Module Programmer's Guide to Local Map Builder for ALVan

Anthony Stentz

Steve Shafer

28 June 1986

CMU Computer Science Department

This work is funded by the US Army Engineer Topographic Laboratories, under contract number DACA76-85-C-0003

1. Introduction

1.1 Conceptual Design

This document describes the local map blackboard component of a mobile robot system, the Autonomous Land Vehicle (ALV), under construction at CMU. This system will utilize diverse sensors including cameras, sonar, and range finders to navigate autonomously on roads; it is composed of several processes performing these tasks and communicating via a Local Map Builder process, which is the subject of this document.

The system architecture consists of four main levels of abstraction: a "virtual vehicle" robot interface, sensor processing, a local map, and high-level cognition as illustrated in figure 1-1:

- *Virtual vehicle:* The lowest level includes the vehicle, control programs, and interface routines to sensor modules. The virtual vehicle handles the details of getting the vehicle to move, point its sensors, ship back data, and perform some data correction such as correction for vehicle motion during sensing. Higher-level modules will be able to command a velocity and trajectory, and have the vehicle move and the sensor data appear with no further effort on their part.

- *Sensor processing:* Sensor processing turns signals into symbols. It takes raw data and produces 3-D coordinate symbols. The output of this level is sensor-dependent and has no semantic interpretation. For example, vision modules take images and produce 3-D lines and surfaces. Sonar modules take raw sonar data and produce 3-D blobs. Range modules take raw range data and produce 3-D discontinuities, flat regions, and rough regions.

- *Local map:* This is the level at which sensor fusion occurs. Sensor modules generate unlabeled and sensor-dependent data objects for storage in the local map. Knowledge Sources (KS) work with this data to combine it with data from other sensors and from predictions to produce symbolically labeled data. The local map level is the subject of this document.

- *Cognition:* This level handles path planning, mission assessment, safety, goal seeking, etc. It includes geometric information, such as global maps, and symbolic information, such as labeled objects. The details of particular sensors are masked.

This document deals with the local map level, which is responsible for maintaining a description of the environment around the vehicle. It receives data from the sensor interpretation processes, which is in map coordinates but has no semantic labels. The data is passed to KS processes and is eventually used by cognition and control processes. The components of the local map level are the local map builder, that orchestrates the processing, a set of knowledge sources that do the work, and a blackboard that stores the data (see figure 1-2):

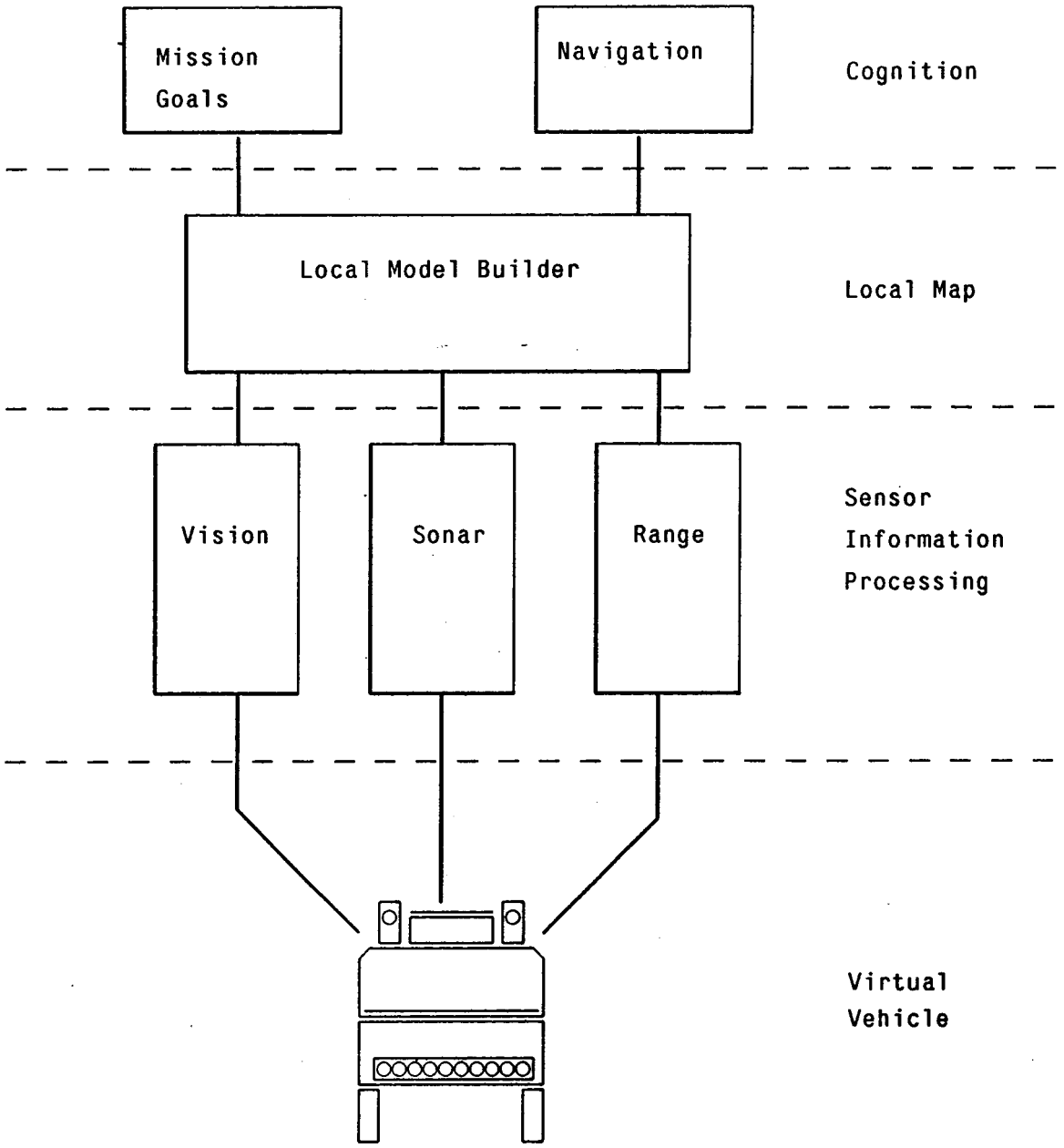

Figure 1-1: The four levels of the CMU system architecture.

- *Local map builder (LMB:)* This component controls the local map. The LMB is responsible for taking requests from navigation and goal- seeking modules, listening to data from the sensor interpretation, and then selecting knowledge sources to run. This gives the LMB a dual role, as both an interface and data channel, and as a scheduler for the knowledge sources.

- *Knowledge Sources (KS:)* This component consists of expert or specialized modules that retrieve partial descriptions from the local map, generate higher-level descriptions, and write back into the local map. Possible KS modules include a road finder, an obstacle finder, a vehicle position predictor, and a landmark identifier.

- *Blackboard:* This component is a data structure into which the LMB stores sensor data interpretations and hypotheses about objects in the local map. Conceptually, it is split into three levels: sensor-dependent data, information that is highly dependent on the sensor that generated it; partial descriptions, such as parts of roads or hypotheses about obstacle location; and fully-labeled objects, such as roads, houses, or traffic signs.

This document describes the LMB, the Blackboard, and the interface to the LMB used by KS, sensor, and cognition modules. It provides detailed specifications for programmers implementing these types of modules.

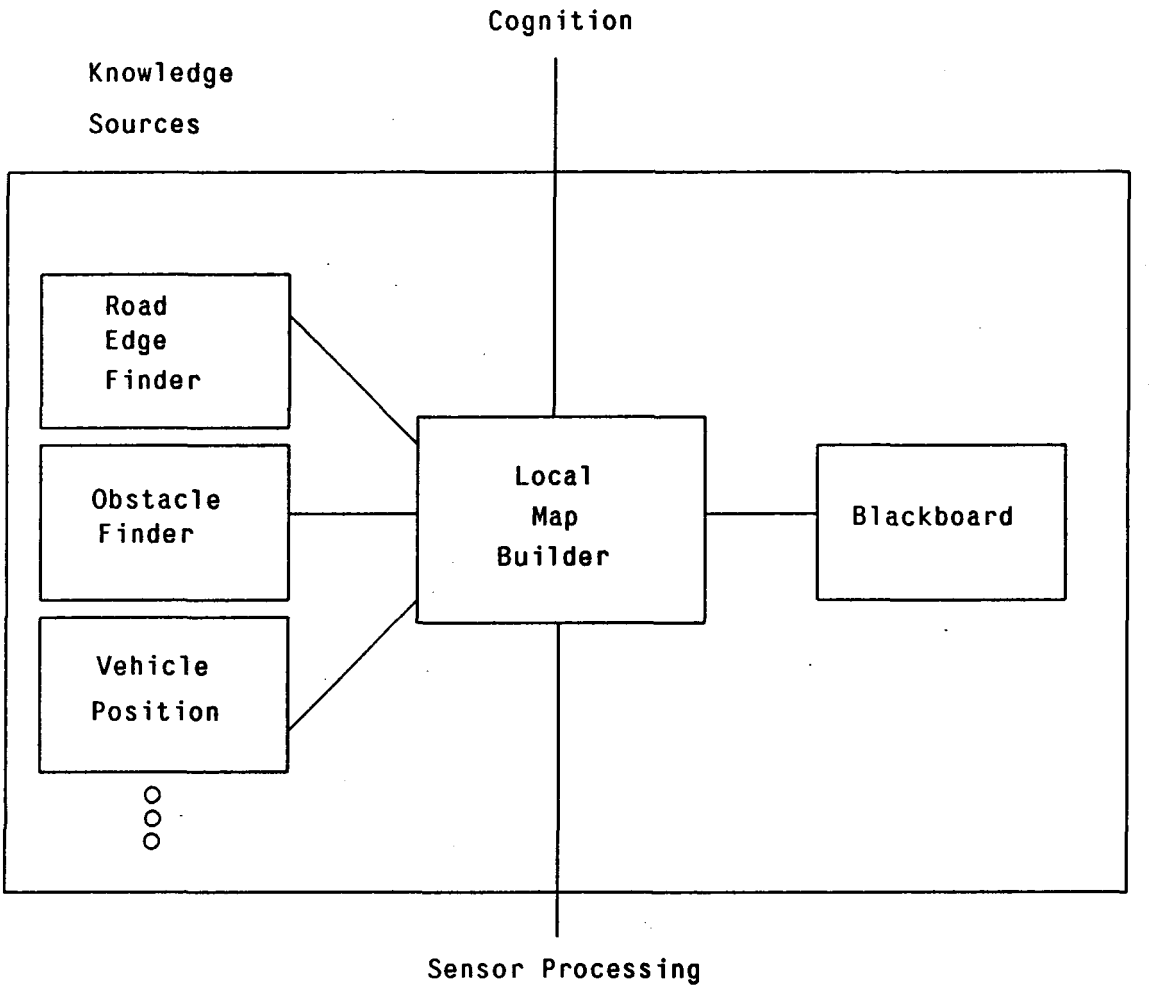

Figure 1-2: The Local Map Level

1.2 The Blackboard Software Package

The blackboard software package consists of software for implementing all three components of the local map level, including conventions and facilities for communication between the three components as well as processes in the cognition, sensor processing and virtual vehicle levels.

Blackboard and LMB

The blackboard is a database of *tokens*, data objects consisting of a set of *attribute-value pairs* determined by the token's *type*. The LMB manages the blackboard, servicing requests from KS, sensor, and cognition modules to store and retrieve tokens from the blackboard. The LMB is equipped with a pattern-matching mechanism, used by KS and other processes for recovering tokens. Additional tasks performed by the LMB include scheduling token requests, expiring old and uninteresting tokens, and transforming spatial data from one coordinate frame to another. The LMB is implemented as an independent module, separately compiled, running as a stand-alone process, complete with network and inter-process communication channels and primitives for communicating with other processes. The blackboard is implemented as a collection of data structures residing in the address space of the LMB.

Interface to LMB

The blackboard software package includes a user interface to the LMB that consists of a library of functions and data structures to be compiled with user-written modules, enabling them to store and retrieve tokens from the blackboard. The package includes mechanisms for defining, allocating, and deleting tokens and for reading and writing their attributes. Mechanisms are also provided for storing these tokens in the blackboard database, constructing patterns (*specifications*) for matching tokens, and retrieving tokens from the blackboard database using these specifications. This package includes the network and inter-process communication channels and primitives necessary for communication with the LMB, hiding these details from the module implementors. The sensor, KS, and cognition modules run as separate processes, communicating to the LMB through their module interfaces. Figure 1-3 illustrates the blackboard software configuration.

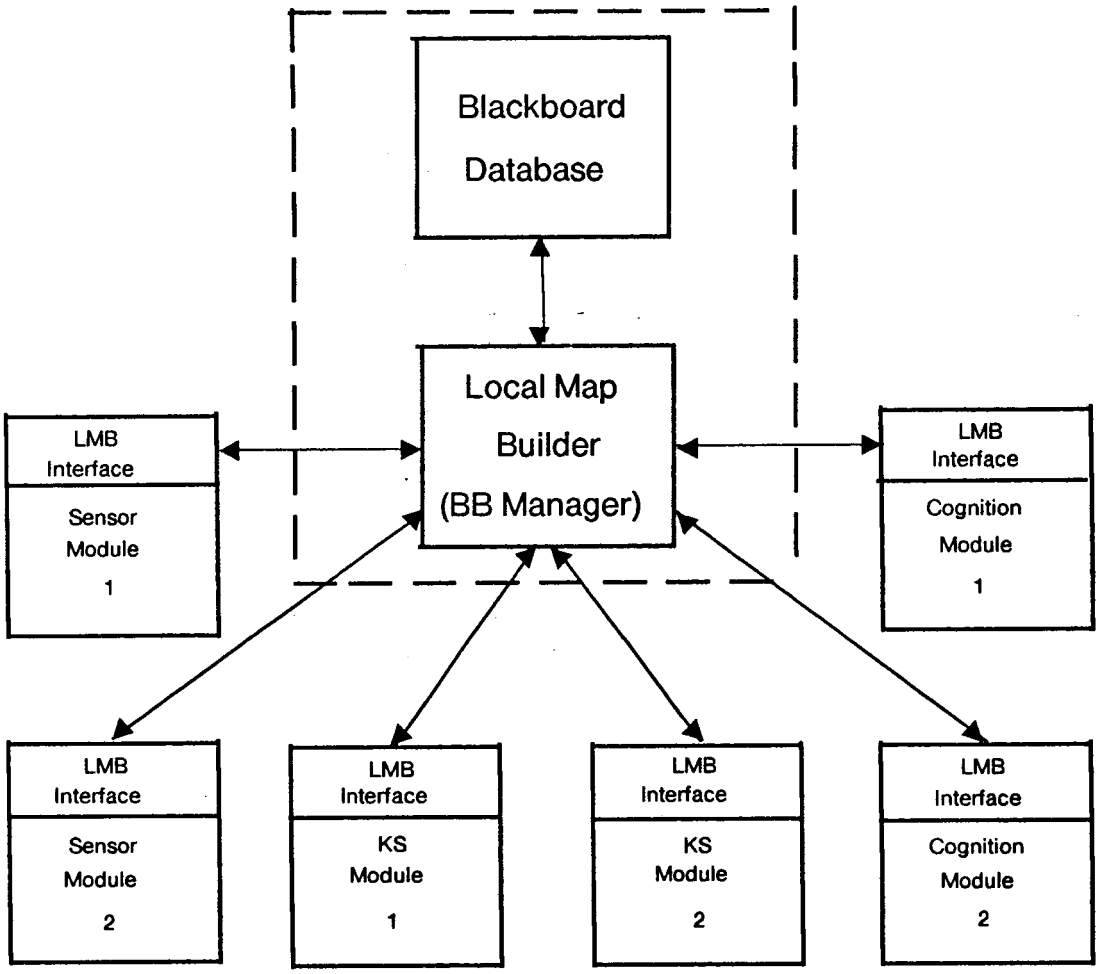

Figure 1-3: Blackboard software configuration

2. The User Interface

The blackboard software package provides the user with a library of C data structures and functions necessary for writing sensor, KS, and cognition modules for use in the blackboard system. More specifically, the package provides the following facilities:

- **Token manipulation:** mechanisms are provided for declaring, allocating, deleting, reading, and writing composite data objects (tokens) residing in the user's address space. Each token consists of a list of attribute-value pairs.

- **Coordinate frame manipulation:** mechanisms are provided for defining, changing, transforming, and deleting coordinate frames in which the world, vehicle, sensors, and physical objects are expressed.

- **Specification manipulation:** mechanisms are provided for building patterns used for matching and recovering tokens residing in the blackboard database. Each pattern is a boolean expression of functions and relations defined over data attributes. A token matches a pattern if its attributes satisfy the boolean expression.

- **Blackboard Communication:** mechanisms are provided for depositing tokens into the blackboard database and for retrieving them, either by pattern-matching specification or direct token addressing.

The inclusion file for user-written modules, *module.h*, can be found in Appendix I. In this document, "user" refers to a programmer implementing a module in the ALVan system.

2.1 Blackboard Tokens

2.1.1 Token Structure and Declarations

A *token* is a data unit capable of representing an object of any type (such as a road, an intersection, an obstacle, or a landmark) or instructions or status information to be passed between modules. Each token is composite, consisting of a set of subobjects known as *attributes*, or characteristics of the token. Attributes fall into three categories:

- *Internal attributes:* are common to all tokens regardless of type and are used by the system to manage the token. These attributes include a unique identification number, a token type, a generation number, a time stamp indicating which coordinate system was used to record the token's data, the time the token was deposited in the database, the time the token was last modified, a pointer to the module that created the token, and the token's location.

- *Local attributes:* are specific to a single token type. The number and types of local attributes vary from one token type to another. For example, tokens of type CAR might have a local attribute NUMBEROFDOORS, an integer defined to be the number of car doors. NUMBEROFDOORS has meaning only for tokens of type CAR.

- *Global attributes:* are common to more than one token type. For example, the global attribute SURFACEAREA might be common to tokens of types INTERSECTION, ROADUNIT, GRASSFIELD, etc.

Attributes have names and values. Attribute names are identifiers; the values may be of primitive, composite, or user-defined (UDT) types. Primitive types include integers, floating point numbers, booleans, enumerated types, and strings of characters. UDT's are streams of bytes unstructured to the blackboard package and interpreted by the user's modules. The two composite types are location and array. A location is defined to be set of three-dimensional coordinates (three floating point numbers) denoting a point, line, polygon, or "blob" and a coordinate frame giving the position and orientation of this point set relative to the world. The location data type and supporting data structures are described in section 2.2. An array is defined to be an ordered set of elements of any type (including array), which are indexed by integers. See section 4.2.8 for a description of the array type.

The set of attributes for a given token is determined by the token's *type*, which is a list of all of the attributes that define a kind of token. Token types are defined by the user and must be described in the template file, *template.init*. Attribute names and types, as well as enumerated types and UDTs, must also be defined in this file. This file is read by the BB manager at start-up to enable it to interpret tokens sent by sensor, KS, and cognition modules. See section 4.2.1 for a detailed description of *template.init*.

Figure 2-1 illustrates an example of a template file. In this example, SURFACETYPE and CONTROLTYPE are enumerated types. IMPORTANCETYPE is a UDT (user-defined type). Within the token type declaration for INTERSECTION, the attributes AREA, CONTROL, IMPORTANCE, and ROADS are local and are of types FLOAT, enumerated types CONTROLTYPE and IMPORTANCETYPE, and ARRAY of four INTs respectively. SURFACE and TRAVERSED are global attributes, and appear in the ROADUNIT token type declaration also. Although AREA appears in both token type declarations, the two local attributes are logically distinct. Note that the internal attributes are not declared in *template.init*, as they are automatically common to all token types.

2.1.2 Operations on Tokens

Tokens may be allocated and deleted with the functions Tnewtoken and Tfreetoken respectively. Token attributes (global, local, and internal) may be read and written with the function classes T?read and T?write respectively, where ? is an attribute designator, a single letter (i, f, b, s, e, u, and l) denoting the attribute type (integer, floating point, string, enumerated, UDT, and location). The types

```
/* Define enumerated types SURFACETYPE and CONTROLTYPE */
ENUM SURFACETYPE = { CONCRETE , ASPHALT , GRAVEL , DIRT } ;
ENUM CONTROLTYPE = { STOPSIGNS , TRAFFICLIGHT };

/* Define UDT IMPORTANCETYPE  */
UDT IMPORTANCETYPE;

/* Define global attributes SURFACE and TRAVERSED to be
   of types SURFACETYPE and BOOL respectively */
GLOBAL SURFACE : SURFACETYPE ;
GLOBAL TRAVERSED : BOOL ;

/* Define array ROADARRAY */
ARRAY ROADARRAY [4] OF INT ;

/* Define token type INTERSECTION with global attributes
   SURFACE and TRAVERSED and local attributes AREA,
   CONTROL, IMPORTANCE, and ROADS */
TOKEN INTERSECTION {
SURFACE : GLOBAL ;
AREA : FLOAT ;
TRAVERSED : GLOBAL ;
CONTROL : CONTROLTYPE ;
IMPORTANCE : IMPORTANCETYPE ;
ROADS : ROADARRAY ;
} ;

/* Define token type ROADUNIT with global attributes SURFACE
   and TRAVERSED and local attributes AREA, ROAD1, and ROAD2 */
TOKEN ROADUNIT {
SURFACE : GLOBAL ;
AREA : FLOAT ;
TRAVERSED : GLOBAL ;
ROAD1 : INT ;
ROAD2 : INT ;
} ;
```

Figure 2-1: Example of a token template file

integer and single-precision floating point are primitive types. Strings are arrays of characters. Booleans and enumerated types are stored as integers. UDTs are structures, consisting of an integer denoting the size of the data and a pointer to the data. The functions Unewudt, Uassignudt, and Ufreeudt allocate, assign data to, and de-allocate UDTs respectively. The functions Anewarray, A?read, A?write, and Afreearray allocate, read from, write to, and de-allocate arrays respectively. Operations on location data are discussed in section 2.2.

Figure 2-2 illustrates the use of these functions. Note that the attribute 'area' and token type 'intersection' in this program segment are referenced by ids. All identifiers in *template.init* including token types, attribute names, UDT, enumerated types, and enumerated scalars are assigned unique

```
TOKEN *t;        /* Declare pointer to token */
int intersectionid, areaid, digtime, areaval;

/* Initialize -- load the id's (see section 2.4.1) */
   ...

/* Allocate a new token of type 'intersection' for an
   image digitized at time 'digtime' */
t = Tnewtoken (intersectionid, digtime);

/* Write the 'area' attribute */
Tfwrite (t, areaid, 5.2);

/* Read the 'area' attribute */
areavalue = Tfread (t, areaid);

/* Free the token */
Tfreetoken (t);
```

Figure 2-2: Example of token operations

ids (integers) by the BB manager as the file is parsed. Identifiers naming user-defined functions are also assigned ids. These ids are passed to all modules during initialization for referencing identifiers. This procedure is discussed extensively in section 2.4.1.

2.2 Locations and Coordinate Frames

For a moving vehicle that avoids obstacles and navigates using landmarks, spatial data is very important; therefore, all tokens have an internal attribute TLOCATION of type *location*. A location is a collection of three-dimensional points to describe the shape of an object expressed in some coordinate frame. Depending on the nature of the represented object, one coordinate frame may be more appropriate another. For example, stationary objects such as landmarks are most suitably expressed in a geocentric, or world coordinate frame, while sensors mounted on the vehicle are best expressed in a vehicle-based frame. The following sections describe the structure of locations along with a powerful set of functions for manipulating them.

2.2.1 Location Data Type

A location is a set of three-dimensional points (a *point set*) expressed in a coordinate frame. A location's point set describes the shape of the represented object, while the coordinate frame specifies its position and orientation. Four types of point sets are provided for representing points, line segments, polygons, and point scatters. A *point* is a single three-dimensional (x, y, z) coordinate denoting a single position in space; a *line segment* is a pair of three-dimensional endpoints possibly denoting a road edge or traffic line; a *polygon* is a set of coplanar vertices possibly denoting the

two-dimensional perimeter of an obstacle or a section of a road; and a point *scatter* is an aggregate of three-dimensional points representing arbitrary shapes ranging from amorphous "blobs" to detailed objects. Coordinate frames are the topic of the next section.

```
LPOINTSET *pset1, *pset2;

/* Create a point set for a triangle */
pset1 = Lnewpointset (LPOLYGON, 3);

/* Add the three vertices */
Laddpoint (pset1, 1.0, 0.0, 0.0);
Laddpoint (pset1, 0.0, 1.0, 0.0);
Laddpoint (pset1, -1.0, 0.0, 0.0);

/* De-allocate the point set */
Lfreepointset (pset1);

/* Create a point set for a "blob" */
pset2 = Lnewpointset (LSCATTER, 6);

/* Add the six points */
Laddpoint (pset2, 3.2, 5.4, 2.1);
Laddpoint (pset2, 4.2, 2.1, 2.1);
Laddpoint (pset2, 5.5, 4.3, 7.1);
Laddpoint (pset2, 5.4, 2.0, 2.0);
Laddpoint (pset2, 3.5, 3.5, 1.0);
Laddpoint (pset2, 1.1, 1.2, 1.0);

/* De-allocate the point set */
Lfreepointset (pset2);
```

Figure 2-3: Making point sets

A number of functions are provided for defining shapes or point sets as illustrated in figure 2-3. The function Lnewpointset is used to create two shapes: a triangle and a "blob". The three vertices of the triangle and the six points of the blob are defined using the function Laddpoint. Finally, the shapes are de-allocated using Lfreepointset. See section 4.2.7 for details.

2.2.2 Coordinate Frames and Families

In addition to a point set for shape, each location has a coordinate frame or frames to describe the position and orientation of the location. There are three pre-defined frames: *world*, *vehicle*, and *null*. The world frame is geocentric, that is, affixed to the earth. It is useful for representing map data such as landmarks, roads, and intersections. The vehicle frame is egocentric, that is, affixed to the vehicle. It is useful for defining the positions and orientations of sensors such as cameras and sonar mounted on the vehicle. The vehicle frame is actually not a single coordinate frame, but is instead a *family* of frames. As the vehicle moves about, the position and orientation of the vehicle frame relative to the

world frame changes. The BB manager keeps a history of the vehicle's motion as a function of time. By specifying a particular time t_1, the transformation from the world frame to the vehicle frame at t_1 can be retrieved; thus locations expressed in the world frame can be compared to those in a vehicle frame. The null frame is simply an indicator that the position and orientation of a particular location is not known.

The user is not restricted to expressing all locations in world or vehicle frames; instead, the user may define *local* frames. A local frame is a coordinate frame defined relative to any other frame or family of frames, called the base frame. A local frame has meaning only within the context of the module that defined it, that is, no other module can express locations in such a frame unless they too define it locally. Local frames are convenient for expressing data in a "natural way", that is, in the same coordinate frame in which they were recorded. Figure 2-4 illustrates the declaration of a local frame to represent a sonar sensor. In order to make a local frame, a base frame and a transformation to this base frame must be specified. Since the sonar sensor is mounted on the vehicle, the vehicle frame family (VEHICLEFF) is appropriate. The transformation to this frame family is given by the pose 'ppose'. The function FmakelocalfromFF creates the frame. See section 4.2.6 for details.

```
FRAME *frm;
POSE ppose = { /* Pose definition */ };

/* Declare a local frame */
frm = FmakelocalfromFF (VEHICLEFF, ppose);
```

Figure 2-4: Declaring a local frame

Global frames are identical to local frames with the exception that the coordinate frame is sent to the blackboard and is available to any other module. Global frames are useful for expressing locations that are accessed by more than one module, such as the structure of a building or other landmarks. Like local frames, global frames defined by a transformation relative to a base frame or family (which must not be local). Figure 2-5 illustrates the declaration of a global frame for a building. The frame is declared relative to the world frame using the function FmakeglobalfromF. See section 4.2.6 for details.

```
FRAME *frm;
POSE ppose = { /* Transformation pose */ };

/* Declare the global frame family */
frm = FmakeglobalfromF (WORLDF, ppose);
```

Figure 2-5: Declaring a global frame

Frames are useful for expressing objects such as a building or sonar sensor that remain fixed relative to a reference frame, but for objects such as a camera on a pan/tilt mount or a moving car, frame families are needed. Consider first the movable camera. The camera's position and orientation relative to the vehicle is a function of the roll, pan, and tilt angles of the camera mount. Rather than declaring each desired configuration as an individual frame (there is a continuum), a local frame *family* parameterized by the roll, pan, and tilt angles of the mount can be used. A local frame family declaration requires a base frame and a user-defined function, that returns a transformation pose from one of the frames in the family to the base. This transformation function "selects a member" of the family as a function of the parameters. Figure 2-6 illustrates the declaration of a local frame family for a movable camera. The family 'frmfam' is local relative to the vehicle (VEHICLEFF). The transform function 'transform' takes '3' arguments (roll, pan, and tilt) and returns a pose. This transform function does not require time as a parameter. See section 4.2.5 for details.

```
FRAMEFAMILY *frmfam;
POSE *transform ();

/* Declare a local frame family relative to the vehicle
   as a function of roll, pan, and tilt angles of the
   camera mount */
frmfam = FFmakelocalfromFF (VEHICLEFF, transform, 3, LNOTIME);

...

/* Declare the transformation function */
transform (roll, pan, tilt)
float roll, pan, tilt; {

/* Body of the function */

}
```

Figure 2-6: Declaring a local frame family

In addition to declaring frame families locally for sensors, the user may need to declare frame families that are accessible to other modules. For example, a vision module detects a moving car and needs to make this information available to an obstacle avoidance module. For such tasks, global frame families are provided. Like local frame families, global families have a base or reference frame; the transformation, however, differs considerably. Instead of a local function defined over a continuum of parameter values, transformations for global families reside in a look-up table. This table or *parameter list*, contains some number of time-pose pairs. A time-pose pair is a global time value 't' and a pose 'p' that provides the transformation from a single frame in the global family at time 't' to the base frame. Note that global frame families have only one parameter (time) and are defined over a finite number of values of time. Modules may recover a global frame family from the blackboard and "update" the transformation by adding time-pose pairs.

Figure 2-7 illustrates the declaration and modification of a global frame family for representing a moving car. The car is seen initially at time t = 1.0 with a transformation 'pose1' from the world frame. A global frame family 'frmfam' is created and is sent to the blackboard. At time t = 2.0, the car is seen relative to the world with a transformation 'pose2'. The global frame family is recovered from the blackboard, and the new time-pose pair is added to the parameter list.

```
FRAMEFAMILY *frmfam;
FFPARAMLIST *plist;
POSE pose1 = { /* Transformation matrix */ };
POSE pose2 = { /* Transformation matrix */ };

/* Make a parameter list */
plist = FFmakeparamlist ();

/* Add a time-pose pair */
FFaddtoparamlist (plist, 1.0, pose1);

/* Make a global frame family using plist */
frmfam = FFmakeglobalfromF (WORLDF, plist);

/* Retrieve the frame family */
FFgetglobalupdate (frmfam, FFLOCK);

/* Retrieve the parameter list */
plist = FFreadparamlist (frmfam);

/* Add a time-pose pair */
FFaddtoparamlist (plist, pose2, 2.0);

/* Write the parameter list */
FFwriteparamlist (frmfam, plist);

/* Replace the frame family */
FFputglobal (frmfam);
```

Figure 2-7: Declaring and modifying a global frame family

As previously described, local and global frames can be created by specifying a transformation and a base frame or family. Frames can also be created by "selecting" a member of a frame family. For example, assume that a camera frame relative to the vehicle is needed for the mount tilted at 20 degrees with no roll nor pan. Instead of creating a frame relative to the vehicle with the appropriate transform, a frame can be selected from the camera frame family by specifying values for the roll, pan, and tilt parameters as illustrated in figure 2-8. Now assume that a camera frame relative to the world frame is needed. Since the vehicle is moving with respect to the world frame, the user must select frames from both the camera family AND the vehicle family. As illustrated in figure 2-8, adding a value for the vehicle's parameter (time) to the function call does the job. See section 4.2.6 for details.

```
FRAME *frm1, *frm2;
FRAMEFAMILY *frmfam;

/* Declare frmfam as a camera frame family */
...

/* Select a frame relative to the vehicle from the camera
   frame family for roll=0.0, pan=0.0, tilt=20.0 */
frm1 = FselectlocalfromFF (frmfam, FFTIME, 3, 0.0, 0.0, 20.0);

/* Select a frame relative to the world from the camera
   and vehicle frame families for roll=0.0, pan=0.0,
   tilt=20.0, and time=47 */
frm2 = FseleclocalfromFF (frmfam, FFTIME, 47, 0.0, 0.0, 20.0)
```

Figure 2-8: Declaring frames from frame families

2.2.3 Operations on Locations

Locations are created by binding a shape (pointset) to a coordinate frame or family in which the shape is expressed. The function LmakelocationF creates locations as illustrated in figure 2-9. In this figure a location is made for a line segment expressed in the world frame. The function Lconvertloc converts locations from one coordinate frame to another. In figure 2-9 a new location is created (loc2) in which the original line segment is expressed in a new global frame. If a location cannot be converted to another coordinate frame, that is, no known transformation exists between the base frames or the transformation is ambiguous, a value of NULL is returned.

```
FRAME *frm;
LPOINTSET *pset;
LOCATION *loc1, *loc2;
POSE pose = {2.2,2.4,5.4,8.6,2.1,9.8,9.8,3.5,1.1};

/* Make a point set for a line segment */
pset = Llineseg (0.0, 0.2, 0.1, 4.2, 4.3, 2.9);

/* Make a location using this point set and the
   world frame */
loc1 = LmakelocationF (WORLDF, pset);

/* Make a local frame based on the world frame */
frm = FmakelocalfromF (WORLDF, pose);

/* Convert loc1 to this new frame */
loc2 = Lconvertloc (loc1, frm);
```

Figure 2-9: Creating and converting a location

In addition to making and converting locations, a number of functions exist for comparing locations and computing secondary features of them. Functions are provided for computing the distance

between two locations or their overlap. Other functions compute the centroid, area, diameter, orientation, convex hull, and minimum bounding rectangle of a location. This latter class of functions perform two-dimensional operations in the x-y plane; thus, for three-dimensional locations the z-coordinate is ignored. See section 4.2.7 for details.

2.3 Pattern-Matching Specifications

2.3.1 Building Specifications

Specifications are patterns for matching and recovering tokens stored in the blackboard database. Each pattern is a boolean expression represented as a tree. The vertices of the tree are functions or relations whose sons are input parameters and whose parent receives the output. The leaf nodes of the tree are constants or token attribute names. The root node must return a boolean. A token "matches" a pattern when the values of its attributes satisfy the pattern, that is, after the attribute values are inserted into the corresponding leaf nodes, the tree evaluates to TRUE.

Specifications (specs) are implemented as a header node of type SPEC pointing to a tree of SPECNODES. A spec node is a structure containing a type field and a body field. The type field indicates the function, pattern-matching attribute, or constant denoted by the specnode. The body field contains either a function id, an attribute id, or constant data. The function Snewspec returns a pointer to a spec header node which points to the root of the given spec node tree. Constant leaf nodes are constructed using the functions S?const, where '?' is a single character (i, f, b, s, e, u, and l) denoting integer, floating point, boolean, string, enumerated, UDT, and location types. These functions return spec nodes for constants of each type. The function Sattribute returns a spec node for a given token attribute. See sections 4.3.1 and 4.3.2 for details.

The functions for contructing vertices in the spec tree are numerous and can be found in sections 4.3.3 through 4.3.7. Functions exist for boolean operations (AND, OR, NOT), algebraic operations (ADD, SUBTRACT, MULTIPLY, DIVIDE), comparative operations (EQUAL, NOT EQUAL, LESS THAN, GREATER THAN), string operations (SUBSTRING, REGULAR EXPRESSION SEARCH), and location operations (AREA, CENTROID, DISTANCE, etc.). Each of these functions takes some number of spec nodes as input and returns a single specnode.

Figure 2-10 illustrates the construction of two specs. In the first spec, sp1, there are two attributes to match: surface-id and intrsctn-area-id. Surface-id is the id for the global attribute SURFACE which can match tokens of any type having this attribute. Intrsctn-area-id is the id for the local attribute AREA of token type INTERSECTION. Therefore, spec sp1 can match only tokens of type

```
SPEC     *sp1, *sp2;
LPOINTSET *pset;
LOCATION *loc;
int intrsctn-area-id, surface-id, concrete-id, location-id;

/* Initialize -- load ids (see Section 2.4.1) */
   ...

/* Build a spec for matching all concrete intersections
   larger than 100 square feet. */
sp1 = Snewspec (Sand (Sgreater (Sattribute (intrsctn-area-id),
                              Sfconstant (100.0)),
                      Sequal (Sattribute (surface-id),
                              Seconstant (concrete-id))));

/* Build a spec for matching a token of any type within
   5 feet of world location (x=25, y=32, z=0). */

/* First create a point set and location. */
pset = Lpoint (25.0, 32.0, 0.0);
loc = Lmakelocation (WORLDF, pset);

/* Build the spec. */
sp2 = Snewspec (Sless (Sdistance (Sattribute (location-id)),
                              Slocation (loc)),
                       Sfloat (5.0));
```

Figure 2-10: A program segment that constructs two specifications

INTERSECTION. Spec sp2, however, can match tokens of *any* type, since location-id is the id for the internal attribute LOCATION, common to tokens of all types. Note that the functions for building specs do not actually compute their named operations; instead, they build a structure that indicates to the BB manager which functions to invoke on which parameters.

Figure 2-11 shows a token of type intersection matching spec sp1 from figure 2-10. The attribute values 200.0 and CONCRETE are inserted into the spec tree leaf nodes AREA and SURFACE respectively. Since 200.0 is greater than 100.0, the function Sgreater returns TRUE. Likewise, since CONCRETE equals CONCRETE, the function Sequal returns TRUE. Sand returns TRUE with two TRUE inputs. Since the spec tree returns TRUE, the token matches the spec. Note that the spec matches all tokens of type intersection with an area greater than 100.0 and a surface of type concrete.

Suppose a user needs to build a number of similar specs. Instead of building each spec separately, a "generic" spec can be built with "slots". Slots are place-holders that point to other specs. By changing the value of a slot, a spec can be changed without building a new one (i.e. for items that change over time). The functions Sindirect and Spointer allow the user to embed "slots" in a spec. The function Sindirect "attaches" the slot to the tree, and Spointer "points" it to a filled spec node.

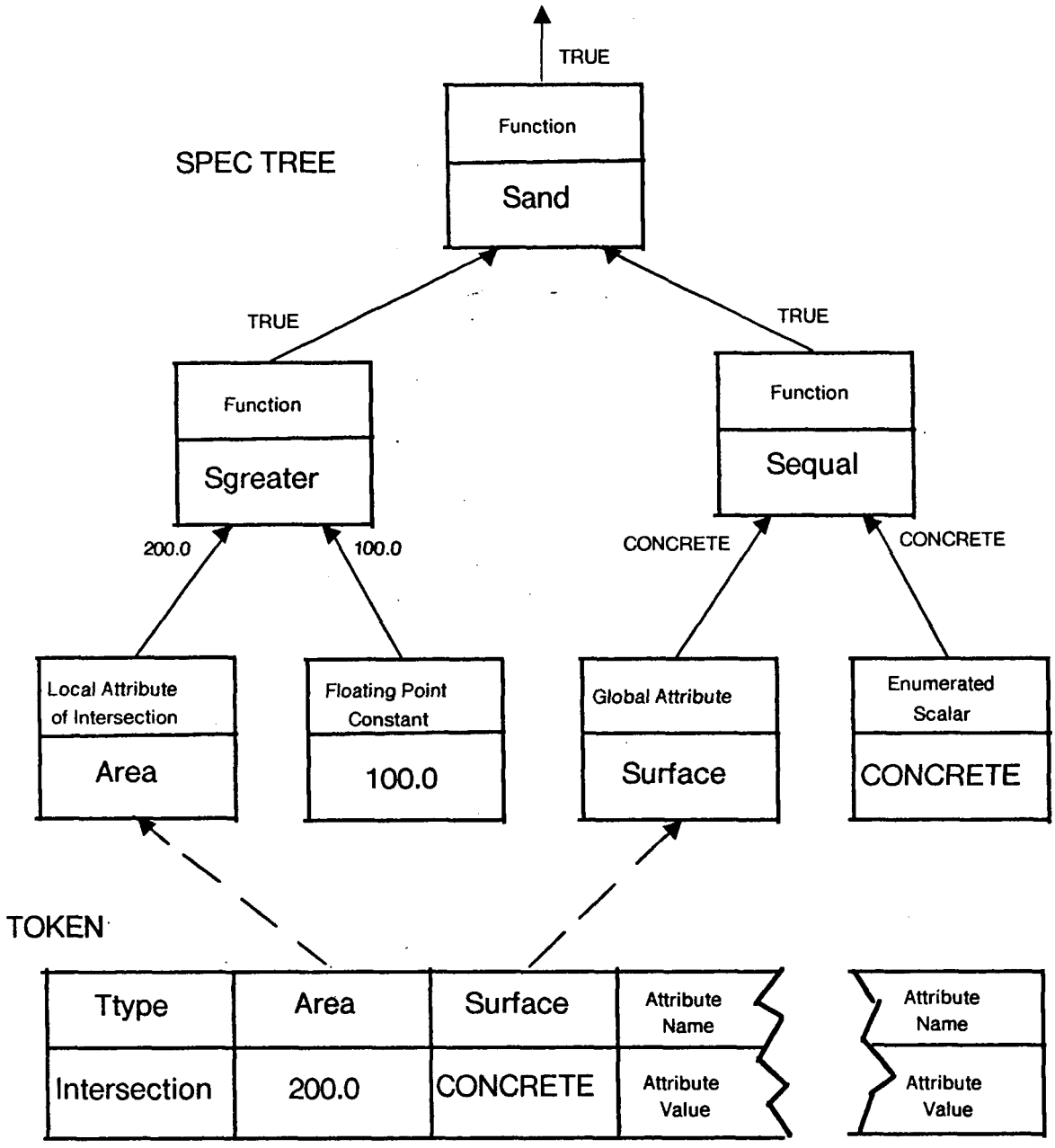

Figure 2-11: Example of a token matching a spec

Figure 2-12 illustrates this method. In this example, a spec is repeatedly changed to recover intersections with surface types matching a set of road unit tokens. See section 4.3.9 for details.

The module writer is not constrained to using the spec functions provided by the blackboard software package. The package provides a facility for handling user-defined functions (UDF's), that

```
SPEC *sp1, *sp2;
SPECNODE *slot;
int intrsctn-area-id, surface-id, concrete-id,
    roadunit-id, ttype-id;

/* Initialize -- load ids (see section 2.4.1) */
...

/* Build a spec to recover all road units */

sp1 = Snewspec (Sequal (Sattribute (ttype-id),
                        Siconst (roadunit-id)));

/* Build a spec to recover all intersections with
   a surface of type <x>, where <x> is a slot */
sp2 = Snewspec (Sequal (Sattribute (surface-id),
                        slot = Sindirect ()));

/* Send spec sp1 (see section 2.3.2) */
...

/* Loop -- for each matched token 't' do: */

        /* Store the surface type of the road into the slot */
        Spointer (slot, Seconst (Teread (t, surface-id)));

        /* Send spec sp2 and recover and process all
           intersections of the same surface type */
        ...
```

Figure 2-12: *Example of building a spec with a slot*

is, functions written by the user and invoked by the BB manager during the matching process. UDF's are defined in the file *functions.c*. This file is compiled separately and then linked with the BB manager. During module initialization, the BB manager passes ids for the function names to the module (see section 2.4.1). Passing these ids to the function Sudf enables the user to reference a UDF from within a spec tree. See section 4.3.10 for details.

2.3.2 Managing Specification Lists

Modules request tokens from the blackboard via specification lists. A specification list is a set of specs ORed together, that is, a token satisfies a specification list if it matches at least one of the specifications in the list. The function Snewlist allocates spec lists, Sfreelist frees a list, and Saddspec adds a spec to a spec list. See figure 2-13 for a program segment that uses spec lists.

In this program segment, LIST1 is passed to Snewlist. This user-supplied id number is bound to the spec list and is used to refer to the spec list in subsequent communication with the BB manager. The user must take care not to assign the same id number to more than one spec list. The user also

```
#define LIST1    1

SPECLIST         *sl;
SPEC             *sp1, *sp2;

/* Build specs sp1 and sp2 */

...

/* Create a spec list and add specs sp1 and sp2 to it */

sl = Snewlist (LIST1);
Saddspec (sl, sp1, 1, BBLOCK);
Saddspec (sl, sp2, 2, BBNOLOCK);

/* Free the spec list */

Sfreelist (sl);
```

Figure 2-13: Program segment that uses spec lists

supplies ids 1 and 2 for specs sp1 and sp2 respectively. These ids are used by the BB manager to inform the module which spec in the list a given token matches. The option BBLOCK instructs the BB manager to lock a token that matches the registered specification, that is, refuse other modules permission to modify the token until the locking module unlocks it. The BBNOLOCK option instructs the BB manager to allow modifications. See section 4.3.11 for details.

2.4 Communicating with the BB

2.4.1 Establishing Connections to the BB

The function BBinit is provided to open communication with the blackboard manager. BBinit sends a message to the blackboard manager, providing it with a module name and a network address. The BB manager logs the name and address in the module table and sends a list of identifiers and their corresponding identification numbers back to the module. These identifiers are those defined in the *template.init* file (see Appendix II). All token type, attribute type, enumerated type, UDT, UDF, and scalar names are bound in this way to IDs (integers), because IDs offer a faster and more efficient mechanism for referencing these items than strings.

BBinit must be invoked before any other BB function. The module is responsible for allocating storage for these ids as well as a table for mapping the identifiers to the id addresses. IDs should be declared as integers; a set of macros is provided in *module.h* for building the table. The macros BINDTOKENTYPEID, BINDLOCALATTRID, BINDGLOBALATTRID, BINDINTERNALATTRID, BINDFUNCTION, BINDUDTID, BINDENUMTYPEID, and BINDSCALARID bind names for token types,

local, global, and internal attribute types, UDF, UDT, enumerated types, and enumerated-type scalars (respectively) to IDs. See sections 4.4.1 and 4.4.2 for details. See figures 2-14 and 2-15 for a program segment that uses these macros.

```
/* Allocate integers for the ids of interest */

int     intersection-id, roadunit-id, surface-id,
        traversed-id, intrsctn-area-id,
        intrsctn-control-id, intrsctn-location-id,
        roadunit-area-id, surfacetype-id,
        controltype-id, concrete-id, asphalt-id,
        gravel-id, dirt-id, stopsigns-id,
        trafficlight-id;

/* Build the mapping table */

BEGINIDLIST

/* Get the id for the intersection token type */
BINDTOKENTYPEID("intersection",intersection-id)

/* Get the ids for the interesting attributes of token
   type intersection */
BINDLOCALATTRID(intersection-id,"area",intrsctn-area-id)
BINDLOCALATTRID(intersection-id,"control",intrsctn-control-id)

/* Get the id for the road unit token type */
BINDTOKENTYPEID("roadunit",roadunit-id)

/* Get the id for the interesting attribute of token
   type road unit */
BINDLOCALATTRID(roadunit-id,"area",roadunit-area-id)

/* Get the ids for the global attributes */
BINDGLOBALATTRID("surface",surface-id)
BINDGLOBALATTRID("traversed",traversed-id)

/* Get the id for the internal attribute of interest */
BINDINTERNALATTRID("location",intrsctn-location-id)

/* Get the id for the enumerated type surfacetype */
BINDENUMTYPEID("surfacetype",surfacetype-id)
```

Figure 2-14: Program segment that uses the macro facility for recovering ids. The segment is continued in figure 2-15

As shown in this program segment, the user need not request the ids for all identifiers in *template.init*. The macro BINDLOCALATTRID requires a token type id along with the attribute name. This id is required to distinguish between local attributes with the same name in different token types. Although internal attribute names do not appear in *template.init*, they can be recovered nonetheless with the macro facility (as seen with the location attribute).

```
/* Get the ids for the scalars of surfacetype */
BINDSCALARID(surfacetype-id,"concrete",concrete-id)
BINDSCALARID(surfacetype-id,"asphalt",asphalt-id)
BINDSCALARID(surfacetype-id,"gravel",gravel-id)
BINDSCALARID(surfacetype-id,"dirt",dirt-id)

/* Get the id for the enumerated type controltype */
BINDENUMTYPEID("controltype",controltype-id)

/* Get the ids for the scalars of controltype */
BINDSCALARID(controltype-id,"stopsigns",stopsigns-id)
BINDSCALARID(controltype-id,"trafficlight",trafficlight-id)

/* End the list of macros *

ENDIDLIST

main () {

/* Module declarations */
...

/* Establish communication with the BB manager and bind ids */
BBinit();

/* Module body */
...
```

Figure 2-15: Program segment that illustrates the macro facility continued from figure 2-14

2.4.2 Depositing New Tokens in BB

New tokens are deposited in the blackboard database with the function BBputtoken. The BB manager assigns the token a unique id number, which is returned by the function. The BB manager writes this id into the token's internal attribute Tid. All of the remaining internal attributes are also written at this time, including a generation number (Tgen) of 1, the time the token was inserted into the blackboard (Titime), and the time it was last modified (Tmtime). Tmtime is initially set to Titime. See section 4.4.4 for details.

Figure 2-16 illustrates the use of this function. A token is created, attributes are written, and the token is sent to the blackboard.

```
TOKEN *t;
LPOINTSET *pset;
LOCATION *l;
int intersectionid, digtime, tlocationid;

/* Initialize -- load ids (see section 2.4.1) */
...

/* Allocate a new token of type 'intersection' for an
   image digitized at time 'digtime' */
t = Tnewtoken (intersectionid, digtime);

/* Create a polygon location with four vertices */
pset = Lnewpointset (LPOLYGON, 4);

/* Load the four points */
Laddpoint (pset, 10.2, 11.5, 0.0);
Laddpoint (pset, 10.6, 23.2, 0.0);
Laddpoint (pset, -11.9, 10.4, 0.0);
Laddpoint (pset, -10.9, 27.1, 0.0);

/* Make the location */
l = Lmakelocation (WORLDF, pset);

/* Write the 'Tlocation attribute */
Tlwrite (t, tlocationid, l);

/* Send the token to the BB */
tid = BBputtoken (t);
```

Figure 2-16: Send a new token to the BB

2.4.3 Recovering Tokens from the Blackboard Using Specs

As previously discussed, specs and spec lists are used for matching and recovering tokens from the blackboard. The software package provides functions for sending spec lists to the BB manager, along with options that dictate the mode of synchronization for token recovery between the BB manager and the module and options that instruct the BB manager when to expire spec lists. There are two primary protocols:

- **Standing spec lists:** The calling module sends a standing spec list to the BB manager and then resumes execution without blocking. Whenever a token appears in the blackboard that matches the spec list, it is immediately sent to the module. The module suspends its current task and jumps to an interrupt routine to process the token, then resumes execution normally. The spec list remains active to match new or modified tokens. Standing specs lists are a convenient facility for matching urgent tokens, that is, tokens that require immediate action regardless of when or how often arrive in the local map.

- **One-shot spec lists:** The calling module sends a one-shot spec list to the BB manager and blocks. All tokens matching the one-shot spec list at the time of its arrival at the BB

are sent back to the module and deposited in token queue. The calling module is unblocked, resumes execution, and is free to recover the tokens from the queue. If there are no matching tokens in the BB database at the time the spec list arrives, the BB manager unblocks the calling module either immediately or after a matching token is eventually deposited, depending on whether the user specifies the BBNOWAIT or BBWAIT option respectively. Once a token or set of tokens matches a one-shot spec list, the BB manager deletes the spec list. One-shot spec lists are convenient for specifying what tokens a module needs *currently*. They force tightly-coupled synchronization between calling modules and the BB manager.

The functions for sending standing and one-shot specs lists are BBsendstandingspec and BBsendoneshotspec. In addition to a spec list, BBsendstandingspec requires a function for processing matching tokens and BBsendoneshotspec requires a waiting option of BBWAIT or BBNOWAIT.

Tokens matching a one-shot spec list are deposited in the token queue. The function BBgettoken retrieves tokens from the queue. It returns a value of BBEMPTY if the queue is empty. See sections 4.4.3 and 4.4.4 for details. Figure 2-17 a program segment that sends spec lists and receives tokens.

2.4.4 Sending and Receiving Tokens by ID

Thus far we have discussed the means for depositing new tokens in the database and recovering existing tokens via specifications. If a module knows the ID of a token in the database (i.e. the module deposited it or examined it previously), it can use the function BBgetidtoken to recover the token, thus avoiding the overhead of matching specs to tokens. As with specifications, the module must specify BBLOCK or BBNOLOCK with the function call.

Once a token has been recovered from the database (either by spec or id) with the locking option, the module may modify the token and replace it with the function BBreplacetoken. Note that this differs from BBputtoken, as it overwrites an existing token rather than creating a new one. If the module decides not to modify the token, it may unlock it with BBunlocktoken. Finally, a module may delete a token with BBdeletetoken. See section 4.4.5 for details.

2.5 Implementation Strategies

2.5.1 Module Structure

The module interface is a set of routines and data structures for compilation with the user's module, providing the module writer with routines for manipulating tokens and specifications and for communicating with the BB manager. A block diagram for the implementation of the module interface is depicted in figure 2-18. The components of figure 2-18 are individually described below:

```
#define TRUE    1

int sid, slid;

/* Initialize -- get ids */
...

/* Build standing spec list sl1 */
...

/* Send sl1 to the BB manager */
BBsendstandinglist (sl1, urgentfunc);

/* Begin primary task */
while (TRUE) {

        /* Build one-shot spec list sl2 */
        ...

        /* Send sl2 to the BB manager and return
           with or without a match */
        BBsendoneshotlist (sl2, BBNOWAIT);

        /* Recover all available tokens */
        while ((t = BBgettoken (&sid, &slid)) != BBEMPTY) {

                /* Process the token */
                ...
        }
}

/* Function for processing urgent tokens */
urgentfunc (t, sid, slid)
TOKEN *t;
int sid, slid; {
        /* Process the token */
        ...
}
```

Figure 2-17: Program segment that sends spec lists and receives tokens

- *ID list:* a block of storage defined by the user for holding ID numbers. This list is downloaded by the *input message processor* when it receives the initialization message from the BB manager.

- *User routines:* the heart of the user program. This block contains the code that implements the sensor, KS, or cognition module. These routines access the module interface through *token, spec, and BB function* calls.

- *Urgent spec handling functions:* the functions specified by the user for processing urgent tokens, that is, tokens that match a standing spec list.

- *Spec functions:* the group of functions for building specifications.

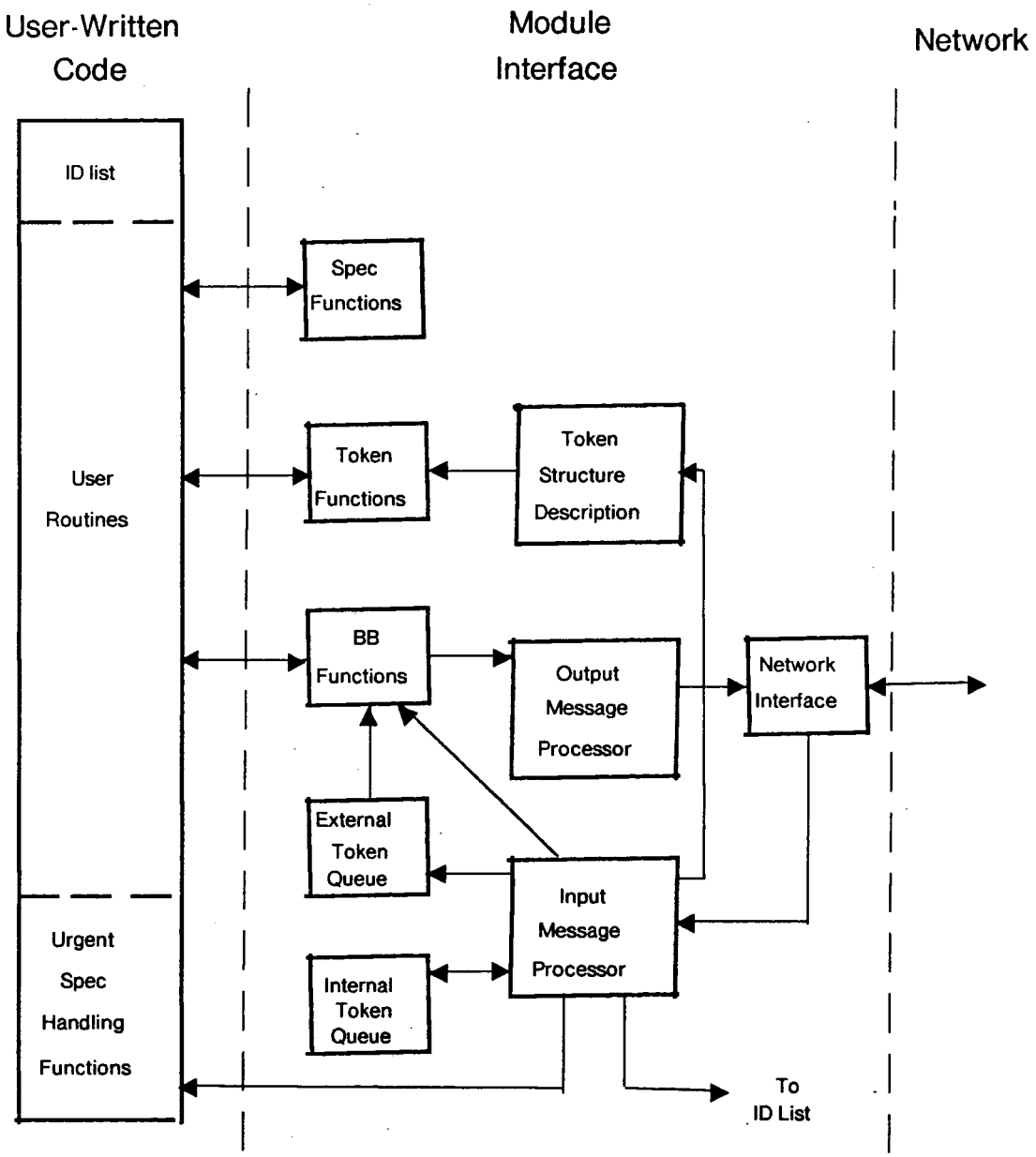

Figure 2-18: Module interface block diagram

- *Token functions:* the group of functions for allocating, deleting, reading, and writing tokens. These functions access the *token structure description* data structure.

- *BB functions:* the group of functions that sends and receives tokens, specs, and other messages to and from the BB manager. All output messages are routed through the *output message handler*. Tokens that match a one-shot spec are recovered from the *external token queue*. Individually addressed tokens and error messages are recovered from the *input message processor*.

- *Token structure description:* a data structure that encapsulates all the information in *template.init*. *Token functions* access for reading and writing token attributes. The structure is downloaded by the BB manager via the *input message processor* during initialization.

- *External token queue:* a queue of tokens sent by the BB manager that matched a one-shot spec list. The queue is loaded by the *input message processor*. *BB functions* extract tokens from the queue.

- *Internal token queue:* a queue of tokens sent by the BB manager that matched a standing spec list. The *input message processor* uses this queue as temporary storage before passing the urgent tokens on to an *urgent spec handling function*.

- *Output message processor:* a set of routines that receives tokens, specs, and other data from the *BB functions*, packs them into messages, and passes them on to the *network interface* for transmission to the BB manager.

- *Input message processor:* a set of routines for handling all input messages. These routines download the *ID list* and the *token structure description* from the initialization message. Individually addressed tokens are passed directly to the calling *BB function*, while tokens matching a one-shot spec list are inserted into the *external token queue*. Tokens matching a standing spec list are inserted into the *internal token queue* to await processing by an *urgent spec handling routine*.

- *Network interface:* a set of routines for sending and receiving messages to and from the BB manager over a network or through inter-process communication channels.

2.5.2 The Algorithm

The user initializes the module interface by calling BBinit. BBinit sends the module's name and address and a list of identifiers to resolve to the BB manager via the *output message processor* and blocks. The BB manager replies with a list of ID numbers and token type descriptions. The arrival of the return message generates a software interrupt and control is passed to the *input message processor*. The input message processor downloads the *ID list* and the *token type descriptions*. BBinit unblocks and returns.

The user accesses the module interface via the *token*, *spec*, and *BB* functions. The *BB functions* that send one-shot spec lists or access tokens individually block until a return message is received. All return messages are handled by the *input message processor*. Whenever a message arrives from the BB manager, a software interrupt is generated, and control is passed to the *input message processor*. The following action is taken, depending on the message type:

- *Individually addressed token:* the token is passed directly to the calling *BB function*, and the function returns.

- *Error message or status information:* the message is passed directly to the calling *BB function*, and the function returns.

- *Frames or frame family information:* the message is passed directly to the calling frame or frame family function and execution resumes.

- *Token matching a one-shot spec list:* the token is inserted into the *external token queue*, the calling *BB function* returns, and the user is free to recover the token using the appropriate *BB function*.

- *Token matching a standing spec list:* the token is inserted into the *internal token queue* to await processing by an *urgent spec handling function*.

Once the *input message processor* has processed all incoming messages, it extracts each token from the *internal token queue* one by one and invokes the appropriate *urgent spec handling function* to process the token. The *input message processor* relinquishes control once the queue is empty.

3. The Blackboard Manager

The blackboard manager is a self-contained module, compiled separately and linked with the user-defined functions (UDF's). It manages the blackboard database, a collection of data structures that resides in the blackboard manager's address space. The blackboard database provides storage for tokens, inserted by knowledge sources, conceptual, and virtual vehicle processes. The BB manager services requests to insert, modify, and delete tokens in this database. Processes requesting tokens can do so using patterns (specifications), requiring the BB manager to attempt to match these patterns with tokens in the database. Other tasks include scheduling token requests which arrive asynchronously over the network or through inter-process channels, expiring old tokens too far from the vehicle to be of interest, and recording the history of vehicle positions over time.

3.1 BB Manager Structure

Figure 3-1 shows a block diagram for the structure of the BB manager. The individual components are:

- *Network interface:* a set of routines for sending and receiving messages to and from user-written modules over a network or through inter-process channels.

- *Input message processor:* a set of routines for handling all input messages. All messages including tokens, specifications, error, and control are routed through this module.

- *Output message processor:* a set of routines that transmits messages from the *input message processor* to user-written modules.

- *BB initialization routine:* a routine that reads *template.init*, loads *token structure description* and *ID list*, and instructs the *input message processor* to accept incoming messages.

- *Pattern-matching functions:* a large collection of functions used by the *input message processor* to match specifications to tokens.

- *UDFs:* a module of user-defined functions linked with the BB manager and used in the same way as *pattern-matching functions*.

- *Token database:* all tokens residing in the blackboard database. Tokens are stored and retrieved via the *input message processor*.

- *Pending speclist table:* a table of all pending speclists awaiting a token to match. This table is managed by the *input message processor*.

- *Token structure description:* a data structure that encapsulates all information in *template.init*. This data structure is downloaded by the *BB initialization routine* and is used by the *input message processor*.

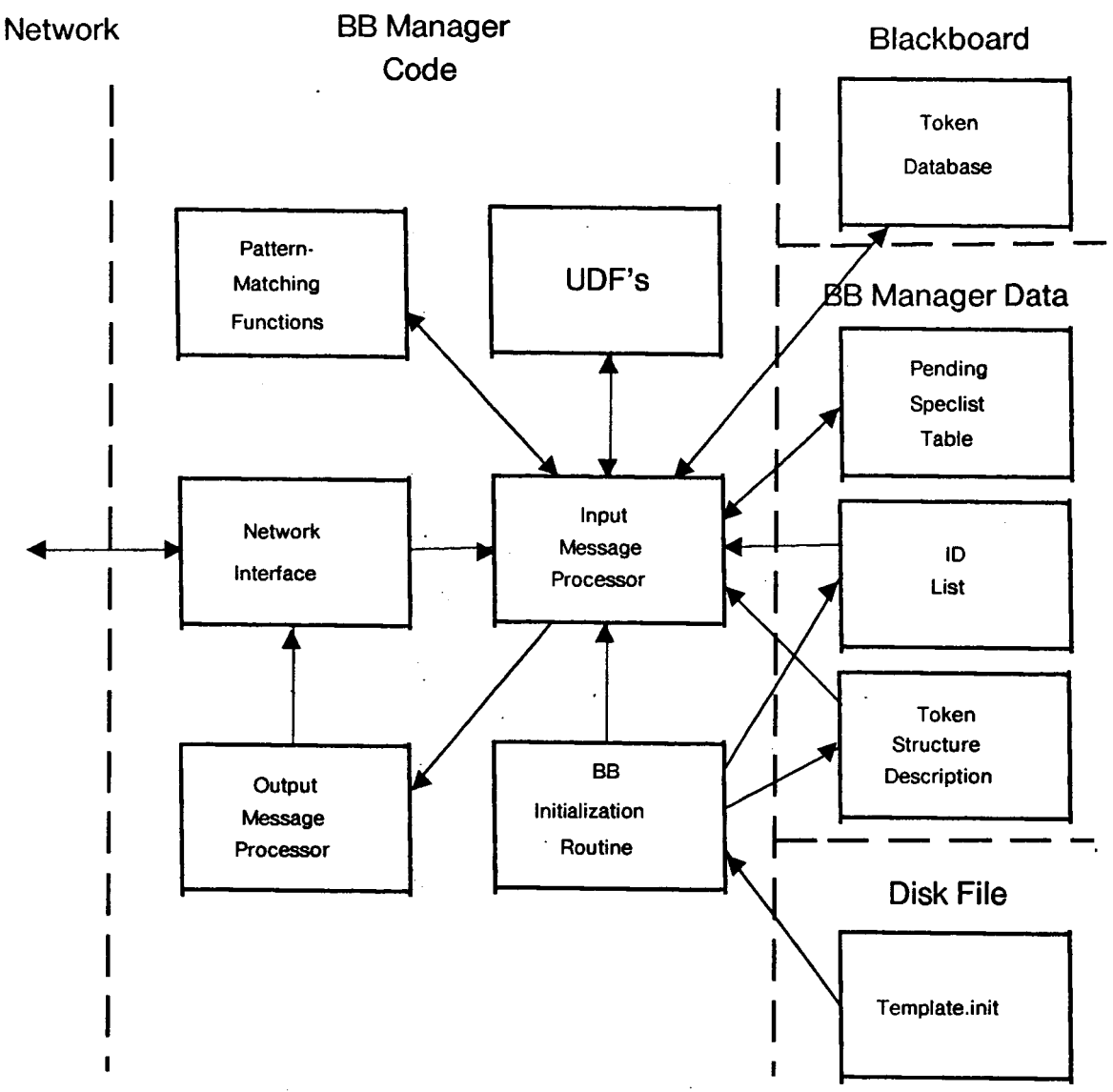

Figure 3-1: BB Manager Structure

- *ID list:* a list that maps identifiers to ID numbers. The list is downloaded by the *BB initialization routine* and is used by the *input message processor*.

3.2 Overview of Algorithms

At start-up the *BB initializer* loads the file *template.init* from disk, assigns IDs to all identifiers in the file, stores these IDs in the *ID list*, stores the token and attribute structure in the *token structure description*, and instructs the *input message processor* to begin accepting messages.

The BB manager receives requests as messages sent over a network or through inter-process channels. Arriving messages are passed by the *network interface* to the *input message processor*. The message type determines the action taken by the *input message processor*. The four message types are initialization, specification, new token, and direct addressing:

3.2.1 Initialization Messages

Each process (KS, sensor, or virtual vehicle) sends an intialization message to the BB manager. This message contains the name and net address of the process and a list of identifiers for token type names, attribute names, enumerated-type names, scalar names, UDF names, and UDT names (see section 2.4.1). The *input message processor* logs the name and address into the *module address table* assigns the process an ID, and looks up the IDs for each of the identifiers in the *ID list*. A message containing the process ID and the list of indentifier IDs is sent back to the calling process via the *output message processor*. The calling process tags future messages to the BB manager with the process ID number for easy identification. In some implementations the name and address of each process in the system may need to be logged with the BB manager before any messages can be sent.

3.2.2 Specification Messages

Specifications are bound together in list and shipped to the BB manager in a message (see section 2.4.3). Each spec list has an ID and a type (either standing or one-shot). The individual specs in the list have IDs. The *input message processor* logs this information into the *pending speclist table* tagged with the ID of the calling process. For each specification in the list, the *input message processor* examines all embedded attributes and determines the set of token types it can match. The intersection of all of these sets is the set of token types a given spec can match (the type set).

After compiling this information, the *input message processor* examines the spec list's type. If the list is standing, the *input message processor* invokes the *pattern-matching functions* and *UDF's* to attempt to match the spec list to tokens in the *token database*, constrained by the type set. All matching tokens are tagged with the spec ID and sent back to the calling process via the *output message processor* in a single message. If specified, matching tokens in the database are locked. Standing specs are retained for future matching. The matching procedure for one-shot spec lists is

the same; however, the life of a one-shot spec list differs from that of a standing list. If the calling process specifies the NOWAIT option, the *input message processor* deletes the spec list after attempting to match it to the tokens in the database, whether the list matched any tokens or not. If the WAIT option is specified, the BB manager sends all matching tokens back to the calling process and deletes the spec list, unless no tokens match. In this case, the spec list is not deleted until a new or modified token matching the list is inserted into the *token database*.

3.2.3 New Token Messages

When a new token arrives from a calling process, the *input message processor* assigns it a unique ID number, sets its generation number to one, and records the time it arrived and the ID number of the process that deposited it. The ID number is sent back to the calling process. The token is placed into the *token database*. with the same type. The *input message processor* attempts to match the token to all specs in the *pending speclist table*. The token is sent to all processes whose matching spec did NOT request a lock. The token is also sent to the process with highest priority that requests a lock. Process priorities are determined before the system is booted and are written into a file to be read by the BB manager.

3.2.4 Direct Addressing Messages

This class of messages includes requests to get, replace, unlock, and delete existing tokens in the database. The calling processes address tokens by their ID numbers. For requests to get a token, the *input message processor* locks the token (if requested) and sends it back to the calling process. Errors are returned if the token is already locked or does not exist. For requests to replace a token, the *input message processor* overwrites the existing token with the new token, provided the calling process has the token locked. The replaced token is unlocked and treated as a new token (see the previous section). Errors are returned if the token does not exist or if the calling process does not have the token locked. For requests to unlock a token, the *input message processor* unlocks it and attempts to match the token against all specs in the *pending speclist table* requesting a lock. The token is sent to the process with the highest priority. Errors are returned if the token to unlock does not exist or if the calling process does not have it locked. For requests to delete tokens, errors are returned if the token does not exist or if another process has it locked; otherwise, the token is deleted.

4. Reference Manual

This chapter describes all spec, token, and BB functions available to the user in the module interface. The syntax and effect of each function is given, indexed by function name. The auxillary files *template.init* and *functions.c* are also described. Examples of the functions and files in this chapter can be found in chapter 2.

4.1 System Data Structures

All of the data structures *tokens*, *arrays*, *locations*, *strings*, *UDTs*, *poses*, *point sets*, *frames*, *frame families*, *parameter lists*, *specs*, and *spec lists* are created dynamically. Care must be exercised in including these data structures in other structures, deleting them, and inserting and retrieving them from the blackboard database to ensure that they are not corrupted. The following sections explain how to manipulate such structures.

4.1.1 Tokens

Tokens created with *Tnewtoken* are *local* only; that is, a block of memory of the right size (depending on the type) is created in the user's address space. All of the attributes are initialized to zero with the exception of *TCTIME* and *TTYPE*, which are set according to the arguments to *Tnewtoken*. NULL pointers are written into the attribute slots for the aggregate types (*array*, *location*, *UDT*, and *string*). When the token is deposited into the blackboard with *BBputtoken*, space is allocated in the LMB's address space, and all attributes are copied from the local copy into the global blackboard copy. All attributes except *TCTIME*, *TTYPE*, and *TLOCATION* are written by the LMB into the global (blackboard) copy of the token. The internal attributes of the local copy remain *unaffected*. *BBputtoken* returns a unique global ID for the token. In order to change a global copy of a token, it must first be retrieved with *BBgetidtoken*. This function makes a local copy of the token and copies all attributes (including internal) into the local copy. *BBreplacetoken* creates a new global token, copies the local token to the global, deletes the old global token and *replaces* it with the new one.

Once a local token has been copied to the blackboard (i.e. with *BBputtoken* or *BBreplacetoken*), it is advisable to delete the local copy (to avoid wasting memory). Likewise, local copies of tokens retrieved from the blackboard (i.e. via *BBgettoken*), should be deleted once they are no longer needed. Local tokens are deleted with *Tfreetoken*. Note that the blackboard copy is *not* deleted. *Tfreetoken* also deletes all aggregate attributes contained in the token. Therefore, it is unwise to include such structures in more than one token, spec, location, array, etc. Instead, these structures should be copied. Tokens in the blackboard database can be deleted by ID with *BBdeletetoken*.

4.1.2 Locations and Substructures

Locations are created locally with *LmakelocationF* and *LmakelocationFF*. Locations consist of a point set and a pointer to a frame or frame family in which the points are expressed. Point sets are created with *Lnewpointset*, *Lpoint*, and *Llineseg*. Local locations are deleted using *Lfreelocation*. This function also deletes the location's point set, but does *not* delete its coordinate frame or frame family.

When locations are sent to the blackboard (either via tokens or specs), the point set is transformed to the first global or system (world, vehicle, or null) frame encountered in the reference chain. Therefore, tokens retrieved from the blackboard will always contain location data expressed in nonlocal frames. If the user wishes to manipulate this data in a local frame, he/she may do so by first invoking *Lconvertloc* on the data.

Frames and frame families are intended to be included in more than one location, frame, or frame family, and for this reason they are not deleted when the parent structure is deleted. Local frames are created with *FselectlocalfromFF*, *FmakelocalfromFF*, and *FmakelocalfromF*. Local frames are known only to the module that defines them. For this reason, no global frame may have a local base frame. Global frames are created with *FselectglobalfromFF*, *FmakeglobalfromFF*, and *FmakeglobalfromF*. Like the functions that create local frames, these functions allocate space *locally* for the global frames; additionally, this information is sent to the blackboard and a global ID is returned and stored in the local structure. In order to make changes to a global frame, the function *Fgetglobalupdate* should be invoked (using the pointer to the local structure refering to it), the frame is altered using *Fwritebase* and/or *Fwritepose*, and the update sent to the blackboard using *Fputglobal*.

The set of all global and local frames in the blackboard system during execution should at all times form a forest of trees, that is, there may be no cycles in the frame reference graph. Any such cycles will crash the system. In order to compare location data expressed in two different coordinate frames, the system finds the ancestor of the frames in the tree, and transforms the data to this frame.

The user must exercise *extreme* caution when deleting either local (via *Ffreelocal*) or global (via *Ffreeglobal*) frames. If a deleted frame is referenced by another frame or location, havoc will result. Admittedly, this feature is undesirable; future implementations will only permit a frame to be deleted when the reference count is zero.

4.2 Token Declarations and Operations

4.2.1 Token Template File Description

Token template file

The file *template.init* contains four types of declarations in any order:

- *Enumerated type declarations*
- *UDT declarations*
- *Global attribute declarations*
- *Token type declarations*

The file may also contain *include statements* to include other declaration files. The exact syntax is given in appendix II.

Enumerated type declarations

Each declaration is of the form:

```
ENUM <enum-type-name> = { <enum-scalar>,
                          <enum-scalar>,
                           ...
                        };
```

where <enum-type-name> and <enum-sclar> are *identifiers* denoting the name for the enumerated type and its scalars respectively. There may be any number of scalars.

UDT declarations

Each declaration is of the form:

```
UDT <user-type-name>;
```

where <user-type-name> is an *identifier* denoting the name for the UDT.

Array type declarations

An array is an ordered set of elements of any *attribute type* and indexed by positive integers. The syntax for the array type is:

```
ARRAY <arrayname> [<num-of-elem>] OF <attribute-type> ;
```

where <arrayname> is an *identifier* denoting the string name of the array, <num-of-elem> is a *number* denoting the number of elements in the array, and <attribute-type> is any *attribute type*. Arrays are indexed by integers in the range 0 through <num-of-elem> - 1.

Global attribute declarations

Each declaration is of the form:

```
GLOBAL <global-attribute-name> : <global-attribute-type>;
```

where <global-attribute-name> is an *identifier* denoting the name of the global attribute and <global-attribute-type> is the *attribute type*.

Token type declarations

Each declaration is of the form:

```
TOKEN <token-type-name> {
        <attribute-statement>
        <attribute-statement>
        ...
};
```

where <token-type-name> is an *identifier* denoting the name of a token type, and <attribute-statement> is either a *local attribute declaration* or an *global attribute reference*. Since global attributes are defined elsewhere, they need only be referenced for inclusion in a token type. Tokens may have any number of global and/or local attributes.

Local attribute declarations

Each declaration is of the form:

```
<local-attribute-name> : <local-attribute-type> ;
```

where <local-attribute-name> is an *identifier* denoting a local attribute name and <local-attribute-type> is an *attribute type*. Local attributes may be declared only within a *token type declaration*.

Include statements

Each statement is of the form:

```
INCLUDE <file>;
```

where <file> is a *string* denoting the name of an inclusion file.

Global attribute references

Each reference is of the form:

```
<global-attribute-name> : GLOBAL ;
```

where <global-attribute-name> is an *identifier* denoting the name of a global attribute declared in a *global attribute declaration*.

Attribute types Possible values for attribute types are INT, FLOAT, BOOL, STRING, LOCATION, *enumerated type name*, *UDT name*, and *array type name*, denoting integer, floating point number, boolean, string, location, UDT, and array respectively. Integers and booleans are four-byte data quantities. Floating point numbers are single precision (four-byte quantities). A location is a pointer to a structure containing a location size (number of coordinate points) and a pointer to an array of coordinate points. A coordinate point is a structure of two floating point numbers denoting x and y. A UDT is a pointer to an unstructured stream of bytes. Enumerated types are four-byte integers for the scalars declared in the corresponding *enumerated type declaration*.

Internal attributes

Internal attributes are common to all token types and therefore don't need to be explicitely declared. Internal attributes are all four-byte integers, with the exception of TLOCATION, which is a pointer to a location structure. Just like global and local attributes, they have names. The internal attributes are:

- **TTYPE:** the token type id number

- **TID:** a unique ID number for the token

- **TGEN:** the generation number of the token
- **TCTIME:** the time of the coordinate system used in this token
- **TITIME:** the time the token was inserted into the BB
- **TMTIME:** the time the token was last modified
- **TCREATOR:** the ID number of the module which created the token
- **TLOCATION:** a LOCATION for storing the token's position on the local map

All internal attributes (with the exception of *TTYPE*, *TCTIME*, and *TLOCATION*) are written by the LMB into the blackboard copy of the token when it is inserted with *BBputtoken*. The attributes *TGEN* and *TMTIME* are updated in the blackboard copy of the token when replaced with *BBreplacetoken*.

Identifiers Legal identifiers are sequences of upper and lower case letters, digits, and the minus sign '-', up to a maximum of 64 characters. Letters are case-folded, so that 'NAME' is the same as 'name'.

Strings Strings are sequences of up to 128 characters, delimited by double quotes. Any characters with the exception of the delimiters may be included in the string.

Numbers The number constants permitted are positive integers, up to 16 characters (digits in the inclusive range of 0 to 9) in length.

Whitespace The characters space, tab, carriage return, and newline may appear anywhere in the file and are discarded by the parser.

Comments Comments are permitted anywhere in the file and are delimited by the prefix '/*' and the postfix '*/'.

4.2.2 Token and Attribute Functions

The following functions create, read, write, and delete *local* copies of tokens. These functions do not affect tokens stored in the blackboard database. See section 4.4 for functions that read and write tokens into the database.

Tnewtoken This function allocates space (locally) for a token of type 'ttype' with a time stamp of 'ctime', and returns a pointer to it. The attribute fields of the token are initialized to null values.

```
t = Tnewtoken (ttype, ctime);
TOKEN *t;
int ttype, ctime;
```

Tfreetoken This function frees a token (local copy only) pointed to by 't'. All array, UDT, and location attributes in the token are also deleted.

```
Tfreetoken (t);
TOKEN *t;
```

T?read This class of functions returns an attribute value for the attribute with id 'aid' in the token 't'. The symbol '?' is an attribute designator, a single character (i, f, b, s, e, u, l, or a) denoting the type of the attribute returned (integer, floating point, boolean, string, enumerated, UDT, location, or array respectively).

```
ival = Tiread (t, aid);
fval = Tfread (t, aid);
bval = Tbread (t, aid);
sval = Tsread (t, aid);
eval = Teread (t, aid);
uval = Turead (t, aid);
lval = Tlread (t, aid);
aval = Taread (t, aid);
int ival, bval, eval, aid;
float fval;
char *sval;
UDT *uval;
LOCATION *lval;
ARRAY *aval;
TOKEN *t;
```

T?write This class of functions writes '?val' into the attribute with id 'aid' in token 't'. The symbol '?' is an attribute designator, a single character (i, f, b, s, e, u, l, a) denoting the type of the attribute returned (integer, floating point, boolean, string, enumerated, UDT, location, or array respectively).

```
Tiwrite (t, aid, ival);
Tfwrite (t, aid, fval);
Tbwrite (t, aid, bval);
Tswrite (t, aid, sval);
Tewrite (t, aid, eval);
Tuwrite (t, aid, uval);
Tlwrite (t, aid, lval);
Tawrite (t, aid, aval);
int ival, bval, eval, aid;
float fval;
char *sval;
UDT *uval;
LOCATION *lval;
ARRAY *aval;
TOKEN *t;
```

4.2.3 UDT functions

Unewudt This function allocates a new UDT. A pointer to the UDT is returned. A UDT should NOT be written into more than one token, spec, or array; instead, copies of the UDT should be used.

```
u = Unewudt ();
UDT *u;
```

Uassignudt This function loads UDT 'u' with the unstructured data of size 'dsize' pointed to by 'data'.

```
Uassignudt (u, data, dsize);
UDT *u;
char *data;
int dsize;
```

Ucopyudt This function makes a copy of the UDT pointed to by 'udt1' and returns a pointer to the copy. This is the "correct" way to include a UDT in two tokens, specs, arrays, etc.

```
udt2 = Ucopyudt (udt1);
UDT *udt1, *udt2;
```

Ufreeudt This function de-allocates the UDT pointed to by 'u'. UDT's which have been written into tokens, specs, or arrays should NOT be deleted with *Ufreeudt*, as this operation will corrupt any structure containing the UDT.

```
Ufreeudt (u);
UDT *u;
```

4.2.4 Pose Functions

Pose Data Type This data structure is a 4 by 4 matrix 'M' of floating point numbers denoting an affine transformation (translation, rotation, and scaling) of points from frame 'a' to a base frame 'b'. The last row of the matrix is assumed to be [0, 0, 0, 1]. If X_a is the homogeneous representation of a point in frame 'a' and X_b represents the same point in frame 'b', then:

$$X_b = M X_a$$

Poses used in frames and frame families describe the transformation from the frame or frame family to its base frame or family. Poses must be nonsingular. See Jerry Agin's *pose* package in the *stripe* library.

Pnewpose This function allocates a new pose. All elements of the new pose are initialized to zero, except for the element in the fourth row and fourth column, which is initialized to one. A pointer to the new pose is returned.

```
pose = Pnewpose ();
POSE *pose;
```

Pwriteelem This function writes the value 'fval' into row 'row' and column 'col' of pose 'pose'.

```
Pwriteelem (pose, row, col, fval);
POSE *pose;
int row, col;
float fval;
```

Preadelem This function returns the value of the element in row 'row' and column 'col' of pose 'pose'.

```
fval = Preadelem (pose, row, col);
float fval;
POSE *pose;
int row, col;
```

Pfreepose This routine de-allocates the pose. Havoc will result if a deleted pose is referenced by a frame or frame family.

```
Pfreepose (pose);
POSE *pose;
```

4.2.5 Frame Family Functions

FFmakeparamlist

This function allocates a parameter list 'plist' (list of time-pose pairs).

```
plist = FFmakeparamlist ();
FFPARAMLIST *plist;
```

FFaddtoparamlist

This function adds the time and transform pair 'ptime, ppose' to the parameter list 'plist'.

```
FFaddtoparamlist (plist, ppose, ptime);
FFPARAMLIST *plist;
int ptime;
POSE *ppose;
```

FFfreeparamlist This function de-allocates the parameter list 'plist'.

```
FFfreeparamlist (plist);
FFPARAMLIST *plist;
```

FFmakeglobalfromFF

This function allocates a global frame family 'frmfam' with a parameter list 'plist' and a base frame family 'bfrmfam'. The base frame family must not be local, but may be a user-defined global frame family or the vehicle (VEHICLEFF) frame family. The new frame family is sent to the blackboard, and 'frmfam' refers to it locally. Whenever 'frmfam' is accessed, a new copy is retrieved from the blackboard.

```
frmfam = FFmakeglobalfromFF (bfrmfam, plist);
FRAMEFAMILY *frmfam, *bfrmfam;
FFPARAMLIST *plist;
```

FFmakeglobalfromF

This function allocates a global frame family 'frmfam' with a parameter list 'plist' and a base frame 'bfrm'. The base frame must be user-defined global frame, the world frame (WORLDFF), or the null frame (NULLF). The new frame is sent to the blackboard, and 'frmfam' refers to it locally. Whenever 'frmfam' is accessed, a new copy is retrieved from the blackboard.

```
frmfam = FFmakeglobalfromF (bfrm, plist);
FRAMEFAMILY *frmfam;
FRAME *bfrm;
FFPARAMLIST *plist;
```

FFmakelocalfromFF

This function creates a local frame family 'frmfam' with base frame family 'bfrmfam', and a transform function named 'transform' which takes 'numofpars' floating point parameters (excluding a time parameter). The base frame family may be local, user-defined global, or vehicle (VEHICLEFF). The transform

function returns a pose for a given set of parameters, which is used to transform from a member of the base frame family to a member of the new family. If 'timearg' is FFTIME, the transform function will be invoked with time (an integer) as the *first* argument, and an array of 'numofpars' parameters as the *second* argument. If 'timearg' is FFNOTIME, only the array will be passed (as the first argument).

```
frmfam = FFmakelocalfromFF (bfrmfam, transform,
                            numofpars, timearg);
FRAMEFAMILY *frmfam, *bfrmfam;
POSE (*transform) ();
FFTIMEARG timearg;
int numofpars;
```

FFmakelocalfromF

This function creates a local frame family 'frmfam' with base frame 'bfrm', and a transform function named 'transform' which takes 'numofpars' parameters. This function differs from *FFmakelocalfromFF* in that the base is a frame instead of a frame family.

```
frmfam = FFmakelocalfromF (bfrm, transform,
                           numofpars, timearg);
FRAMEFAMILY *frmfam;
FRAME *bfrm;
POSE (*transform) ();
FFTIMEARG timearg;
int numofpars;
```

FFgetglobalid

This function retrieves the ID number from the global frame family 'frmfam'.

```
id = FFgetglobalid (frmfam);
int id;
FRAMEFAMILY *frmfam;
```

FFgetglobalfromid

This function retrieves the global frame family with ID number 'id' and stores it in a newly allocated frame family 'frmfam'. Thereafter, 'frmfam' can be used to refer locally to the frame family. If 'lock' is BBLOCK, the frame family is locked and no other module may change it. If 'lock' is BBNOLOCK, the frame family is not locked. BBPROTECTED is returned in 'status' if the frame family is already locked; BBGONE returned if the frame family has been deleted; BBSUCCESS is returned if the operation succeeds.

```
frmfam = FFgetglobalfromid (id, lock, status);
FRAMEFAMILY *frmfam;
int id, lock;
BBRESULT *status;
```

FFgetglobalupdate

This function retrieves an updated version of the global frame family 'frmfam' (if changed) from the blackboard. If 'lock' is BBLOCK, the frame family is locked and no other module may change it. If 'lock' is BBNOLOCK, the frame family is not locked. BBPROTECTED is returned if the frame family is already locked; BBGONE is returned if the frame family has been deleted; BBSUCCESS is returned if the operation succeeds.

```
status = FFgetglobalupdate (frmfam, lock);
BBRESULT status;
FRAMEFAMILY *frmfam;
int lock;
```

FFputglobal This function replaces an updated version of global frame family frmfam to the blackboard and unlocks the family. BBPROTECTED is returned if the frame family is already locked; BBGONE is returned if the frame family has been deleted; BBSUCCESS is returned if the operation succeeds.

```
status = FFputglobal (frmfam);
BBRESULT status;
FRAMEFAMILY *frmfam;
```

FFunlockglobal This function unlocks the global frame family 'frmfam'. BBPROTECTED is returned if the frame family is already locked; BBGONE is returned if the frame family has been deleted; BBSUCCESS is returned if the operation succeeds.

```
status = FFunlockglobal (frmfam);
BBRESULT status;
FRAMEFAMILY *frmfam;
```

FFwritebaseFF This function loads the base frame family in frame family 'frmfam' with the frame family 'bfrmfam'. It is assumed that the base frame was previously null (NULLF).

```
FFwritebaseFF (frmfam, bfrmfam);
FRAMEFAMILY *frmfam, *bfrmfam;
```

FFwritebaseF This function loads the base frame in frame family 'frmfam' with the frame 'bfrm'. It is assumed that the base frame was previously null (NULLF).

```
FFwritebaseF (frmfam, bfrm);
FRAMEFAMILY *frmfam, *bfrm;
```

FFcheckbase This function returns either BFRM if the base of global frame family 'frmfam' is a frame or BFRMFAM if it's a frame family.

```
test = FFcheckbase (frmfam);
BASETYPE test;
FRAMEFAMILY *frmfam;
```

FFreadbaseFF This function returns a pointer to the base frame family for global frame family 'frmfam1'. The user is responsible for checking (via *FFcheckbase*) that the base for 'frmfam1' is a frame family and not a frame.

```
frmfam2 = FFreadbaseFF (frmfam1);
FRAMEFAMILY *frmfam2, *frmfam1;
```

FFreadbaseF This function returns a pointer to the base frame for global frame family 'frmfam'. The user is responsible for checking (via *FFcheckbase*) that the base for 'frmfam' is a frame and not a frame family.

```
frm = FFreadbaseF (frmfam);
FRAME *frm;
FRAMEFAMILY *frmfam;
```

FFwriteparamlist This function writes the parameter 'plist' into the global frame family 'frmfam'.

```
FFwriteparamlist (frmfam, plist);
FRAMEFAMILY *frmfam;
FFPARAMLIST *plist;
```

FFreadparamlist This function reads the parameter list 'plist' from the global frame family 'frmfam'.

```
plist = FFreadparamlist (frmfam);
FFPARAMLIST *plist;
FRAMEFAMILY *frmfam;
```

FFfreelocal This function de-allocates a local frame family 'frmfam'. Havoc results if a deleted local frame family is referenced by another frame, frame family, or location.

```
FFfreelocal (frmfam);
FRAMEFAMILY *frmfam;
```

FFfreeglobal This function de-allocates a global frame family 'frmfam'. All frames, frame families, and locations referencing a deleted global frame subsequently reference to NULL. BBGONE is returned if the frame family has already been deleted, and BBPROTECTED is returned if the frame family is locked by another module; otherwise, BBSUCCESS is returned.

```
status=FFfreeglobal(frmfam);
BBRESULT status;
FRAMEFAMILY *frmfam;
```

4.2.6 Frame Functions

FselectlocalfromFF

This function creates a frame 'frm' by selecting frames from a chain of frame families starting with local or global frame family 'bfrmfam' and working backwards. The 'n' parameters 'par1, par2, ..., parn' are extracted left to right and inserted into the frame family transform functions as needed until the entire list is used. Extra parameters are discarded. If a time value is specified, 'time – arg' must be set to FFTIME and the value is passed in 'time'. All transform functions requiring time as a input parameter are invoked with the 'time' value as the first argument. If no time value is specified, 'time – arg' must be set to FFNOTIME. The frame 'frm' is local.

```
frm = FselectlocalfromFF (bfrmfam, time_arg, time, n,
                          par1, par2, ..., parn);
FRAMEFAMILY *bfrmfam;
FRAME *frm;
FFTIME time_arg;
int n, time;
float par1, par2, ..., parn;
```

FselectglobalfromFF

This function creates a frame 'frm' in the same manner as *FselectlocalfromFF*, except that the base frame 'bfrmfam' must be global. Global base frames are parameterized only by 'time'. The frame 'frm' is global.

```
frm = Fselectglobalfromff (bfrmfam, time);
FRAME *frm;
FRAMEFAMILY *bfrmfam;
int time;
```

FmakeglobalfromFF

This function makes a global frame 'frm' with respect to base frame family 'bfrmfam' and a pose of 'ppose'. The frame family 'bfrmfam' must be user-defined global or the vehicle frame family (VEHICLEFF). The new frame is sent to the BB and 'frm' refers to it locally.

```
frm = FmakeglobalfromFF (bfrmfam, ppose);
FRAMEFAMILY *bfrmfam;
FRAME *frm;
POSE *ppose;
```

FmakeglobalfromF

This function makes a global frame 'frm' with respect to base frame 'bfrm' and a pose of 'ppose'. The frame 'bfrm' must be user-defined global or the world frame (WORLDF). The new frame is sent to the BB and 'frm' refers to it locally.

```
frm = FmakeglobalfromF (bfrm, ppose);
FRAME *frm, *bfrm;
POSE *ppose;
```

FmakelocalfromFF

This function creates a local frame 'frm' with respect to base frame family 'bfrmfam' and a pose of 'ppose'.

```
frm = FmakelocalfromFF (bfrmfam, ppose);
FRAMEFAMILY *bfrmfam;
FRAME *frm;
POSE *ppose;
```

FmakelocalfromF

This function creates a local frame 'frm' with respect to base frame 'bfrm' and a pose of 'ppose'.

```
frm = FmakelocalfromF (bfrm, ppose);
FRAME *frm, *bfrm;
POSE *ppose;
```

Fgetglobalid This function retrieves the ID number from the global frame 'frm'.

```
id = Fgetglobalid (frm);
int id;
FRAME *frm;
```

Fgetglobalfromid This function retrieves the global frame with ID number 'id' and stores it in a newly allocated frame 'frm'. Thereafter, 'frm' can be used to refer locally to the frame. If 'lock' is BBLOCK, the frame is locked and no other module may change it. If 'lock' is BBNOLOCK, the frame is not locked. BBPROTECTED is returned if the frame family is already locked; BBGONE is returned if the frame family has been deleted; BBSUCCESS is returned if the operation succeeds.

```
frm = Fgetglobalfromid (id, lock, status);
FRAME *frm;
int id, lock;
BBRESULT *status;
```

Fgetglobalupdate

This function retrieves an updated version of the global frame 'frm' (if changed) from the blackboard. If 'lock' is BBLOCK, the frame is locked and no other module may change it. If 'lock' is BBNOLOCK, the frame is not locked. BBPROTECTED is returned if the frame family is already locked; BBGONE is returned if the frame family has been deleted; BBSUCCESS is returned if the operation succeeds.

```
status = Fgetglobalupdate (frm, lock);
BBRESULT status;
FRAME *frm;
int lock;
```

Fputglobal

This function writes an updated version of global frame 'frm' to the blackboard and unlocks it. BBPROTECTED is returned if the frame family is already locked; BBGONE is returned if the frame family has been deleted; BBSUCCESS is returned if the operation succeeds.

```
status = Fputglobal (frm);
BBRESULT status;
FRAME *frm;
```

Funlockglobal

This function unlocks the global frame 'frm'. BBPROTECTED is returned if the frame family is already locked; BBGONE is returned if the frame family has been deleted; BBSUCCESS is returned if the operation succeeds.

```
status = Funlockglobal (frm);
BBRESULT status;
FRAME *frm;
```

FwritebaseFF

This function loads the base frame family in the locked frame 'frm' with 'frmfam'. The base frame is assumed to have been NULLF.

```
FwritebaseFF (frm, frmfam);
FRAMEFAMILY *frmfam;
FRAME *frm;
```

FwritebaseF

This function loads the base frame in the locked frame 'frm1' with 'frm2'. The base frame is assumed to have been NULLF.

```
FwritebaseF (frm1, frm2);
FRAME *frm1, *frm2;
```

Fcheckbase

This function returns either BFRM if the base of global frame 'frm' is a frame or BFRMFAM if it's a frame family.

```
test = Fcheckbase (frm);
BASETYPE test;
FRAME *frm;
```

FreadbaseFF

This function returns a pointer to the base frame family for global frame 'frm'. The

user is responsible for checking (via *Fcheckbase*) that the base for 'frm' is a frame family and not a frame.

```
frmfam = FreadbaseFF (frm);
FRAMEFAMILY *frmfam;
FRAME *frm;
```

FreadbaseF This function returns a pointer to the base frame for global frame'frm1'. The user is responsible for checking (via *Fcheckbase*) that the base for 'frm1' is a frame and not a frame family.

```
frm2 = FreadbaseF (frm1);
FRAME *frm1, *frm2;
```

Fwritepose This function loads the pose 'ppose' into the locked global frame 'frm'.

```
Fwritepose (frm, ppose);
FRAME *frm;
POSE *ppose;
```

Freadpose This function reads the pose 'ppose' from the locked global frame 'frm'.

```
ppose = Freadpose (frm);
POSE *ppose;
FRAME *frm;
```

Ffreelocal This function de-allocates a local frame 'frm'. Havoc results if a deleted local frame is referenced by another frame, frame family, or location.

```
Ffreelocal (frm);
FRAME *frm;
```

Ffreeglobal This function de-allocates a global frame 'frm'. All frames, frame families, and locations referencing a deleted global frame subsequently reference to NULL. BBGONE is returned if the frame has already been deleted, and BBPROTECTED is returned if the frame is locked by another module; otherwise, BBSUCCESS is returned.

```
status = Ffreeglobal (frm);
BBRESULT status;
FRAME *frm;
```

4.2.7 Location Functions

Currently, all functions operating on tokens are two-dimensional (with the exception of *Ldistance3*). All functions taking a single location as input ignore the z-coordinate of all points in the location. Functions taking two locations as input transform the points into the common ancestor frame or frame family of both locations, and then ignore the z-coordinates. *Ldistance3* provides the user with the minimum of support needed for three-dimensional operations. Eventually all of the location functions will support three-dimensional operations if meaningful.

Currently, the software does not distinguish between convex and non-convex polygons. Functions

such as *Ldistance* and *Lintersection* assume all polygons are non-convex and thus use brute-force, inefficient algorithms. Future releases will distinguish between the two types and use efficient algorithms whenever possible.

Location Data Type

A location is a set of three-dimensional points expressed in a coordinate frame or frame family. There are four location subtypes:

- *LPOINT:* a single point in 3-space with x, y, and z coordinates
- *LLINESEG:* a line segment in 3-space with two endpoints (x1, y1, z1) and (x2, y2, z2)
- *LPOLYGON:* a simple planar polygon described by a set of vertices (three-dimensional points) such that the interior of the polygon lies the right when traversing the vertices in order
- *LSCATTER:* a scattering of points in no particular order

Lnewpointset This function creates a new point set 'pset' of type 'type' and a maximum size of 'size'. The parameter 'type' may be LPOINT, LLINESEG, LPOLYGON, or LSCATTER.

```
pset = Lnewpointset (type, size);
LPOINTSET *pset;
LPOINTSETTYPE type;
int size;
```

Lwritepoint This function writes point 'x,y,z' into the 'ith' point of point set 'pset'. The index 'i' ranges from 0 to the size of the point set minus 1.

```
Lwritepoint (pset, i, x, y, z);
LPOINTSET *pset;
int i;
float x, y, z;
```

Lreadpoint This function reads the 'ith' point from the point set 'pset' into 'x,y,z'. The index 'i' ranges from 0 to the size of the point set minus 1.

```
Lreadpoint (pset, i, x, y, z);
LPOINTSET *pset;
int i;
float *x, *y, *z;
```

Lpointsettype This function returns the point set type for point set 'pset'.

```
type = Lpointsettype (pset);
LPOINTSETTYPE type;
LPOINTSET *pset;
```

Lpointsetsize This function returns the size (number of points) of point set 'pset'.

```
size = Lpointsetsize (pset);
int size;
LPOINTSET *pset;
```

Lfreepointset This function de-allocates the point set 'pset'.

```
Lfreepointset (pset);
LPOINTSET *pset;
```

Lpoint This function creates a point set of type LPOINT and stores the point 'x,y,z' into it.

```
pset = Lpoint (x, y, z);
LPOINTSET *pset;
float x, y, z;
```

Llineseg This function creates a point set of type LLINESEG and stores the endpoints 'x1,y1,z1' and 'x2,y2,z2' into it.

```
pset = Llineseg (x1, y1, z1, x2, y2, z2);
LPOINTSET *pset;
float x1, y1, z1, x2, y2, z2;
```

LmakelocationF This function creates a location 'loc' from the point set 'pset' with respect to coordinate frame 'frm'. A location should NOT be written into more than one token, spec, or array; instead, copies of the location should be used.

```
loc = LmakelocationF (frm, pset);
LOCATION *loc;
FRAME *frm;
LPOINTSET *pset;
```

LmakelocationFF This function creates a location 'loc' from the point set 'pset' with respect to coordinate frame family 'frmfam'. A location should NOT be written into more than one token, spec, or array; instead, copies of the location should be used.

```
loc = LmakelocationFF (frmfam, pset);
LOCATION *loc;
FRAMEFAMILY *frmfam;
LPOINTSET *pset;
```

Lcopylocation This function makes a copy of the location pointed to by 'loc1' and returns a pointer to the copy. This is the "correct" way to include a location in two tokens, specs, arrays, etc.

```
loc2 = Lcopylocation (loc1);
LOCATION *loc1, *loc2;
```

Lfreelocation This function de-allocates location 'loc'. The location's point set is also deleted. Locations which have been written into tokens, specs, or arrays should NOT be deleted with *Lfreelocation*, as this operation will corrupt any structure containing the location.

```
Lfreelocation (loc);
LOCATION *loc;
```

Lconvertloc This function creates a location 'loc2' by converting location 'loc1' to frame 'frm' if 'btype' is BFRM or to a frame family 'frmfam' if 'btype' is BFRMFAM. NULL is returned if the location cannot be converted.

```
loc2 = Lconvertloc (loc1, btype, frm, frmfam);
LOCATION *loc1, *loc2;
BASETYPE btype;
FRAME *frm;
FRAMEFAMILY *frmfam;
```

Lcheckbase This function returns the base type for location 'loc'. BFRM is returned if the location is expressed in a frame, BFRMFAM is returned if the location is expressed in a frame family. The functions *LgetbaseF* and *LgetbaseFF* retrieve the base frame and frame family respectively of a location. frame or frame family bases.

```
btype = Lcheckbase (loc);
BASETYPE btype;
LOCATION *loc;
```

LgetbaseF This function returns the base frame 'frm' for location 'loc'. The function *Lcheckbase* determines the location's base type.

```
frm = LgetbaseF (loc);
FRAME *frm;
LOCATION *loc;
```

LgetbaseFF This function returns the base frame family 'frmfam' for location 'loc'. The function *Lcheckbase* determines the location's base type.

```
frmfam = LgetbaseFF (loc);
FRAMEFAMILY *frmfam;
LOCATION *loc;
```

Lgetbaseid This function returns the ID number a location 'loc' expressed in a global frame or frame family. The function *Lcheckbase* determines the base type for a location.

```
id = Lgetbaseid (loc);
int id;
LOCATION *loc;
```

Lgetpointset This function returns the point set for location 'loc'.

```
pset = Lgetpointset (loc);
LPOINTSET *pset;
LOCATION *loc;
```

Ldistance This function computes the distance between the locations 'loc1' and 'loc2'. *Ldistance* returns the minimum Euclidian distance between the two locations. For points, this distance is measured from the point itself; for line segments, it is measured from a point on the segment; for polygons, it is measured from an interior or boundary point of the polygon or on; and for point sets, it is measured from a point in the set. The minimum distance between intersecting locations is zero.

```
dist = Ldistance (loc1, loc2);
LOCATION *loc1, *loc2;
float dist;
```

Ldistance3 This function computes the distance between between the two *point* locations 'loc1' and 'loc2'. The two points are treated as three-dimensional (the z-

coordinate is meaningful). *Ldistance3* is undefined for line segments, polygons, and point scatters (a value of MAXFLOAT is returned).

```
dist = Ldistance3 (loc1, loc2);
LOCATION *loc1, *loc2;
float dist;
```

Lcentroid

This function computes the centroid of location 'loc1'. The centroid of a point is the point itself. The centroid of a line segment is the midpoint. The centroid of a polygon is its "center of mass". The centroid of a point set is the average of the points in the set.

```
cent = Lcentroid (loc1);
LOCATION *cent, *loc1;
```

Larea

This function computes the area of location 'loc1'. The areas of a point and a line segment are zero. The area of a polygon is the area enclosed by the perimeter. The area of a point set is the area of the best-fitting ellipse.

```
area = Larea (loc1);
LOCATION *loc1;
float area;
```

Ldiameter

This function computes the diameter of location 'loc1'. The diameter of a location is the maximum distance between two points of the location.

```
diam = Ldiameter (loc1);
LOCATION *loc1;
float diam;
```

Lorientation

This function computes the orientation of location 'loc1'. The orientation of a location is the angle (in radians) of the location's major axis to the positive x-axis. Angles tending toward the positive y-axis are positive. The angles range from -PI/2 to PI/2. The major axis of a point is undefined, hence an error value of MAXFLOAT is returned. For line segments, the major axis is angle between the segment and the x-axis. For polygons and point sets, the orientation is the angle between the axis of the best-fitting ellipse and the x-axs.

```
orien = Lorientation (loc1);
float orien;
LOCATION *loc1;
```

Lchull

This function computes the convex hull of location 'loc1'. The convex hull of a point is the point itself. The hull of a line is the line itself. The hull of a polygon is rigorously defined elsewhere. The hull of a point set is the smallest convex polygon containing all of the points.

```
chull = Lchull (loc1);
LOCATION *loc1, *chull;
```

Lbox

This function computes the minimum bounding rectangle (MBR) of location 'loc1'.

```
box = Lbox (loc1);
LOCATION *loc1, *box;
```

Loverlap

This function returns TRUE if locations 'loc1' and 'loc2' overlap, i.e. if they intersect; FALSE otherwise.

```
test = Loverlap (loc1, loc2);
LOCATION *loc1, *loc2;
int test;
```

4.2.8 Array Functions

Anewarray This function creates a new array of 'size' elements of type 'type' (where type is AINTEGER, AFLOAT, ABOOLEAN, ASTRING, ALOCATION, AENUMERATED, AUDT, or AARRAY). An array should NOT be written into more than one token, spec, or array; instead, copies of the array should be used.

```
array = Anewarray (type, size);
ARRAY *array;
ARRAYTYPE type;
int size;
```

Acopyarray This function makes a copy of the array pointed to by 'array1' and returns a pointer to the copy. This is the "correct" way to include an array in two tokens, specs, arrays, etc.

```
array2 = Acopyarray (array1);
ARRAY *array1, *array2;
```

Afreearray This function deallocates array 'array'. All array, UDT, and location elements are also deleted. Arrays which have been written into tokens, specs, or arrays should NOT be deleted with *Ufreearray*, as this operation will corrupt any structure containing the array.

```
Afreearray (array);
ARRAY *array;
```

A?read This class of functions returns an array value with index 'index' in the array 'array'. The symbol '?' is a type designator, a single character (i, f, b, s, e, u, l, or a) denoting the type of the array element returned (integer, floating point, boolean, string, enumerated, UDT, location, or array respectively).

```
ival = Airead (array, index);
fval = Afread (array, index);
bval = Abread (array, index);
sval = Asread (array, index);
eval = Aeread (array, index);
uval = Auread (array, index);
lval = Alread (array, index);
aval = Aaread (array, index);
int ival, bval, eval, index;
float fval;
char *sval;
UDT *uval;
LOCATION *lval;
ARRAY *aval, *array;
```

A?write This class of functions writes '?val' into the element of array 'array' with index 'index'. The symbol '?' is a type designator, a single character (i, f, b, s, e, u, l, or a) denoting the type of the array element returned (integer, floating point, boolean, string, enumerated, UDT, location, or array respectively).

```
Aiwrite (array, index, ival);
Afwrite (array, index, fval);
Abwrite (array, index, bval);
Aswrite (array, index, sval);
Aewrite (array, index, eval);
Auwrite (array, index, uval);
Alwrite (array, index, lval);
Aawrite (array, index, aval);
int ival, bval, eval, index;
float fval;
char *sval;
UDT *uval;
LOCATION *lval;
ARRAY *aval, *array;
```

Asizeof This function returns the size (in elements) of array 'array'.

```
size = Asizeof (array);
int size;
ARRAY *array;
```

Atypeof This function returns the type of array 'array'.

```
type = Atypeof (array);
ARRAYTYPE type;
ARRAY *array;
```

Aextend This function extends array 'array' by 'increase' elements.

```
Aextend (array, increase);
ARRAY *array;
int increase;
```

A?append This class of functions extends array 'array' by one element and stores '?val' in this element. The symbol '?' is a type designator, a single character (i, f, b, s, e, u, l, or a) denoting the type of '?val' (integer, floating point, boolean, string, enumerated, UDT, location, or array respectively).

```
Aiappend (array, ival);
Afappend (array, fval);
Abappend (array, bval);
Asappend (array, sval);
Aeappend (array, eval);
Auappend (array, uval);
Alappend (array, lval);
Aaappend (array, aval);
int ival, bval, eval;
float fval;
char *sval;
UDT *uval;
LOCATION *lval;
ARRAY *aval;
```

A?member This class of functions returns a boolean value of TRUE if '?val' is an element of the array 'array'. The symbol '?' is a type designator, a single character (i, f, b, s, e, u, l, or a) denoting the type of '?val' (integer, floating point, boolean, string,

enumerated, UDT, location, or array respectively). If '?val' is not a member of 'array', FALSE is returned.

```
test = Aimember (array, ival);
test = Afmember (array, fval);
test = Abmember (array, bval);
test = Asmember (array, sval);
test = Aemember (array, eval);
test = Aumember (array, uval);
test = Almember (array, lval);
test = Aamember (array, aval);
int test, ival, bval, eval;
float fval;
char *sval;
UDT *uval;
LOCATION *lval;
ARRAY *aval;
```

Aintersection This function returns an array 'intersection' which is the set intersection of arrays 'array1' and 'array2'. (Note: comment on arrays of arrays.)

```
intersection = Aintersection (array1, array2);
ARRAY *intersection, *array1, *array2;
```

Aunion This function returns an array 'union' which is the set union of arrays 'array1' and 'array2'. (Note: comment on arrays of arrays.)

```
union = Aunion (array1, array2);
ARRAY *union, *array1, *array2;
```

Aequal This function returns the boolean value TRUE if 'array1' and 'array2' are equal, that is, if 'array1' includes every member of 'array2' and vice versa (set equality).

```
test = Aequal (array1, array2);
int test;
ARRAY *array1, *array2;
```

Amax This function returns the lowest index of the maximum element of array 'array'. This function is not defined over arrays of type location, UDT, and array.

```
index = Amax (array);
int index;
ARRAY *array;
```

Amin This function returns the lowest index of the minimum element of array 'array'. This function is not defined over arrays of type location, UDT, and array.

```
index = Amin (array);
int index;
ARRAY *array;
```

Asort This function sorts the elements of 'array' in ascending order and returns a pointer to the sorted array. The original array remains unchanged. This function is not defined over arrays of type location, UDT, and array.

```
sorted = Asort (array);
ARRAY *sorted, *array;
```

4.3 Specifications and Specification Lists

4.3.1 Specification Functions

Snewspec - This function allocates a spec header node and points it to the spec node 'sn'. A pointer to the spec header node is returned. The spec node 'sn' is assumed to be the root node of a spec tree.

```
sp = Snewspec (sn);
SPEC *sp;
SPECNODE *sn;
```

Sfreespec This function frees spec 'sn'. All spec nodes in the tree are also deleted, as well as embedded array, UDT, and location constants. This function should not be invoked on specs which are included in a spec list; otherwise havoc will result.

```
Sfreespec (sp);
SPEC *sp;
```

4.3.2 Constant and Attribute Spec Node Functions

S?const This class of functions converts constants '?val' into spec nodes, where '?' is a single character (i, f, b, s, e, u, l, or a) denoting the constant's type (integer, floating point, boolean, string, enumerated, UDT, or location respectively).

```
sn = Siconst (ival);
sn = Sfconst (fval);
sn = Sbconst (bval);
sn = Ssconst (sval);
sn = Seconst (eval);
sn = Suconst (uval);
sn = Slconst (lval);
sn = Saconst (aval);
SPECNODE *sn;
int ival, bval, eval;
float fval;
char *sval;
UDT *uval;
LOCATION *lval;
ARRAY *aval;
```

Sattribute This function creates a spec node that refers to the attribute with id 'aid'.

```
sn = Sattribute (aid);
SPECNODE *sn;
int aid;
```

Sarrelemattr This function creates a spec node that refers to the array element with index 'index' in the attribute with id 'aid'. The referenced attribute must be of type array.

```
sn = Sarrelemattr (aid, index);
SPECNODE *sn;
int aid, index;
```

4.3.3 Boolean Spec Node Functions

Sand[n]

These two functions build AND specnodes (boolean), that is, nodes that instructs the BB manager to AND the input operands together. Sand takes two input specnodes (also booleans), while Sandn takes 'n' specnodes.

```
sn = Sand (sn1, sn2);
sn = Sandn (n, sn1, sn2, ... , snn);
SPECNODE *sn, *sn1, *sn2, ..., *snn;
int n;
```

Sor[n]

These two functions build OR specnodes (boolean), that is, nodes that instructs the BB manager to OR the input operands together. Sor takes two input specnodes (also booleans), while Sorn takes 'n' specnodes.

```
sn = Sor (sn1, sn2);
sn = Sorn (n, sn1, sn2, ... , snn);
SPECNODE *sn, *sn1, *sn2, ..., *snn;
int n;
```

Snot

This function builds a NOT spec node 'sn', which instructs the BB manager to negate its operand 'sn1' (a boolean). No excuses are given for this function's name.

```
sn = Snot (sn1);
SPECNODE *sn, *sn1;
```

4.3.4 Relational Spec Node Functions

Sequal

This function builds a spec node 'sn' (a boolean), instructing the BB manager to test the equality of the input operands sn1 and sn2. The operands must be floating point numbers, integers, booleans, enumerated scalars, or strings.

```
sn = Sequal (sn1, sn2);
SPECNODE *sn, *sn1, *sn2;
```

Snequal

This function builds a spec node 'sn' (a boolean), instructing the BB manager to test the inequality of the input operands sn1 and sn2. Specnodes sn1 and sn2 must hold data of the same type and must be floating point numbers, integers, booleans, enumerated scalars, or strings.

```
sn = Snequal (sn1, sn2);
SPECNODE *sn, *sn1, *sn2;
```

Sless[eq]

These functions build a spec node 'sn' (a boolean), instructing the BB manager to test if sn1 is less than sn2 or if sn1 is less than or equal to sn2 for Sless and Slesseq respectively. Specnodes sn1 and sn2 must be integers or floating point numbers.

```
sn = Sless (sn1, sn2);
sn = Slesseq (sn1, sn2);
SPECNODE *sn, *sn1, *sn2;
```

Sgreater[eq]

These functions build a spec node 'sn' (a boolean), instructing the BB manager to test if sn1 is greater than sn2 or if sn1 is greater than or equal to sn2 for Sgreater and Sgreatereq respectively. Specnodes sn1 and sn2 must be integers or both floating point numbers.

```
sn = Sgreater (sn1, sn2);
sn = Sgreatereq (sn1, sn2);
SPECNODE *sn, *sn1, *sn2;
```

4.3.5 Arithmetic Spec Node Functions

Srange This function builds a spec node 'sn' (a boolean), instructing the BB manager to test if 'sn1' lies between 'sn2' and 'sn3' inclusively. Spec nodes sn1, sn2, and sn3 must be integers or floating point numbers.

```
sn = Srange (sn1, sn2, sn3);
SPECNODE *sn, *sn1, *sn2, *sn3;
```

Sadd[n] These two functions build an ADD spec node 'sn' of type integer or floating point, which instructs the BB manager to add the input operands together (also of type integer or floating point). Sadd takes two operands, and Saddn takes 'n'.

```
sn = Sadd (sn1, sn2);
sn = Saddn (n, sn1, sn2, ... , snn);
SPECNODE *sn, *sn1, *sn2, ..., *snn;
```

Ssubtract This function builds a SUBTRACT sec node 'sn' of type integer or floating point, which instructs the BB manager to subtract sn2 from sn1 (also of type integer or floating point).

```
sn = Ssubtract (sn1, sn2)
SPECNODE *sn, *sn1, *sn2;
```

Smultiply[n] These two functions build an MULTIPLY spec node 'sn' of type integer or floating point, which instructs the BB manager to multiply the input operands together (also of type integer or floating point). Smultiply takes two operands, and Smultiplyn takes 'n'.

```
sn = Smultiply (sn1, sn2);
sn = Smultiplyn (n, sn1, sn2, ... , snn);
SPECNODE *sn, *sn1, *sn2, ..., *snn;
```

Sdivide This function builds a DIVIDE spec node 'sn' of type integer or floating point, which instructs the BB manager to divide sn1 by sn2 (also of type integer or floating point). Integer division is truncated.

```
sn = Sdivide (sn1, sn2)
SPECNODE *sn, *sn1, *sn2;
```

4.3.6 String Spec Node Functions

Ssubstring This function builds a SUBSTRING spec node 'sn' of type boolean, which instructs the BB manager to test whether string spec node sn1 is a substring of string spec node sn2.

```
sn = Ssubstring (sn1, sn2);
SPECNODE *sn, *sn1, *sn2;
```

Ssregexp This function builds a REGULAR EXPRESSION SEARCH spec node 'sn' of type boolean, which instructs the BB manager to test whether string spec node 'sn1'

denotes a regular expression matching string spec node 'sn2'. See *recomp* in section 3 of the *UNIX* manual.

```
sn = Ssregexp (sn1, sn2);
SPECNODE *sn, *sn1, *sn2;
```

4.3.7 Location Spec Node Functions

Sdistance

This function builds a DISTANCE spec node 'sn' of type float, which instructs the BB manager to compute the minimum distance between location spec nodes 'sn1' and 'sn2'. For points, this distance is measured from the point itself; for line segments, it is measured from a point on the segment; for polygons, it is measured from an interior or boundary point of the polygon or on; and for point sets, it is measured from a point in the set. The minimum distance between intersecting locations is zero.

```
sn = Sdistance (sn1, sn2);
SPECNODE *sn, *sn1, *sn2;
```

Sdistance3

This function builds a DISTANCE3 spec node 'sn' of type float, which instructs the BB Manager to compute the minimum Euclidian distance between two point location spec nodes 'sn1' and 'sn2'. The two points are treated as three-dimensional (the z-coordinate is meaningful). *Sdistance3* is undefined for line segments, polygons, and point scatters (a value of MAXFLOAT is returned).

```
sn = Sdistance3 (sn1, sn2);
SPECNODE *sn, *sn1, *sn2;
```

Scentroid

This function builds a CENTROID spec node 'sn' of type location, which instructs the BB manager to compute the centroid of location spec node 'sn1'. The centroid of a point is the point itself. The centroid of a line segment is the midpoint. The centroid of a polygon is its "center of mass". The centroid of a point set is the average of the points in the set.

```
sn = Scentroid (sn1);
SPECNODE *sn, *sn1;
```

Sarea

This function builds an AREA spec node 'sn' of type float, which instructs the BB manager to compute the area of location spec node 'sn1'. The areas of a point and a line segment are zero. The area of a polygon is the area enclosed by the perimeter. The area of a point set is the area of the best-fitting ellipse.

```
sn = Sarea (sn1);
SPECNODE *sn, *sn1;
```

Sdiameter

This function builds an DIAMETER spec node 'sn' of type float, which instructs the BB manager to compute the diameter of location spec node 'sn1'. The diameter of a location is the maximum distance between two points of the location.

```
sn = Sdiameter (sn1);
SPECNODE *sn, *sn1;
```

Sorientation

This function builds an ORIENTATION spec node 'sn' of type float, which instructs the BB manager to compute the orientation of location spec node 'sn1'. The orientation of a location is the angle (in radians) of the location's major axis to the

positive x-axis. Angles tending toward the positive y-axis are positive. The angles range from -PI/2 to PI/2. The major axis of a point is undefined, hence an error value of MAXFLOAT is returned. For line segments, the major axis is angle between the segment and the x-axis. For polygons and point sets, the orientation is the angle between the axis of the best-fitting ellipse and the x-axs.

```
sn = Sorientation (sn1);
SPECNODE *sn, *sn1;
```

Schull This function builds a CONVEX HULL spec node 'sn' of type location, which instructs the BB manager to compute the convex hull of location spec node 'sn1'. The convex hull of a point is the point itself. The hull of a line is the line itself. The hull of a polygon is rigorously defined elsewhere. The hull of a point set is the smallest convex polygon containing all of the points.

```
sn = Schull (sn1);
SPECNODE *sn, *sn1;
```

Sbox This function builds a BOX spec node 'sn' of type location, which instructs the BB manager to compute the minimum bounding rectangle (MBR) of location spec node 'sn1'.

```
sn = Sbox (sn1);
SPECNODE *sn, *sn1;
```

Soverlap This function builds an OVERLAP spec node 'sn' of type boolean, which instructs the BB manager to test if location spec nodes 'sn1' and 'sn2' overlap, i.e. if they intersect.

```
sn = Soverlap (sn1, sn2);
SPECNODE *sn, *sn1, *sn2;
```

4.3.8 Array Spec Node Functions

Smember This function builds a MEMBER spec node 'sn', which instructs the BB manager to test if data spec node 'sn2' is a member of the array spec node 'sn1'.

```
sn = Smember (sn1, sn2);
SPECNODE *sn, *sn1, *sn2);
```

Sintersection This function builds an INTERSECTION spec node 'sn', which instructs the BB manager to return the intersection of array spec nodes 'sn1' and 'sn2'.

```
sn = Sintersection (sn1, sn2);
SPECNODE *sn, *sn1, *sn2;
```

Sunion This function builds an UNION spec node 'sn', which instructs the BB manager to return the union of array spec nodes 'sn1' and 'sn2'.

```
sn = Sunion (sn1, sn2);
SPECNODE *sn, *sn1, *sn2;
```

Saequal This function builds an AEQUAL spec node 'sn', which instructs the BB manager to test if array spec nodes 'sn1' and 'sn2' are equal (contain the same elements regardless of order).

```
sn = Saequal (sn1, sn2);
SPECNODE *sn, *sn1, *sn2;
```

Smax This function builds a MAX spec node 'sn', which instructs the BB manager to return the maximum element in array spec node 'sn1'. This function is not defined over data of type location, UDF, and array.

```
sn = Smax (sn1);
SPECNODE *sn, *sn1;
```

Smin This function builds a MIN spec node 'sn', which instructs the BB manager to return the minumum element in array spec node 'sn1'. This function is not defined over data of type location, UDF, and array.

```
sn = Smin (sn1);
SPECNODE *sn, *sn1;
```

4.3.9 Slot Functions

Sindirect This function creates an indirect spec node that points to another spec node. By changing the value of this pointer (see *Spointer*), part of a spec tree can be changed without rebuilding the entire tree.

```
sn = Sindirect ();
SPECNODE *sn;
```

Spointer This function loads an indirect spec node 'sn1' (see *Sindirect*) with the address of a spec node 'sn2'.

```
Spointer (sn1, sn2);
SPECNODE *sn1, *sn2;
```

4.3.10 User-defined Functions

functions.c All user-defined functions (UDF's) are defined in this file. It is compiled separately and then linked with the BB manager. The file has the following structure:

```
#include <functions.h>
UDF name table
UDF definitions
```

See Appendix III for *functions.h*.

UDF name table This table is a data structure listing the names, addresses, and number of arguments for each function defined in the file. A macro facility is provided for building the table. The table has the following form:

```
BEGINFUNCLIST
FUNCTION(funcname1,func1,numberofargs1)
FUNCTION(funcname2,func2,numberofargs2)
...
FUNCTION(funcnamen,funcn,numberofargsn)
ENDFUNCLIST
```

where 'funcname' is a string for a function's name, 'func' is the function's address, and 'numberofargs' is the number of arguments to the function as shown below:

```
char *funcname1, *funcname2, ..., *funcnamen;
void (*func1)(), (*func2)(), ..., (*funcn)();
int numberofargs1, numberofargs2, ..., numberofargsn;
```

Every UDF defined in the file must have a FUNCTION entry in the table.

UDF definitions This block contains all of the UDF definitions. Each definition has the following form:

```
SPECNODE *function (numofargs, arguments)
int numofargs;
SPECNODE **arguments;
{ /* body of function */ }
```

Upon invocation, the BB manager passes to the function an integer (numofargs) denoting the number of arguments and a pointer (arguments) to an array of argument pointers. The arguments to the function must be specnodes. The function must return a specnode. See Appendix III for details.

Sudf This function builds a spec node for invoking the user-defined function with id 'fid'. The integer 'n' indicates the number of input parameters to the user-defined function, and 'sn1', 'sn2', ..., 'snn' are spec nodes for these parameters.

```
sn = Sudf (fid, n, sn1, sn2, ... , snn);
SPECNODE *sn, *sn1, *sn2, ... , *snn;
int fid, n;
```

4.3.11 Specification List Functions

Snewlist This function allocates a new spec list with id 'slid' and returns a pointer to it in 'sl'.

```
sl = Snewlist (slid);
SPECLIST *sl;
int slid;
```

Saddspec This function adds spec 'sp' to spec list 'sl' with spec id 'spid' and locking option 'lq'. The locking option may be BBLOCK or BBNOLOCK.

```
Saddspec (sl, sp, spid, lq);
SPECLIST *sl;
SPEC *sp;
int spid;
BBLOCKOPT lq;
```

Sfreelist This function frees the spec list pointed to by 'sl'. All specs in the list are also deleted.

```
Sfreelist (sl);
SPECLIST *sl;
```

4.4 BB Functions

4.4.1 Macro for Recovering ID Numbers

Macro Usage - Macros are provided for building a data structure for mapping identifiers in *template.init* and *functions.c* to ids. Ids must be declared and recovered before invoking most other functions in the BB software package. Modules using the macro facility must have the following structure:

1. *ID declarations*
2. *Macro block*
3. *BBinit()*
4. *Module body*

ID declarations IDs are declared as integers as follows:

```
int id1, id2, ..., idn;
```

Macro block This block contains all macros for binding identifier names to ids. It has the following structure:

```
BEGINIDLIST
<macro1>
<macro2>
...
<macron>
ENDIDLIST
```

where <macro1-n> can be the macros *BINDTOKENTYPEID*, *BINDLOCALATTRID*, *BINDGLOBALATTRID*,*BINDINTERNALATTRID*, *BINDUDFID*, *BINDUDTID*, *BINDENUMTYPEID*, or *BINDSCALARID*.

BINDTOKENTYPEID

This macro binds token type name 'ttname' to id 'ttid'. The macro has the following syntax:

```
BINDTOKENTYPEID(ttname,ttid)
char *ttname;
int ttid;
```

BINDLOCALATTRID

This macro binds local attribute name 'atname' in the token type with id 'ttid' to the attribute id 'aid'. The macro has the following syntax:

```
BINDLOCALATTRID(ttid,atname,aid)
char *atname;
int ttid, aid;
```

BINDGLOBALATTRID

This macro binds global attribute name 'atname' to the attribute id 'aid'. The macro has the following syntax:

```
BINDGLOBALATTRID(atname,aid)
char *atname;
int aid;
```

BINDINTERNALATTRID

This macro binds internal attribute name 'atname' to the attribute id 'aid'. The macro has the following syntax:

```
BINDINTERNALATTRID(atname,aid)
char *atname;
int aid;
```

BINDUDFID This macro binds the user-defined function (UDF) name 'fname' to the id 'fid'. The macro has the following syntax:

```
BINDUDFID(fname,fid)
char *fname;
int fid;
```

BINDUDTID This macro binds the user-defined type (UDT) name 'uname' to the id 'uid'. The macro has the following syntax:

```
BINDUDTID(uname,uid)
char *uname;
int uid;
```

BINDENUMTYPEID

This macro binds the enumerated type name 'etname' to the id 'etid'. The macro has the following syntax:

```
BINDENUMTYPEID(etname,etid)
char *etname;
int etid;
```

BINDSCALARID This macro binds the scalar name 'sname' in the enumerated type with id 'etid' to the id 'sid'. If 'etid' is TTYPE, the scalar 'sname' is a token type name. The macro has the following syntax:

```
BINDSCALARID(etid,sname,sid)
char *sname;
int etid,sid;
```

4.4.2 Function for Initialization

BBinit This routine establishes communication with the BB manager and binds ids to identifiers. BBinit must be called before any routines that require id numbers. The syntax is:

```
BBinit ();
```

4.4.3 Functions for Sending and Cancelling Spec Lists

BBsendstandinglist

This function sends the standing spec list 'sl' to the BB manager. The function call does not block. Whenever a token is inserted into the blackboard or when an existing token is unlocked that matches 'sl' the function 'func' is invoked to process the token. Standing spec lists remain active in the BB after matching a token. The syntax is:

```
BBsendstandinglist (sl, func);
SPECLIST *sl;
void (*func) ();
```

The function should be declared as:

```
func (t, sid, slid)
TOKEN *t;
int sid, slid; {
/* Body of function */
}
```

where 't' is a token matching the spec with id 'sid' from the spec list with id 'slid'.

BBsendoneshotlist

This function sends the one-shot spec list 'sl' to the BB manager. If 'wo' is BBNOWAIT, the calls returns whether the spec list matches an existing, unlocked token or not; if 'wo' is BBWAIT, the call blocks until a token is deposited or until an existing one is unlocked that matches the spec list. In both cases, the one-shot spec list is deleted by the BB manager. Matched tokens are deposited in the token queue. The syntax is:

```
BBsendoneshotlist(sl, wo);
SPECLIST *sl;
BBWAITOPT wo;
```

BBcancelspeclist

This function instructs the BB manager to delete the spec list with id 'spid'. The syntax is:

```
BBcancelspeclist (slid);
int slid;
```

4.4.4 Functions for Getting and Putting Tokens

BBgettoken

This function recovers a token from the token queue. Tokens matching one-shot spec lists are deposited in this queue. If the queue is empty, NULL is returned. The pointers 'ptrslid' and 'ptrspid' provide the ids of the spec list and spec within the spec list respectively that matched the token. The syntax is:

```
t = BBgettoken (ptrslid, ptrspid);
TOKEN *t;
int *ptrslid, *ptrspid;
```

BBputtoken

This function deposits a local token 't' into the BB database. A unique id number, assigned to the token by the BB manager, is returned by the function. The internal attributes of the local token are **not** updated. The syntax is:

```
id = BBputtoken (t);
TOKEN *t;
int id;
```

4.4.5 Functions for Accessing Tokens by ID Number

BBgetidtoken This function returns the token with id number 'id' from the BB database. If 'lq' is BBLOCK, the BB manager locks the token (BBNOLOCK leaves the token unlocked). BBGONE is returned if the token has been deleted, and BBPROTECTED is returned if a lock is requested and the token is already locked by another module; otherwise, BBSUCCESS is returned. The syntax is:

```
t = BBgetidtoken (id, lq, &status);
TOKEN *t;
int id;
BBLOCKOPT lq;
BBRESULT status;
```

BBreplacetoken This function sends the local token 't' to the BB, replacing the token with the same id number. Before a global token can be replaced, it *must* be retrieved with *BBgetidtoken* or *BBgettoken*. The token is unlocked. BBGONE is returned if the id fails to match an existing token, BBPROTECTED is returned if the module did not have the token locked, and BBSUCCESS is returned if the operation succeeded. The syntax is:

```
b = BBreplacetoken (t);
TOKEN *t;
BBRESULT b;
```

BBunlocktoken This function unlocks the token in the BB with id number 'id'. BBGONE is returned if the token does not exist, BBPROTECTED is returned if the module did not have the token locked, and BBSUCCESS is returned if the operation succeeded. The syntax is:

```
b = BBunlocktoken (tid);
int tid;
BBRESULT b;
```

BBdeletetoken This function deletes the token in the BB with id number 'id'. BBGONE is returned if the token does not exist, BBPROTECTED is returned if the module did not have the token locked, and BBSUCCESS is returned if the operation succeeded. The syntax is:

```
b = BBdeletetoken (tid);
int tid;
BBRESULT b;
```

4.5 Position Correction Functions in the Blackboard

4.5.1 World Map Based Corrections

BBcorrect3 This function corrects the vehicle's position to three degrees of freedom relative to the world map at time 't'. The parameters 'x' and 'y' give the translation from the origin of the world frame, and 'phi' is the orientation measured counter clockwise from the positive x-axis of the world frame.

```
BBcorrect3 (t, x, y, phi);
int t;
float x, y, phi;
```

BBcorrect2 This function corrects the vehicle's position to two degrees of freedom relative to the world map at time 't'. The vehicle is constrained to lie on the line defined by 'r' and 'th', where 'r' is the minimum distance to the line and 'th' is the angle between 'r' and the positive x-axis measured counterclockwise. The vehicle's orientation is set by 'phi'. The LMB corrects the position by straightening the vehicle's path since the last three-degree correction and intersecting it with the given line.

```
BBcorrect2 (t, r, th, phi);
int t;
float r, th, phi;
```

BBcorrect1 This function corrects the vehicle's position to degrees of freedom relative to the world map at time 't'. The vehicle's orientation is changed to 'phi' measured counterclockwise from the positive x-axis. The x and y parameters are left unchanged.

```
BBcorrect1 (t, phi);
int t;
float phi;
```

4.5.2 Vehicle Based Correction

BBvehcorrect3 This function corrects the vehicle's position at time 't2' to three degrees of freedom relative to the vehicle at time 't1'. The parameters 'x' and 'y' give the translation from the origin of the world frame, and 'phi' is the orientation measured counter clockwise from the positive x-axis of the vehicle frame.

```
BBvehcorrect3 (t1, t2, x, y, phi);
int t1, t2;
float x, y, phi;
```

BBvehcorrect2 This function corrects the vehicle's position at time 't2' to two degrees of freedom relative to the vehicle at time 't1'. The vehicle is constrained to lie on the line defined by 'r' and 'th', where 'r' is the minimum distance to the line and 'th' is the angle between 'r' and the positive x-axis measured counterclockwise. The vehicle's orientation is set by 'phi'. The LMB corrects the position by straightening the vehicle's path since the last three-degree correction and intersecting it with the given line.

```
BBvehcorrect2 (t1, t2, r, th, phi);
int t1, t2;
float r, th, phi;
```

BBvehcorrect1 This function corrects the vehicle's position at time 't2' to degrees of freedom

relative to the vehicle at time 't1'. The vehicle's orientation is changed to 'phi' measured counterclockwise from the positive x-axis. The x and y parameters are left unchanged.

```
BBvehcorrect1 (t1, t2, phi);
int t1, t2;
float phi;
```

BBsetdistance This function sets the distance travelled by the vehicle from time 't1' to time 't2' to 'dist'. Exactly how this measurement will be incorporated into the model is not yet specified.

```
BBsetdistance(t1, t2, dist);
int t1, t2;
float dist;
```

4.5.3 Dead Reckoning Command

BBmotionparams This function notifies the LMB that the vehicle's course has been altered at time 't'. The parameters are 'l-speed' for speed of the left wheels, 'r-speed' for speed of the right wheels, and 'width' for the width of the vehicle (length of axle).

```
BBmotionparams (t, l_speed, r_speed, width)
int t;
float l_speed, r_speed, width;
```

4.5.4 Status Command

BBpositionest This function returns an estimate of the vehicle's position with respect to the world coordinate frame at time 't', that is, the translation from the world frame in 'x' and 'y', and the orientation 'phi' measured counter clockwise from the positive x-axis of the world frame. If the position of the vehicle with respect to the world is unknown, FALSE is returned; otherwise, TRUE is returned.

```
BBpositionest (t, &x, &y, &phi);
int t;
float *x, *y, *phi;
```

4.6 Global Time Function

BBglobaltime This function returns the global time, that is, the time of the LMB.

```
t = BBglobaltime ();
int t;
```

5. Future Versions

The following capabilities and support tools will be added to the blackboard software package in future releases:

- *Lisp interface:* Module writers are constrained to using the C programming language. Future versions will include a module interface package for Lisp.

- *Debugging aids:* No mechanism is provided for testing the software short of "firing up" the entire system. Future versions will include an TTY-based module emulator.

I. Include File for Modules

```
/*************************************************************
*
*       File: module.h
*
*************************************************************/

/* Define TRUE and FALSE */
#define TRUE            1
#define FALSE           0

typedef enum {
        BBLOCK,
        BBNOLOCK
} BBLOCKOPT;

typedef enum {
        BBGONE,
        BBPROTECTED,
        BBSUCCESS
} BBRESULT;

typedef enum {
        BBWAIT,
        BBNOWAIT
} BBWAITOPT;

/*************************************************************
*
*       Pose data type
*
*************************************************************/

struct POSEHEADER {
        float matrix[4][4];
};

typedef struct POSEHEADER POSE;

/*************************************************************
*
*       POSE *pose;
*       int col, row;
*       float val;
*
*       Routines for handling poses:
*
*       pose = Pnewpose ();     allocates new pose
*       val = Preadelem (pose, row, col);
*                               reads the pose value in row and col
*       Pwriteelem (pose, row, col, val);
*                               writes the pose value into row and col
*       Pfreepose (pose);       de-allocates pose
```

```
*
*************************************************************/

POSE *Pnewpose ();
float Preadelem ();

/*************************************************************
*
*       Frame family data type
*
*************************************************************/

typedef enum {
        BFRM,
        BFRMFAM
} BASETYPE;

typedef enum {
        FFGLOBAL,
        FFLOCAL,
        FFVEHICLE
} FFTYPE;

struct FFPARAMBLKHEADER {
        int     FFtime; /* time value */
        POSE    *FFpose; /* pose value */
        struct FFPARAMBLKHEADER *FFnextblk; /* pointer to next
                                               param block */
};

typedef struct FFPARAMBLKHEADER FFPARAMBLK;

struct FFPARAMLISTHEADER {
        FFPARAMBLK      *FFfirstblk, /* pointer to first param block */
                        *FFlastblk; /* pointer to last param block */
        int             FFsize; /* number of blocks in linked list */
};

typedef struct FFPARAMLISTHEADER FFPARAMLIST;

typedef enum  {
        FFNOTIME,
        FFTIME
} FFTIMEARG;

struct FFHEADER {
        BASETYPE FFbasetype; /* base frame or frame family */
        union {
                struct FRAMEHEADER *FFbasefrm; /* if base frame */
                struct FFHEADER *FFbasefrmfam; /* if base frame family */
        } FFbase;
        FFTYPE FFnewtype; /* new frame family type */
        union {
                struct {
                        POSE    (*FFtransform)(); /* function to transform
```

```
                                                coordinates */
                        int     FFnumofparms; /* number of parms to trans */
                        FFTIMEARG FFtimearg; /* flag indicating time
                                                as parm */
                } FFlocalfrmfam;
                struct {
                        int     FFframefamid; /* global id */
                        FFPARAMLIST *FFparamlist; /* pointer to list of
                                                <time><pose> pairs */
                } FFglobalfrmfam;
        } FFdata;
        int     FFmark; /* Mark for internal usage */
};

typedef struct FFHEADER FRAMEFAMILY;

/*************************************************************
*
*       FRAMEFAMILY *bfrmfam, *frmfam, *frmfam1, *frmfam2;
*       FFPARAMLIST *plist;
*       FRAME *bfrm;
*       POSE (*transform) (), *ppose;
*       int numofpars, ptime, id;
*       BBRESULT status;
*       BBLOCKOPT lock;
*
*       Routines for handling frame families:
*
*       plist = FFmakeparamlist ();
*                               allocates a parameter list plist.
*       FFaddtoparamlist (plist, ppose, ptime);
*                               adds the pair <ptime><ppose> to
*                               parameter list plist.
*       FFfreeparamlist (plist);
*                               frees parameter list plist.
*       frmfam = FFmakeglobalfromFF (bfrmfam, plist);
*                               makes a global frame family frmfam with base
*                               frame family bfrmfam and a parameter list of
*                               plist.  The frame family bfrmfam must not be
*                               local.  The new frame family is sent to the
*                               BB and frmfam refers to it locally.
*                               Whenever frmfam is accessed, a new copy
*                               is recovered from the BB.
*       frmfam = FFmakeglobalfromF (bfrm, plist);
*                               makes a global frame family frmfam with
*                               base frame bfrm and a param list of plist
*                               The frame bfrm must be global.  The new
*                               frame family is sent to the BB and
*                               frmfam refers to it locally.
*                               Whenever frmfam is accessed, a new copy
*                               is recovered from the BB.
*       frmfam = FFmakelocalfromFF (bfrmfam, transform, numofpars);
*                               creates a local frame family frmfam with
*                               base frame family bfrmfam, and a transform
*                               function transform which takes numofpars
```

```
*                                parameters.
*       frmfam = FFmakelocalfromF (bfrm, transform, numofpars);
*                                creates a local frame  family frmfam with
*                                base frame bfrm, and a transform function
*                                transform which takes numofpars parameters.
*       id = FFgetglobalid (frmfam);
*                                retrieves the ID number id from global
*                                frame family frmfam.
*       frmfam = FFgetglobalfromid (frmfam, lock, status);
*                                retrieves the global frame family with ID
*                                number id and stores it in frmfam.  If
*                                lock is BBLOCK, the frame family is locked;
*                                if it's BBNOLOCK, the frame family is not
*                                locked.  BBPROTECTED is returned in status
*                                the frame family is already locked,
*                                otherwise BBSUCCESS.
*       FFgetglobalupdate (frmfam, lock);
*                                recovers the global frame family frmfam
*                                from the BB.  If lock is BBLOCK, the
*                                frame family is locked and no other
*                                module may change it.  If lock is BBNOLOCK,
*                                the frame family is read but not locked.
*                                BBPROTECTED is returned if the frame family
*                                is already locked, otherwise BBSUCCESS
*                                is returned.
*       FFputglobal (frmfam);
*                                writes an updated version of global frame
*                                family frmfam to the BB and unlocks it.
*                                BBPROTECTED is returned if the frame family
*                                is locked by another module, otherwise
*                                BBSUCCESS is returned.
*       FFunlockglobal (frmfam);
*                                unlocks the global frame family frmfam.
*                                BBPROTECTED is returned if the frame family
*                                is locked by another module, otherwise
*                                BBSUCCESS is returned.
*       FFwritebaseFF (frmfam1, frmfam2);
*                                loads the base frame family in frmfam1
*                                with frmfam2.  The base frame family
*                                is assumed to have been NULLFF.
*       FFwritebaseF (frmfam, bfrm);
*                                loads the base frame family in frmfam
*                                with global frame bfrm.  The base frame
*                                family is assumed to have been NULLFF.
*       FFcheckbase (frmfam);    returns BFRM if base in frmfam
*                                is a frame or BFRMFAM if a frame
*                                family.
*       FFreadbaseFF (frmfam);   returns base frame family of frmfam.
*       FFreadbaseF (frmfam);    returns base frame of frmfam.
*       FFwriteparamlist (frmfam, plist);
*                                writes parameter list plist into frame
*                                family frmfam.
*       FFreadparamlist (frmfam);
*                                returns the parameter list for frmfam.
*
```

```
*************************************************************/

FRAMEFAMILY *FFmakeglobalfromFF (), *FFmakeglobalfromF (),
            *FFmakelocalfromFF (), *FFmakelocalfromF (),
            *FFreadbaseFF (), *FFgetglobalfromid ();
FFPARAMLIST *FFmakeparamlist (), *FFreadparamlist ();
BBRESULT FFgetglobalupdate (), FFputglobal (), FFunlockglobal ();
struct FRAMEHEADER *FFreadbaseF ();
BASETYPE FFcheckbase ();

/* Declare the system frame family */
extern FRAMEFAMILY *VEHICLEFF;

/*************************************************************
*
*       Frame data type
*
*************************************************************/

typedef enum {
        FSELECTFRAME, /* frame made by selecting from family */
        FPOSEFRAME /* frame made by specifying a pose */
} FTRANSTYPE;

typedef enum {
        FGLOBAL,
        FLOCAL,
        FWORLD,
        FNULL
} FTYPE;

struct FRAMEHEADER {
        BASETYPE Fbasetype; /* base frame or frame family */
        union {
                struct FRAMEHEADER *Fbasefrm; /* if base frame */
                FRAMEFAMILY *Fbasefrmfam; /* if base frame family */
        } Fbase;
        FTYPE Fnewtype; /* new frame type */
        int     Ffrmid; /* frame id if global */
        FTRANSTYPE Ftranstype; /* transform type */
        union {
                POSE *Fpose; /* transform from base frame or family */
                struct {
                        int     Fnumofpars; /* number of parameters to
                                               transforms */
                        float   *Fparameters; /* pointer to a block of
                                                 parameters */
                        FFTIMEARG Ftimearg; /* indicates a time value
                                               was specified */
                        int     Ftimevalue; /* the time value */
                } Fselectframe;
        } Fdata;
        int     Fmark; /* mark for internal usage */
};
```

```
typedef struct FRAMEHEADER FRAME;

/*************************************************************
*
*       FRAME *frm, *bfrm, *frm1, *frm2;
*       FRAMEFAMILY *bfrmfam;
*       POSE *ppose;
*       FFTIME time_arg;
*       BBRESULT status;
*       BBLOCKOPT lock;
*       int numofpars, id, time;
*       float par1, par2, ..., parn;
*
*       Routines for handling frames:
*
*       frm = FselectlocalfromFF (bfrmfam, time_arg, time, numofpars,
*                            par1, par2, ..., parn);
*                               creates a frame frm by evaluating the
*                               numofpars par1, par2, ..., parn and time
*                               in local base frame family bfrmfam.
*       frm = FselectglobalfromFF (bfrmfam, time);
*                               creates a frame frm by evaluating the
*                               time argument in global base frame family
*                               bfrmfam.
*       frm = FmakeglobalfromFF (bfrmfam, ppose);
*                               makes a global frame frm with base frame
*                               family bfrmfam and a pose of ppose.  The
*                               frame family bfrmfam must not be local.
*                               The new frame is sent to the BB and frm
*                               refers to it locally.
*       frm = FmakeglobalfromF (bfrm, ppose);
*                               makes a global frame frm with base frame
*                               bfrm and a pose of ppose.  The frame
*                               bfrm must be global.  The new frame is sent
*                               to the BB and frm refers to it locally.
*       frm = FmakelocalfromFF (bfrmfam, ppose);
*                               makes a local frame frm with base frame
*                               family bfrmfam and a pose of ppose.
*       frm = FmakelocalfromF (bfrm, ppose);
*                               makes a local frame frm with base frame
*                               bfrm and a pose of ppose.
*       id = Fgetglobalid (frm);
*                               retrieves the ID number id from global
*                               frame frm.
*       frmfam = Fgetglobalfromid (frm, lock, status);
*                               retrieves the global frame with ID
*                               number id and stores it in frm.  If
*                               lock is BBLOCK, the frame is locked;
*                               if it's BBNOLOCK, the frame is not
*                               locked.  BBPROTECTED is returned in status
*                               the frame is already locked,
*                               otherwise BBSUCCESS.
*       Fgetglobalupdate (frm, lock);
*                               recovers the global frame frm from the
*                               BB. If lock is BBLOCK, frm is locked.
```

```
*                                If lock is BBNOLOCK, the frm is not locked.
*                                BBPROTECTED is returned if the frame is
*                                already locked, otherwise BBSUCCESS is
*                                returned.
*       Fputglobal (frm);        writes an updated version of global frame
*                                frm to the BB and unlocks it.  BBPROTECTED
*                                is returned if the frame is locked by
*                                another module, otherwise BBSUCCESS is
*                                returned.
*       Funlockglobal (frm); unlocks the global frame frm.  BBPROTECTED
*                                is returned if the frame is locked by
*                                another module, otherwise BBSUCCESS is
*                                returned.
*       FwritebaseFF (frm1, frmfam);
*                                loads the base frame family in frm1
*                                with frmfam.  The base frame family
*                                is assumed to have been NULLFF.
*       FwritebaseF (frm1, frm2);
*                                loads the base frame in frm1 with frm2.
*                                The base frame is assumed to have been
*                                NULLF.
*       Fcheckbase (frm);        returns BFRM if the base of frm is
*                                a frame and BFRMFAM if it's a family.
*       FreadbaseFF (frm);       returns the base frame family of frm.
*       FreadbaseF (frm);        returns the base frame of frm.
*       Fwritepose (frm, ppose);
*                                writes pose ppose into frame frm.
*       Freadpose (frm);         reads the pose from frame frm.
*       Fconvertfrm (frm1, ppose1, frm2);
*                                returns the pose from frame frm2 equivalent
*                                to pose ppose1 from frame frm1.
*
************************************************************/

FRAME *FselectlocalfromFF (), *FselectglobalfromFF (),
      *FmakeglobalfromFF (), *FmakeglobalfromF (),
      *FmakelocalfromFF (), *FmakelocalfromF (), *FreadbaseF (),
      *Fgetglobalfromid ();
FRAMEFAMILY *FreadbaseFF ();
FTRANSTYPE Freadtranstype ();
POSE *Freadpose (), *Fconvertfrm ();
BBRESULT Fgetglobalupdate (), Fputglobalupdate (), Funlockglobal ();
BASETYPE Fcheckbase ();
int Freadtime ();

/* Declare the system frames */
extern FRAME *WORLDF, *NULLF;

/************************************************************
*
*       Location data type
*
************************************************************/

typedef enum {
```

```
        LPOINT,
        LLINESEG,
        LPOLYGON,
        LSCATTER
} LPOINTSETTYPE;

struct LPOINTCOORS {
        float   Lcoors[4]; /* Column vector of x,y,z,w values */
};

typedef struct LPOINTCOORS LSPOINT;

struct LPOINTSETHEADER {
        LPOINTSETTYPE ltype; /* type of location */
        int     Lsize; /* number of points in location */
        LSPOINT *Lpoints; /* pointer to block of points */
};

typedef struct LPOINTSETHEADER LPOINTSET;

struct LOCATIONHEADER {
        BASETYPE Lbasetype; /* base is frame or frame family */
        union {
                FRAME   *Lframe; /* if frame */
                FRAMEFAMILY *Lframefam; /* if frame family */
        } Lbase;
        LPOINTSET *Lpointset; /* point set */
};

typedef struct LOCATIONHEADER LOCATION;

/*************************************************************
*
*       LPOINTSET *pset;
*       FFTYPE btype;
*       FRAME *frm;
*       FRAMEFAMILY *frmfam;
*       LOCATION *loc, *loc1, *loc2;
*       LPOINTSETTYPE ltype;
*       float x, y, z, x1, y1, z1, x2, y2, z2;
*       int lsize, test, id;
*
*       Routines for handling point sets and locations:
*
*       pset = Lnewpointset (ltype, lsize);
*                               creates a new point set pset of type
*                               ltype and maximum size lsize.
*       Lreadpoint (pset, i, &x, &y, &z);
*                               reads the point x,y,z from the ith point
*                               in point set pset
*       Lwritepoint (pset, i, x, y, z);
*                               adds the point x,y,z to the ith point
*                               in point set pset
*       Lpointsettype (pset);   returns type of point set pset
*       Lpointsetsize (pset);   returns size of point set pset
```

```
*       Lfreepointset (pset);   de-allocates the point set pset.
*       pset = Lpoint (x, y, z);
*                               creates a point set of type LPOINT
*                               and stores the point x,y,z in it.
*       pset = Llineseg (x1, y1, z1, x2, y2, z2);
*                               creates a point set of type LLINESEG
*                               and stores the endpoints x1,y1,z1 and
*                               x2,y2,z2 in it.
*       loc = LmakelocationF (frm, pset);
*                               creates a location loc from the points in
*                               pset with respect to coordinate frame frm.
*       loc = LmakelocationFF (frmfam, pset);
*                               creates a location loc from the points in
*                               pset with respect to coordinate frame
*                               family frmfam.
*       Lfreelocation (loc);    de-allocates location loc.
*       loc2 = Lconvertloc (loc1, btype, frm, frmfam);
*                               creates a location loc2 by converting
*                               loc1 to frame frm or frame family
*                               frmfam.  NULL is returned
*                               if the location cannot be converted.
*       btype = Lcheckbase (loc);
*                               returns the base type for location loc
*       frm = LgetbaseF (loc);  returns the base frame for location loc
*       frmfam = LgetbaseFF (loc);
*                               returns the base frame family for
*                               location loc
*       id = Lgetbaseid (loc);
*                               gets the ID number for global base frame
*                               or family of location loc.
*       pset = Lgetpointset (loc);
*                               gets the point set for location loc.
*
************************************************************/

LOCATION *LmakelocationF (), *LmakelocationFF (), *Lconvertloc ();
LPOINTSET *Lnewpointset (), *Lpoint (), *Llineseg (), *Lgetpointset ();
LPOINTSETTYPE Lpointsettype ();
BASETYPE Lcheckbase ();
FRAME *LgetbaseF ();
FRAMEFAMILY *LgetbaseFF ();

/************************************************************
*
*       Routines for comparing locations and calculating
*       secondary features:
*
*       LOCATION *l, *loc1, *loc2, *cent, *chull, *box, *axis;
*       float dist, area, diam, orien;
*       int test;
*
*       dist = Ldistance (loc1, loc2);
*                               computes distance between loc1 and loc2
*       dist = Ldistance3 (loc1, loc2);
*                               computes 3D distance between point
```

```
*                                   loc1 and loc2
*       cent = Lcentroid (loc1);
*                                   computes centroid of loc1
*       area = Larea (loc1);        computes area of loc1
*       diam = Ldiameter (loc1);
*                                   computes diameter of loc1
*       orien = Lorientation (loc1);
*                                   computes the orientation of loc1
*       chull = Lchull (loc1);      computes the convex hull of loc1
*       box = Lbox (loc1);          computes the minimum bounding rectangle
*                                   of loc1
*       axis = Laxis (loc1);        computes the axis of loc1
*       test = Loverlap (loc1, loc2);
*                                   tests whether loc1 overlaps loc2
*
************************************************************/

LOCATION *Lcentroid (), *Lchull (), *Lbox (), *Laxis ();
float Ldistance (), Ldistance3 (), Larea (), Ldiameter (), Lorientation ();

/************************************************************
*
*       User-defined types (UDT)
*
************************************************************/

struct UDTHEADER {
        int     Usize; /* size of the UDT in bytes */
        char    *Udata; /* user-defined data */
};

typedef struct UDTHEADER UDT;

/************************************************************
*
*       UDT routines:
*
*       UDT *u;
*       char *data;
*       int dsize;
*
*       u = Unewudt ();             allocates a new UDT u
*       Uassignudt (u, data, dsize);
*                                   loads UDT u with data of size
*                                   dsize pointed to by data
*       Ufreeudt (u);               de-allocates the UDT u
*
************************************************************/

UDT *Unewudt ();

/************************************************************
*
*       Array data type
*
```

```
*************************************************************/

typedef enum {
        AINTEGER,
        AFLOAT,
        ABOOLEAN,
        ASTRING,
        ALOCATION,
        AENUMERATED,
        AUDT,
        AARRAY
} ARRAYTYPE;

struct ARRBLKHEADER {
        int             Asize; /* size of block in elements */
        struct ARRBLKHEADER *Anextblk; /* next array block */
        union { /* pointer to a block of data of each type */
                int     *Aiblk;
                float   *Afblk;
                int     *Abblk;
                char    **Asblk;
                LOCATION **Alblk;
                int     *Aeblk;
                UDT     **Aublk;
                struct ARRAYHEADER **Aablk;
        } Aelem;
};

typedef struct ARRBLKHEADER ARRAYBLOCK;

struct ARRAYHEADER {
        ARRAYTYPE       Atype; /* type of elements in array */
        int             Atotalsize; /* total number of elements in array */
        ARRAYBLOCK      *Afirstblk, /*first block in array */
                        *Alastblk; /* last block in array */
};

typedef struct ARRAYHEADER ARRAY;

/*************************************************************
*
*       char *sval;
*       int eval, ival, bval, size, index, test, increase;
*       float fval;
*       LOCATION *lval;
*       UDT *uval;
*       ARRAY *aval, *array, *array1, *array2, *union, *intersection;
*
*       Routines for handling arrays:
*
*       Anewarray (type, size); creates a new array with the given
*                               type and size
*       Afreearray (array);     deallocates an array
*       Airead (array, index);  reads integer element of array with given
*                               index
```

```
*       Afread (array, index);  reads float element of array with given
*                               index
*       Abread (array, index);  reads boolean element of array with given
*                               index
*       Asread (array, index);  reads string element of array with given
*                               index
*       Aeread (array, index);  reads enumerated element of array with given
*                               index
*       Auread (array, index);  reads UDT element of array with given
*                               index
*       Alread (array, index);  reads location element of array with given
*                               index
*       Aaread (array, index);  reads array element of array with given
*                               index
*       Aiwrite (array, index, ival);
*                               writes integer element ival into array
*                               with given index
*       Afwrite (array, index, fval);
*                               writes float element fval into array
*                               with given index
*       Abwrite (array, index, bval);
*                               writes boolean element bval into array
*                               with given index
*       Aswrite (array, index, sval);
*                               writes string element sval into array
*                               with given index
*       Aewrite (array, index, eval);
*                               writes enumerated element eval into array
*                               with given index
*       Auwrite (array, index, uval);
*                               writes UDT element uval into array
*                               with given index
*       Alwrite (array, index, lval);
*                               writes location element lval into array
*                               with given index
*       Aawrite (array, index, aval);
*                               writes array element aval into array
*                               with given index
*       Asizeof (array);        returns the size of the array
*       Atypeof (array);        returns the type of the array
*       Aextend (array, increase);
*                               increases array by increase elements
*       Aiappend (array, ival); appends ival to array
*       Afappend (array, fval); appends fval to array
*       Abappend (array, bval); appends bval to array
*       Asappend (array, sval); appends sval to array
*       Aeappend (array, eval); appends eval to array
*       Auappend (array, uval); appends uval to array
*       Alappend (array, lval); appends lval to array
*       Aaappend (array, aval); appends aval to array
*       Aimember (array, ival); determines if ival is a member of array
*       Afmember (array, fval); determines if fval is a member of array
*       Abmember (array, bval); determines if bval is a member of array
*       Asmember (array, sval); determines if sval is a member of array
*       Aemember (array, eval); determines if eval is a member of array
```

```
*       Aumember (array, uval); determines if uval is a member of array
*       Almember (array, lval); determines if lval is a member of array
*       Aamember (array, aval); determines if aval is a member of array
*       Aintersection (array1, array2);
*                               returns intersection of array1 and array2
*       Aunion (array1, array2);
*                               returns union of array1 and array2
*       Aequal (array1, array2);
*                               determines if array1 equals array2
*       Amax (array);           returns index of maximum element of array
*       Amin (array);           returns index of minimum element of array
*       Asort (array);          sorts the elements of array in ascending
*                               order
*
************************************************************/

int Airead (), Abread (), Aeread (), Asizeof (), Amax (), Amin (),
    Aimember (), Afmember (), Abmember (), Asmember (), Aemember (),
    Aumember (), Almember (), Aamember (), Aequal ();
ARRAY *Anewarray (), *Aaread (), *Aintersection (), *Aunion (), *Asort ();
ARRAYTYPE Atypeof ();
float Afread();
char *Asread ();
LOCATION *Alread ();
UDT *Auread ();

/************************************************************
*
*       Token data structures
*
************************************************************/

/* Define ID types */
typedef enum {
        TOKENTYPEID,
        LOCALATTRID,
        GLOBALATTRID,
        INTERNALATTRID,
        UDFID,
        UDTID,
        ENUMTYPEID,
        SCALARID,
        LASTID
} IDTYPE;

struct ids {
        IDTYPE  idtype; /* ID type */
        int     *idclass; /* context for ID */
        char    *idname; /* string name of ID */
        int     *id; /* address of ID */
};

typedef struct ids IDENTRY;

#define BEGINIDLIST     IDENTRY idlist [] = {
```

```
#define ENDIDLIST       {LASTID, 0, "", 0}};

#define BINDTOKENTYPEID(tokentypename,tokentypeid) \
                        {TOKENTYPEID, 0, tokentypename, &tokentypeid},
#define BINDLOCALATTRID(tokentypeid,attrtypename,attrtypeid) \
                        {LOCALATTRID, &tokentypeid, attrtypename, \
                         &attrtypeid},
#define BINDGLOBALATTRID(attrtypename,attrtypeid) \
                        {GLOBALATTRID, 0, attrtypename, &attrtypeid},
#define BINDINTERNALATTRID(attrtypename,attrtypeid) \
                        {INTERNALATTRID, 0, attrtypename, &attrtypeid},
#define BINDUDFID(functionname,functionid) \
                        {UDFID, 0, functionname, &functionid},
#define BINDUDTID(udtname,udtid) \
                        {UDTID, 0, udtname, &udtid},
#define BINDENUMTYPEID(enumtypename,enumtypeid) \
                        {ENUMTYPEID, 0, enumtypename, &enumtypeid},
#define BINDSCALARID(enumtypeid,scalarname,scalarid) \
                        {SCALARID, &enumtypeid, scalarname, &scalarid},

union ATTRIBUTEHEADER {
        int     Aival;
        float   Afval;
        int     Abval;
        char    *Asval;
        int     Aeval;
        UDT     *Auval;
        LOCATION *Alval;
        ARRAY   *Aaval;
};

typedef union ATTRIBUTEHEADER ATTRIBUTE;

typedef ATTRIBUTE TOKEN;


/* Temporary definition of MAXINT and MAXFLOAT (replace these
   with real declarations!!!!) */
#define MAXINT          (1 << (sizeof(int) * 8 - 1)) -1
#define MININT          0 - MAXINT
#define MAXFLOAT        1e37
#define MINFLOAT        0 - MAXFLOAT

/* Define NULL values for each primitive type */
#define NULLINT         MAXINT
#define NULLFLOAT       MAXFLOAT
#define NULLBOOL        MAXINT
#define NULLSTRING      NULL
#define NULLENUM        MAXINT
#define NULLUDT         NULL
#define NULLLOC         NULL
#define NULLARR         NULL

/* Define other NULL data structures */
#define NULLTOKEN       NULL
```

```
#define NULLPOSE        NULL
#define NULLPSET        NULL
#define NULLPARAMLIST   NULL

/*************************************************************
*
*       Token routines:
*
*       char *ttname, *atname, *fname, *uname,
*            *etname, *vname, *sval;
*       int ttid, aid, fid, uid, etid, vid, eval, ival, bval;
*       float fval;
*       TOKEN *t;
*       LOCATION *lval;
*       ARRAY *aval;
*       UDT *uval;
*
*       Macros for recovering id numbers:
*
*       BEGINIDLIST             begins list of id bindings
*       BINDTOKENTYPEID(ttname,ttid)
*                               binds token type name ttname to id ttid
*       BINDLOCALATTRID(ttid,atname,aid)
*                               binds local attribute name atname in token
*                               type with id ttid to attribute id aid.
*       BINDGLOBALATTRID(atname,aid)
*                               binds global attribute name atname to
*                               attribute id aid
*       BINDINTERNALATTRID(atname,aid)
*                               binds internal attribute name atname to
*                               attribute id aid
*       BINDUDFID(fname,fid)
*                               binds function name fname to id fid
*       BINDUDTID(uname,uid)    binds user-defined type name uname
*                               to id uid
*       BINDENUMTYPEID(etname,etid)
*                               binds enumerated type name etname to
*                               id etid
*       BINDSCALARID(etid,vname,vid)
*                               binds enumerated scalar name vname in
*                               enumerated type with id etid to id vid
*       ENDIDLIST               ends list of id bindings
*
*       Routines for creating and deleting tokens:
*
*       t = Tnewtoken (type, ctime);
*                               allocates a new token t of type
*                               type and data collection time
*                               ctime
*       Tfreetoken (t);         de-allocates token t
*
*       Routines for reading and writing attribute values:
*
*       ival = Tiread (t, aid); reads integer attribute aid for token t
*       fval = Tfread (t, aid); reads float attribute aid for token t
```

```
*       bval = Tbread (t, aid); reads boolean attribute aid for token t
*       sval = Tsread (t, aid); reads string attribute aid for token t
*       eval = Teread (t, aid); reads enumerated attribute aid for token t
*       uval = Turead (t, aid); reads udt attribute aid for token t
*       lval = Tlread (t, aid); reads location attribute aid for token t
*       aval = Taread (t, aid); reads array attribute aid for token t
*       Tiwrite (t, aid, ival); writes integer ival into attribute aid of
*                               token t
*       Tfwrite (t, aid, fval); writes fload fval into attribute aid of
*                               token t
*       Tbwrite (t, aid, bval); writes boolean bval into attribute aid of
*                               token t
*       Tswrite (t, aid, sval); writes string sval into attribute aid of
*                               token t
*       Tewrite (t, aid, eval); writes enumerated type eval into attribute
*                               aid of token t
*       Tuwrite (t, aid, uval); writes udt uval into attribute aid of
*                               token t
*       Tlwrite (t, aid, lval); writes location lval into attribute aid of
*                               token t
*       Tawrite (t, aid, aval); writes array aval into attribute aid of
*                               token t
*
************************************************************/

int Tiread (), Tbread (), Teread ();
TOKEN *Tnewtoken ();
float Tfread ();
char *Tsread ();
LOCATION *Tlread ();
ARRAY *Taread ();
UDT *Turead ();

/************************************************************
*
*       Specification List data structures
*
************************************************************/

typedef enum {
        SEMPTY,
        SINDIRECT,
        SFUNCTION,
        SUDF,
        SATTRIBUTE,
        SARRELEMATTR,
        SINTEGER,
        SFLOAT,
        SBOOLEAN,
        SSTRING,
        SLOCATION,
        SENUMERATED,
        SUDT,
        SARRAY
} SPECNODETYPE;
```

```
struct SPECNODEHEADER {
        SPECNODETYPE type;
        union {
                struct SPECNODEHEADER *indirect; /* for indirect node */
                struct {
                   int  funcid, /* function id */
                        num; /* number of parms */
                   struct SPECNODEHEADER **parameters; /* list of parms*/
                } funcnode;
                struct {
                   int  attrid, /* attribute id */
                        index; /* index if attribute is an array */
                } attrnode;
                int     ival; /* each of the data types */
                float   fval;
                int     bval;
                char    *sval;
                LOCATION *lval;
                int     eval;
                UDT     *uval;
                ARRAY   *aval;
        } nodecontents;
};

typedef struct SPECNODEHEADER SPECNODE;

struct SPECHEADER {
        int     specid; /* spec id number */
        BBLOCKOPT speclock; /* locking option */
        SPECNODE *spectree; /* pointer to spec tree */
};

typedef struct SPECHEADER SPEC;

struct SPECLINKHEADER {
        struct SPECLINKHEADER *nextlink; /* next link in list */
        SPEC    *specnode; /* spec node in list */
};

typedef struct SPECLINKHEADER SPECLINK;

struct SPECLISTHEADER {
        int     Sslid; /* speclist id */
        void    (*func) (); /* function to process list */
        int     Ssize; /* number of specs in list */
        SPECLINK *SLfirstspec, /* pointer to the first speclink in list */
                 *SLlastspec; /* pointer to the last speclink in list */
};

typedef struct SPECLISTHEADER SPECLIST;


/*************************************************************
*
*       Specification routines:
```

```
*
*       SPECLIST *sl;
*       SPEC *sp;
*       BBLOCKOPT lq;
*       SPECNODE *sn, *sn1, *sn2;
*       int n, spid, slid, i, b, e;
*       float f;
*       char *s;
*       UDT *u;
*       LOCATION *l;
*
*       Routines for managing spec lists:
*
*       sl = Snewlist (slid);   allocates a new spec list sl with id slid
*       Sfreelist (sl);         frees spec list sl
*       Saddspec (sl, sp, spid, lq);
*                               adds spec sp to speclist sl with
*                               lockq lq
*
*       Routines for managing specs:
*
*       sp = Snewspec (sn);     makes a spec out of the spectree
*                               with root specnode sn
*       Sfreespec (sn);         de-allocates specnode sn
*
*       Routines for converting constants into specnodes (leaves):
*
*       sn = Siconst (i);       converts integer i to specnode sn
*       sn = Sfconst (f);       converts float f to specnode sn
*       sn = Sbconst (b);       converts boolean b to specnode sn
*       sn = Ssconst (s);       converts string s to specnode sn
*       sn = Seconst (e);       converts enumerated scalar with id e
*                               to specnode sn
*       sn = Slconst (l);       converts location l to specnode sn
*       sn = Suconst (u);       converts udt u to specnode sn
*       sn = Saconst (a);       converts array a to a specnode sn
*
*       Routines for converting attributes into specnodes (leaves):
*
*       sn = Sattribute (aid);  converts attribute with id aid to
*                               specnode sn
*       sn = Sarrelemattr (aid, index);
*                               converts array element with index
*                               in attribute with id aid to
*                               specnode sn
*
*       Routines for building function specnodes (vertices):
*
*       sn = Sand (sn1, sn2);   returns AND node for two booleans
*       sn = Sandn (n, sn1, sn2, ...);
*                               returns AND node for n booleans
*       sn = Sor (sn1, sn2);    returns OR node for two booleans
*       sn = Sorn (n, sn1, sn2, ...);
*                               returns OR node for n booleans
*       sn = Snot (sn1)         returns NOT node for boolean sn1
```

```
*       sn = Sequal (sn1, sn2); returns EQUAL node for any types
*       sn = Snequal (sn1, sn2);
*                               returns NOTEQUAL node for any types
*       sn = Sless (sn1, sn2);  returns LESSTHAN node for sn1 less than sn2
*       sn = Slesseq (sn1, sn2);
*                               returns LESSTHANOREQUALTO node for sn1
*                               less than or equal to sn2
*       sn = Sgreater (sn1, sn2);
*                               returns GREATERTHAN node for sn1
*                               greater than sn2
*       sn = Sgreatereq (sn1, sn2);
*                               returns GREATERTHANOREQUALTO node for sn1
*                               less than or equal to sn2
*       sn = Srange (sn1, sn2, sn3);
*                               returns RANGE node for sn1 between sn2
*                               and sn3 inclusive
*       sn = Sadd (sn1, sn2);   returns ADD node for sn1 + sn2
*       sn = Saddn (n, sn1, sn2, ...);
*                               returns ADDN node for sn1 + sn2 + ... + snn
*       sn = Ssub (sn1, sn2);   returns SUB node for sn1 - sn2
*       sn = Smultiply (sn1, sn2);
*                               returns MULTIPLY node for sn1 * sn2
*       sn = Smultiplyn (n, sn1, sn2, ...);
*                               returns MULTIPLYN node for sn1 * sn2 *
*                               ... * snn
*       sn = Sdivide (sn1, sn2);
*                               returns DIVIDE node for sn1 / sn2
*       sn = Ssubstring (sn1, sn2);
*                               returns SUBSTRING node for sn1 a substring
*                               of sn2
*       sn = Ssregexp (sn1, sn2);
*                               returns REGEXP node for sn1 a regular
*                               expression matching sn2
*       sn = Sdistance (sn1, sn2);
*                               returns DISTANCE node for minimum distance
*                               between sn1 and sn2
*       sn = Sdistance3 (sn1, sn2);
*                               returns DISTANCE3 node for minimum 3D
*                               distance between points sn1 and sn2
*       sn = Scentroid (sn1);   returns CENTROID node for the centroid
*                               or sn1
*       sn = Sarea (sn1);       returns AREA node for the area of sn1
*       sn = Sdiameter (sn1);   returns DIAMETER node for the diameter or
*                               largest width of sn1
*       sn = Sorientation (sn1);
*                               returns ORIENTATION node for orientation of
*                               principal axis of sn1
*       sn = Schull (sn1);      returns CHULL node for the convex hull
*                               of sn1
*       sn = Sbox (sn1);        returns BOX node for minimum bounding
*                               rectangle of sn1
*       sn = Saxis (sn1);       returns principal axis of sn1
*       sn = Soverlap (sn1, sn2);
*                               returns OVERLAP node for the overlap of
*                               sn1 and sn2
```

```
*
*       Function for invoking UDFs
*
*       sn = Sudf (fid, n, sn1, sn2, ...);
*                               returns user-defined function node
*                               for n operands
*
*       Functions for handling arrays
*
*       Smember (sn1, sn2);     tests if sn2 is a member of sn1
*       Sintersection (sn1, sn2);
*                               returns the intersection of sn1 and sn2
*       Sunion (sn1, sn2);      returns the union of sn1 and sn2
*       Saequal (sn1, sn2);     tests if sn1 and sn2 are equal
*                               (same elements)
*       Smax (sn1);             returns maximum element of sn1
*       Smin (sn1);             returns minimum element of sn1
*
*       Functions for building and using "slots"
*
*       sn = Sindirect ();      allocates a slot node sn
*       Spointer (sn1, sn2);    points sn1 to sn2
*
*************************************************************/

SPECLIST *Snewlist();
SPEC     *Snewspec ();
SPECNODE *Siconst (), *Sfconst (), *Sbconst (), *Ssconst (), *Slconst (),
         *Suconst (), *Saconst (), *Sudf (), *Sand (), *Sandn (), *Sor (),
         *Snot (), *Sequal (), *Snequal (), *Sgreater (), *Srange (),
         *Sadd (), *Saddn (), *Ssub (), *Ssubn (), *Smult (), *Smultn (),
         *Sdivide (), *Ssubstring (), *Ssregexp (), *Sdistance (), *Sorn (),
         *Scentroid (),  *Sarea (), *Sdiameter (), *Sorientation (),
         *Schull (), *Sbox (), *Saxis (), *Soverlap (), *Sattribute (),
         *Sindirect (), *Sarrelemattr (), *Smember (), *Sintersection (),
         *Sunion (), *Saequal (), *Smax (), *Smin (), *Sdistance3 ();

/*************************************************************
*
*       BB routines:
*
*       SPECLIST *sl;
*       BBLOCKOPT lq;
*       BBWAITOPT wo;
*       BBSTATUS status;
*       TOKEN *t;
*       BOOLEAN b;
*       LOCATION lval1, lval2;
*       void (*func) ();
*       int id, n, t1, t2, spid, slid, *ptrspid, *ptrslid;
*
*       Routine for initialization:
*
*       BBinit ();              establishes communication and loads ids
*
```

```
*       Routine for sending standing spec lists:
*
*       BBsendstandinglist (sl, func);
*                               sends standing spec list sl
*                               and function func to the BB
*
*       Routine for sending one-shot spec lists:
*
*       BBsendoneshotlist (sl, wo);
*                               sends one-shot spec list sl to the BB;
*                               if wo is BBNOWAIT, the call returns whether
*                               an existing token is matched or not; if
*                               wo is BBWAIT, the call blocks until a token
*                               is deposited that matches the speclist.
*
*       Routine for deleting spec lists:
*
*       BBcancelspeclist (slid);
*                               cancels spec list with id slid
*
*       Routines for getting and putting tokens:
*
*       t = BBgettoken (ptrslid, ptrspid);
*                               returns a pointer to the next token
*                               available from the BB; returns NULL
*                               if no tokens are available.  The speclist
*                               and spec ids which matched the token
*                               are pointed to by ptrslid and ptrspid.
*       BBputtoken (t);         sends token pointed to by t to the BB
*
*       Routines for accessing tokens by id number:
*
*       t = BBgetidtoken (id, lq, status);
*                               returns a pointer to the token with
*                               id number id; the token is locked if
*                               lq is BBLOCK.
*       b = BBreplacetoken (t); sends token t to the blackboard, replacing
*                               the token with the same id number.  BBGONE
*                               is returned if the id fails to match an
*                               existing token; BBPROTECTED is returned
*                               if another module has the token locked;
*                               otherwise, BBSUCCESS is returned.  Replaced
*                               tokens are unlocked
*       b = BBunlocktoken (id); unlocks token in BB with id number id;
*                               BBGONE is returned if id fails to match
*                               an existing token; BBPROTECTED is returned
*                               if another module has the token locked;
*                               otherwise, BBSUCCESS is returned
*       b = BBdeletetoken (id); deletes token in BB with id number id;
*                               BBGONE is returned if id fails to match
*                               an existing token; BBPROTECTED is returned
*                               if another module has the token locked;
*                               otherwise, BBSUCCESS is returned
*
*************************************************************/
```

```
int BBpoll (), BBputtoken ();
BBRESULT BBreplacetoken (), BBdeletetoken (), BBunlocktoken ();
TOKEN *BBgettoken (), *BBgetidtoken ();
```

II. Grammar for Token Template File

The syntax of the *template.init* file is given by the following grammar.

```
! Template file

<template-file> := <enum-type-dec> <template-file> |
                   <user-type-dec> <template-file> |
                   <array-type-dec> <template-file> |
                   <global-attribute-dec> <template-file> |
                   <token-type-dec> <template-file> |
                   <include-statement> <template-file> | e

! Token type declarations

<token-type-dec> := TOKEN <token-type-name> { <attribute-dec-list> } ;
<token-type-name> := <identifier>
<attribute-dec-list> := <attribute-dec> <attribute-dec-list> | e
<attribute-dec> := <attribute-name> : <attribute-type> ;
<attribute-name> := <identifier>
<attribute-type> := GLOBAL | <data-type>

! Global attribute declarations

<global-attribute-dec> := GLOBAL <global-attribute-name> :
                          <global-attribute-type> ;
<global-attribute-type> := <data-type>

! User defined type declarations

<user-type-dec> := UDT <user-type-name> ;
<user-type-name> := <identifier>

! Enumerated type declarations

<enum-type-dec> := ENUM <enum-type-name> = { <enum-value> <enum-list>
<enum-list> := , <enum-value> <enum-list> | } ;
<enum-type-name> := <identifier>
<enum-value> := <identifier>

! Array type declarations

<array-type-dec> := ARRAY <array-type-name> [ <number> ] OF <data-type> ;
<array-type-name> := <identifier>

! Data type declaration

<data-type> := INT | FLOAT | BOOL | STRING | LOCATION |
               <user-type-name> | <enum-type-name> |
               <array-type-name>

! Include statements

<include-statement> := INCLUDE <file-name> ;
```

```
<file-name> := <string>
```

A *number* is a sequence of up to 16 digits (0-9). An *identifier* is a sequence of up to 64 upper or lower case letters, exclamation point '!', number sign '#', minus sign '-', or underscore '_'. *Identifiers* must not begin with a digit. Letters in *identifiers* are case-folded when parsed. A *string* is a sequence of up to 128 ascii characters delimited by quotation marks '"'. Extra whitespace characters including spaces, tabs, carriage returns, and linefeeds are ignored by the parser. Comments are permitted and must be delimited by '/*' and '*/'.

III. Include File for User-defined Functions

```
/*************************************************************
*
*       File: functions.h
*
*************************************************************/

#ifndef SPECNODE
#include <module.h> /* for the SPECNODE type def */
#endif

/*************************************************************
*
*       Macro for building function name list
*
*************************************************************/

struct BBtableentry {
        char *BBfuncname;
        void (*BBfunc) ();
        int BBnumargs;
};

typedef BBtableentry BBtable;

#define BEGINFUNCLIST   BBtable BBfunclist [] =
#define ENDFUNCLIST     { "", 0, 0}};
#define FUNCTION(functionname,function,numberofargs) \
                        {functionname, function, numberofargs},

/*************************************************************
*
*       BEGINFUNCLIST           begins a list of function names and
*                               the number of their arguments
*       FUNCTION(functionname,function,numberofargs)
*                               stores the function name 'functionname'
*                               with the function's address 'function' and
*                               the 'numberofargs' number of arguments
*                               in the function list
*       ENDFUNCLIST             ends the list
*
*       Function declaration:
*
*       SPECNODE *userdeffunc (numofargs, arguments)
*       int numoargs;           /* number of arguments in list */
*       SPECNODE **arguments    /* pointer to a list of argument pointers */
*       { /* body of function */ }
*
*************************************************************/
```

IV. A Complete Example of a KS

The following example is a KS for detecting obstacles. The KS uses a standing spec for matching "sonar blobs" large enough to be obstacles (over 20cm tall). For each large blob, an obstacle token is created and inserted into the blackboard. The C code for the KS along with the *template.init* file are given.

```
GLOBAL HEIGHT : FLOAT;
GLOBAL CONFIDENCE : FLOAT;

TOKEN SONARBLOB {
        HEIGHT : GLOBAL;
        CONFIDENCE: GLOBAL;
};

TYPE OBSTACLE {
        HEIGHT : GLOBAL;
        CONFIDENCE : GLOBAL;
        SONAR-EVIDENCE : INT;
        VISION-EVIDENCE : INT;
        RED : INT;
        GREEN : INT;
        BLUE : INT;
};

------------------------------

/*****************
*
* KS for detecting obstacles
*
*****************/

#include <module.h>

#define SPECLISTID      1
#define SPECID          1

int ttypeid, tlocationid, sonarblobtypeid, tctimeid,
    ttidid, heightid, sonarblobid, confidenceid,
    obstacleid, sonarevidenceid, slid, spid;

BEGINIDLIST
BINDATTRID(INTERNALATTR, "ttype", ttypeid)
BINDSCALARID(TTYPE,"sonarblob",sonarblobtypeid)
BINDATTRID(INTERNALATTR,"tlocation",tlocationid)
BINDATTRID(INTERNALATTR,"tctime",tctimeid)
BINDATTRID(INTERNALATTR,"ttid", ttidid)
BINDATTRID(GLOBALATTR,"height",heightid)
BINDATTRID(GLOBALATTR,"confidence",confidenceid)
BINDTOKENTYPEID("sonarblob",sonarblobid)
BINDTOKENTYPEID("obstacle",obstacleid)
```

```
BINDATTRID(obstacleid,"sonar-evidence",sonarevidenceid)
ENDIDLIST

main () {

SPECLIST *sl;
SPEC *sp;
TOKEN *t, *tnew;

BBinit ();
sp = Snewspec (Sand (Sequal (Sattribute (ttypeid),
                             Seconst (sonarblobtypeid)),
                      Sgreater (Sattribute (heightid),
                                Sfconst (0.20))));
sl = Snewlist (SPECLISTID);
Saddspec (sl, sp, SPECID, BBNOLOCK);
while (TRUE) {
        BBsendoneshotlist (sl, WAIT);
        while ((t = BBgettoken (&slid, &spid)) != BBEMPTY) {
                tnew = Tnewtoken (obstacleid,
                                Tiread (t, tctimeid));
                Tlwrite (tnew, tlocationid, Tlread (t, tlocationid))
                Tfwrite (tnew, heightid, Tfread (t, heightid));
                Tfwrite (tnew, confidenceid,
                                Tfread (t, confidenceid));
                Tiwrite (tnew, sonarevidenceid,
                                Tiread (t, ttidid));
                BBputtoken (tnew);
                Tfreetoken (t);
                Tfreetoken (tnew);
        }
}}
```

V. Instructions for Using the Blackboard Package

The blackboard package currently resides on the SUN systems in the "/usr/alv/exp/bb" directory. As of yet, it is not configured to run on VAXEN.

V.1 Compiling Modules

The LMB Interface package resides in "/usr/alv/exp/bb/lib/liblmbint.a", and the inclusion file resides in "/usr/alv/exp/bb/include/module.h". In order to access the LMB interface library, the module programmer must add the following lines to his/her ".login" file:

```
setpath LPATH -i999 /usr/alv/exp/bb/lib
setpath CPATH -i999 /usr/alv/exp/bb/include
```

Thereafter, the inclusion file can be accessed with:

```
#include <module.h>
```

and the programmer's module can be compiled with:

```
cc -o module module.c -llmbint -lm -lcs -limg
```

The math, cs, and image libraries currently reside in "/usr/lib/libm.a", "/usr/cs/lib/libcs.a", and "/usr/alv/lib/libimg.a" respectively. These paths must also be added to LPATH.

V.2 Executing the System

The LMB resides in "/usr/alv/exp/bb/bin/lmb". In order to access this program, the module programmer must add the following line to his/her ".login" file:

```
setpath PATH -i999 /usr/alv/exp/bb/bin
```

Thereafter, the LMB can be executed by typing "lmb" to the shell. In order to boot the blackboard system, the user must follow following procedure:

1. Choose a machine on which to run the LMB (i.e. IUSA).

2. Login to IUSA and "cd" to a directory containing the "template.init" file.

3. Run the LMB by typing "lmb" to the shell. Enter the number of modules (N) that will interface to the LMB. The LMB will proceed to compile the "template.init" file and report any errors. If there are no errors, the LMB will wait until it receives N connections.

4. On any of the SUN systems, login and execute each module (in any order). The module

will prompt the user for the name of the machine on which the LMB is running. Enter the name (i.e. IUSA). Once all the modules have established a connection with the LMB, the entire system will begin synchronous execution.

5. In order to shutdown the system, interrupt each module and LMB. Future versions of the software will provide a more graceful shutdown procedure.

Table of Contents

List of Figures