Vehicle Level World Model Manager Interface Description

Stephen Balakirsky, stephen@nist.gov
NIST

1.Abstract

Developed over the course of two decades at the National Institute of Standards and Technology (NIST) and elsewhere, the Real-time Control System (RCS) provides a reference architecture and engineering methodology to aid in the design of complex control systems. In order to accomplish complex behaviors, RCS provides guidelines for the decomposition of control problems into a set of hierarchical control nodes each of which follows a "sense-model-act" paradigm. In this paradigm, sensory processing (SP) functions feed information into a world model (WM) and a behavior generation (BG) module makes decisions or "acts" based on commands from the level above and input from a value judgement (VJ) module in cooperation with the WM.

RCS may be further specialized into application-specific versions. One such version is 4-D/RCS [Albus, 1997] that is aimed at the design and implementation of control systems for intelligent autonomous vehicles for military scout missions.

An important part of the 4-D/RCS hierarchy is the world model. The goal of this paper is to describe the world model manager system that provides client applications access to a comprehensive world model. These client applications may perform such tasks as planning or display generation. This paper concentrates on the details of a "vehicle level node" implementation of the world model for the Demo III program [Shoemaker et al., 1998]. Emphasis is placed on interface flexibility and modularity.

1.1. Keywords

4D/RCS, world model, planning, behavior generation, Demo III, intelligent control.

2.Introduction

If we define intelligent control as control that causes a system to successfully perform complex physical tasks in the presence of uncertainty and unpredictability, then a long standing goal of NIST has been to create a reference model architecture that creates intelligent control. The RCS reference model architecture is one such architecture and it has been successfully applied to multiple diverse systems [Albus, 1995]. In order to provide intelligent control of complex systems, guidelines are provided for the decomposition of the problem into a set of hierarchical control nodes. The decomposition into the hierarchy is guided by control theory that takes into account such items as system response times and planning horizons. RCS has been further specialized into the application specific 4-D/RCS that is aimed at the design and implementation of control systems for intelligent autonomous vehicles for military scout missions. The "4-D" portion of the name refers to the integration of the VaMoRs [Dickmanns et al., 1994] approach to dynamic machine vision.

Each node of the 4-D/RCS hierarchy follows a "sense-model-act" paradigm and includes sensory processing, world modeling, value judgement, and behavior generation as depicted in Figure 1 [Albus, 1999]. In this paradigm, sensory processing functions filter and extract information to feed into a world model. The WM maintains this information over an area that matches the planning horizon of the module in a Knowledge Database (KD). The KD includes symbols and data structures containing information about entities, events, and knowledge of how the world behaves. In addition, the WM may provide simulation facilities to estimate the state of the world at the present or some future time. The behavior generation module utilizes these facilities in cooperation with the value judgment module to compute possible plans or courses of action. The role of the VJ module in the planning process is to compute costs, risks, and benefits of these courses of action. Finally, the BG module makes decisions or "acts" based on input from the VJ module in cooperation with the WM and transmits these decisions to the next lower level of the hierarchy.

The 4-D/RCS hierarchy contains seven levels, each of which is modeled after the generic level depicted in Figure 1. The levels range in planning scope from "servo control" planning for the moves of an individual actuator, to "battalion control" planning for the movement of a large group of vehicles. Each level is designed to function in a particular spatial and temporal scope. For each level, the temporal scope is based on the response time required for control of the vehicle and the spatial scope is based on the required planning horizon. As one moves higher up the hierarchy, the temporal and spatial scope increase while the resolution decreases. By using this scheme 4-D/RCS strives to maintain a constant level of complexity throughout the hierarchy.

The Vehicle level World Model Manager (VWMM) is responsible for the updating and maintenance of the vehicle level's model of the world and the self. The world model is based on a scrolling rectangular grid and supports the storage of multiple attributes for a given cell. A variety of point, line, and area query/set functions are provided that allows multiple applications to read or set data. More complex functions, such as methods for the computation of route cost, are also provided. The VWMM may be decomposed into three separate components that map into 4-D/RCS; the Vehicle Attribute Map (VAM) that maps to KD, the Vehicle World Model (VWM) that maps to the Simulator/Predictor, and the Vehicle Cost Generator (VCG) that maps to VJ. The three layers may be implemented in one or more actual programs. A block diagram for a system that contains two attribute maps may be seen in Figure 2.

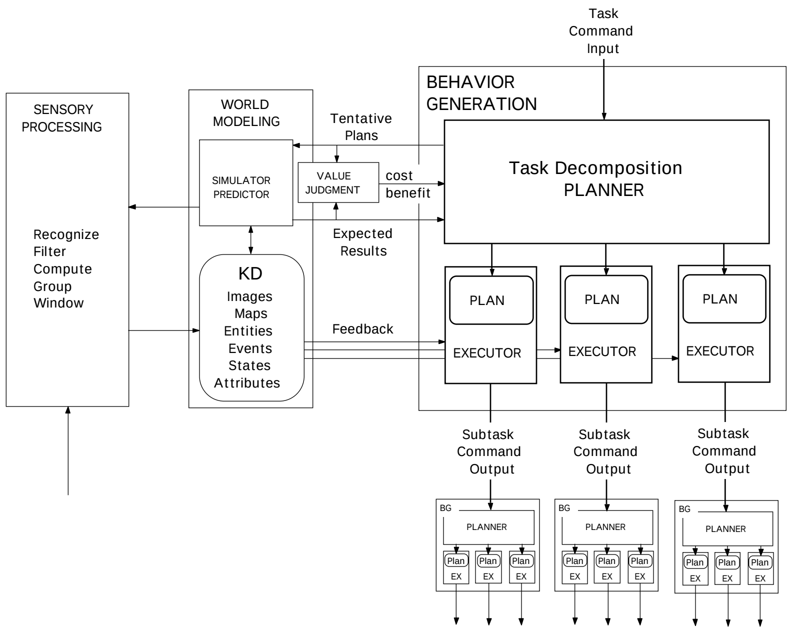

Figure 1: A typical 4-D/RCS computational node.

2.1. Vehicle Attribute Map

A vehicle attribute map is an array of pixels (cells) that contains information about a single class of attributes for the map area, such as elevation, hydrology, roads, or obstacles. Feature information that resides in the vehicle's long-term data store is loaded into the corresponding attribute layers during initialization. The VAM is implemented as a scrolling array in which pointers move instead of the data as the vehicle traverses terrain. At database instantiation, the extent (or size), resolution, and content type of the VAM is set. The VAM data cells currently supports C++ types of char, int, float, and double. Each VAM contains a generic header that contains such information as the map lower left coordinate, an offset value for every cell in the map, cell resolution, and map extent. Each VAM supports point, line, and area accesses (set/query). In addition, each VAM supports modifiers on the queries that allow queries such as "return location of all cells in the area > (<)(=) 'x'".

The vehicle location in the VAM's scrolling map is a parameter that may be changed. The location of the vehicle in the VAM sets the look-ahead/behind area of the VAM. For example, a centrally located vehicle will maintain as much information ahead of the vehicle as behind. A vehicle located at the trailing edge of the database will contain almost no data for behind the vehicle, but a maximum amount of data ahead of the vehicle.

A complete VWMM system contains several individual VAM "layers". Each VAM represents a single feature or a cluster of features from a single sensor. Figure 2 depicts a system that contains two VAM layers with one layer receiving data from an a priori data store and the other by subsystem level sensor processing. The Vehicle Obstacle VAM receives its data from the higher resolution (smaller spatial extents) subsystem level and contains vehicle resolution obstacles. The Vehicle Level Road VAM receives its data from a long-term data store (the a priori maps) and contains information about the road network. As shown in the example, several segment queries have been sent to the VAM layers from the planner through the WMM. The VCG portion of the individual VAM layers evaluate these segments and sends back a value that represents the "Roadness" or "Obstacleness" of the path segment. Note that since path segments are directional, the attribute computed may also be directional in nature (i.e. a one way road). The WMM will then use these values for determining a final cost value to pass back to the planner.

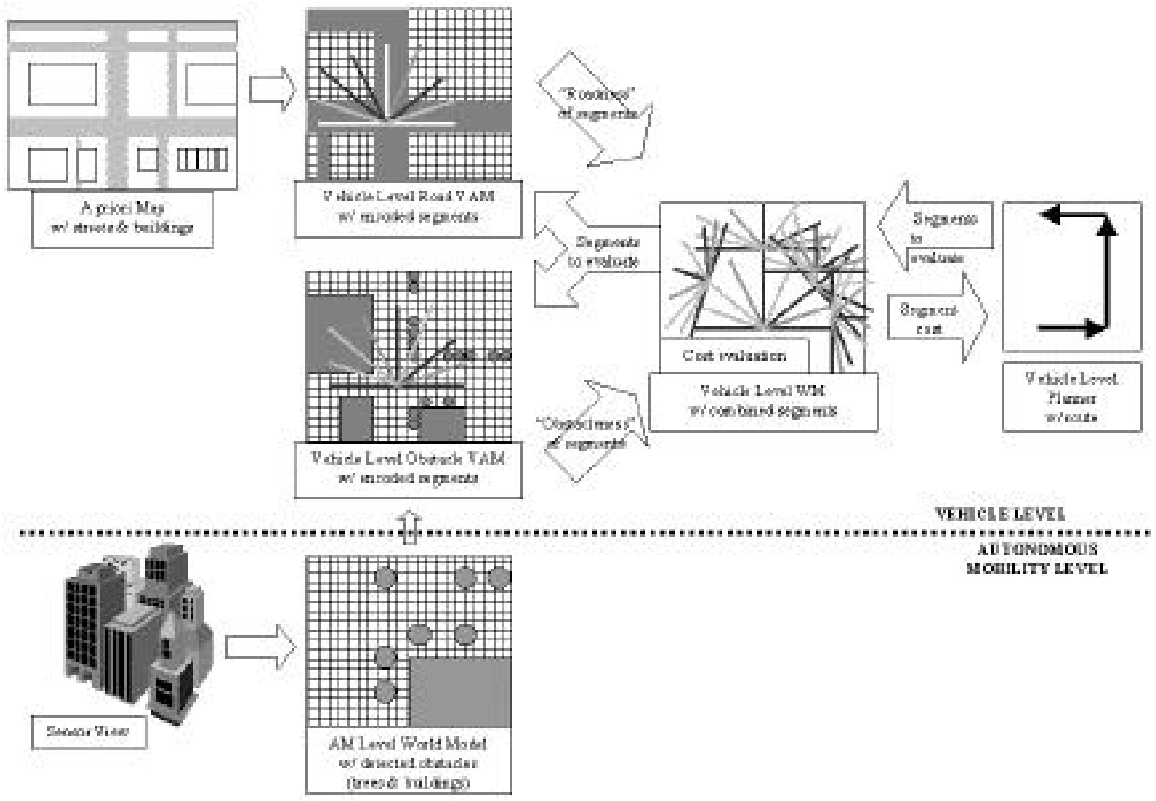

Figure 2: Simplified WMM component layout.

2.2. *Vehicle World Model*

The VWM is a collection of several components. They include the Vehicle Cost Map (VCM), Vehicle Entity Database, and Vehicle World Model Communications Interface (VWMCI). This paper will only address the Vehicle Cost Map and Communications Interface. The VCM is a data cache for the KD area of the 4-D/RCS hierarchy. Certain features that need to be stored in the WM are static in nature. For example, the cost of driving in a minefield does not change with vehicle speed or direction. These features are combined to form a single cost, and that cost is stored in the VCM. This eliminates the need to reevaluate static cell costs for multiple route segments that pass through the same cell. When a final cost is computed, this non-directional cost is combined with the dynamic component (dependent on direction, speed, etc.) by the VCG. In order to aid in dynamic re-planning, the VCM also contains a list of "dirty" cells. A dirty cell is a cell whose value has changed. Through the use of "dirty" cells, a planner will only need to recompute costs and trajectories in areas where sensor readings have changed.

The VWMCI is the central location from which all of the data in the database may be accessed. It hides all of the communication with the VAM layers from the user. As shown in Figure 2, the VWMCI receives queries from the planner and passes these queries to the individual VAM layers for evaluation. The WMM then uses the VCG to combine the returned attributes into a final cost value that is stored in the VCM and passed back to the behavior generator. By maintaining a list of all of the segments that have previously been given to BG, the WMM is able to notify BG if a segment cost changes and the segment should be marked as invalid due to the new sensor information.

2.3. Vehicle Cost Generator

The VCG is a family of routines used to calculate the cost of traversing a linear path segment. In the 4-D/RCS hierarchy, the VCG resides as a value judgement function. A path segment is defined as a start point and an end point (they may be the same point). The VCG computes both a static cost for each cell in a segment, which is stored in the VCM for possible later use, and a dynamic cost that is dependent on the direction of travel, vehicle speed, and other dynamic features. These costs are then combined, and the cost for the entire path segment is returned to the calling program. The current cost computation function forms the cost of the segment through the application of rules. The function allows for varying costs to be assigned to a segment based on the combination of attributes that are returned from the individual VAM layers. Using this technique, we have been able to demonstrate road following or road avoidance based on simple changes to the cost function.

2.4. Programing Language

All of the above systems are written in C++.

2.5. Operating System

All of the above systems are designed to run on Sun Solaris[1] systems. The systems will be ported at a later time to VxWorks.

3.World Model Manager – Planner Interaction

The vehicle level sensory processing, which feeds data to the WMM, is designed to work on a predetermined grid based layout. This grid is significantly different in form from the graph of randomly placed nodes that the Vehicle level Planner (VP) is designed to work on. The VP’s graph is considerably more sparse then a traditional four or eight connected grid structure, and may be viewed as containing path segments rather then adjacent grid elements. This gives rise for the need of a translator to translate the grid-based costs from the sensory processing to the path-based costs of the VP. Every effort has been made to utilize the VWMM as the interface layer between the grid-based sensors and the graph-based VP.

3.1. Communication Flow

The VWMM maintains a single input channel and several data channels. All command input and data sent to the VWMM travels through the input channel. Responses to the commands are returned over an assigned data channel. The Neutral Messaging Language (NML) [NIST,99] is used for all communications.

3.1.1. Initialization

Upon start-up, the VP will need to establish an NML connection to the VWMM command channel and to the data channel that matches its “connection number” field from the NML configuration file. The VP will then send an “WMInitialize” command to the VWWM. This command causes the VWMM to flush any stale buffers that are associated with the given channel number, mark all “dirty” flags as “not cached, value unchanged” for this channel, and prepare to service requests. For a complete description of the meaning of the “dirty” flags data see section 4.1.2. A “WMDone” message is returned as a result of this call.

[1] Certain commercial equipment, instruments, or materials are identified in this paper in order to adequately specify the experimental procedure. Such identification does not imply recommendation or endorsement by NIST, nor does it imply that the materials or equipment identified are necessarily best for the purpose.

3.1.2. Determination of node points

There are two classes of nodes in the graph that the VP is interested in gathering for its initial planning. These are random nodes, and critical nodes. The VP will submit a list of random nodes for evaluation as point costs to the VWMM (one or more "WMQueryPoints" message containing a variable number of points up to a maximum number). The VWMM will evaluate the cost of these points and return either a cost or error message for each point. A point not lying in the current extent of the database would cause a point error, but will not affect other points.

The VP will also query the VWMM for a list of critical nodes with a "WMQueryCritPoints" message. A critical node is defined as a location that may be of critical importance to the generation of a proper route. For example, the entrance to a bridge over a river would be designated as a critical point. Upon receipt of this request, the VWMM will send a list of all of the critical nodes in a "WMPointValues" message.

3.1.3. Segment cost evaluation

The VP sends "WMQuerySegments" requests to the VWMM to evaluate the cost of traversing path segments. The VP message will contain segment start and end points in vehicle relative coordinates as well as a segment id. This segment id will be used for all future communications about the segment between the VP and VWMM. The VWMM will compute the cost of traversing the segment, and send this cost along with the id back to the VP in a "WMSegmentValues" message. A higher cost than normal will be generated if both endpoints do not lie in the current map extents. This higher cost represents the unknown nature of some of the data. The segment cost will be a combination of a "dynamic" cost function that takes into account cost factors that vary with the direction, speed, position of the enemy, etc. and a "static" cost function that takes all other factors into account.

3.1.4. Planning cycle reset

In preparation to begin the next planning cycle, the VP will send a list of discarded segments to the VWMM in a "WMDiscardSegs" message. The VWMM will delete these segments from memory and then check to see which segments it has that may be invalid because of new sensor input or map extents. A list of these invalid segments will be returned to the VP over the data channel in a "WMDiscardSegs" message. The planning cycle it then repeated from the beginning (3.1.2).

3.2. Communication Messages

The BG system will communicate directly with the VWM. The actual communications messages used are detailed in the section 4.2 Vehicle World Model.

4.Detailed Description of the VWMM

The following section discusses in detail the complete implementation of the VWMM system. The VWMM was designed to be brought up in phases. Phase I is complete and has been delivered to the Demo III operator control unit integration contractor. Phase II is currently under development. Known benefits and limitations of the chosen implementation are discussed, along with possible alternatives and the phases of implementation.

The chosen implementation for the VWMM breaks the VWMM into two separate applications. The first includes the VAM and a section of the VCG. The second includes the VCM and the remaining section of the VCG. Depending on the number and nature of the subsystem level sensory processing elements, there may be several VAM modules in a given system.

The VCG was divided into two sections in an effort to increase processing efficiency. The most logical place for the expertise of computing cost on a given attribute would be in that attribute's cost map (the VAM) where all of the data is already contained. However, the expertise of how to combine several individual costs is best left to a module that has a more global sense of the environment: the VCM component of the VWM. Therefore the "Roadness", "Obstaclesness", or "Hilltopness" of a point or segment is computed in the VAM layers. These attributes are then combined into a cost that is based on mission goals and objectives in the WM.

4.1. *Vehicle Attribute Map (VAM)*

The VAM is closely coupled with the vehicle level sensory processing, and has the responsibility of acting as the bridge between two systems that are functioning on different time constants. It must gather data from the output of the subsystem level sensory processing (at the subsystem level rate) and publish this data for consumption by the VCM and section level processes at the vehicle level rate.

There are many different implementation options that are available for the VAM. This paper will focus on a distributed implementation of the VAM database. This has the advantage of allowing the system designer to place each VAM on the same CPU as the subsystem level sensory processing that it is so closely tied to. By residing on the sensory processing CPU, the communication bandwidth required over system wide buses will be reduced. This reduction comes from the slower data rate and lower resolution of the vehicle level data that will be transmitted from the VAM. In addition, the highly compute intensive cost map generation will be distributed among several processors. The main disadvantage of the distributed approach is that it will increase system latency when responding to planner requests. There is also the possibility of increased data storage requirements that may be created by implementing a data cache to decrease system latency.

Given that we are going to take a distributed approach, there are still at least two very different implementation approaches that could be used for the VAM. In the first implementation, the VAM operates as a control system on a fixed cycle time. It reads the subsystem level a fixed number of times, processes the information, and then publishes all of its data. This cycle is then repeated.

In the second implementation option, the VAM operates as a data server. The VAM still reads data from the subsystem level, but between reads it services client requests for data. There is no fixed cycle time for when outputs will be available, and some of the computations are performed on an as needed basis. This is the implementation that will be used for the VAM.

4.1.1. Data Flow

The VAM is a special hybrid case of an application that requires a constant time loop (data in from lower level sensory processing) and a blocking read on a command channel to quickly service clients (requests for data from the VWM). To adapt to this, the VAM uses a combination of blocking and non-blocking data reads. The VAM performs a blocking read on its command input channel with a time out set to a length of time shorter then that of the sensor update rate. If no commands are received, the VAM will return from its read and enter its sensor update loop. If a command is received, the VAM will service the command and then enter its sensor update loop. If the VAM enters the sensor update loop prematurely, a "no new data" message will be returned from the NML buffer and the VAM will once again block on the command channel.

The sensor update loop is where data is read in and processed from the subsystem level of the vehicle. This data is rolled up into vehicle resolution size cells and inserted into the database. Upon insertion, the appropriate database flags are set. Since some sensors send multiple reports of the same data, a check may be used to assure that new data is truly new and not merely repeated.

Details on the VAM command servicing are given in the sections below.

4.1.2. Database Implementation

The heart of the VAM is a fixed size scrolling database. Data is scrolled by providing access functions that use modulo arithmetic (remainder value functions). Through this technique, it is possible to keep the moving vehicle anywhere in the database (center, bottom, etc.) without ever moving data. The contents of the database include the rolled-up subsystem level sensory processing and a set of "dirty" flags.

Data that comes into the VAM from the subsystem level must be reduced to the vehicle level of resolution, and must have vehicle level attributes extracted from it. For example, 0.4-meter square obstacles from the subsystem level must be grouped and interpreted to fit into 4-meter squares on the vehicle level. It is envisioned that each VAM will perform unique processing algorithms to perform this reduction.

Two "dirty" flags are provided for each VWMM client to reduce the amount of VAM computations and the communications between the server and clients. The first flag is a cache flag and represents whether or not the client has a copy of the data in its cache. The second flag is a valid flag and represents whether or not the data value has changed since the client viewed it (making the data invalid). The detail of the operation of the dirty flags is provided in table Table 1. These flags allow for database queries based on whether or not data has been reported to the client in the past or has changed since the last report.

Table 1: Use of "dirty" flags.[2]

Action In	Current State		Next State		Action Out
	Data Cache	Data Valid	Data Cache	Data Valid	
Initialize	X	X	Clear	Set	None
Data In	Clear	Set	Clear	Set	None
	Set	X	Set	Clear	None
Update Cache	Clear	Set	Clear	Set	None
	Set	Clear	Clear	Set	Data to Client
	Set	Set	Set	Set	None
Read Data	Clear	Set	Set	Set	Data to Client
	Set	Clear	Set	Clear	Data to Client
	Set	Set	Set	Set	Data to Client

4.1.3. Communications to VAM

Communications with each VAM takes place over NML channels. Each VAM has a connection to a data input source, a single command channel, and multiple output data channels. Each VAM may service one or more attributes. For example, the LADAR processing develops both a map of obstacles and a map of areas of cover (areas that the vehicle can not be seen from above). The LADAR VAM will service requests for both of these attributes. The commands that are accepted by the VAM are described below. For a complete description of the command channel (in the form of "self-documenting" code) please see the appendices.

4.1.3.1.AMCenter

This command is used to center the VAM on the given location.

4.1.3.1.1.Parameters

center	– The new center of the database.
connectionNum	– The number of the client's connection to the VAM. This number must match the connection number in the NML configuration file.

4.1.3.1.2.Return over data channel

An AMDone message is returned over the data channel upon completion.

4.1.3.2.AMClose

This command is used to close a client connection to the VAM.

4.1.3.2.1.Parameters

connectionNum	– The number of the client's connection to the VAM. This number must match the connection number in the NML configuration file.

4.1.3.2.2.Return over data channel

An AMDone message is returned over the data channel upon completion.

4.1.3.3.AMInitialize

This command is used to connect a client to the VAM. It is called on initial connection to the VAM and whenever the client wishes to re-initialize or flush all existing data.

[2] Critical points may behave differently than the state table. See section 4.1.3.5 and 4.1.3.7 for details.

4.1.3.3.1.Parameters

connectionNum	– The number of the client’s connection to the VAM. This number must match the connection number in the NML configuration file. In the case of the VWMM, the NML channel referred to by this connection will be the VWMM input channel.
flushData	– If true, the database flushes all of its data. If false, the data valid flag is set to true and the data cache flag is set to false for all data for the given client, and the data is maintained.
bufferName	– Name of client command buffer.

4.1.3.3.2.Return over data channel

An AMDone message is returned over the data channel upon completion.

4.1.3.4.AMUpdateCache

This command is used to retrieve a list of data cells that have their data cache flag set and data valid flag cleared. The data cache flag is cleared and the data valid flag is set for all returned points.

4.1.3.4.1.Parameters

connectionNum	– The number of the client’s connection to the VAM. This number must match the connection number in the NML configuration file.
worldModelClient	– Information used by world model. Not used internally by VAM.

4.1.3.4.2.Return over data channel

An AMCacheReturn message is returned over the data channel on completion.

4.1.3.5.AMQueryCritPoints

This command is used to retrieve a series of critical point costs from the VAM. Sensor processing will have identified which points are critical. Critical points are such items as entrance and exit points for a mountain pass or bridge, or a gate in a fence. The client’s data cache flag is set for all critical points by this call.

4.1.3.5.1.Parameters

connectionNum	– The number of the client’s connection to the VAM. This number must match the connection number in the NML configuration file.

4.1.3.5.2.Return over data channel

An AMPointValues message is returned over the data channel on completion.

4.1.3.6.AMQueryPoints

This command is used to retrieve a series of point costs from the VAM. The point costs are returned on the assigned data channel. The data cache flag is set for any point that is returned.

4.1.3.6.1.Parameters

numPoints	– The number of points to return.
points	– An array of dblPoint structures not to exceed “MAX_QUERY_POINTS”. The Waypoint structure contains the UTM Northing and Easting.
connectionNum	– The number of the client’s connection to the VAM. This number must match the connection number in the NML configuration file

4.1.3.6.2.Return over data channel

An AMPointValues message is returned over the data channel on completion.

4.1.3.7. AMQuerySegments

This command is used to retrieve a series of segment costs from the VAM. The segment costs are returned on the assigned data channel. The algorithm outlined in section 4.2.1 is used to decompose the segment into individual points for the cost computation. All of these individual points will have their data cache flags set.

4.1.3.7.1. Parameters

numSegments	– The number of segments to return.
segments	– An array of amSegment structures not to exceed "MAX_QUERY_SEGMENTS".
connectionNum	– The number of the client's connection to the VWMM. This number must match the connection number in the NML configuration file.

4.1.3.7.2. Return over data channel

An AMSegmentValues message is returned over the data channel on completion.

4.1.3.8. AMQueryValues

This command is used to find points in the VAM that are greater then, equal to, or less then a given value. The data cache flag is set for any point that is returned.

4.1.3.8.1. Parameters

searchType	– Less than, equal to, or greater than.
connectionNum	– The number of the client's connection to the VAM. This number must match the connection number in the NML configuration file.
value	– The value to search for.

4.1.3.8.2. Return over data channel

An AMPointValues message is returned over the data channel on completion.

4.1.3.9. AMSetPoints

This command is used to set a series of point costs in the VAM. If the point's data cache flag is set, the point's data valid flag will be cleared.

4.1.3.9.1. Parameters

numPoints	– The number of points to set.
points	– An array of dblPointValue not to exceed "MAX_QUERY_POINTS".
connectionNum	– The number of the client's connection to the VWMM. This number must match the connection number in the NML configuration file. If the connectionNum is '0', no return message is generated.

4.1.3.9.2. Return over data channel

An AMDone message is returned over the data channel on completion.

4.1.4. Communications from VAM

4.1.4.1. AMDone

This message is returned to the calling application when the WM is finished with a command. It is only sent in the cases specified above.

4.1.4.1.1. Parameters

error	– Contains any error information from the command.

layerNum	– Layer identifier of VAM.

4.1.4.2.AMPointValues

This message is returned over the data channel as the result of point query to the VAM.

4.1.4.2.1.Parameters

layerNum	– Layer identifier of VAM.
numPoints	– The number of points returned.
points	– An array of dblPointValue not to exceed "MAX_QUERY_POINTS". The pointValues structure contains north and east offsets from the pointBase and a cost value.

4.1.4.3.AMSegmentValues

This message is returned over the data channel as the result of segment query to the VAM.

4.1.4.3.1.Parameters

layerNum	– Layer identifier of VAM.
numSegments	– The number of segments returned.
segmentValue	– An array of SegValues not to exceed "MAX_QUERY_SEGMENTS".

4.1.5. Phases of Implementation

In an effort to get the overall system up and running as quickly as possible, the VAM will be introduced in various phases. Each phase will add to and enhance the capabilities of the previous phase.

4.1.5.1.Phase I – Complete

The phase I implementation concentrates on providing a single, simple VAM for the VWMM to communicate with. The VAM supports the complete VAM command set but features greatly simplified algorithms for the computation of cell and segment cost.

4.1.5.2.Phase II – Currently Under Development

The phase II implementation will develop a second VAM layer. The most probable layers under consideration are either cover or roads. The second VAM layer will allow for more complete testing of the VWMM to include such things as VCG logic that combines the VAM layers. The exact choice for which VAM layer will be implemented will be based on data availability.

4.1.5.3.Phase III

The phase III implementation will introduce more robust algorithms for the computation of cell and segment cost for both VAM layers. This will greatly increase the cost map's robustness and accuracy.

4.2. Vehicle World Model (VWM)

The VWM is the main data interface to clients. It receives data requests, computes final cost values (in the Vehicle Cost Map component), and returns these costs to the clients. The overall data flow of the VWM is styled after a client/server architecture. In this architecture, the VWM blocks until a data request is received. It then performs the request and returns to its blocked state. Individual data flows for commands are detailed below.

The VWM may itself be viewed as consisting of several modules. They include the Vehicle Cost Map (VCM), Vehicle Entity Database, and Vehicle World Model Communications Interface. This paper will only address the VCM and Communications Interface. The VCM may be additionally decomposed into modules that include the Point-Segment Mapper and Cost Function.

4.2.1. Point-Segment Mapper

The vehicle planning system functions on a graph of connected nodes. A single numeric identifier labels these graph segments. As shown in Figure 3, the sensor systems function on an array (grid) of pixels that represent a plane in the 3-D space around the vehicle. This plane may be transformed through projections into a grid that conforms to the surface that the vehicle is traversing. An x-y coordinate pair represents the spatial location of the array cell in this projected plane. In order to translate between the two systems, a mapping is needed that maps an individual array cell to each graph segment that contains it and that maps each graph segment to a list of its cells. This is the function of the Point-Segment Mapper (PSM).

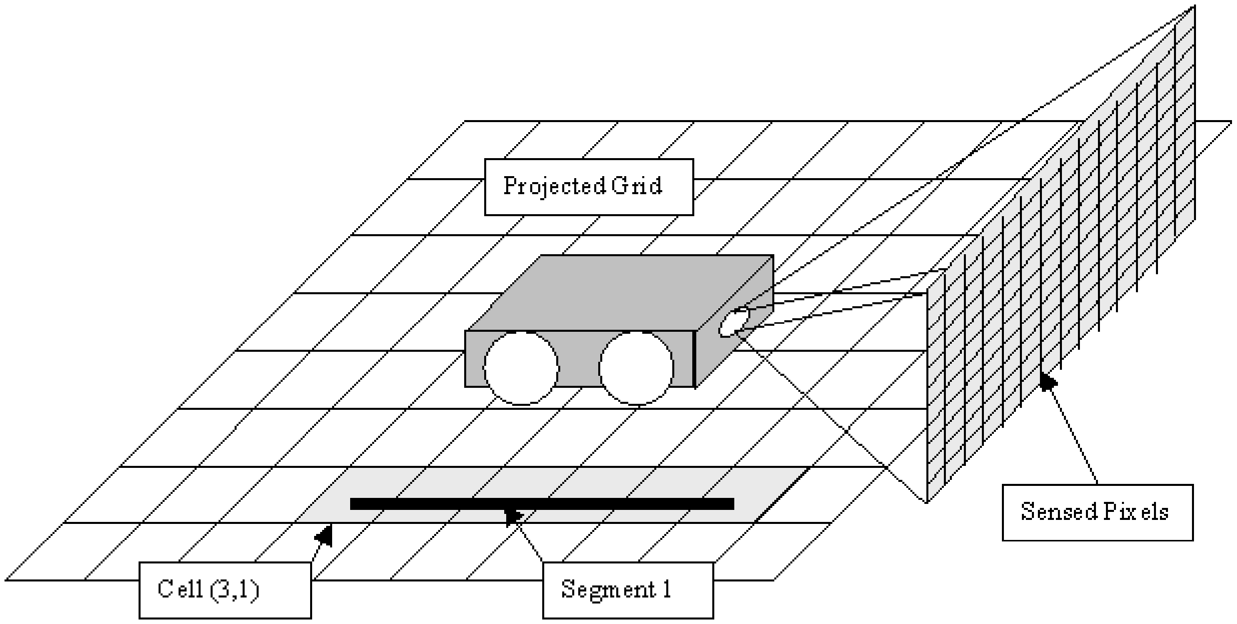

Figure 3: Point-Segment mapper definitions.

During WM initialization, memory is allocated for the storage of two two-dimensional tables. The table extents are determined by parameters input into the initialization call. The first table is the segment-mapping table (Table 3). This table is of dimension "*maxSegs*" by "*maxPixPerSeg*" and allows for quick access of pixels belonging to any segment. In addition to the pixel mapping values, this array contains a "cost" field and "received from" field which are used by the cost function. The cost field contains all of the directional costs associated with the segment and the received from field contains bit fields that represent which VAM layers have reported their costs. The second table is the pixel-mapping table and is implemented as a VAM layer. Each cell of the VAM contains a vector of length "*maxSegPerPix*" and allows for the quick access of segments that a pixel is a member of (Table 2). In addition to the segment mappings, the VAM also contains a "cost" field and "received from" field that are used by the cost function. This cost field contains all of the non-directional costs from the individual VAM layers and the received from field contains bit fields that represent which VAM layers have reported their costs.

Table 2: Pixel Mapping Table

Pixel #	Cost	Recv. From	Seg 1	Seg 2	---	maxSegPerPix
1	80	0xFF	10	23		
2						
---						
numOfMapPixels						

Table 3: Segment Mapping Table

Segment #	Cost	Recv. From	Pix 1	Pix 2	---	maxPixPerSeg
1	80	0xFF	10	23		
2						
---						
maxSegs						

This scheme requires a large amount of memory in exchange for the convenience of not having to manage linked lists of data. A trade-off that must still be examined is the processing overhead of linked-list maintenance verses the memory overhead of static memory.

4.2.1.1.Building the Tables

To update a path segment in the PSM, the start point and end point of the path are passed in. The PSM will then map the path to pixels, update both tables, and pass back a list of pixels that were mapped. The algorithm used for traversing the grid is detailed below.

1) Compute the direction of traversal to get from the start point (S) to end point (E) (north vs. south and east vs. west)
2) Initialize the first point in the output array.

$$i = 0$$

$$O_N(i) = S_N,\ O_E(i) = S_E$$

3) Check for end conditions.

$$if\,(O_N(i) == E_N(i)\ \&\ \&O_E(i) == E_E(i))END;$$

4) Determine if distance to end is greater in the north or east direction. If the distance is greater in the north direction, the next move will be due north or south. If the distance is greater in the east direction, the next move will be due east or west.
5) Add increment to direction determined above. Increment counter *i*.
6) Go to step 3.

4.2.1.2.Deleting Segments from the Table

There are two cases for freeing data from the table structures. The first occurs when a path segment is no longer valid or needed. The steps for removing a path segment are detailed below.

1) Obtain address of pixel from segment mapping table.
2) If address is “null” then end.
3) Go to pixel mapping table element pointed to by address and remove segment number from list.
4) Re-order the pixel mapping table’s segment list.
5) Remove pixel from segment list.
6) Repeat from (1).

In the second case, a pixel has been invalidated and all associated path segments must be removed. The steps to accomplish this are detailed below.

1) Obtain segment number from pixel mapping table.
2) If address is “null” then end.
3) Perform above procedure to remove segment.
4) Repeat from (1).

4.2.2. Cost Function

The cost function is used to compute the segment costs that are passed onto the planning system. This function will vary depending on the mission specifications, and is implemented as a function pointer that may point to one of many different mission specific functions. Commands from the supervising level will determine which cost function is applied, and the values of attributes used by the particular cost function.

The cost function must have the ability to calculate incremental cost on a given segment. The reason behind this is that the data for a particular segment may be obtained in an asynchronous manner. For example, the request for calculating the cost of a given segment may be delivered to several different VAM layers that respond to the request at different times. The cost value will need to be updated as each VAM provides its input. Whether or not a particular VAM layer has provided its cost input is stored in the “received from” flag area. The cost function may utilize the “cost” data area in any manner, as long as a final cost value is available once all of the inputs have been received.

4.2.2.1.Weighted Average

The simplest form of cost function will be a weighted average of all of the reporting VAM layers. Each VAM layer will report values in a predetermined range for each segment. These values will be positive to cause the planner to stay away from an area, and negative to cause the planner to be attracted to an area. In this scheme, the data from the VAM layer will be multiplied by the layer’s weight and added to the total in the “cost” data area. The final cost will automatically be available once all VAM layers have reported.

4.2.2.2.Complex Combination

There are many other means of computing a cost function. For these techniques, the “cost” data area may be used as a temporary storage area until all of the VAM layers have reported their data. For example, a certain number of bits may be used for each VAM. Once all VAM layers have reported their data (as indicated by the “received from” flags) the cost function would compute the final cost. The use of bit fields allows for complex rules to be created that need a combination of data from various VAM layers to compute the final cost.

4.2.3. Communications Interface

The VWM operates as a data server. It does not have a fixed cycle time, and blocks on reads of its command channel. Figure 4 and Figure 5 depict the overall communication flow of the program. Items enclosed by dashed lines will be implemented in later releases of the software. Specific algorithms for each of the commands along with a text description of the command and attributes for communicating with the VWM are given in the following sections. For a complete description of the implementation (in the form of “self-documenting” code) please see the appendixes.

4.2.3.1.WMClose

A client uses this command to close its connection to the VWM. All resources that were allocated specifically for that client are freed.

4.2.3.1.1.Parameters

connectionNum	– The number of the client’s connection to the VWMM. This number must match the connection number in the NML configuration file.

4.2.3.1.2.Return over data channel

A WMDone message is returned over the data channel on completion.

4.2.3.2.WMDiscardSegs

This command is used to inform the VWMM that certain SegmentIDs are no longer valid or needed. These segments are deleted, and then a garbage collection cycle is begun. The purpose of the garbage collection cycle is to update the planner’s cache.

The garbage collection cycle begins by the VWM sending “AMUpdateCache” commands to all of the VAM layers. The vehicle location is then updated by scrolling the VWM pixel mapping array database to correspond to the new vehicle location. Since the database is designed as a circular scrolling entity, any data that “falls off” the edge of the database must be cleared so that the memory may be reused. In order to clear this memory, segments must become invalid and are removed from the database. This data, however, is not truly invalid. The sensors have not updated it for some period of time, but the cost of the segment is still valid. Therefore, the planner is not told to delete the segments from its cache. It is allowed to keep them as valid entities.

As VAM layers report back from the cache updates, segments which have had cost changes due to new sensor input are removed. Once all VAM layers have reported, the final list of invalid segments is reported back to the planner.

4.2.3.2.1.Parameters

connectionNum	– The number of the client's connection to the VWMM. This number must match the connection number in the NML configuration file.
numSegs	– The number of segments to be deleted. Passing a zero for this number will cause the VWMM to only perform a garbage collection cycle.
segments	– An integer array of SegIDs to delete. The number of segments shall not exceed MAX_QUERY_SEGMENTS.
vehPosition	– A dblPoint structure that contains the current vehicle position. This is used to scroll the database.

4.2.3.2.2.Return over data channel

A WMDiscardSegs message is returned over the data channel on completion. The WMDiscardSegs command contains segments that the VWMM has determined are no longer valid, and should be deleted from the client. This message may contain zero segments.

4.2.3.3.WMInitialize

This command is used to connect a client to the VWMM. It is called on initial connection to the VWMM and whenever the client wishes to re-initialize or flush all existing data. The command causes the VWM to allocate memory for the Segment Mapping Array, Pixel Mapping Array, and the Segment Pending List.

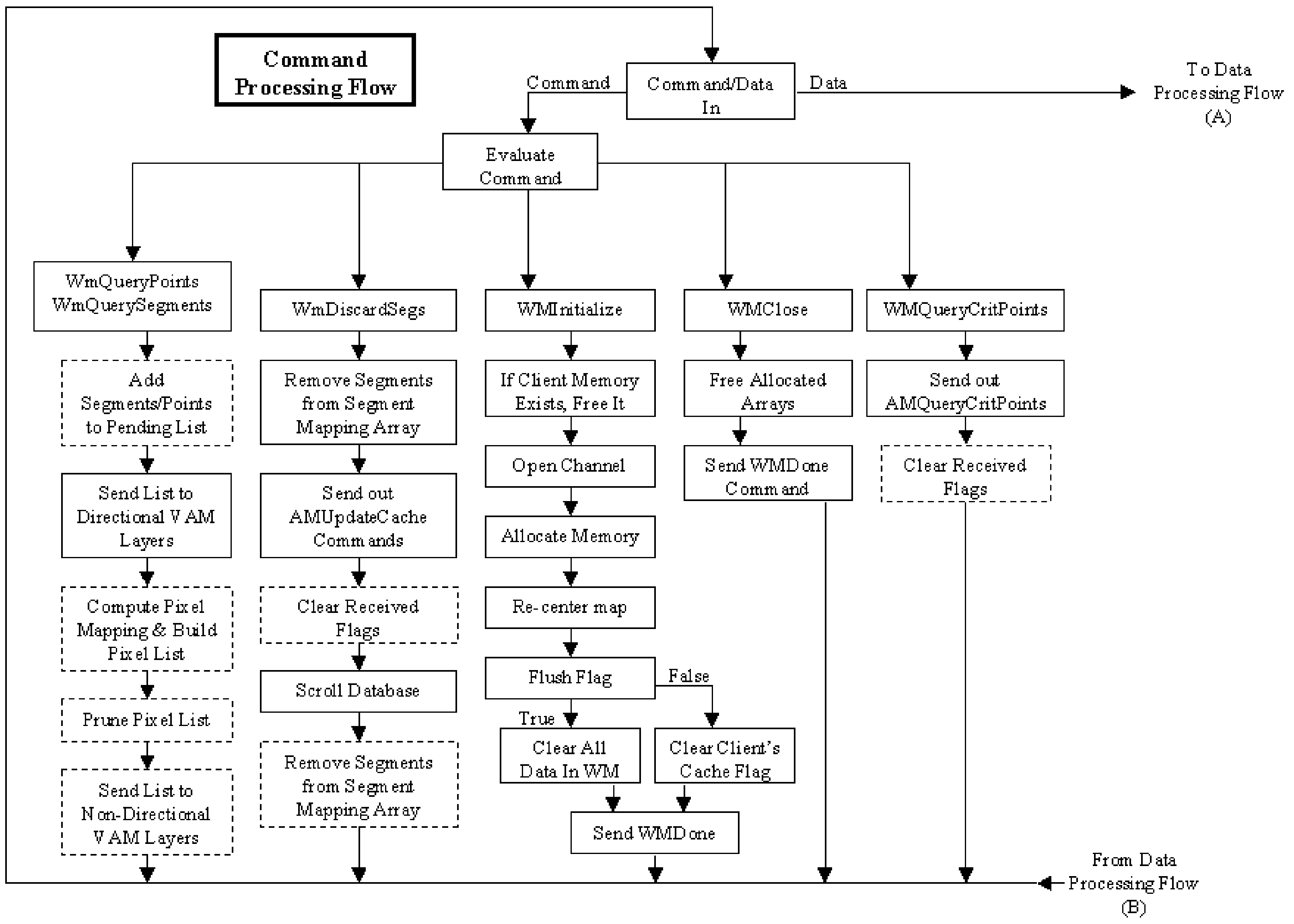
Command Processing Flow
Command/Data In
Command
Data
To Data Processing Flow (A)
Evaluate Command
WmQueryPoints WmQuerySegments
Add Segments/Points to Pending List
Send List to Directional VAM Layers
Compute Pixel Mapping & Build Pixel List
Prune Pixel List
Send List to Non-Directional VAM Layers
WmDiscardSegs
Remove Segments from Segment Mapping Array
Send out AMUpdateCache Commands
Clear Received Flags
Scroll Database
Remove Segments from Segment Mapping Array
WMInitialize
If Client Memory Exists, Free It
Open Channel
Allocate Memory
Re-center map
Flush Flag
True
False
Clear All Data In WM
Clear Client's Cache Flag
Send WMDone
WMClose
Free Allocated Arrays
Send WMDone Command
WMQueryCritPoints
Send out AMQueryCritPoints
Clear Received Flags
From Data Processing Flow (B)

Figure 4: Command Processing Flow

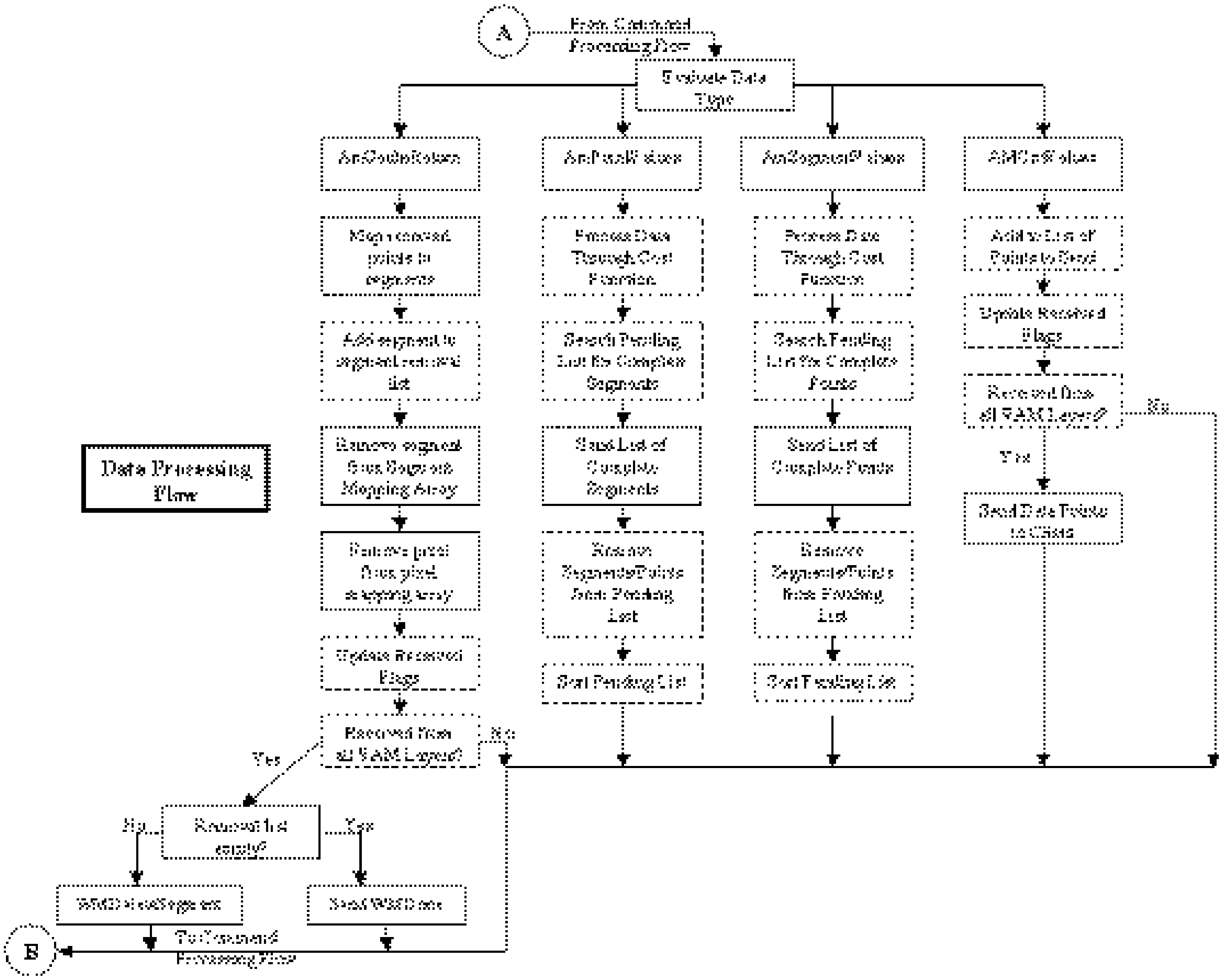
A
From Calculated Processing Flow
Evaluate Data Type
Process Data Through Cost Function
Search Pending List for Complete Segments
Search Pending List for Complete Points
Send List of Complete Segments
Send List of Complete Points
Remove Segments/Points from Pending List
Sort Pending List
Add to List of Points to Send
Update Received Flags
Received from all VAM Layers?
Send Data Points to Client
Data Processing Flow
Add segment to segment received list
Remove segment from Segment Mapping Array
Remove pixel from pixel mapping array
Yes
No
Received list empty?
Send WMD Data
B
To Command Processing Flow

Figure 5: Data Processing Flow

4.2.3.3.1.Parameters

connectionNum	– The number of the client's connection to the VWMM. This number must match the connection number in the NML configuration file.
flushData	– If true, the database flushes all of its VAM data. If false, the cache flag for all data is cleared for the given client, and the data is maintained.
maxSegments	– The maximum number of graph segments that the WM will be required to maintain.
maxPixPerSeg	– The maximum number of pixels contained in any segment.
maxSegPerPix	– The maximum number of segments that a pixel may belong to.
vehPosition	– The position of the vehicle.
bufferName	– The name of the NML buffer to return results to.

4.2.3.3.2.Return over data channel

A WMDone message is returned over the data channel on completion.

4.2.3.4.WMQueryCritPoints

This command is used to retrieve a series of critical point costs from the VWMM. The individual VAM layers determine what points are critical points.

4.2.3.4.1.Parameters

connectionNum	– The number of the client's connection to the VWMM. This number must match the connection number in the NML configuration file.

4.2.3.4.2.Return over data channel

A WMPointValues message is returned over the data channel on completion.

4.2.3.5.WMQueryPoints

This command is used to retrieve a series of point costs from the VWMM. The point costs are returned on the assigned data channel.

4.2.3.5.1.Parameters

numPoints	– The number of points to return.
points	– An array of dblPoint structures not to exceed "MAX_QUERY_POINTS".
connectionNum	– The number of the client's connection to the VWMM. This number must match the connection number in the NML configuration file.

4.2.3.5.2.Return over data channel

A WMPointValues message is returned over the data channel on completion.

4.2.3.6.WMQuerySegments

This command is used to retrieve a series of segment costs from the VWMM. The segment costs are returned on the assigned data channel.

4.2.3.6.1.Parameters

numSegments	– The number of segments to return.
segments	– An array of amSegment structures not to exceed "MAX_QUERY_SEGMENTS". The Segment structure contains two Waypoint structures (one each for the origin and the end) and a SegID.
connectionNum	– The number of the client's connection to the VWMM. This number must match the connection number in the NML configuration file

4.2.3.6.2.Return over data channel

A WMSegmentValues message is returned over the data channel on completion.

4.2.3.7.WMStatus

This command is used to retrieve status information from the world model.

4.2.3.7.1.Parameters

connectionNum	- The number of the client's connection to the VWMM. This number must match the connection number in the NML configuration file.

4.2.3.7.2.Return over data channel

A WMStatusValues message is returned over the data channel on completion.

4.2.4. Data Distribution Interface

4.2.4.1.WMDone

This message is returned to the calling application when the WM is finished with a command. It is only sent in the cases specified above.

4.2.4.1.1.Parameters

error	– Contains any error information from the command.

4.2.4.2.WMPointValues

This message is returned over the data channel as the result of point query to the VWMM.

4.2.4.2.1.Parameters

numPoints	– The number of points to return.
points	– An array of pointValues not to exceed "MAX_QUERY_POINTS". The pointValues structure contains north and east offsets from the pointBase and a cost value.

4.2.4.3.WMSegmentValues

This message is returned over the data channel as the result of segment query to the VWMM.

4.2.4.3.1.Parameters

numSegments	– The number of segments to return.
segmentValue	– An array of SegValues not to exceed "MAX_QUERY_SEGMENTS". The SegValues structure contains the cost of the segment and the SegID assiciated with that cost.

4.2.4.4.WMStatusValues

This message is returned over the data channel as the result of a status query to the VWMM.

4.2.4.4.1.Parameters

maxSegments	– The maximum number of graph segments that the WM will be required to maintain.
maxPixPerSeg	– The maximum number of pixels contained in any segment.
maxSegPerPix	– The maximum number of segments that a pixel may belong to.
vehPosition	– The position of the vehicle.
dbResolution	– The resolution of the database.

4.2.5. Phases of Implementation

In an effort to get the overall system up and running as quickly as possible, the VWM has been introduced in various phases. Each phase adds to and enhances the capabilities of the previous phase.

4.2.5.1.Phase I - Complete

The phase I implementation concentrated on providing a simple, VWM that is capable of supporting a single client and a single VAM. The VWM supports the complete VWM command set. However, internally everything was implemented as segments, no internal point structures were implemented.

4.2.5.2.Phase II – Under Development

The phase II implementation will develop the logic necessary for communicating with a second VAM layer. Internal point structures will also be implemented. The internal point structures should reduce the communications bandwidth of the system.

4.2.5.3.Phase III

The phase III implementation will introduce support for multiple clients.

5.Deliveries

Phase I was delivered to the Demo III integration contractor in December of 1999 for evaluation and integration. A second delivery (Phase II, date to be determined by the integration contractor) will be made which contains bug fixes and modifications requested by the contractor as well as the additional Phase II features. Phase III will be delivered at a date to be determined by the integration contractor. This will contain bug fixes and the additional features mentioned above.

6.References

Albus, J. S. 1995. The NIST Real-time Control System (RCS): An Application Survey. Proc. of the AAAI 1995 Spring Symposium Series, Stanford University, Menlo Park, CA.

Albus, J. S. 1997. 4-D/RCS: A Reference Model Architecture Demo III. National Institute of Standards and Technology, Gaithersburg, MD, NISTIR 5994.

Albus, J.S. 1999, 4-D/RCS Reference Model Architecture for Unmanned Ground Vehicles, Proc. SPIE AeroSense, Orlando FL, Vol. 3693.

Dickmanns, E. D. et al. 1994. The Seeing Passenger Car “VaMoRS-P,” Proc. International Symposium on Intelligent Vehicles ‘I94, Paris, France, pp. 68-73.

NIST 1999. http://isd.cme.nist.gov/projects/rcs_lib/

Shoemaker, C. M. and Bornstein, J. A. 1998. Overview of the Demo III UGV Program. Proc. Of the SPIE Robotic and Semi-Robotic Ground Vehicle Technology, Vol. 3366, pp. 202-211.

7.Appendices

7.1. Acronyms

4-D/RCS:	4-D Real-time Control System, where “4-D” are the three spatial dimensions and the dimensions of time.
BG:	Behavior Generation
KD:	Knowledge Database
NIST:	National Institute of Standards and Technology
NML:	Neutral Machine Language
PSM:	Point-Segment Mapper
RCS:	Real-time Control System
SP:	Sensory Processing
VAM:	Vehicle Attribute Map

VCG:	Vehicle Cost Generator
VCM:	Vehicle Cost Map
VJ:	Value Judgement
VP:	Vehicle Planner
VWM:	Vehicle level World Model
VWMCI:	Vehicle World Model Communications Interface
VWMM:	Vehicle level World Model Manager
WM:	World Model

7.2. *Attribute map NML command channel*

```
/* file: amNML.hh
 * purpose: NML command set for attribute maps
 * author: Stephen Balakirsky
 *         NIST
 * date: Nov 1998
 */
#ifndef __amNML__
#define __amNML__

#include <rcs.hh>
#include <pointValue.hh>

#define AMCenter_TYPE           101
#define AMClose_TYPE            102
#define AMInitialize_TYPE       103
#define AMQueryCritPoints_TYPE  104
#define AMQueryPoints_TYPE      105
#define AMQuerySegments_TYPE    106
#define AMQueryValues_TYPE      107
#define AMSetPoints_TYPE        108
#define AMUpdateCache_TYPE      109
#define AMMapOutput_TYPE        110

/* search types for AMQueryValues */
#define AMSearchLessThan    0
#define AMSearchEqualTo     1
#define AMSearchGreatorThan 2

class AMCenter: public RCS_CMD_MSG
{
public:
  AMCenter():RCS_CMD_MSG(AMCenter_TYPE,sizeof(AMCenter)) {};
  void update(CMS *);

  dblPoint center;
  int connectionNum;
};

class AMClose: public RCS_CMD_MSG
{
public:
  AMClose():RCS_CMD_MSG(AMClose_TYPE,sizeof(AMClose)) {};

  void update(CMS *);

  int connectionNum;
};

class AMInitialize: public RCS_CMD_MSG
{
public:
  AMInitialize():RCS_CMD_MSG(AMInitialize_TYPE,
sizeof(AMInitialize)){};

  void update(CMS *);

  int connectionNum;
  int flushData;
  char bufferName[80]; // name of client command channel
};

class AMQueryCritPoints: public RCS_CMD_MSG
{
public:

AMQueryCritPoints():RCS_CMD_MSG(AMQueryCritPoints_TYPE,sizeof(AMQueryCr
itPoints)){};

  void update(CMS *);

  int connectionNum;
};

class AMQueryPoints: public RCS_CMD_MSG
{
public:
  AMQueryPoints():RCS_CMD_MSG(AMQueryPoints_TYPE,
sizeof(AMQueryPoints)) {};

  void update(CMS *);

  //  int numPoints;
  //  dblPoint points[MAX_QUERY_POINTS];
  DECLARE_NML_DYNAMIC_LENGTH_ARRAY( dblPoint, points, MAX_QUERY_POINTS
);
  int connectionNum;
};

class AMQuerySegments: public RCS_CMD_MSG
{
public:

AMQuerySegments():RCS_CMD_MSG(AMQuerySegments_TYPE,sizeof(AMQuerySegmen
ts)){};

  void update(CMS *);

  //  int numSegments;
  //  amSegment segments[MAX_QUERY_SEGMENTS];
  DECLARE_NML_DYNAMIC_LENGTH_ARRAY( amSegment, segments,
MAX_QUERY_SEGMENTS );
  int connectionNum;
};
```

```
class AMQueryValues: public RCS_CMD_MSG
{
public:
  AMQueryValues():RCS_CMD_MSG(AMQueryValues_TYPE,
sizeof(AMQueryValues)){};

  void update(CMS *);

  int searchType;
  double value;
  int connectionNum;
};

class AMSetPoints: public RCS_CMD_MSG
{
public:
  AMSetPoints():RCS_CMD_MSG(AMSetPoints_TYPE, sizeof(AMSetPoints)){};

  void update(CMS *);

  int connectionNum;
  //  int numPoints;
  //  dblPointValue points[MAX_QUERY_POINTS];
  DECLARE_NML_DYNAMIC_LENGTH_ARRAY( dblPointValue, points,
MAX_QUERY_POINTS );
};

class AMUpdateCache: public RCS_CMD_MSG
{
public:
  AMUpdateCache():RCS_CMD_MSG(AMUpdateCache_TYPE,
sizeof(AMUpdateCache)){};

  void update(CMS *);

  int connectionNum;
  int worldModelClient;
};

class AMMapOutput: public RCS_CMD_MSG
{
public:
  AMMapOutput():RCS_CMD_MSG(AMMapOutput_TYPE,sizeof(AMMapOutput)) {};
  void update(CMS *);

  int state; // 1-on 0-off
};

////////////////////////////////////////////////////////////////
// Message formatting function
int AM_format(NMLTYPE type, void *buf, CMS *cms);

#endif
```

7.3. World Model NML command channel

```
/* file: wmNML.hh
 * purpose: NML command set for world model manager
 * author: Stephen Balakirsky
 *         NIST
 * The #define xx_TYPE represent the commands that the world model
accepts.
 * Each xx_TYPE has a corresponding xx class which contains the
variables
 * necessary for each command.
 * date: Nov 1998
 */
#ifndef __wmNML__
#define __wmNML__

#include <rcs.hh>
#include <pointValue.hh>

#define MAX_WM_CLIENTS         1   // max. clients for the world model

#define WMClose_TYPE           201 // close a connection with the world
model
#define WMDiscardSegs_TYPE     202 // tell the world model to discard
segments
#define WMInitialize_TYPE      203 // initialize a connection to the wm
#define WMQueryCritPoints_TYPE 204 // get cost of critical points from
the wm
#define WMQueryPoints_TYPE     205 // get cost of specified points from
wm
#define WMQuerySegments_TYPE   206 // get cost of specified segments
from wm
#define WMStatus_TYPE          207 // get the status from the world
model

// messages sent back to WM as response to queries to AM
#define AMCacheReturn_TYPE     220 // am sending back cache updates to
wm
#define AMCritValues_TYPE      221 // am sending back critical points
#define AMDone_TYPE            222 // am done with given command
#define AMPointValues_TYPE     223 // am sending back point values
#define AMSegmentValues_TYPE   224 // am sending back segment values

// class structures
class WMClose: public RCS_CMD_MSG
{
public:
  WMClose():RCS_CMD_MSG(WMClose_TYPE, sizeof(WMClose)) {};
  void update(CMS *);

  int connectionNum; // the connection number of the client (in config
file)
};

class WMDiscardSegs: public RCS_CMD_MSG
{
public:
  WMDiscardSegs():RCS_CMD_MSG(WMDiscardSegs_TYPE,
sizeof(WMDiscardSegs)) {};
  void update(CMS *);

  int connectionNum; // the connection number of the client (in config
file)
  //  int numSegments;   // number of segments to discard
  //  int segments[MAX_QUERY_SEGMENTS]; // segment ids
  DECLARE_NML_DYNAMIC_LENGTH_ARRAY( int, segments, MAX_QUERY_SEGMENTS
);
  dblPoint vehPosition; // requested location of database center
};

class WMInitialize: public RCS_CMD_MSG
{
public:
  WMInitialize():RCS_CMD_MSG(WMInitialize_TYPE, sizeof(WMInitialize))
{};
  void update(CMS *);

  int connectionNum;    // the connection number of the client (in
config file)
  int flushData;        // flush all data from all layers of wm (1=yes)
  int maxSegments;      // max. number of segments in wm
  int maxPixPerSeg;     // max. length of segment in pixels
  int maxSegPerPix;     // max. number of segments that a pixel belongs
to
  dblPoint vehPosition; // requested location of database center
  char bufferName[80];  // name of client command channel
};

class WMQueryCritPoints: public RCS_CMD_MSG
{
public:
  WMQueryCritPoints():RCS_CMD_MSG(WMQueryCritPoints_TYPE,
sizeof(WMQueryCritPoints)) {};
  void update(CMS *);
  int connectionNum; // the connection number of the client (in config
file)
};

class WMQueryPoints: public RCS_CMD_MSG
{
public:
  WMQueryPoints():RCS_CMD_MSG(WMQueryPoints_TYPE,sizeof(WMQueryPoints))
{};
  void update(CMS *);

  //  int numPoints;     // number of points to query
(<MAX_QUERY_POINTS)
  //  dblPoint points[MAX_QUERY_POINTS]; // location of points
  DECLARE_NML_DYNAMIC_LENGTH_ARRAY( dblPoint, points, MAX_QUERY_POINTS
);
  int connectionNum; // the connection number of the client (in config
file)
};
```

```
class WMQuerySegments: public RCS_CMD_MSG
{
public:
WMQuerySegments():RCS_CMD_MSG(WMQuerySegments_TYPE,sizeof(WMQuerySegmen
ts)) {};
  void update(CMS *);

  //  int numSegments;   // number of segments to query
(<MAX_QUERY_SEGMENTS)
  //  amSegment segments[MAX_QUERY_SEGMENTS]; // location of segments
  DECLARE_NML_DYNAMIC_LENGTH_ARRAY( amSegment, segments,
MAX_QUERY_SEGMENTS );
  int connectionNum; // the connection number of the client (in config
file)
};

class WMStatus: public RCS_CMD_MSG
{
public:
  WMStatus():RCS_CMD_MSG(WMStatus_TYPE, sizeof(WMStatus)) {};
  void update(CMS *);

  int connectionNum; // the connection number of the client (in config
file)
};

class AMCacheReturn: public RCS_CMD_MSG
{
public:
  AMCacheReturn():RCS_CMD_MSG(AMCacheReturn_TYPE,
sizeof(AMCacheReturn)) {};
  void update(CMS *);

  int requestingClient;             // wm client that requested update
  int layerNum;                     // layer number of VAM
  //  int numPoints;                    // number of points to update
  //  dblPoint points[MAX_QUERY_POINTS]; // location of actual points
  DECLARE_NML_DYNAMIC_LENGTH_ARRAY( dblPoint, points, MAX_QUERY_POINTS
);

};

class AMCritValues: public RCS_CMD_MSG
{
public:
  AMCritValues():RCS_CMD_MSG(AMCritValues_TYPE, sizeof(AMCritValues))
{};
  void update(CMS *);

  int layerNum;                         // layer number of VAM
  //  int numPoints;                        // number of points
returned
  //  dblPointValue points[MAX_QUERY_POINTS]; // actual points and
values
  DECLARE_NML_DYNAMIC_LENGTH_ARRAY( dblPointValue, points,
MAX_QUERY_POINTS );
};

class AMDone: public RCS_CMD_MSG
{
public:
  AMDone():RCS_CMD_MSG(AMDone_TYPE, sizeof(AMDone)) {};
  void update(CMS *);

  int layerNum;                        // layer number of VAM
  int error;                           // error return
};

class AMPointValues: public RCS_CMD_MSG
{
public:
  AMPointValues():RCS_CMD_MSG(AMPointValues_TYPE,
sizeof(AMPointValues)) {};
  void update(CMS *);

  int layerNum;                             // layer number of VAM
  //  int numPoints;                            // number of points
returned
  //  dblPointValue points[MAX_QUERY_POINTS]; // actual points and
values
  DECLARE_NML_DYNAMIC_LENGTH_ARRAY( dblPointValue, points,
MAX_QUERY_POINTS );
};

class AMSegmentValues: public RCS_CMD_MSG
{
public:
  AMSegmentValues():RCS_CMD_MSG(AMSegmentValues_TYPE,
sizeof(AMSegmentValues)) {};
  void update(CMS *);

  int layerNum;                                  // layer number of VAM
  //  int numSegments;                               // number of
segments returned
  //  amSegmentValue segments[MAX_QUERY_SEGMENTS]; // actual segments
and values
  DECLARE_NML_DYNAMIC_LENGTH_ARRAY( amSegmentValue, segments,
MAX_QUERY_SEGMENTS );
};

////////////////////////////////////////////////////////////////
// Message formatting function
int WM_format(NMLTYPE type, void *buf, CMS *cms);

#endif
```

7.4. WM NML status channel

```
/* file: wmStatNML.hh
 * purpose: NML command status set for world model manager
 * author: Stephen Balakirsky
 *         NIST
 * date: Nov 1998
 */
#ifndef __wmStatNML__
#define __wmStatNML__

#include <rcs.hh>
#include <pointValue.hh>

#define WMCritValues_TYPE    301 // returned critical points
#define WMDeleteSegment_TYPE 302 // delete segments from planner
#define WMDone_TYPE          303 // WM done with command
#define WMPointValues_TYPE   304 // returned values of points
#define WMSegmentValues_TYPE 305 // returned values of segments
#define WMStatusValues_TYPE  306 // returned values of status

// error messages
#define WM_SUCCESS             0
#define WM_ALLOC_ERROR         1

class WMCritValues: public RCS_CMD_MSG
{
public:
  WMCritValues():RCS_CMD_MSG(WMCritValues_TYPE, sizeof(WMCritValues))
{};
  void update(CMS *);

  //  int numPoints;                          // number of points
returned
  //  dblPointValue points[MAX_QUERY_POINTS]; // actual points and
values
  DECLARE_NML_DYNAMIC_LENGTH_ARRAY( dblPointValue, points,
MAX_QUERY_POINTS );
};

class WMDeleteSegment: public RCS_CMD_MSG
{
public:

WMDeleteSegment():RCS_CMD_MSG(WMDeleteSegment_TYPE,sizeof(WMDeleteSegme
nt)) {};
  void update(CMS *);

  //  int numSegments;                  // number of segments to delete
  //  int segments[MAX_QUERY_SEGMENTS]; // actual segments
  DECLARE_NML_DYNAMIC_LENGTH_ARRAY( int, segments, MAX_QUERY_SEGMENTS
);

};

class WMDone: public RCS_CMD_MSG
{
public:
  WMDone():RCS_CMD_MSG(WMDone_TYPE, sizeof(WMDone)) {};
  void update(CMS *);

  int error; // error return
};
```

```
class WMPointValues: public RCS_CMD_MSG
{
public:
  WMPointValues():RCS_CMD_MSG(WMPointValues_TYPE,
sizeof(WMPointValues)) {};
  void update(CMS *);

  //  int numPoints;                          // number of points
returned
  //  dblPointValue points[MAX_QUERY_POINTS]; // actual points and
values
  DECLARE_NML_DYNAMIC_LENGTH_ARRAY( dblPointValue, points,
MAX_QUERY_POINTS );
};

class WMSegmentValues: public RCS_CMD_MSG
{
public:

WMSegmentValues():RCS_CMD_MSG(WMSegmentValues_TYPE,sizeof(WMSegmentValu
es)) {};
  void update(CMS *);

  //  int numSegments;                            // number of
segments returned
  //  amSegmentValue segments[MAX_QUERY_SEGMENTS]; // actual segments
and values
  DECLARE_NML_DYNAMIC_LENGTH_ARRAY( amSegmentValue, segments,
MAX_QUERY_SEGMENTS );

};

class WMStatusValues: public RCS_CMD_MSG
{
public:
  WMStatusValues():RCS_CMD_MSG(WMStatusValues_TYPE,
sizeof(WMStatusValues)) {};

  void update(CMS *);

  int maxSegments;      // max. number of segments in wm
  int maxPixPerSeg;     // max. length of segment in pixels
  int maxSegPerPix;     // max. number of segments that a pixel belongs
to
  dblPoint vehPosition; // requested location of database center
  int dbResolution;     // The resolution of the database
};

////////////////////////////////////////////////////////////////
// Message formatting function
int WMStat_format(NMLTYPE type, void *buf, CMS *cms);

#endif
```