event

```
extern CALLBACK_EVENT_PTR unitorg_task_organized_changed_event;
```

unitorg_task_organized_changed_event is a libcallback event which can be accessed after libunitorg is initialized. Applications may attach a unitorg_changed_event_handler to this event via (see section 'callback_register_handler' in LibCallback Programmer's Manual). This event will be fired when the task organized superior for a unit has changed.

3.2 unitorg_function_organized_changed_event

```
extern CALLBACK_EVENT_PTR unitorg_function_organized_changed_event;
```

unitorg_function_organized_changed_event is a libcallback event which can be accessed after libunitorg is initialized. Applications may attach a unitorg_changed_event_handler to this event via (see section 'callback_register_handler' in LibCallback Programmer's Manual). This event will be fired when the functionally organized superior for a unit has changed.

3.3 UNITORG_MAX_BREADTH

The macro UNITORG_MAX_BREADTH is a typical maxmimum number of direct subordinates in a

unit. This is a reasonable number to use as a max for the arrays in `unitorg_get_context` and `unitorg_get_capable_vehicles`.

LibUnits

Table of Contents

1 Overview 1

2 Functions 3

2.1 units_init 3
2.2 units_create_editor 3
2.3 units_class_init 4
2.4 units_create 4
2.5 units_destroy 5
2.6 units_changed 5

1 Overview

Libunits manages the editing of unit class persistent objects. The display of these objects performed via libBGRDB icons. present).

2 Functions

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

2.1 units_init

```
void units_init()
```

units_init initializes libunits. Call this function before calling any other libunits functions.

2.2 units_create_editor

```
int32 units_create_editor(data_path, reader_flags, dialog_parent,
                          gui, tactmap, tcc, ctdb, map_erase_gc,
                          sensitive, refresh_event, db, select, pvd, addr)
    char               *data_path;
    int32               reader_flags;
    Widget              dialog_parent;
    SGUI_PTR            gui;
    TACTMAP_PTR         tactmap;
    COORD_TCC_PTR       tcc;
    CTDB               *ctdb;
    GC                  map_erase_gc;
    SMSTVE_WINDOW_PTR   sensitive;
    CALLBACK_EVENT_PTR refresh_event;
    PO_DATABASE        *db;
    SELECT_TOOL_PTR     select;
    PVD_GUI_PTR         pvd;
    SimulationAddress  *addr;
```

'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 shell which should parent popup dialogs

'gui'
: Specifies the SAF GUI

'pvd' Specifies the PVD

'tactmap' Specifies the tactical map

'tcc' Specifies the map coordinate system

'ctdb' Specifies the terrain database

'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

'pvd' Specifies the PVD GUI

'addr' Specifies the Simulation address of the workstation

units_create_editor creates the units editor. The data file ('units.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.

2.3 units_class_init

```
void units_class_init(parent_class)
    CLASS_PTR parent_class;
```

'parent_class'
Specifies the parent class (probably c2obj_class)

units_class_init creates a handle for attaching units class information to entries. The parent_class is one created with class_declare_class.

2.4 units_create

```
void units_create(entry)
    PO_DB_ENTRY *entry;
```

'entry' Specifies the unit entry

units_create creates the units class information for a entry and attaches it to the entry's libclass user data.

2.5 units_destroy

```
void units_destroy(entry)
    PO_DB_ENTRY *entry;
```

'entry' Specifies the unit entry

units_destroy frees the units class information for a entry.

2.6 units_changed

```
void units_changed(entry)
    PO_DB_ENTRY *entry;
```

'entry' Specifies the unit entry

units_changed updates displayed graphic in response to a libpo object_changed event.

LibUnitUtil

Table of Contents

1 Overview 1

2 Functions 3

2.1 unitutil_create 3

1 Overview

Libunitutil provides UI-independent utilities for supporting units. Currently support is provided for creating units and all appropriate subunits.

2 Functions

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

2.1 unitutil_create

```
PO_DB_ENTRY *unitutil_create(new, db)
    UUTIL_UNIT_PARAMS *new;
    PO_DATABASE       *db;
```

unitutil_create creates a unit from the attributes passed in the **new** **UUTIL_UNIT_PARAMs** argument in the **PO_DATABASE** identified by the **db** argument. All appropriate subunits are created automatically. A pointer to the unit's **PO_DB_ENTRY** is returned.

Libuoccpos

Table of Contents

1 Overview 1
1.1 Task Parameters 1

2 Functions 3
2.1 uoccpos_init 3
2.2 uoccpos_class_init 3
2.3 uoccpos_create 3
2.4 uoccpos_destroy 4
2.5 uoccpos_init_task_state 4

1 Overview

Libuoccpos implements a unit-level Occupy Position task. The Preparatory task (see section 'Overview' in Libupoccpos Programmer's Manual). gets the subordinates in covered positions. The Occupy Position task keeps the subordinates in these positions until different orders are assigned. 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).

1.1 Task Parameters

There are no parameters in the SM_UOcpyPos task.

2 Functions

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

2.1 uoccpos_init

```
void uoccpos_init()
```

uoccpos_init initializes libuoccpos. Call this before any other libuoccpos function.

2.2 uoccpos_class_init

```
void uoccpos_class_init(parent_class)
    CLASS_PTR parent_class;
```

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

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

2.3 uoccpos_create

```
void uoccpos_create(vehicle_id, params)
    int32                       vehicle_id;
    UOCCPOS_PARAMETRIC_DATA  *params;
```

'vehicle_id'
Specifies the vehicle ID

'params' Specifies initial parameter values

uoccpos_create creates the uoccpos class information for a vehicle and attaches it vehicle's block of libclass user data.

2.4 uoccpos_destroy

```
void uoccpos_destroy(vehicle_id)
    int vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

uoccpos_destroy frees the uoccpos class information for a vehicle. This should be called before freeing the class user data with class_free_user_data.

2.5 uoccpos_init_task_state

```
void uoccpos_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_UOcpyPos task that is about to be created, uoccpos_init_task_state initializes the model size, and state variables.