KR: an Efficient Knowledge Representation System

Dario Giuse

CMU-RI-TR-87-23

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

October 1987

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory under contract F33615-84-K-1520. The views and conclusions are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Table of Contents

List of Figures

Abstract

KR is a very efficient semantic network knowledge representation language implemented in Common Lisp. It provides basic mechanisms for knowledge representation which include user-defined inheritance, relations, and the usual repertoire of knowledge manipulation functions. The system is simple and compact and does not include some of the more complex functionality often found in other knowledge representation systems. Because of its simplicity, however, KR is highly optimized and offers good performance. These qualities make it suitable for many applications which require a mixture of good performance and flexible knowledge representation.

1. Introduction

The Dante Project [Giuse 86] is a broad investigation into the issue of building man-machine interfaces. Having developed and refined user-interface techniques, we now intend to combine what we have learned into one integrated environment: the Uniform Workstation Interface. The Uniform Workstation Interface will provide an integrated interface system for a very heterogeneous environment of workstations and multiprocessors connected through a local area network.

Work within the Dante Project proceeds along several dimensions. One such dimension is the development of so-called *interactions styles,* i.e., the basic components that together form the Uniform Workstation Interface. Each interaction style corresponds to a traditional user interface paradigm such as form filling or menu selection. Another dimension is the integration of the different interaction styles into a coherent unit. Our strategy is to achieve this goal through a *common representation system* that underlies the different components of the Uniform Workstation Interface. This representation system serves two purposes: It allows the different components to communicate with one another, and it provides an explicit representation of the user and the computing environment.

The knowledge representation system we are developing is named KR and is described in this document. It is implemented in Common Lisp[Steele 84] and provides compact, very efficient knowledge representation. KR was specifically developed with interactive applications in mind, and thus good performance was the primary design concern.

In spite of its simplicity, KR has many potential applications outside the immediate Dante environment. It is fairly well integrated with LISP and provides a natural extension of the LISP philosophy. Because of its efficiency, KR is ideally suited for a number of applications which require flexible representation of knowledge but cannot afford the performance overhead often associated with full-fledged knowledge representation systems. Moreover, KR is entirely written in portable Common Lisp and thus can be used for a wide variety of situations.

The first section of the document presents the general principles behind the design of KR and its relationship with similar knowledge representation systems. The following section describes the main concepts of the system, including the notions of schema, slot, value, inheritance, and relation. The central portion of the document describes in detail the functional interface to the system and gives a complete specification of ail the functions that comprise the interface. The following section describes how to load the system and presents several points about the best way to use KR to represent knowledge. Finally, the last section presents an evaluation of the actual performance of the system.

2. The KR Philosophy

This section briefly describes the most significant design choices in KR. Such choices have a profound effect on both the internal design of the system and on the appearance of user-level code that uses it. In this section we will assume that the reader is familiar with at least the general concepts of knowledge representation, and especially semantic network systems.

2-1 Structure of the System

KR is a knowledge representation system implemented in Common Lisp. It can be described as a semantic network system, since it stores knowledge as a network of chunks of information. Such systems are often referred to as Irame systems*.

The main feature of semantic network systems is the flexibility they provide in representing knowledge. Unlike more traditional data-storage systems, such as for instance relational data bases, semantic networks are built out of completely unstructured chunks. Each chunk (known as a *frame* or *schema*) can store any arbitrary piece of information and is not in any way restricted to a particular format or data structure. The general way to represent information is as *attribute-value pairs*.

A program or user is free to use a schema in any given way and to store as much information as needed in it. Moreover, schemata[1] can be modified as needed, even after they have been created. Relational data bases, by comparison, force each chunk to be in one of a small group of possible formats, and the format of a chunk cannot be modified after creation.

The other important property that KR shares with most semantic network systems is that certain values in a schema can be interpreted as links to other schemata. This enables the system to support very complex network structures, which can be freely extended and modified by application programs. KR provides simple mechanisms that enable an application program to specify the structure of a network and how the system should handle the existing knowledge.

2.2 Simple versus Complex

KR is a very simple knowledge representation system. Simplicity results in two desirable properties: The system is easy to maintain and extend, and it performs fairly well. While the first property is intuitive, the second property deserves a little explanation.

It is certainly true that fine-tuning a simple system for performance is easier than fine-tuning a complex system. This is indeed what happened with KR, which we first implemented in a straightforward way and then fine-tuned very extensively to achieve good performance.

A common objection to this approach, however, maintains that where a simple system fails to implement a particular functional capability the application program must implement that capability itself. This might conceivably introduce an overall loss of efficiency. Advocates of this objection conclude that a knowledge representation system must implement all possible functions that will ever be required.

We believe that this argument is flawed. Our personal observation has been that it is quite difficult to provide simultaneously the right type of extended functionality and the right performance. The crucial problem is that system implementors often cannot anticipate exactly how the extended functionality will be used; as a result, they have to implement it in a completely general fashion. In most situations, unfortunately, complete generality means poor performance. Given any particular problem, system-defined general purpose solutions are typically inferior to solutions that use problem-specific information. In some sense, system-defined general purpose solutions are equivalent to brute force algorithms, since they have no information whatsoever about the particular problem.

The ironical consequence is that the "extended" functionality often gets bypassed completely for performance reasons, and users end up implementing it differently. What was supposed to alleviate the problem ends up making things worse: Application programmers go through the frustration of first basing their code on system-defined functionality, then finding out that it is too slow, and finally having to re-implement it in an ad-hoc fashion.

KR takes an entirely different approach. It recognizes that extended functionality cannot be implemented efficiently without detailed knowledge of how it will be used. KR, therefore, makes it easy for an

---

[1]The plural of *schema* is *schemata*

application program to implement particular solutions, but does not try to provide a "complete" set of solutions for all possible problems. Rather than providing a monolithic system, complete unto itself, KR simply extends the LISP language. Functions expressible in LISP are never duplicated, and the system only implements the lowest level of knowledge representation.

A consequence of this design choice is that the application developer must be more involved with the details of the implementation. This seems entirely logical, however, in view of the previous considerations: The application developer has much better knowledge of the particular problem, and can ultimately provide a more efficient solution.

2.3 Error Handling

Most Algol- and Pascal-like programming languages perform type checking at compile time, the idea being to catch errors as soon as possible. One could say that such systems assume that the programmer is in error, unless proven otherwise. This idea is reasonable for novice programmers but is overly restrictive for experienced programmers. Significant portions of most large Pascal programs, for instance, are purely devoted to type conversions among similar objects (such as arrays of the same basic type that simply happen to have a different number of elements).

LISP, on the other hand, takes an entirely different approach: type checking is performed at run-time. Rather than trying to prevent errors at any cost, LISP gives the programmer more freedom and simply informs him or her when an error does occur. To put it differently, LISP assumes that the programmer is right unless proven otherwise.

A similar dichotomy exists in knowledge representation systems. Some systems (such as SRL [Wright and Fox 83] and CRL [Carnegie Group 86], for instance) take the position that the programmer is wrong unless proven otherwise. In CRL, for example, one cannot assign a value to a slot in a schema[2] without first creating the slot. Failure to do so causes an error, unless the programmer explicitly overrides the default.

Unlike those systems, KR follows the LISP philosophy when handling errors: it assumes that what the programmer is doing is correct, and tries to do the reasonable thing if possible. One could view this as a simple form of "Do What I Mean" (DWIM) behavior. As an example, consider again the case where a programmer tries to assign a value to a non-existing slot in a schema. Rather than generating an error, KR first creates the slot and then gives it the new value. This is almost always the behavior the programmer intended.

A consequence of this approach is that traditional patterns of usage become simplified. To continue our example, the typical pattern for assigning a value to a slot in CRL is as follows:

- Create an empty slot;
- Assign a new value.

or (even worse in terms of performance and code legibility):

- Check to see if the slot exists;
- If not, create an empty slot;
- Assign a new value.

The complexity arises purely from the desire to prevent error messages, rather than from the problem itself. The corresponding code in KR, on the other hand, is simply a value assignment. The programmer can assume that if the slot is not there the system will do the right thing and create the slot before using it.

---

[2] *SBQ* betow for an explanation of terms §te rafee, *sht and schema.*

We believe that this approach is more intuitive and leads to a more natural programming style.

3. Main Concepts in KR

3.1 Schema, Slot, Value

Knowledge in KR is represented as a network of schemata. The *schema* data structure is the basic unit of representation and consists of a *name*, a set of *slots*, and a set of *values* for each slot.

The user can assemble a *network* of schemata by using a schema name as the value in the slot of another schema, which causes the two schemata to become linked. Networks that correspond to arbitrarily complex graphs can be constructed this way.

The name of a schema is always a symbol. In particular, we recommend that users employ only *keywords* as schema names. This choice makes any schema directly accessible from any Lisp package. It also has another advantage: it reduces the possibility of conflicts. In the current implementation of KR, slots and values are stored on the property-list of the schema name. Using keywords, which normally have empty property-lists, makes conflicts with existing symbols much less likely.

A schema may have any number of *slots*, which are simply attribute-value pairs. The slot name indicates the attribute name; the slot values (if any) indicate its values. Slot names are also symbols, and again we recommend that keywords be used. All slots in a schema must have distinct names, but different schemata may very well have slots with the same name.

Each slot can contain zero or more *values*. Values are the actual data items stored in the schema, and may be of any Lisp type. KR provides functions to add, delete, and retrieve values from a given slot in a schema.

The printed representation of a schema shows the schema name followed by slot/value pairs, each one on a separate line. The whole schema is surrounded by double curly braces. Consider a schema for John's pet, Fido:

```
{{fido
  :is-a :dog :pet
  :owner :John
  :color :brown
  :age 5
  }}
```

The schema is named :FIDO and contains four slots, named *is-a*, *owner*, *color*, and *age*. The slot *age* contains one value, the integer 5. The slot *color* also contains one value, the keyword :BROWN. The slot *is-a* contains two values, :DOG and :PET.

3.2 Inheritance

The main function of values is to provide information about the object represented by a schema. In the previous example, for instance, a query for Fido's age would return the value "5".

Values can also perform another function: They can establish *connections between schemata*. Consider the *owner* slot in the example above: if we interpret :JOHN as a schema name, than the slot tells us that

the :FIDO schema is somehow related to the :JOHN schema. Given the name of the slot, we might reasonably assume this to mean that John owns Fido.

KR makes it possible to use such connections to perform *inheritance,* i.e., to control the way information is inherited by a particular schema from some other schema to which it is connected. Inheritance allows information to be arranged in a hierarchical fashion, with lower-level schemata inheriting most of their general features from higher-level nodes and possibly providing local refinements or modifications. A connection that enables inheritance of values is called a *relation* (see section 3.3).

3.2.1 An Example of Inheritance

The most common example of inheritance is provided by the *is-a* relation. If schema A is connected to schema B by the *is-a* relation,[3] then values that are not present in A may be inherited from B.

Consider, for instance, the :RDO schema in our previous example. If we were to ask "How many legs does Fido have?[11] we would not be able to find the correct answer by just looking at the :Fioo schema. Let us suppose, however, that we had also defined another schema:

```
{{dog
  :is-a:  :mammal
  :owner:
  :legs:  4}}
```

Since we said that Fido *is-a* dog, the value can be inherited from the :DOG schema through the *is-a* slot. The answer would thus be "Fido has 4 legs." Inheritance is possible in this case because *is-a* is defined by the system to be a relation.

3.2.2 The Role of Inheritance

Inheritance achieves three purposes: It reduces network size, it helps maintain consistency, and it allows local knowledge to override global knowledge. That inheritance reduces network size is obvious, since whenever a piece of information for a schema is the same as in a more general one, we need not repeat it in the more specialized schema. In the example above, we do not say that Fido has four legs, nor that it has a tail or that it barks. :Fioo can inherit all of these properties from the parent concept :DOG.

Inheritance helps maintain consistency because it allows any piece of information to be stored only once. When a change is needed, the information is simply modified in one place. Multiple updates are unnecessary since the change will be immediately apparent in the rest of the network.

Finally, inheritance allows local redefinition of global knowledge. A particular schema can assert a different, local value for some piece of inherited knowledge by simply providing a local slot with the same name and a new value. **Its** children would then inherit the new value, since the inheritance process stops as soon as a value is found.

3.3 Relations

Slots like *is-a* that enable knowledge to be inherited from other parts of a network are called *relations.* Inheritance along a relation is typically defined to proceed depth-first and may include any number of steps (in other words, the search terminates if a value is found or if no other schema can be reached via the relation).

[3] In other *words,* if the name of fie schema B appears as a value in the is-a slot of schema A.

KR allows the user to define new relations as desired. This is achieved through the function **create-relation** (see section 4.4), which performs all the necessary bookkeeping operations.

Any relation, including user-defined ones, may be declared to have an inverse link. If this is the case, KR will automatically generate an inverse link any time the relation is used to link one schema to another. Imagine, for instance, that we defined *pet-of* to be a relation having *has-pet* as its inverse. Writing :JOHN in the *pet-of* slot of :FIDO would automatically add :FIDO to the *has-pet* slot of :JOHN, thereby creating a reverse link.

3.4 Link Maintenance

KR automatically maintains all the links and inverse links described above, and the application programmer does not have to worry about them. This is probably one of the most convenient features of the system.

Imagine, for instance, that the two schemata A and B are linked by a certain relation and inverse relation. This means that schema A will have the name of schema B as the value in one of its slots. If the program decides to delete schema B, then, it is essential that the link from A to B also disappear. Failure to do so would cause the reference in A to be dangling: it would be an error to try to follow the reference, since the schema being pointed to (i.e., B) would no longer exist.

KR carefully keeps track of similar situations whenever they occur and corrects them instantly. The KR function that deletes schema B will automatically follow all the reverse pointers and make sure that any reference to B disappears as well.

In a similar manner, whenever the name of a schema is assigned as a value to a slot which happens to be a relation, KR automatically creates an inverse link. This ensures that the state of the knowledge representation system is completely consistent at any point in time, independent of the particular sequence of operations.

4. Program Interface

The KR program interface allows a program or a user to create and modify schemata, slots, and values. The interface is available as a set of functions defined and exported by the "KR" package. See section 5.1 for instructions on how to load KR onto your system.

4.1 Notation

In order to simplify the notation we will use the following conventions:

- The notation *<object>* indicates any Lisp object, which may or may not be a schema.
- The notation *<schema>* indicates that a function expects a valid schema as an argument. An error will typically be signalled if this is not the case.
- The notation *<slot>* indicates a valid slot name for a schema, i.e., a symbol (and more specifically a keyword).
- The notation *<schema-name>* indicates that a valid name for a schema (i.e., a keyword) must be supplied. It is not necessary for a schema by that name to already exist.

The notation "==>" will indicate the result of evaluating a LISP form. The notation "<==>" will indicate that

two forms are equivalent, i.e., one produces exactly the same effect as the other. Finally, the LISP comment line "; prints:" will be used to indicate the printed output produced by evaluating a LISP expression.

4.2 Example Schemata

The following sections use certain schemata as examples. We present here the definitions of those schemata once and for all:

```
;;; Define the :OWNER relation and its inverse, :HAS-PET.
(create-relation :owner T '(:has-pet))     ; T means inheritance is on.

(create-schema :dog
  (:is-a :mammal)
  (:legs 4))

(create-schema :fido
  (:is-a :dog)
  (:owner :john)
  (:age 4.5))

(create-schema :john
  (:is-a :person :lawyer)
  (:address "13 Elm Street"))
```

Figure 4-1 shows all the user-defined schemata after those definitions have been executed. Relations are indicated as an arrow going from a schema to the one it is related to.

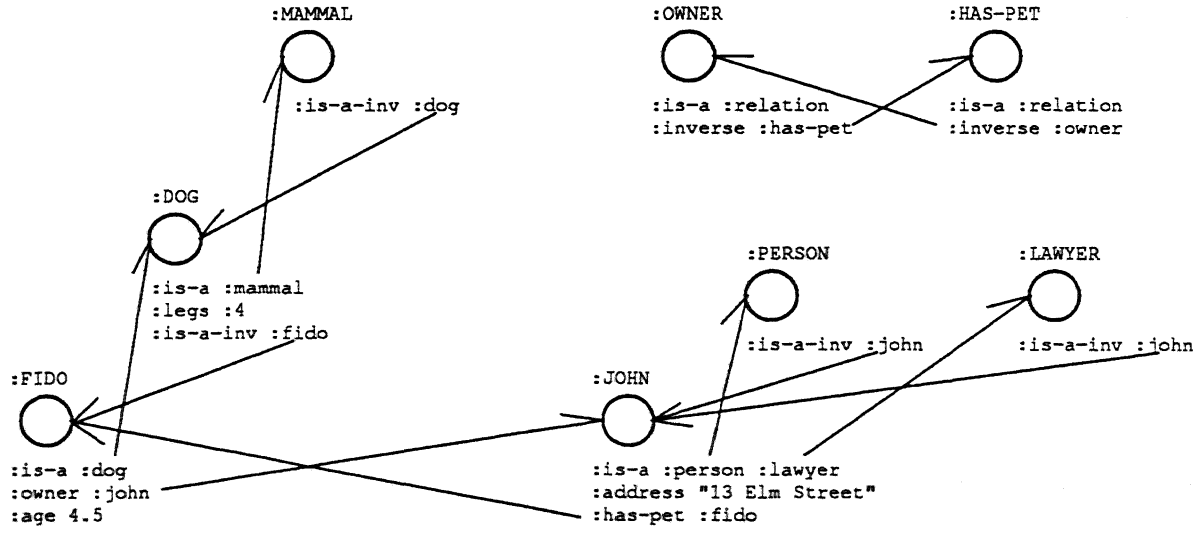


Figure 4-1: The resulting network of schemata

4.3 Predicates and Query Functions

The functions in this group give information about a schema or a slot. All functions whose name ends in "-P" are predicates, i.e., they return a value which is simply used as a Boolean.

(SCHEMA-P *object)* *[Function]*
A predicate that returns NIL if <object> is not a valid schema, non-nil otherwise.
Examples:

```
(schema-p :fido) => T     ; or a system-dependent non-nil value
(schema-p :waffle) => NIL
```

(RELATION-P *object)* *[Mam]*
A predicate that returns NIL if <object> is not a relation, or a non-nil value if it is the name of a schema and the schema is declared to be a relation.
Examples:

```
(relation-p :has-pet) => T
(relation-p :color) => NIL
```

(IS-A-P *schema1 schema?)* *[Function]*
A predicate that returns T if <schema1> is related to <schema2> by the *is-a* relation, either directly or through an inheritance chain.
Examples:

```
(is-a-p :fido :dog) => T
(is-a-p :fido :mammal) => T
(is-a-p rfido :canine) ==> NIL
```

(HAS-SLOT-P *schema slot)* *[FunctM*
A predicate that returns T if the <schema> contains a slot named <slot>, NIL otherwise. Note that <slot> must be local to <schema>, and inherited slots are not considered.
Examples:

```
(has-slot-p :fido :is-a)  => T
(has-siot-p :£ido :legs) ==> NIL   ; Slot is not local
```

(GET-SLOTS *schema)* *[Function*
Returns a list of all the slot names in <schema>. The list only includes locaT slots and does not report slots that might be inherited.
Example:

```
(get-slots rfido) ==> (:AGE :OWNER :XS-A}
```

(GET-ALL-SLOTS *schema!)* *{Macrc:*
Returns a list of all the slot names in <schema>, including slots that are not local but may be fnherïec through an inheritance chain.

Example:

```
(get-all-slots :fido) ==>
  (:LEGS :HAS-PET :IS-A-INV :IS-A :ADDRESS :OWNER :AGE)
```

Note that the example above returns a somewhat surprising list of slot names for :fido (including :has-pet). This is because we declared :owner to be an inheritance relation.

(PS *object*) [*Function*]
Prints out the current schema corresponding to <object>, if one exists, or nothing if <object> is not a valid schema.
Example:

```
(ps :fido)    ; prints out:

{{FIDO
  AGE: 4.5
  OWNER: :JOHN
  IS-A: :DOG
  }}
```

4.4 Schema Manipulation Functions

This group includes functions that create, modify, and delete whole schemata. The function **create-relation** does so implicitly, since relations are also represented by KR as schemata.

(CREATE-SCHEMA *schema-name* &rest *slot-definitions*) [*Macro*]
This macro creates and returns a new schema named <schema-name>. <slot-definitions>, if present, are used to create new slots and values for the schema. Each slot definition should be a list whose CAR is the name of a slot and whose CDR is a (possibly empty) list of values for that slot.
Note: if <schema-name> already exists, the schema is modified in place and will contain the union of its previous slots and the slots specified by create-schema. Previous slots which are mentioned in the call will retain whatever values they had before the operation.
Example:

```
(create-schema :timmy (:is-a :cat) (:age 1.5) (:color :brown :white))
```

(CREATE-FRESH-SCHEMA *schema-name* &rest *slot-definitions*) [*Macro*]
This function is similar to CREATE-SCHEMA, except that it always deletes the schema <schema-name> (if it exists) before creating a new schema. The schema is guaranteed to include only the slots and values specified in the call.

(COPY-SCHEMA *schema*) [*Function*]
Creates and returns a new schema which is an identical copy of <schema>. The newly created schema is automatically given a unique name. All the slots in the new schema contain a copy of the values in the corresponding slot of <schema>. Corresponding lists of values, in other words, will be **equal** in the LISP sense, but not **eq**.
Examples:

```
(copy-schema :fido) ==> T1806
(ps 't1806)      ; prints:

{{T1806
  AGE: 4.5
  OWNER: :JOHN
  IS-A: :DOG
  }}
```

(DELETE-SCHEMA *schema-name*) [*Function*]
Destroys the schema named by <schema-name>. Returns T if the schema was destroyed, NIL if it did not exist.

(CREATE-RELATION *schema-name inherits-p inverses*) [*Function*]
Creates a new schema named <schema-name> and declares it to be a relation. The new relation will have <inverses> (a list of relations) as its inverse relations. If <inherits-p> is non-nil, <schema-name> will become a relation with inheritance, and values may be inherited through it. As a side effect, a schema called <schema-name> is created and linked to the :relation schema through an *is-a* link; all of the <inverses> schemata are also linked to <schema-name>.
Example:

```
(create-relation :has-subsystems nil '(:part-of :subsystem-of))
```

The previous function call defines the non-inheritance relation :HAS-SUBSYSTEMS and its two inverses, :PART-OF and :SUBSYSTEM-OF.

4.5 Slot Manipulation Functions

This group includes functions which create, modify, and delete slots in a schema. It also includes a convenient way to iterate a user-defined function over all the slots in a schema.

(CREATE-SLOT *schema slot-name*) [*Function*]
Creates slot <slot-name> in <schema>. The slot will initially be empty.
Examples:

```
(create-slot :fido :color) ==> NIL
(ps :fido)       ; prints:

{{FIDO
  COLOR:
  AGE: 4.5
  OWNER: :JOHN
  IS-A: :DOG
  }}
```

(DELETE-SLOT *schema slot-name*) [*Function*]
Destroys the slot named <slot-name> from <schema>. Values previously stored in the slot, if any, are lost.

(DO-SLOTS *schema function)* *[Function]*
Iterates <function> over all the slots of the <schema>. The <function>, which should be a LISP function of two arguments, is applied in turn to each of the local slots of the <schema>; the first argument is the schema itself, and the second argument is the name of the slot. The <function> is called purely for side effects, and DO-SLOTS simply returns NIL.
Note that the same result can be achieved with an explicit iteration over the list returned by GET-SLOTS, but in general DO-SLOTS avoids the allocation of storage implicit in the latter.
Example:

```
(do-slots :fido #' (lambda (schema slot)
                     (format t "Slot ~S has values ~S~%"
                             slot (get-values schema slot))))
Slot :COLOR has values NIL
Slot :AGE has values (4.5)
Slot :OWNER has values (:JOHN)
Slot :IS-A has values (:DOG)
```

4.6 Value Manipulation Functions

This group includes the most commonly used KR functions, i.e., the one which retrieve or modify the value(s) in a slot.

(GET-VALUE *schema slot-name)* *[Macro]*
Returns the first value in the slot <slot-name> from the <schema>. If the slot is empty or not present, it returns NIL. Inheritance may be used when looking for a value.
Examples:

```
(get-value :fido :is-a) => :DOG
(get-value :fido :legs) => 4        ; inherit the value from :DOG
(get-value :john :is-a) => :PERSON  ; first value only
```

A setf form is defined for GET-VALUE, so that one can write, for instance,

```
(setf (get-value :fido :owner) :Bill)
```

(GET-VALUES *schema slot-name)* *[Macro]*
Returns all the values in <slot-name> from the <schema>, as a list. If the slot is empty or not present, it returns NIL. Inheritance may be used when looking for values.
Examples:

```
(get-values :fido :is-a) => (:DOG)
(get-values :john :is-a) => (:PERSON :LAWYER)  ; all values
```

A setf form Is also defined for GET-VALUES. For instance,

```
(setf (get-values :fido :owner) '{.-Bill :Jill))
```

(GET-LOCAL-VALUES *schema slot-name)* *[Macro]*
Similar to GET-VALUES, but only local slots are examined and inheritance is never used.
Examples:

```
(get-local-values :fido :is-a) => (:DOG)
(get-local-values :fido :legs) => NIL        ; no inheritance
```

(DOVALUES (*variable schema slot-name)* &rest *body)* *[Macro]*
This macro lets you iterate over all the values in slot <slot-name> for the <schema>. The <body> is repeatedly executed with <variable> bound to each value in turn. It is an error for <body> to modify the structure of the slot.
Example:

```
(devalues (owner-name :£ido : owner)
  (format t "Fido is a pet of -A, who lives at ~A.~%"
          owner-name (get-value owner-name :ADDRESS)))     ; prints:

Fido is a pet of JOHN, who lives at 13 Elm Street.
```

(DO-ALL-VALUES *(variable schema slot-name)* &rest *body)* *[Macro]*
This is similar to DOVALUES, except that in this case when inheritance is used to find the slot. In this case, all the parents of the <schema> are explored, whereas DOVALUES would stop whenever a parent with the slot is reached. The difference is only important when <schema> has multiple parents.

(SET-VALUE *schema slot-name object}* *[Function*
Causes slot <slot-name> in <schema> to contain <object> as its single value. Note that

(set-value s slot value) <==> (setf (get-value s slot) value)

because of the setf form described above.
Example:

```
(set-value rfido :color :brown) => (rbrown)
(ps rfido)   ; prints:

{{FIDO
  COLOR: :BROWN
  AGE: 4.5
  OHMER: :JOHH
  IS-A: :DOG
  H
```

(SET-VALUES *schema sbt-name object-list)* *[Functi&i*
Causes slot <slot-name> in <schema> to contain the values specified by <object-list>.
Note that

(set-values *s* slot values) <=> (setf (get-values s slot) values)

Examples:

```
{set-values :£ido :owner r(:peter :paul :mary))
```

(APPEND-VALUE *schema slot-name objectl* *[Functtr*
Adds one more value, <object>$_s$ to the end of the list of values in <sk>t-name>. The new value will appear

last in the values returned by GET-VALUES.
Examples:

```
(append-value :fido :color :white)      ; and now
{{FIDO
  COLOR: :BROWN :WHITE
  AGE: 4.5
  OWNER: :JOHN
  IS-A: :DOG
  }}
```

(DELETE-VALUE-N *schema slot-name position*) [*Function*]
Deletes the n-th value from a slot. <position>, a 0-based integer, indicates which value should be deleted from slot <slot-name> in the <schema>. <position> must be between 0 and the position of the last value in the slot.
Example:

```
(delete-value-n :fido :color 1)         ; and now
{{FIDO
  COLOR: :BROWN
  AGE: 4.5
  OWNER: :JOHN
  IS-A: :DOG
  }}
```

5. Using the System

5.1 How to Load KR

Until the Dante software is made a part of the official distribution system, you will have to load KR by hand. The easiest method to accomplish this is to execute the following expression from within LISP:

```
(unless (get :dante-modules :kr)
  (setf (lisp::search-list "kr:")
        '("/../herbie/usr/dante/kr/release/code"))
  (load "kr:dante-loader"))
```

Like all Dante subsystems, KR follows a special convention that lets an application program determine whether the subsystem is already loaded. After loading KR, the special keyword **:dante-modules** will have its **:kr** property set to a non-nil value. The typical way to check for this is shown in the expression above.

The system defines a package of its own, namely the "KR" package. All the function names described in the Program Interface section of this document are exported from the KR package, and all you need to do is to add the following line to your program:

```
(use-package "KR")
```

5.2 Internal Representation

This section briefly describes the internal representation for schemata, slots, and values. Only information that may be useful at the application-program level is presented here. Such information can be considered a part of the external contract of the KR system, and application programs can safely rely on the details presented here.

Schemata are simply represented as symbols. No special information is attached to a symbol to indicate that it is a KR schema.

Slots are represented as part of the P-list of a symbol. In particular, each slot corresponds directly to an entry in the P-list. Application programs should never depend on this particular implementation, and should not modify the P-list of a symbol to modify slot information.

Values are represented as a list which is the value of an entry in the P-list. A slot with one value is represented as a list of one element. Values are always internally stored in the same order as shown by the PS function. A list of values is always a simple list, and it contains no additional information whatsoever. Consequently, all the ordinary LISP list-manipulation functions can be used on lists of values. Moreover, the list returned by GET-VALUES is always guaranteed to be EQ to the list of values internally stored in the schema.

5.3 Style Notes

5.3.1 List Representation

The fact that **setf** forms are defined for the two access functions **get-value** and **get-values** makes it possible to obtain quite a few interesting combinations while keeping the functional interface to KR very simple. This is a typical example of how following the LISP philosophy can greatly simplify the external interface of a knowledge representation system.

The operation of adding a new value to the front of a slot, for instance, does not require a special KR function. One simply writes:

```
(push value (get-values schema slot))
```

Similarly, in order to add a value to a slot only if it is not already there, one simply writes one of the following (depending on whether a special test function is required):

```
(pushnew value (get-values schema slot))
     or
(pushnew value (get-values schema slot) :test #'some-test-function)
```

As another example, no special function is needed to eliminate the first value from a slot. One simply writes:

```
(pop (get-values schema slot))
```

Other commonly-used KR idioms also arise from the fact that values are stored as lists. To find out how many values are in a slot, for instance, one uses the function LENGTH:

```
(length (get-values schema slot))
```

To search for a given value in a slot, one can use the functions FIND, POSITION, MEMBER, or any of the variations provided by Common Lisp.

5.3.2 Adding and Deleting Values

One should not use destructive LISP functions to add or delete values from a slot, even though those functions might "work" in some cases. We recommend that the KR access functions (such as SET-VALUE and SET-VALUES, or the SETF methods for GET-VALUE and GET-VALUES) be used in all cases to achieve the same effect.

The reason to avoid direct destructive operations is that such operations may leave the system in an inconsistent state when the slot being operated upon is a relation. Remember that slots that happen to be relations must be handled specially because of the reverse links. Strictly speaking, this only applies to symbols, but we prefer to simply state the following rule of thumb: *Do not use destructive operations to alter the contents of a slot.*

5.4 Usage Hints

5.4.1 Inheritance Relations

KR allows you to freely define new relations that perform inheritance. As a general rule, however, we recommend that you consider carefully whether such relations are really required for your application. Two problems can arise from excessive usage of inheritance relations:

- Poor performance.
- Confusion.

Inheritance relations may affect the system's performance since they turn what is normally a simple hierarchical network into a tangled graph. Every time a slot is accessed and a value is not present locally, KR may have to proceed up the hierarchy following several relations, instead of just the *is-a* relation. There are clearly cases when this is justified by the additional functionality, however, and one should evaluate advantages and disadvantages of the choice on a case-by-case basis.

The second factor, i.e., confusion, is somewhat less intuitive. We will refer back to the example in section 4.3 to illustrate this point. That example is repeated here for convenience:

```
(get-all-slots :fido) ==>
  (:LEGS :HAS-PET :IS-A-INV :IS-A :ADDRESS :OWNER :AGE)
```

Remember that we had linked :FIDO to :JOHN via the *owner* slot, and we had declared *owner* to be an inheritance relation. Getting the list of all slots, then, returned surprising things like *has-pet*, even though :FIDO is a dog and thus is not supposed to have any pets. What happened is that the *owner* relation opened up all the slots in :JOHN for inheritance, and thus :FIDO was suddenly endowed with all the properties that would normally only belong to a person. If :JOHN had had a *salary* slot and a *languages-spoken* slot, those would also have been inherited by :FIDO!

Again, there are cases when user-defined inheritance relations are quite useful. An example occurs when a network is used to represent a situation with multiple hierarchies. In such cases it is natural to define inheritance relations to support the multiple hierarchies, rather than writing special-purpose code to do the same thing.

5.4.2 The IS-A Hierarchy

The IS-A hierarchy constitutes the most natural way to structure a network hierarchically. There are cases, however, where we feel that using the IS-A hierarchy is not appropriate because of stylistic or performance reasons. The most typical example is when the IS-A hierarchy is used to express minor or insignificant differences among certain schemata. In such situations it might be more appropriate to use a

separate slot to express the difference.

As an illustration of this point we will use an example from the Chinese Tutor [Giuse 87], an intelligent language tutor for beginner-level Chinese which uses KR to represent all of its internal data structures. A particular entity of the Chinese language is a *glyph*, i.e., the printed or written representation of a character. Glyphs are represented in the program as KR schemata.

As it turns out, different types of glyphs exist in Chinese (in particular, the complex form and the simplified form of a Chinese character correspond to different glyphs). The very first version of the Chinese Tutor used the *is-a* hierarchy to differentiate among the different types of glyphs, so that the original structure of the network looked like the one shown in figure 5-1.

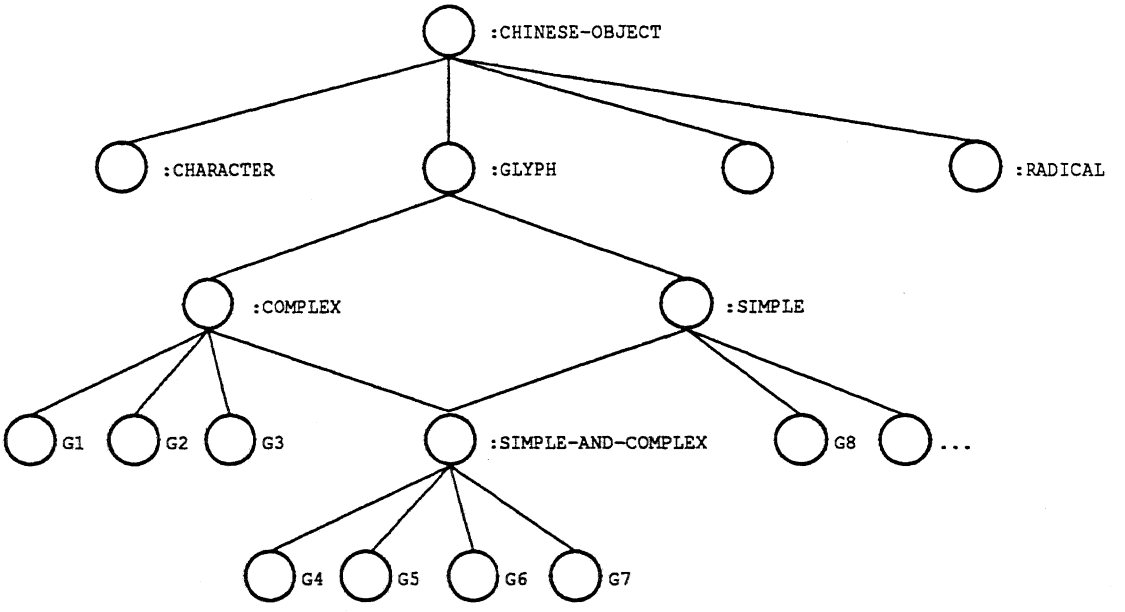


Figure 5-1: The original IS-A hierarchy for glyphs

Note, in particular, how some glyphs appeared two levels below the :GLYPH schema, whereas others appeared three levels below because of the peculiar position of the :SIMPLE-AND-COMPLEX schema. This solution was not ideal. The *is-a* hierarchy was used to represent essentially minor semantical differences, rather than a true hierarchical structure. As a consequence, a common set of operations became unnecessarily complicated and expensive. These operations all followed the same pattern, i.e., they needed to access all the glyphs in the system *independent* of those minor semantical differences. Keeping track of all the glyphs was difficult because they could appear in several subtrees and possibly at different levels in the hierarchy.

The second version of the system eliminated the problem by making all the glyphs immediate children of the :GLYPH schema. This is illustrated in figure 5-2.

Keeping track of all the glyphs, then, simply became a matter of looking into the *is-a-inv* slot of the :GLYPH schema. The minor differences among glyphs are now encoded in a different slot, which does not serve any hierarchical function. The slight increase in space for the extra slot is more than justified in view of better performance and much cleaner code.

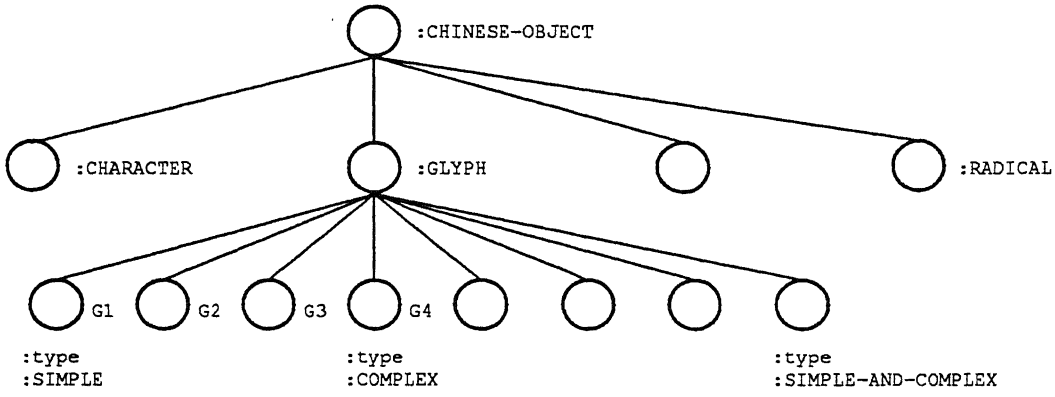


Figure 5-2: The new IS-A hierarchy for glyphs

6. Performance of the KR System

This section presents a simple evaluation the performance of the KR system. These figures were collected on an IBM RT running CMU Common Lisp under the Mach operating system. The RT used to obtain these figures had 10 Mbytes of central memory.

All figures refer to compiled code and are expressed in microseconds per function call. Statistics were collected from within Lisp by executing each function call within a tight loop for a given number of times and timing the loop. A few microseconds should be subtracted from each number to eliminate the loop overhead; 3 or 4 microseconds is probably a reasonable factor.

6.1 Value Access and Modification

GET-VALUE:	40.5
GET-VALUE with 1 level of inheritance:	142.3
GET-VALUE with 2 levels of inheritance:	208.2
GET-VALUE with 3 levels of inheritance:	328.7
GET-VALUES:	40.0
GET-VALUES with 1 level of inheritance:	137.1
GET-VALUES with 2 levels of inheritance:	239.0
GET-VALUES with 3 levels of inheritance:	327.1
GET-LOCAL-VALUES:	10.6
SET-VALUE:	94.4
SET-VALUES:	127.8

6.2 Predicates

RELATION-P:	11.1
HAS-SLOT-P:	100.8
SCHEMA-P:	29.1
IS-A-P:	75.7

6.3 Discussion

The figures above indicate that KR performs quite well. To put those figures in perspective, consider that an empty function call and return in the same environment takes about 14 microseconds. The time to execute the simplest and most commonly used access functions, GET-VALUE and GET-VALUES, is of the order of 3 function calls.

It is also worth mentioning that none of the functions in the tables above allocate any memory at all. This eliminates a common cause of inefficiency, namely, the allocation of temporary storage ("garbage") which has to be eliminated later on.

It is somewhat difficult to provide fair comparisons with other existing knowledge representation systems. Such comparisons are always prone to criticism unless all conditions are absolutely identical, which is difficult to obtain. Just as one point in the spectrum, however, we will mention that the corresponding execution times for CRL running on a Symbolics 3640 Lisp Machine are significantly longer than the ones reported above. Considering that Common Lisp benchmarks typically perform 1.2 to 2.2 times better on a Symbolics than on RT/PC, we might conclude that the above functions in KR are anywhere between 7 and 13 times faster than in CRL. Significantly, it appears that the most expensive functions (such as functions involving inheritance) are even more efficient, relatively speaking, than the simpler ones.

As a final point of comparison we will mention that the time to access a slot in a Common Lisp structure in the same environment is 10.9 microseconds. Compared to this, the corresponding function in KR (i.e., GET-VALUE) is 3.7 times slower. Given that access to Common Lisp structures is very highly optimized in Lisp, it seems that the performance penalty for using KR is amply justified by the much greater flexibility offered by the system.

7. Summary

KR is a simple, very efficient knowledge representation system for Common Lisp. It implements the basic paradigm of semantic network systems and offers such features as inheritance, user-defined relations, and user-defined inheritance.

The main emphasis of the system is on efficiency. Unlike many semantic network knowledge representation systems, KR does not try to provide a monolithic system, but rather aims at extending the Common Lisp philosophy in a natural way. The system is highly optimized and provides a solid substrate upon which application programs can build more elaborate knowledge manipulation algorithms. The program interface to the system consists of a small number of carefully tuned functions; these functions are easy to understand and to use.

Because it adopts the fundamental LISP philosophy, KR fits in very naturally with Common Lisp based application programs. Because of its simplicity, the system is quite small and entirely portable. We feel

that these characteristic make it a useful knowledge representation language, and one whose range of applicability extends well beyond the original environment it was developed for.

References

[Carnegie Group 86]
KnowledgeCraftReferenceManual
Carnegie Group, Inc., Pittsburgh, PA, 1986.

[Giuse 86] Dario Giuse.
Research in Uniform Workstation Interfaces - Research Proposal to DARPA
1986.

[Giuse 87] Dario Giuse.
LISP as a Rapid Prototyping Environment: the Chinese Tutor.
submitted to the International Journal on Lisp and Symbolic Computation, 1987.

[Steeie 84] Guy L. Steele.
Common LISP - The Language.
Digital Press, Burlington, MA, 1984.

[Wright and Fox 83]
M. Wright, M. Fox.
SRL:SchemaRepresentationLanguage.
Technical Report, Carnegie-Mellon University, December, 1983.

NAVLAB: An Autonomous Navigation Testbed

Kevin Dowlling, Rob Guzlkowskl, Jim Ladd
Hennlng Pangels, Jeff Singh, and William Whlttaker

CMU-Rt-TR-87-24

The Robotics Instltute
Camegie Mellon University
Pittsburgh, Pennsylvania 15213

November 1987

This research was sponsored by the Defense Advanced Research Project Agency (DARPA) under contract number DACA76-86-C-0019. The views and conclusions in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the US **government**

Table of Contents

List of Figures

Abstract

The NavLab is a testbed for research in outdoor navigation, image understanding, and the role of human interaction with intelligent systems; it accommodates researchers and all computing onboard. The core of the NavLab is the vehicle controller, a multi-processor computer that controls all locomotion, actuation and physical sensing; it interacts with a computer host and human operator to implement varying degrees of autonomy. The chassis is a modified van with a computer-controllable, hydraulic drivetrain. The NavLab supports a choice of sensing to accommodate many types of navigation research. This technical report details the control computing and physical configuration of the NavLab vehicle.

1. Introduction

The NavLab is a testbed for research in outdoor navigation, image understanding, and the role of human interaction with intelligent systems. A mobile navigation habitat, it accommodates researchers and significant onboard computing. Applications for field navigation vehicles include mapping of hazardous waste sites, off-road haulage, material handling at construction worksites, and exploration of planetary surfaces.

The NavLab is a roadworthy truck modified so that humans or computers can control as occasion demands. Because it is self-contained, it is not subject to telemetry bottlenecks, communication faults or dependence on stationary infrastructure, and can travel to confront navigation problems of interest at any site.

The core of the NavLab is the vehicle controller. In autonomous mode, this multi-processor computer controls all locomotion, actuation and physical sensing. It interacts with a computer host and human operator to implement varying degrees of autonomy. The NavLab controller queues and executes Virtual Vehicle commands originating from a computer or human host. This command set provides high-level motion and control primitives that mask the physical details of the vehicle, and is extensible for control of other mobile systems.

The NavLab configuration consists of a chassis, drivetrain and shell. The chassis is a modified, cut-away van with a computer-controllable, hydraulic drivetrain. Driver's controls allow a human monftor to override automatic control for overland travel, setup and recovery from experimental errors. The shell houses all onboard equipment including computers, controllers, telemetry, and internal sensors. In addition, it provides a working area for operators, allowing research within the confines of the vehicle. Equipment racks, monitors, lighting, air-conditioning, seating and desk space create a mobile environment for research.

Humans can monitor and supervise the NavLab from the operator's console for setup, error recovery and tuning. Interface modes include Virtual Vehicle instructions, joystick motion control, and direct servo motion commands. The console also incorporates several displays to show the current states of both the vehicle and control computer.

The NavLab supports a choice of sensing to accommodate many types of navigation research. Video cameras provide color and intensity images for *scene* interpretation. NavLab vision experiments use a single camera to analyze road edges through intensity, texture, and color segmentation. A scanning rangefinder sweeps the surroundings with a distance-measuring laser that provides useful three-dimensional information about the geometry and reflectivity of the environment. Laser experiments navigate through geometric features like trees and buildings. Taken together, data of color, intensity, range and reflectance provide a rich basis for building natural scene descriptions. Sensor information from several sources can be fused to achieve more robust perception. A blackboard architecture integrates the distributed processes that sense, map, plan and drive.

The NavLab represents continuing evolution in the design of navigation vehicles. Fully self-contained, i is a milestone in mobile robotics.

This technical report details the control computing and physical configuration of the NavLab vehicle. Information on other aspects of the NavLab, Including perception, modeling, planning and blackboard

architectures, can be found in articles listed in Appendix V.

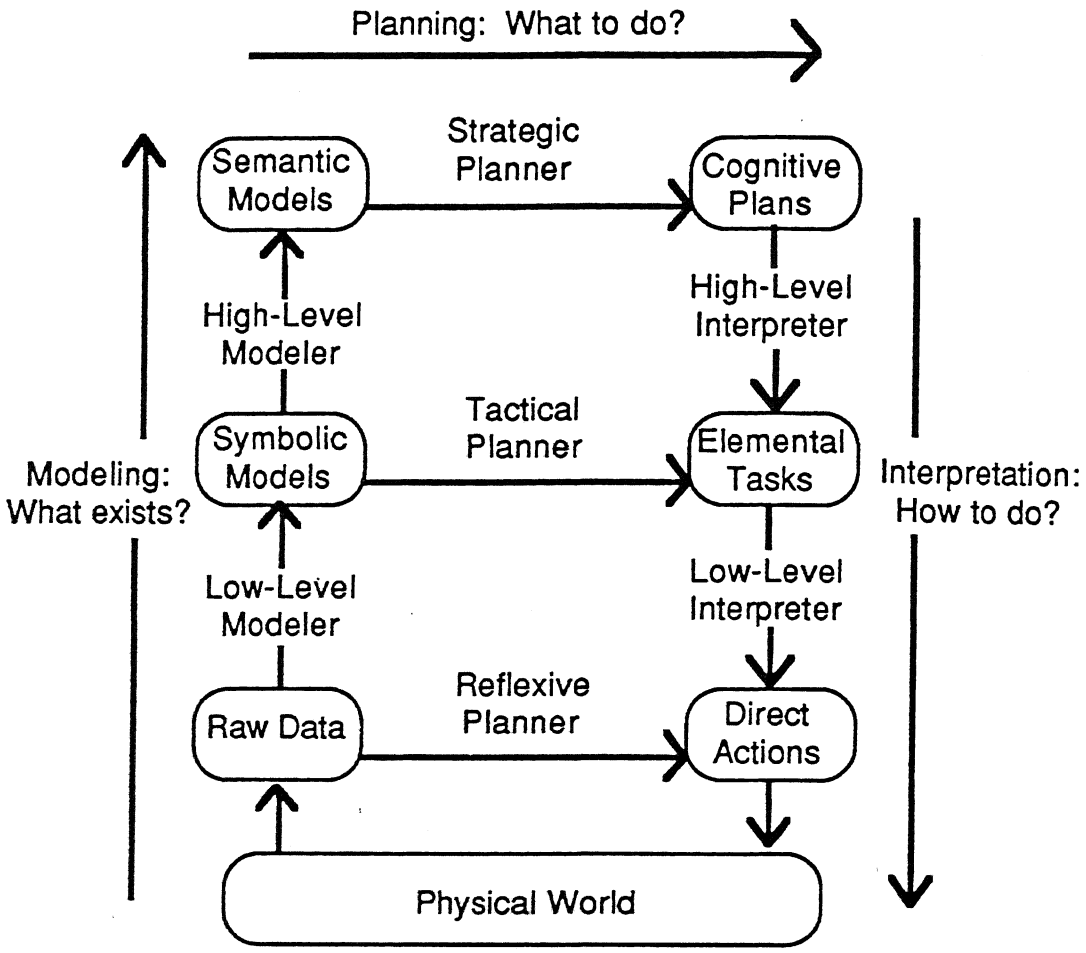


Figure 2-1: The Hierarchical Layering of a System Architecture for Modeling and Planning

2.2 Virtual Vehicle

A Virtual Vehicle is a man-machine interface that accepts conceptual commands and provides a clean separation between the navigation host and vehicle control. This interface masks implementation details of the physical vehicle, facilitating adaptability to future navigation testbeds.

The Host (the computing engine that does planning) communicates with the Virtual Vehicle via ASCII data transmitted over a serial line. The communication falls into three categories:

- *Commands* issued by the Host
- *Queries* by the Host about the status of devices
- *Reports* initiated by the Virtual Vehicle on completion of commands and in case of emergency.

In the current implementation, the vehicle is directed along circular arcs because arcs are quickly computed and absolute position is not critical (the arcs are being upgraded to clothoids). Because it is not possible for a vehicle to switch between an arc of one curvature and another instantly, path transitions

are inexact. Errors are compensated for by dynamically planning arcs to reach subgoal points along the path.

To facilitate synchronization, all drive and steering commands are initiated at the transitions between arcs. The capability is provided to make changes to vehicle motion (e.g., curvature of the arc, vehicle velocity) on the fly. Arcs (specified as [arclength, radius of curvature]) can be queued for sequential implementation.

The Virtual Vehicle and Host interact as follows:

- The Host issues a new arc command before the arc in execution is completed.
- If an immediate condition is specified, the old arc is discarded and the new arc is accepted immediately. Otherwise, the new arc is initiated at the end of the arc being executed.
- When a new arc is initiated, vehicle position is reported to the Host for use in calculating future path plans.
- The Host incorporates the reported position in planning the next arc, thereby compensating for deviations from the desired path.

The Virtual Vehicle instruction set and details of interfacing can be found in Appendix IV.

2.3 Controller Architecture

The NavLab controller is a powerful and flexible multi-processor system. A functional block diagram of the controller is shown in Figure 2-2. A Pilot module, responsible for management and operation of the key peripherals and I/O devices in the system, maintains direct control of all physical action and motion. The Pilot is also responsible for system startup and synchronization and acts as the hub in a star configuration for inter-processor communication. A Command Preprocessor manages I/O between the controller and devices that communicate with it. The Sensor Manager controls a network of 8-bit micro-controllers distributed throughout the vehicle to provide points of intelligent analog and digital I/O. Accommodations are made for an Advisor to set limits on physical motion parameters based on the perceived condition of the mechanical systems of the vehicle. The Advisor incorporates a bump detection subsystem that signals the Pilot if immediate action is necessary.

Each module in the system contains its own operating environment for independent/parallel operation. The operating environments are subsets of those used for system development. Code for each module is down-loadable to permit easy modification to the system.

2.3.1 Hardware

The NavLab controller is designed as a two-tiered multi-processor system. The first tier is responsible for the primary computing, control I/O and motion control. It is comprised of 4 Intel 28612 processor boards residing in a common Multibus backplane. The second tier performs remote data acquisition and control of devices located around the vehicle using a serial network of 8-bit micro-controllers. The Sensor Management System in the first tier is the interface between the two tiers.

2.3.1.1 Primary Computing

Processors in the first tier take advantage of the multiple bus structure of the system to increase processing throughput. Each processor contains a local bus with enough memory resources to support its own execution environment. Processors have bus master capabilities to access and control I/O

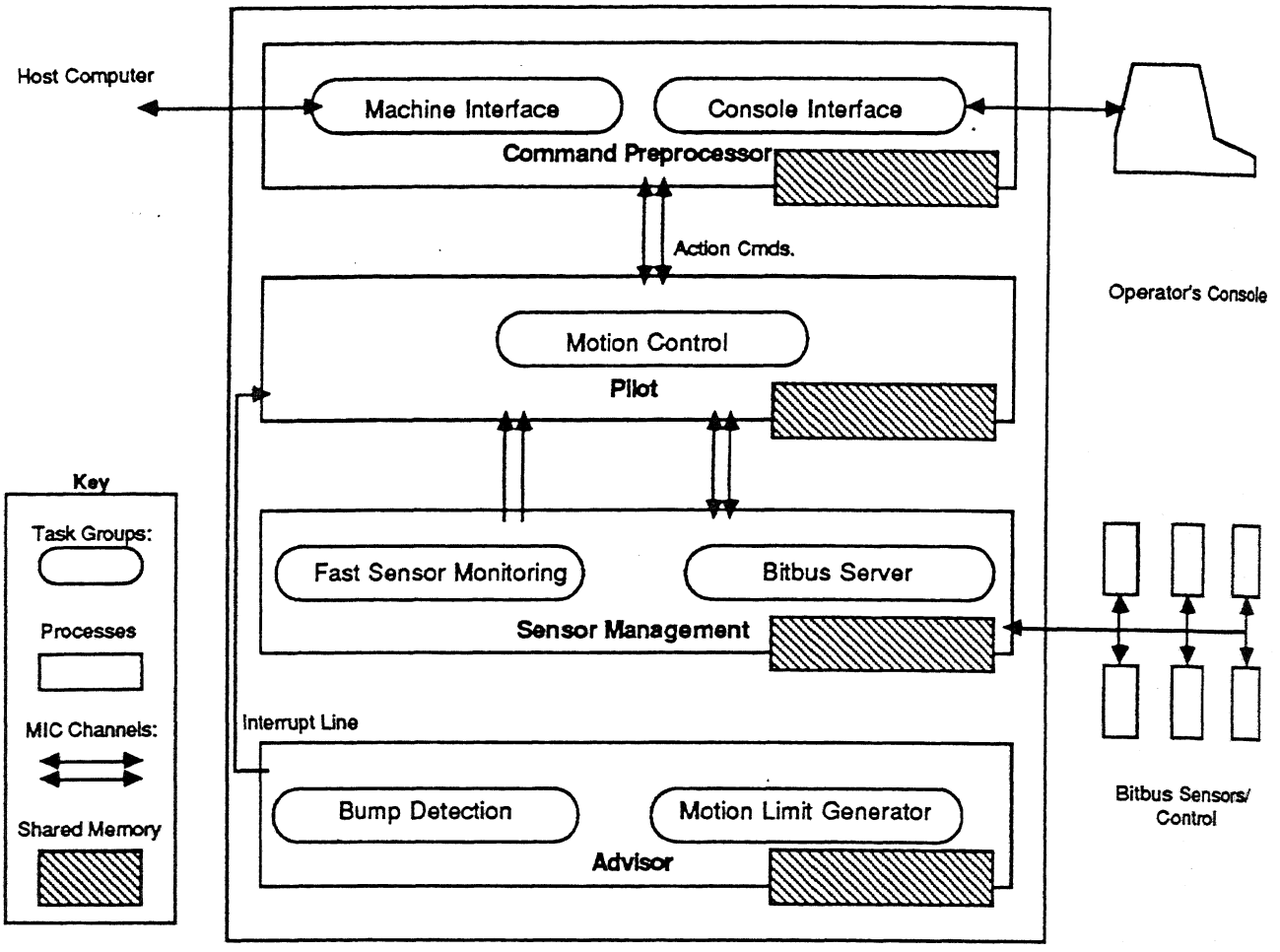


Figure 2-2: Architecture of Controller

boards and shared memory spaces. Critical memory and I/O locations are controlled using a semaphore system while bus contention is arbitrated in hardware. Interrupt lines in the Multibus backplane tie the processors together for inter-processor communications.

Each processor is identically configured with 256 K local ROM, 512 K local RAM, and a 256 K window to the Multibus. The ROMs on the I/O processors only contain operating system software and a download facility to allow loading of applications. The multiple bus structure permits a total system memory of 2.5 MB even though only 1 MB is addressable from each processor.

The controller also contains intelligent slave boards for I/O expansion and servo motor control. These boards may be accessed by any bus master. Often, access is restricted to a specific processor to avoid contention problems.

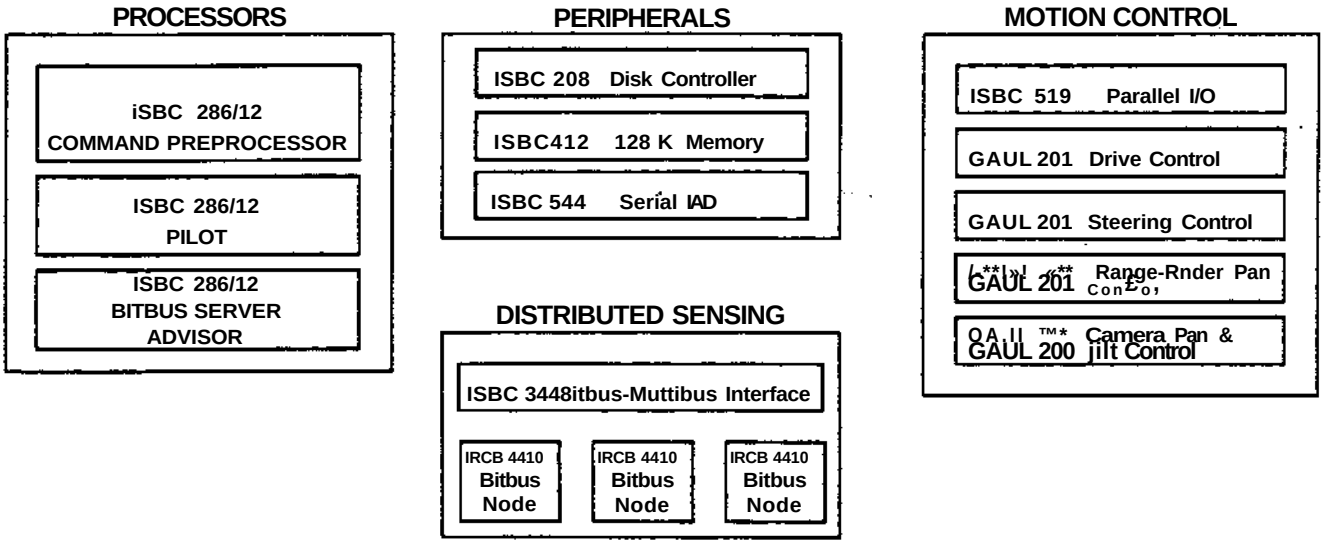


Figure 2-3: Hardware Configuration

2.3.1.2 Secondary Computing

The second computing tier physically distributes sensing and control of devices around the vehicle by using nodes that consist of 8-bit microcontrollers communicating over a high-speed serial bus using a message protocol called Bitbus. The network is controlled by a Master node that either continually polls other nodes to read analog and digital inputs or continuously commands them with reference values. The communication overhead makes Bitbus suitable to tasks that require high-level control and slow data acquisition. The serial bus network is extensible to support up to 250 nodes. Further implementation details of Bitbus can be found in Section 2.3.5.1.

2.3.2 System Software

System software for the controller is based on the iRMX 86 real-time operating system. iRMX is configurable to customize operating environments for each processor in the system. These operating environments are resident in ROM local to each board and are booted from reset. iRMX 86 provides objects to support an event-driven, mufti-tasking environment.

A facility to down-load object code was developed for building and testing multiprocessor systems. A single processor accesses the mass storage device containing object axle files for downloading. This processor, like a system server, toads object axle into shared memory and signals the appropriate target board when a valid record is available. The other processors contain consumer jobs to copy records from shared memory to their local memory. On signal from the server processor, a consumer job releases the server CPU to allow the local Operating System to start the jobs from the newly loaded code. Once the application software is running, the consumer toad job lies idle and wafts for a signal from the server processor to reset and begin the toad sequence again. This flexible toad facility is a valuable tool for building and testing multiprocessor systems.

2.3.2.1 Interprocessor Communication

Processors communicate using shared memory in two different ways. Common variables are accessed by multiple processors to share state information (scratch pad communication). Messages can also be written to specific memory locations on other processor boards and the receiving board is signaled by an interrupt. This method is often used by one board to direct processing on another board.

Scratch Pad Communication

This method is a simple solution to sharing a large amount of data between modules. Processes that acquire data (status of devices, vehicle orientation, speed, etc.) post this information to the scratch pad area instead of sending the data to all modules that need them. Most shared variables are independent of each other; hence contention problems are limited to access of the individual memory locations to read or write. Dependent variables (ones that must be accessed as a group) require a software semaphore to provide mutual exclusion. An indivisible test and set instruction provided by PL/M-86 was used to create the semaphore system.

Module for Interprocessor Communications

The Module for Interprocessor Communication (MIC) was developed to support flexible pipelined communications between tasks running on separate processor boards. MIC provides the applications programmer with a simple set of procedure calls from which a task can queue messages containing a board and task destination. MIC handles the transfer of these messages between boards.

MIC is implemented as a star architecture. All messages are sent through a central node to limit the number of rèquired interrupt lines. This scheme is well suited to the NavLab controller because most interprocessor communications are to a central node (the Pilot).

MIC was built ˙using tools provided by iRMX including inter-task communication, dynamic memory allocation, and FIFO queues. MIC runs as an interrupt-driven task. It responds to signal interrupts to determine the destination of a received message and then sends it to the appropriate task. iRMX system calls permit asynchronous message transfers between tasks.

MIC was designed to be compact (5 K), fast, and capable. MIC is able to dynamically allocate message segments to meet the load of interprocessor communication traffic that varies from processor to processor. This prevents wasting memory and time required by the system programmer to tune buffer sizes for Individual boards. When application code is modified to change message traffic, MIC can adjust to use only the necessary memory resources.

2.3,3 Command Preprocessor

The Command Preprocessor front-ends I/O originating from two sources: the driving Host computer (Host) and the operator's console (Console). At the lowest level, it drives the physical data Erics supporting these command streams. In the NavLab controller, RS232C serial channels are controlled. *M* the next level, it validates data integrity of Host-originated Virtual Vehicle Interface (WI) command packets by checking format correctness, parameter count, and packet size. At the highest level i checfe parameter values against established limits. The Command Preprocessor has the ability to rejefi commands exceeding the current operating limits, but the Pilot has final authority on commanc acceptance. Query commands issued by the Host are handled directly by the Command Preprocessor without Pilot involvement.

The Command Preprocessor communicates primarily with the Pilot module. The other modules are indirectly accessed through value lookups in the Scratch Pad. All commands involving action, such as motion commands or control commands to a device managed by the Bitbus Sensor/Control Network, are first sent through the Pilot to update its knowledge of the vehicle state affected by the controller.

The Command Preprocessor contains two separate subprocesses to service the Host and Console concurrently. The Host Interface is responsible for maintaining communications between the controller and the Host. The Console Interface interprets commands from the operator console keyboard. The Console is given priority over the Host so that it is possible for the operator to override Host commands. Commands are received as ASCII packets. The Host sends only numeric data; each command is given an opcode. The Console allows the operator to enter commands as simple mnemonics.

Communication errors are trapped by syntactic data validation. The Command Preprocessor takes two different actions based on the type of command it receives. For motion commands, the arguments are validated based on the allowable ranges of vehicle motions posted in shared memory by the Advisor. If all the arguments are acceptable, the command is passed on to the Pilot. An acknowledge message is then sent, signaling that the command was accepted and will be executed. If for any reason the command is found to be invalid, a disacknowledge message along with an explanation for rejection is sent to the command initiator.

The Command Preprocessor processes query commands (e.g., heading, position). The requests are satisfied by accessing the shared memory region where the information is updated constantly. This method makes it unnecessary to interrupt other processes. The data is formatted and shipped to the requestor.

The Command Preprocessor also maintains the display on the operator console onboard the NavLab. The screen is divided into three parts:

1. Display -- A window displays vehicle data. The operator can select between 5 different displays:
    - Sensor data shown in graphical form (vertical bars).
    - Sensor data shown in alphanumeric form.
    - Status of switches controlled by the controller shown in alphanumeric form.
    - Command packets between controller and the Host.
    - A help screen that explains how the operator can control the vehicle by using the Virtual Vehicle instruction set.
2. Command line -- Allows the operator to:
    - Enter Virtual Vehicle commands.
    - Enter software joystick commands.
    - Turn on/off switches controlled by the Bitbus network.
3. Information area -- A window is reserved for special messages that may be sent by any process in the controller.

2.3.4 Pilot

The Pilot's main function is controlling or initiating all physical action and motion control. The Pilot also plays the central role in inter-processor communications by acting as the hub in a star configuration. All commands altering the state of the vehicle are filtered through the Pilot, eliminating contention and state ambiguity problems potential to systems altered by multiple independent processes. For the generalized

case, the Pilot module would occupy several processor boards and handle manipulation as well as locomotor control.

The Pilot is composed of a hierarchy of concurrent processes (tasks), each of which is dedicated to maintaining a specific subset of state variables and initiating all actions affecting those variables. At the lowest level, each axis of motion has an individual driver process associated with it that formats motor-controller specific command strings, performs I/O exchanges with the motor-controller board, and maintains the current values of all pertinent variables for that axis in local memory. The axis drivers at this level have no notion of the physical configuration of the overall system. Coordination of motions is handled by higher-level processes.

Action requests can be submitted to the Pilot by the Command Preprocessor at any time. On receipt of such a request, the Pilot returns an acknowledge/disacknowledge message to the Command Preprocessor indicating whether it can execute the command. If the received command can be executed, it is decoded and forwarded to the appropriate subprocess for handling. Depending on the type of action requested, this process may then

- direct motions (via the appropriate axis drivers)
- read or set parallel I/O lines (for example, to select a different transmission gear)
- update the values of some state variables.

Because individual processes each have a specific run-time priority, critical commands (e.g., "STOP") always obtain control of the CPU, even if a lower-priority command is still in progress. Also, because task scheduling is event-driven rather than time-shared, high-priority processes always run uninterrupted, i.e., in constant time.

A special set of tasks within the Pilot maintains and processes a queue of arcs specifying a path for the vehicle. These arcs are executed continuously and a position report is issued to the Command Preprocessor on completion of each arc. Velocity and acceleration parameters can be updated at any time during execution of an arc; in addition, one value for each of these variables may be queued to go into effect with the beginning of the *next* arc execution.

The Pilot has the final responsibility for command acceptance or rejection, command queue management, and implementing established equations to achieve requested arc trajectories. Implementation details of the vehicle are masked by the Pilot.

The NavLab incorporates braking as well as forward and reverse propulsion in a single, bi-directional hydrostatic drive. For the generalized vehicle case, the Pilot would coordinate brake/throttle control to achieve velocity and position objectives. At the servo level, motion is controlled by motion control boards commanded by the Pilot. Emergency stop conditions are signaled to the Pilot by a critical interrupt line controlled by a planned Health Preservation module with bump detection facilities. On assertion of this line, the Pilot is responsible for graceful shutdown, leaving the vehicle ready for recovery actions issued from the operator's console. Because only the Pilot controls the motion, it is always aware of the current motion state.

Finally, a few background processes perform such functions as maintaining the system clock and calculating position coordinates based on sensor measurements.

2.3.5 Sensor/Device Management System

Apart from the five main axes of motion, there are numerous sensors that must be monitored and devices that must be activated. The Sensor/Device Management System manages two classes of sensors. The first class is characterized by sensors and devices that need not be monitored/controlled frequently. For example, a sensor might be dedicated to monitoring hydraulic fluid temperature; while this information is important, it is not essential that it be updated more frequently than once in several seconds. Another class of sensors is that group of devices that must be monitored frequently. An example is a process that must analyze data from inertial devices and post these results in shared memory several times a second.

2.3.5.1 Bitbus System

The Bitbus System is a highly flexible and expandable data acquisition and control system. By taking advantage of the Bitbus distributed control architecture, the Distributed System supports analog status sensors and digital I/O channels using microcontrollers distributed on a serial network. Nodes on this network transfer data to the Bitbus Server module using the Bitbus message passing protocol. The Bitbus nodes are programmable to meet a wide range of sensor and control configurations. Data returned to the Bitbus server are conditioned and scaled at the Bitbus node, reducing computational requirements of the Bitbus server.

The primary responsibility of the Bitbus server is to acquire and move sensor data to shared memory locations recognized by other modules in the controller. When the Pilot sends an action command request, the server must format messages to control any devices supported by a Bitbus node. In support of these functions, the server must also handle node initialization, self-monitoring, and fault recovery for the Bitbus network. The chief advantage of using a Bitbus network is the modular expandability and flexibility that is inherent to the Bitbus architecture. Complex inter-processor message passing facilities are included in the architecture, relieving the programmer of much responsibility.

In simple systems with limited I/O points, the Distributed System could be replaced with a single board computer equipped with the appropriate I/O expansion modules. An effort should be made to keep I/O operations local to the processor to avoid consuming bus bandwidth. With either implementation scenario, the update rates of shared variables should be adjustable to control the bus access frequency of the Distributed System for tuning purposes.

The Bitbus network provides a distributed control structure to service the first class of sensors. A list of sensors and devices on this network can be found in Section 2.5.2.

The Bitbus network is based on a master (Bitbus server) and slave (Bitbus nodes) concept (Figure 2-4). Nodes provide the connection between the sensors/devices and the central Bitbus Server. Because each node operates independently, fast data acquisition can be achieved by distributing the work load among many nodes. Nodes can also be programmed to perform control tasks by reacting immediately to critical conditions as they arise.

The Bitbus Server, one process on the Sensor Management Module, initializes the network and monitors status. Because the nodes cannot initiate communications, the Server must continuously poll each node for output data. When the Server receives a message from a node, it posts the relevant information in shared memory for reference by other processes. When some high-level process needs to control a Bitbus node, a message is sent to the Server. This message is then broadcast on the network where it is

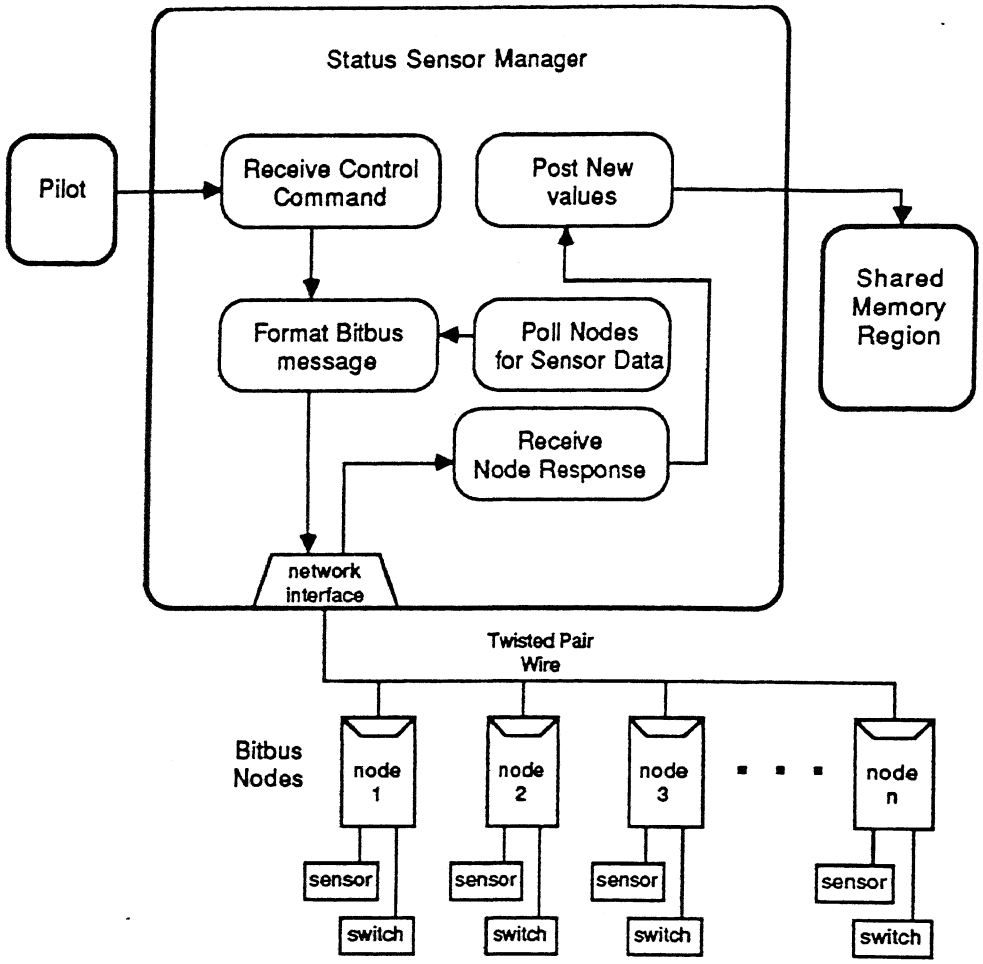


Figure 2-4: Bitbus Server

trapped and processed by the addressed node.

2.3.5.2 Fast Sensor Monitoring

The Sensor Management system also maintains processes to monitor those devices that must be serviced at a high frequency. At present, the only such device envisioned is an Inertial Navigation System anticipated to report position and orientation data about 10 times/sec. The incoming data is parsed and posted in shared memory. Other devices that need to be monitored constantly can be added to the controller simply by allocating a process to them. This method is preferable to the Bitbus method when data must be accessed frequently and must be made available to the entire system quickly.

2.4 Motion Control

Of the 5 axes of motion, only drive and steering can be controlled both manually and automatically. The other three motions of pan and tilt are only used in automatic operation. Figure 2-5 (a) shows the configuration during manual operation. All axes of motion on the NavLab are physically controlled by Galil DMC-200 series motor controllers. These controllers were chosen for:

- Multibus compatibility
- multiple modes of control (position, velocity, torque)

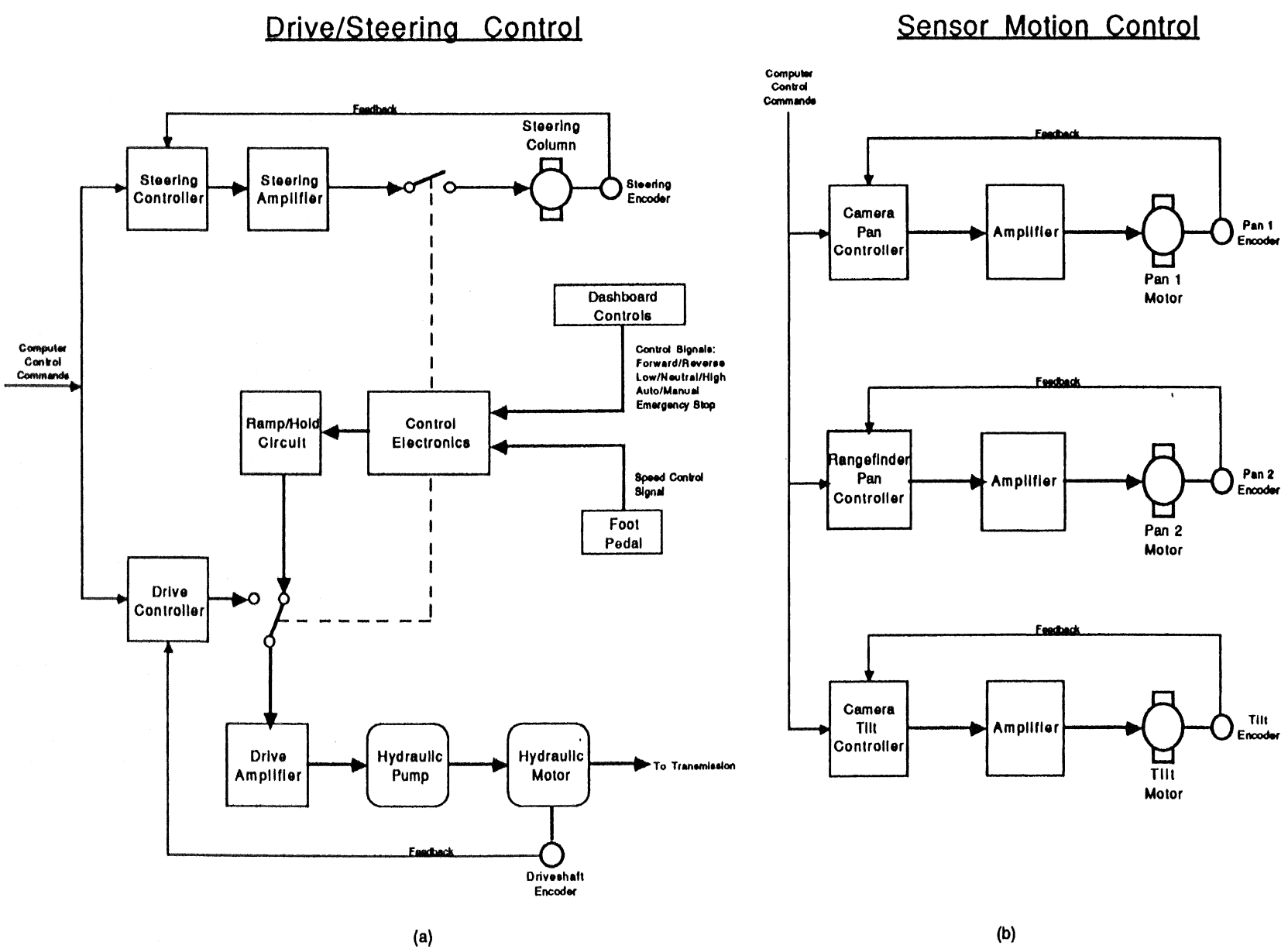


Figure 2-5: Motion Actuation

- coordinated motion of two or more axes (DMC-200 only)
- programmable acceleration and slew rates
- status, position, and error reporting.

A digital phase lead control law with adjustable gain, pole and zero provides a stable closed-loop system for a wide range of plant dynamics. The motor controllers communicate with the Pilot subsystem through Multibus I/O ports for data as well as handshake exchange.

Single axis Galil DMC-201 controllers are used for steering, drive, and laser-ranging pan motions, while a DMC-200 two-axis unit is used for the camera pan and tilt (Figure 2-5 [b]). Each controller is software-calibrated at power-up to match the dynamics of the controlled axis. Thus, motor controller boards can be interchanged simply by selection of appropriate I/O addresses via jumpers.

2.4.1 Dash Panel Control

The vehicle operates manually to simplify transport to and from test sites. Manual operation doesn't require any computing or generator power. The electronic components active during manual operation are powered by the NavLab's 12 V system.

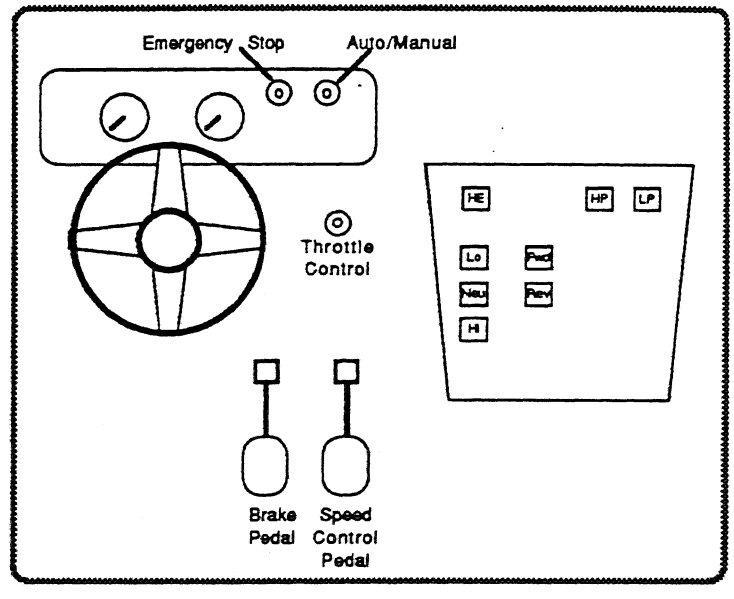


Figure 2-6: Dash Panel Layout

A human interface is incorporated for safe and easy use by drivers of standard automobiles. Figure 2-6 shows illuminated pushbutton controls mounted within reach of the driver.

- **High, Neutral, & Low**: allow the operator to choose gears. Because switching directly from one gear to another produces an unsafe lurching of the vehicle, a hardware logic function allows switching only by first selecting Neutral.
- **Forward, Reverse**: select the direction in which the vehicle moves.
- **Auto/Manual**: a pull-push switch that switches between manual and automatic control.
- **Emergency Stop**: disables autonomous locomotion and brings the vehicle to a rapid, controlled stop. Servo-lock of steering is disabled; steering is returned to manual control.
- **Brake Pedal**: as in commercial cruise control systems, a light touch of the brake pedal brings

the vehicle back under manual control.
- **Speed Control Pedal:** activates a 20K ohm potentiometer to produce a voltage proportional to the angle of deflection of the pedal.
- **Throttle Control:** this dial sets the vehicle engine RPM as detailed in Section 4.2.1.1.
- HE: this switch turns on/off the heat exchanger fan for the hydraulic system.
- HP, LP: These lamps are lit when the dirty oil filters in the high and low pressure hydraulic systems indicate an alarm.

By default, when the vehicle is powered up, it is put into manual mode, neutral gear, and forward direction. It is necessary to provide the ability to override the autonomous mode in a fast but controlled manner if an emergency develops. To ensure reliable operation, manual override is a hardwired electronic circuit with sealed electromagnetic relays instead of sequential logic gates. This design proved to be immune to the noise and power fluctuations common to automotive electrical systems. Because this circuitry is essential to vehicle locomotion, it is powered by the vehicle 12 V system rather than the generator.

An electronic ramp/hold circuit in series with the foot pedal provides adjustable limits on acceleration and deceleration and ensures that abrupt movements of the foot pedal do not cause the vehicle to lurch. This feature was included both for safety and ease of driving. A second ramp/hold unit ensures a smooth deceleration in case of an emergency stop.

2.4.2 **Steering Control**

The steering control system consists of a computer-controlled DC servo motor linked to the steering column by a toothed belt. A single axis motor controller (DMC-201) uses feedback from an optical 1200-line incremental encoder mounted directly on the motor shaft to maintain tight position control over the steering wheel. A servo-amplifier converts the +/-10V control signal from the motor controller to drive the DC motor with up to 11 amps of continuous current. At maximum speed, the steering mechanism can be moved between its two extreme positions in 2 seconds.

Feedback is obtained from an encoder on the motor shaft that is mated to the steering wheel, which is always turned a specified amount. Differences between intended and achieved radius occur due to linkage non-linearities and factors such as friction between the road and the wheels, grade of the road, vehicle speed, and speed with which the steering wheel is turned.

Limit switches on the steering linkage are hardwired inputs to the controller board and provide both a safety stop to protect the steering mechanism and a reference point for roughly calibrating **the** steering control system to a known position on power-up or system reset.

Z4.3 Drive

A single axis motor controller services the drive system. The voltage (-10V to 10V) produced by the motor controller is converted to a current signal (-100mA to 100mA) by an amplifier that directly operates a hydraulic servo valve to set the speed of the hydraulic motor. Acceleration of the vehicle is limited by a ramp/hold circuit, mentioned in Section 2.4.1, in effect providing a low pass filter to the input signal An optical 300-line incremental encoder mounted on the hydraulic motor shaft provides feedback to the motor controller. Because the transmission is downstream of the hydraulic motor (i.e., between the motor and the driveshaft)$_s$ the encoder pulses must be interpreted differently for high and tow gears.

2.5 Sensors/Devices

At present, the controller features for handling sensors are not fully implemented. T expansion are proposed for the near future. An Inertial Navigation System will be incorporat continuous position and orientation information. A Bitbus network will be used to monito devices distributed around the vehicle.

2.5.1 Inertial Navigation

An Inertial Navigation System (INS) to be deployed on the NavLab will receive distance data will provide position and inclination data along the axes specified as output. The INS c heading on its own and provides updates of position and heading.

The following information will be obtained from the INS:

1. True heading of the vehicle -- 0.5 degree resolution.
2. Rate of change of heading -- 0.5 degs/sec resolution.
3. X, Y, Z position in cm -- 10 cm resolution. This will allow movement on a 100 km^2 grid
4. Roll and pitch inclination -- 0.5 degree resolution.

Performance criteria include:

- Dead Reckoning Capability: Speeds along the direction of travel of up to 60 km/hour; tu speeds (change of orientation) of up to 40 degrees/sec.
- Accuracy: Maximum long track error: 1% of distance traveled. Maximum cross track 0.1 degree/hour.
- Updates: Must be able to handle the accuracy requirements above with updates coming once an hour or once in 5km.
- Necessity of Stopping: Must not need more than 5 minutes for the vehicle to be comp stationary on power-up or on recalibration.

At present a device that uses three mechanical gyroscopes and requires an odometer inp considered. A second device being considered is a strap-down system that uses ring laser g This is much more accurate than the first and does not require odometer input.

2.5.2 Sensors/Devices on Bitbus Network

The following is a list of sensors and devices that are monitored and controlled by the Bitbu Scan cycle time indicates the period at which each of the sensors is monitored. Temperatur degrees centigrade. Pressure units are pounds/sq. inch.

Sensor	Scan Cycle	Minimum	Maximum
Thermocouples			
Engine Oil	30 sec	0 deg	175 deg
Engine Coolant	30 sec	0 deg	150 deg
NavLab Cabin	30 sec	-10 deg	40 deg
NavLab External	30 sec	-10 deg	40 deg
Hydraulic Reservoir	30 sec	0 degs	100 degs
Pressure Transducers			
Engine Oil	10 sec	0 psi	60 psi
Hi-pressure System (input)	10 sec	0 psi	3000 psi
Hi-pressure System (output)	10 sec	0 psi	3000 psi

Voltages & Currents			
Battery Voltage	30 sec	0 V	15V
RPH	2 sec	0 V	5 V
GAS Level	60 sec	0 V	5 V
Low Level Reservoir	60 sec	0 V	5 V
Swash Plate Angle	1 sec	-10V	10V
Switch Settings			
Transmission Gear State	1 sec		
Generators (2)	1 sec		
Heat Exchanger	1 sec		

3. Vehicle Shell

NavLab's foundation is a 1985 General Motors Vandura cutaway chassis chosen as a commercial base to simplify development. As acquired, the vehicle consisted of a chassis, a drivetrain and a cab. A custom shell was constructed to house the onboard AC power generation, power distribution, control and computing equipment. Space for operators is provided, allowing research activity within the confines of the vehicle. The original configuration also included a 350 ci V-8, cruise control, an automatic transmission, dual rear wheels, power steering, power brakes and a 33 gal fuel tank.

3.1 Exterior Design

The shell was custom-built with particular attention paid to strength requirements, anticipating needs for extensibility. The roof and cab support air-conditioning, antennas, sensors, and working personnel. The floor of the shell supports about 2000 kg. The shell is dimensioned so that researchers can stand inside; five equipment racks are housed side by side along one side of the vehicle. Figure 3-1 shows a rear and side view of the vehicle.

The shell is made entirely of steel. Heavy gauge was used on the front and back walls while lighter gauges were used along the side walls and roof. A metallic blue paint protects the entire shell. There are compartments for the generators and power-related equipment. Louvered metal doors provide outside access; there is no access to these compartments from inside the vehicle to keep fumes from entering the shell.

A wiring port in the floor behind the driver's seat allows wiring from the underside to enter the vehicle. Another access vent in the shell above the passenger compartment enables wiring from cameras and range sensors to enter the vehicle.

3.2 Interior Design

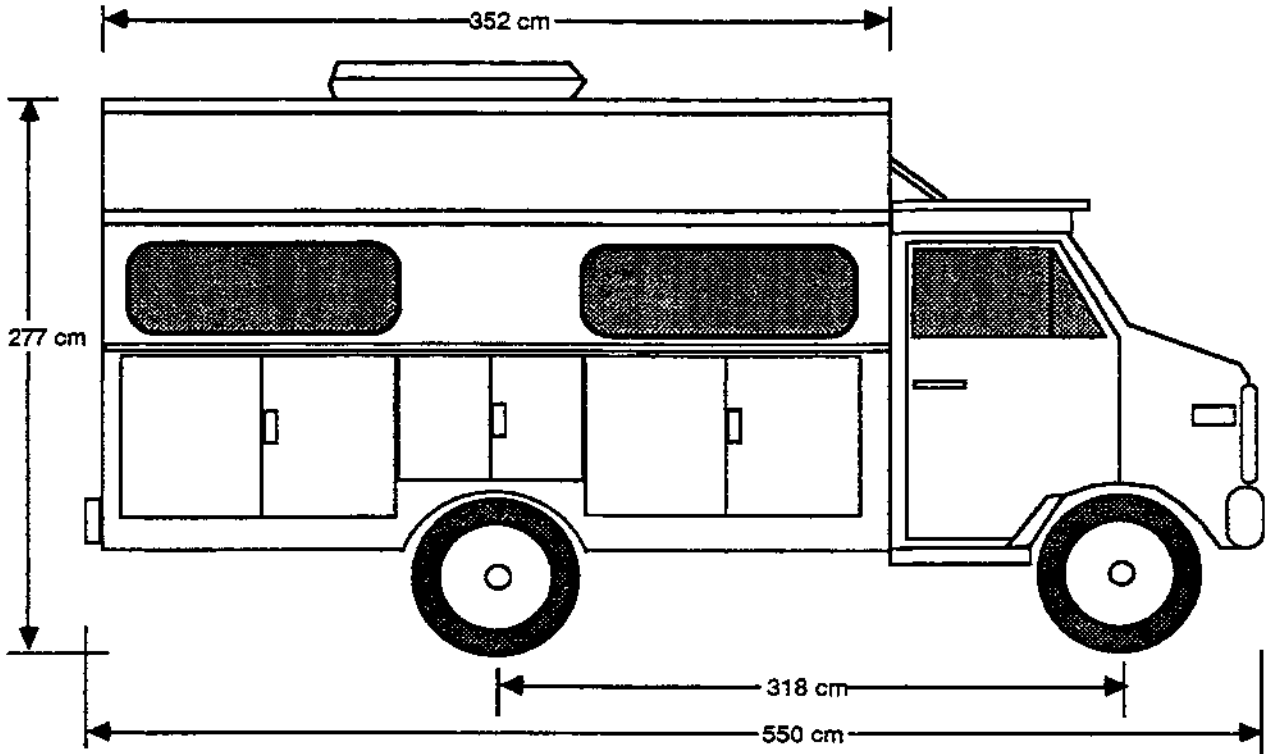
Figure 3-2 shows a topdown view of the NavLab. The cab has two seats, one for a driver and one for a passenger. A console located between the two seats allows the operator to control and monitor the transmission. A research area behind the cab contains computing, sensing, and control equipment, as well as space for two researchers.

Five equipment racks are boated on the left side. A desk area extending the length of the research area is located opposite the racks across the aisle. Three video monitors mounted above the desk area can swivel to a desired viewing position.

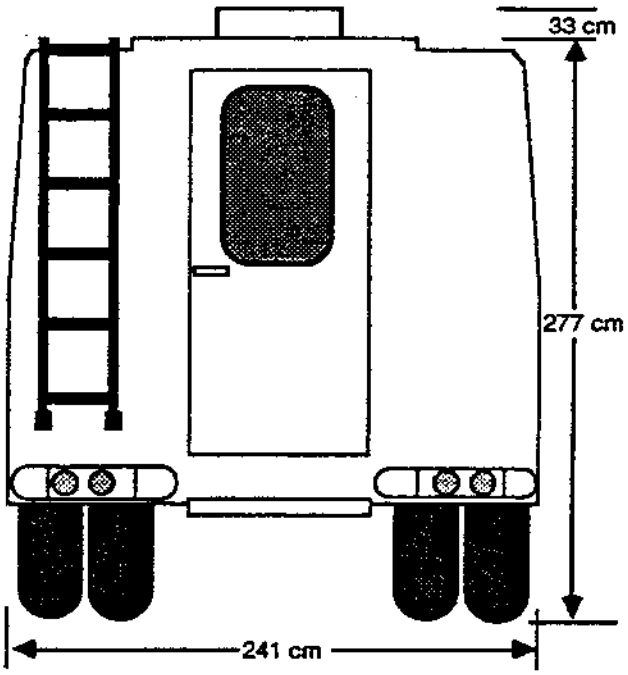
Along the rear edge of the desktop an outlet strip provides power for the various terminals and test equipment. Boft-down inserts with elastic straps prevent computer equipment from sliding on the desktop when the vehicle is moving.

Cabling between devices passes through cable trays mounted close to the ceiling. The tray design securely holds video and communication cables but allows for easy removal or addition. Track-mounted fights above the desk area provide independently aimed illumination.

In addition to the two seats in the cab, a swivel seat, centered in the desk area, is mounted on the waS of a generator compartment. Extra removable seats can be mounted in the aisle for more researchers.



Side View



Rear View

General Navlab Specs

Total Weight: 5449kg
Minimum Turning Radius: 750 cm
Center of Gravity: (112cm, 244cm)
(x,y,) with rear comer on the
driver's side as the origin.

Figure 3-1: Side and Rear View of the Vehicle

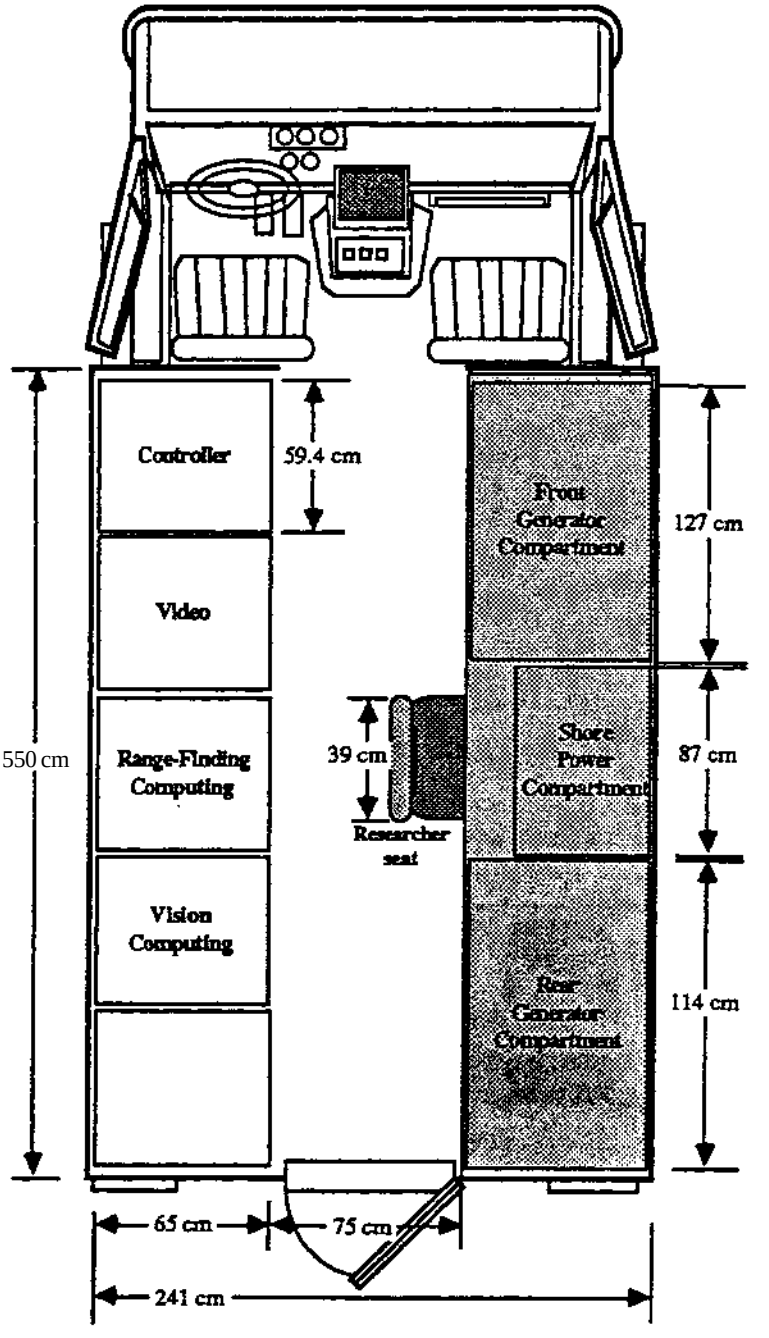


Figure 3-2: Irrterior Layout of Vehicle

3.2.1 Cooling

Hie heat generated by power conditioning, lighting, and electronics would damage some of the experimental compiling. Thus in addition to the air conditioning provided on the van, a standard roof-mounted recreational vehicle air conditioner provides cooling.

Fiber insulation between the shell and interior panels also provides protection from the heat. Insulation inside and outside the shell helps control interior climate. Underfloor insulation keeps heat from the hydrostats and exhaust from entering the interior. Flat sheets of fiberglass covered with thin gauge aluminum are inserted in floor areas between frame members. High temperature silicon-based insulation covered with heavy gauge aluminum foil covers exhaust pipes.

4. Locomotion

Steering and drive motions coordinate to drive the NavLab through planned trajectories. Both axes of motion are controlled by analog signals issued by the controller while in automatic mode or through manual controls.

4.1 Steering

Figure 4-1 shows a front and side view of the NavLab steering mechanism.

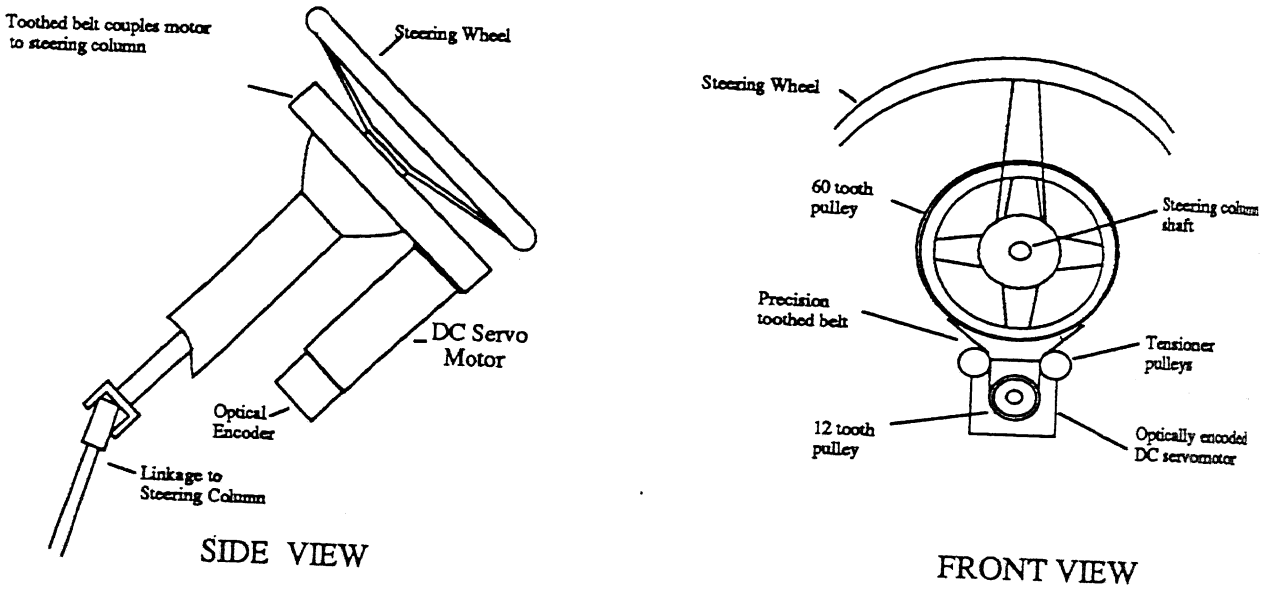


Figure 4-1: Steering Adaptation

The original linkage and steering column are driven by a DC servomotor mounted below the steering column. The motor is connected to the steering column shaft via a precision toothed belt; two toothed pulleys provide a gear reduction of 5 to 1. This configuration provides high enough torque to turn the steering shaft but low enough for the operator to overpower the steering motor in an emergency. A special hub ties all the steering elements together and a safety enclosure houses the moving parts. Limit switches at the extremes of steering travel prevent command error from damaging the system.

4.2 Drive

A hydraulic pump and motor combination comprises NavLab drive. This hydrostatic combination was selected because it provides precise control of position, speed and acceleration. Hydrostatic equipment also has a long history of smooth control and finely adjustable response.

Drive power comes from the main vehicle engine. Engine RPM is limited by a governor to prevent overdriving the attached hydraulic pump. Pump output is controlled by an analog signal.[1] The

[1]This signal originates from either a foot pedal that replaces the standard gas pedal or a drive controller, depending on whether the vehicle is in manual or automatic mode.

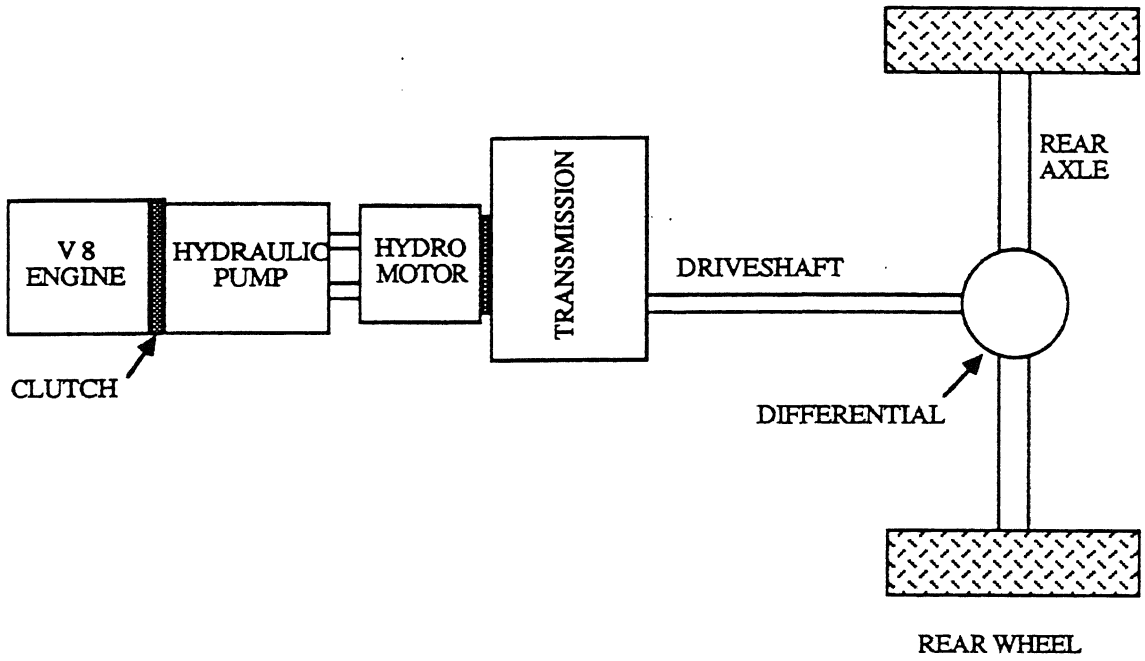


Figure 4-2: Schematic of Vehicle Drivetrain

displacement of the pump (proportional to the signal from the footpedal) determines the speed at which the hydraulic motor moves. The motor in turn powers a two-speed transmission which operates at either a 4:1 (low gear) or 1:1 (high gear) ratio, turning the driveshaft connected to the vehicle differential. Figure 4-2 shows a schematic of the drivetrain.

The configuration described above makes control of vehicle motion simpler than if the vehicle transmission and brakes had to be controlled to produce desired velocity. The standard braking system is intact but is only used in case of emergency because the analog signal to the hydraulic pump controls both accelerations and decelerations.

4.2.1 Vehicle Engine

The standard 350 ci V-8 engine is the main source of driving power. The following modifications were made:

- An electrically actuated clutch was installed to couple the vehicle engine to the hydraulic pump. The clutch is disengaged to isolate it from the engine when the engine is being started.
- The alternator was upgraded to a 120 amp dual output unit to satisfy the additional requirements of the two-battery, 12 volt system onboard.
- One stock emission control air pump was substituted by a hydraulic power takeoff unit. It is driven from the crankshaft end and shares a stock V-belt with the power steering pump.
- An engine oil cooler was installed to reduce oil deterioration caused by the constant high engine temperature.

4.2.1.1 Engine RPM Control

An engine RPM control keeps the vehicle engine running at a determined range of RPM irrespective of grade and speed. A magnetic pickup on the output shaft of the engine provides feedback to a specialized controller that maintains a constant RPM by moving an actuator linked to the engine carburetor. Figure 4-2 shows a schematic of the mechanism.

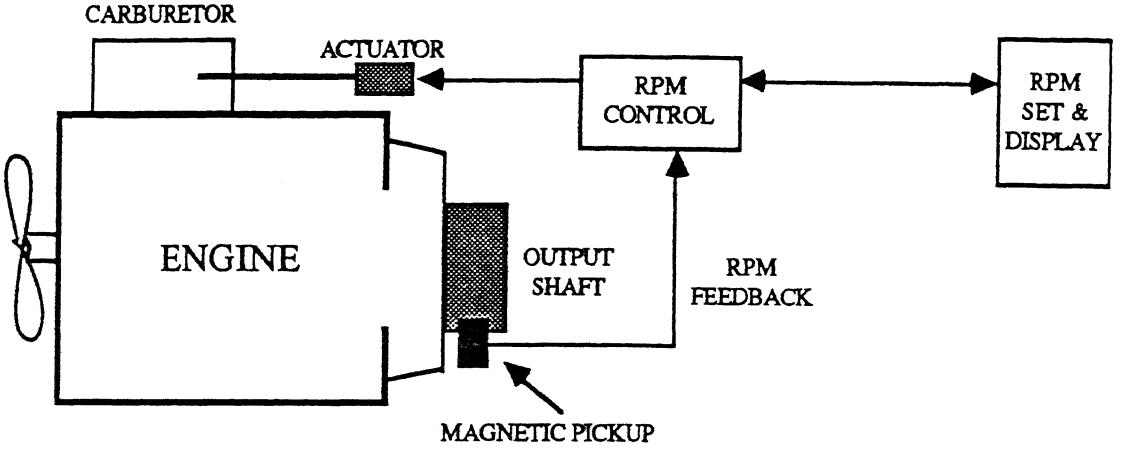


Figure 4-3: Mechanism for Engine RPM Control

4.2.2 Hydraulic Pump

The output of the engine drives a hydraulic pump through a flexible coupling. The pump is suspended from a frame crossmember with rubber shock mounts to allow movement with the engine.

The pump, a Sunstrand axial piston pump, is equipped with an electronic displacement control valve that alters the angle of an internal swashplate between 0 and 18 degrees, depending on an input signal that varies from -10 to 10 V. Negative voltages cause the pump to turn in the reverse direction. At 0 volts the pump has a holding torque to keep the vehicle stationary. At 10 V the displacement is maximum, corresponding to maximum driving speed.

Hydraulic fluid is supplied to the pump from the reservoir by an integral charge pump to replace the fluid pumped to the motor while an equal amount of surplus hot oil is drained from the pump case and passed through the main heat exchanger.

4.2.3 Hydraulic Motor

Hydraulic power from the pump is transmitted to a matching fixed displacement motor attached directly to the transmission. High-pressure flexible hose couples the motor and pump. Because the motor is a fixed displacement type, it always turns the same amount for every unit volume of fluid pumped in, resulting in an RPM of the motor that is directly proportional to the input signal of the pump.

A 10 micron filter cleans the return leg of the high pressure system. An additional crossmember supports

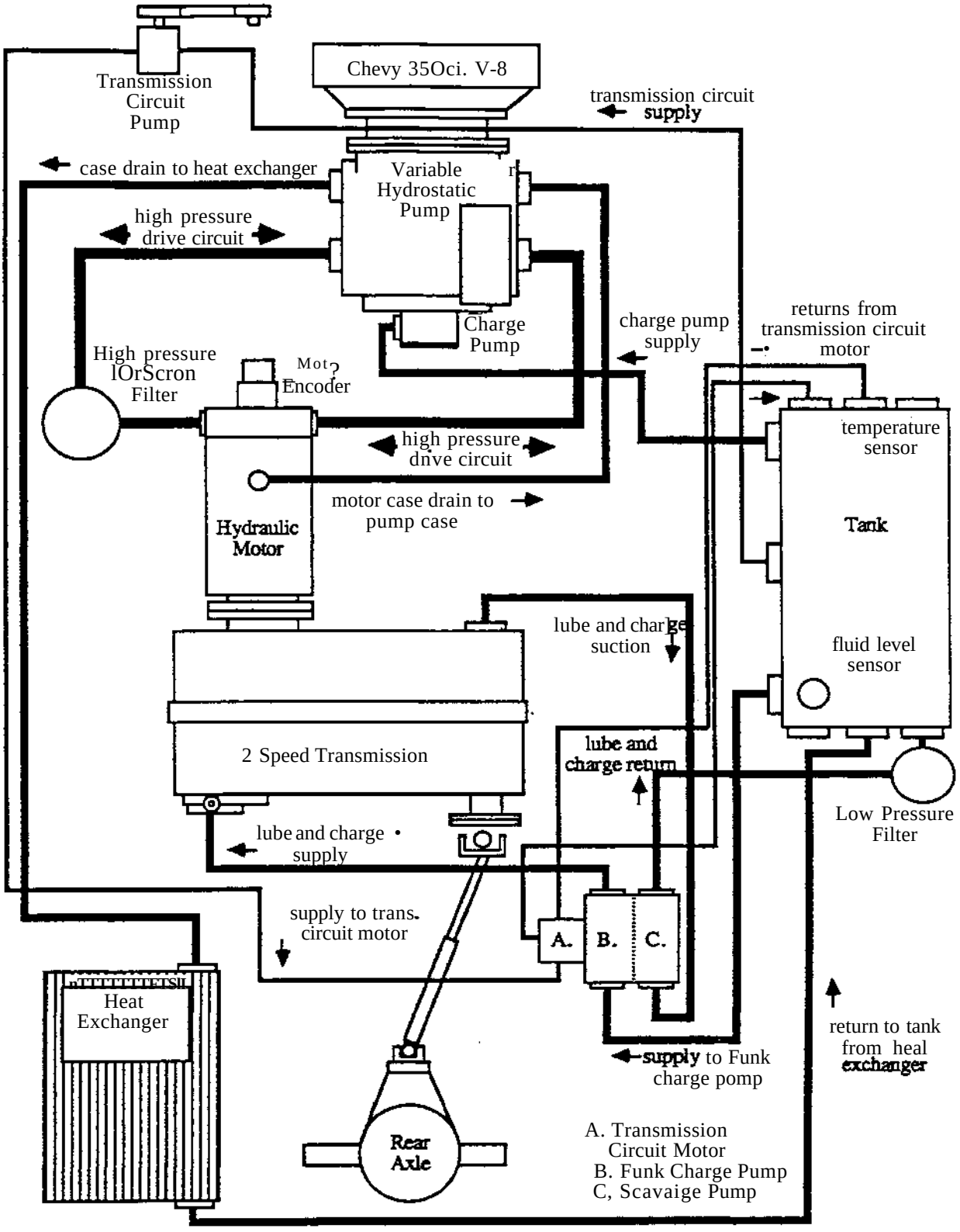


Figure 4-4: Hydro Drive System

the hydrostatic motor. The motor unit has an SAE standard shaft mounted to a mating flange on the driveshaft's forward universal joint yoke.

4.2.4 Transmission

The Funk transmission, an electrically shifted two-speed gear box installed between the hydraulic motor and the driveshaft, is bolted to a reinforced frame member. It is mechanically coupled to the motor on the input side and to the driveshaft on the output side. The transmission provides a ratio of 3.950:1 in low gear and 1.0441:1 in high gear. Low gear supports low-speed experimentation (0-20 km/h); high gear (0-40 km/h) transports the vehicle along public roads in manual mode. The gear is selected electronically by applying a voltage to one of two solenoids on the transmission; if neither solenoid is activated, the transmission is in neutral gear. It is necessary to shift into neutral when changing from one gear to another.

A flow-through lubrication and charge system was added to the transmission to circulate hydraulic fluid. The fluid is returned to the reservoir through a low-pressure filter.

4.2.5 Reservoir, Heat Exchanger, and Filters

A reservoir holds about 80 liters of hydraulic fluid. Because seals and bearing surfaces are sensitive to temperature and contamination of the hydraulic fluid, oil returned to the reservoir must be allowed enough time to de-aerate and cool. Heat is removed by passing oil from the pump case drain through a heat exchanger. Cooled oil is directed back to the reservoir. Dirt in the oil is filtered at two points: in the return leg of the high-pressure system and between the transmission and the reservoir.

A series of valves assist in the cooling and circulation of working fluid. A shuttle valve and a low pressure bleed-off valve act together to allow a small portion of the working fluid to circulate through the oil cooler and reservoir. A make-up pump replenishes the fluid that is removed via a bleed-off valve.

The reservoir is equipped with a thermistor and a level gauge to relay tank status to the vehicle controller.

4.2.6 Hydrostat Sensor and Control System

Figure 4-5 shows the sensing and control system associated with the hydraulic drive system. All the components are located on the underside of the vehicle so all lines enter the vehicle through a wiring port in the floor behind the driver's seat.

Control lines include:

1. Hydrostatic pump displacement: This line controls the swash plate angle in the pump regulating the displacement of hydraulic fluid to the motor.
2. Gear selection: This line controls the gear (high or low) of the transmission.
3. Heat exchanger fan control: This line controls the on/off state of a fan that cools the hydraulic fluid.

Sensors include:

1. Dirty filter sensors: one dirty filter sensor is installed in each of the high- and low-pressure legs of the hydraulic system. These sensors trigger an alarm when they become clogged.
2. Pressure transducers: These read system pressure at input and output of the hydraulic motor.

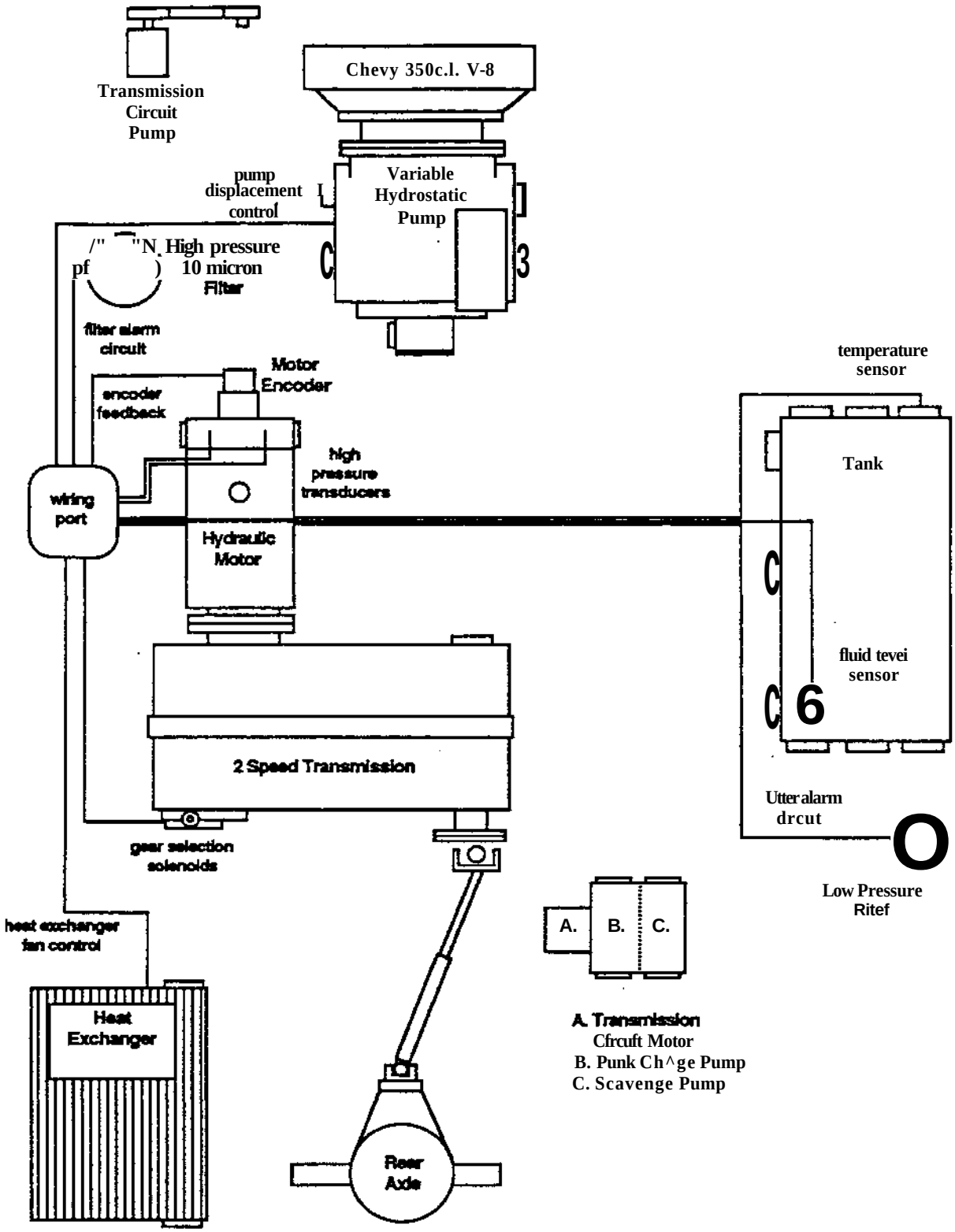


Figure 4-5: Hydrostat Sensor and' Control Lines

3. Reservoir sensors: These measure fluid level and temperature of hydraulic fluid in the reservoir.
4. Motor encoder feedback: An optical encoder mounted on the shaft of the hydraulic motor provides feedback to the drive controller.
5. Steering limit switches: Limit switches are installed on the steering mechanism to signal an alarm if the wheels are cranked beyond acceptable limits.

5. Electrical System

All the electrical power needed by the NavLab is available onboard the vehicle. Electrical power can, however, be brought in from a shore power plug while the vehicle is in a fixed location. Power is distributed such that the generators are not needed to drive the NavLab manually.

5.1 AC Power

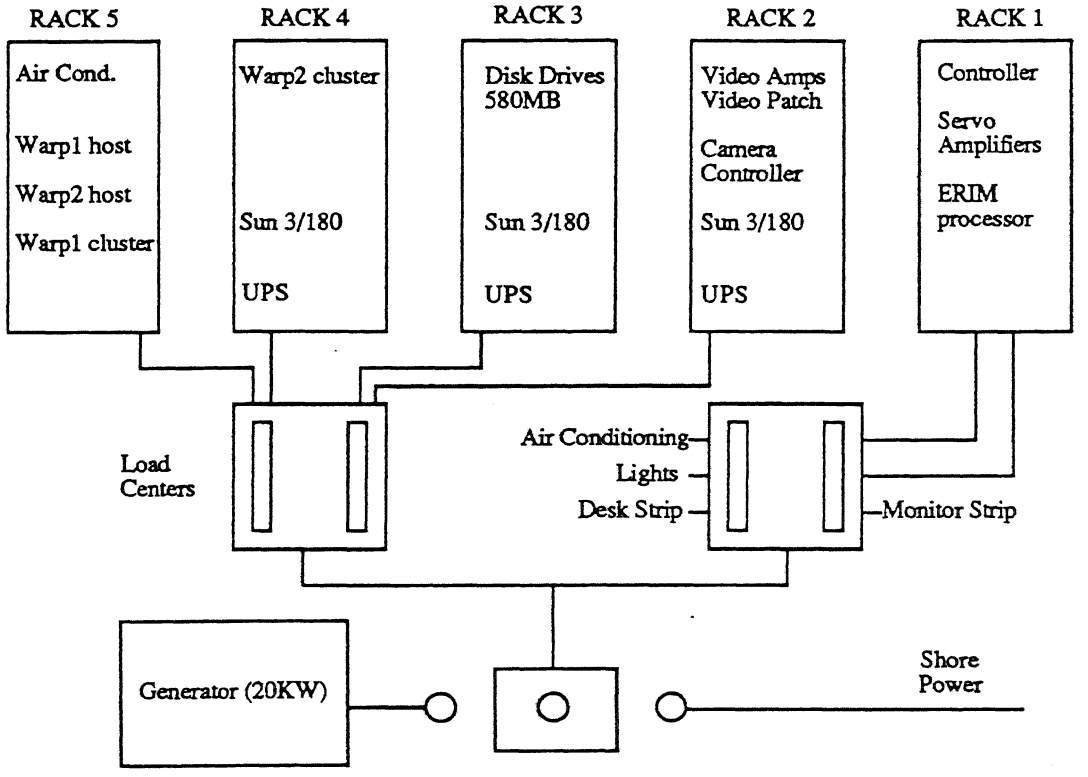


Figure 5-1: Wiring Schematic for AC Power

Figure 5-1 shows a schematic of the AC power system onboard the NavLab.

5.1.1 Generators

The generator supplies 100 VAC power to the variety of devices on the NavLab. The generator resides in a compartment accessible only from the outside of the vehicle, insuring the separation of noxious fumes from the interior.

Two compartments house an engine that is hydraulically coupled to a hydraulic generator unit in the forward compartment. This arrangement allows a single source of power up to 20 KW. Fuel to supply the engine comes from the vehicle fuel lines and the electrical power to start the generator is supplied by the vehicle 12 VDC system. The unit can be stopped and started by a panel switch.

5.1.2 Shore Power

The NavLab can plug into power from a building when stationary, alleviating constant generator operation. An extension cord from a nearby power outlet (220 VAC 50A) mates to a 220 VAC single-phase plug mounted in the outside center compartment.

5.1.3 Power Conditioning

Because variations in load and temperature affect generator power output, the power from the generators must be be conditioned to protect sensitive machinery from spikes and brown-outs. This is done by passing power through Uninterruptable Power Supplies (UPS). These devices not only condition the power from the generators but also provide full-load backup for up to 15 minutes, even if the generators or shore power are shut down, allowing a graceful system shutdown if power fails. Three UPS devices provide a total of 6KW of conditioned power, which-will more than suffice for a complete configuration of computing equipment. The lights, air conditioner, video monitors, and servo-amplifiers do not receive conditioned power because they are much less prone to fluctuations in generator output.

5.2 DC Power

Because many of the devices onboard use DC power, the standard vehicle 12 V system was extended by adding an extra battery and replacing the alternator with a dual output 120 amp unit that charges both batteries.

Figure 5-2 shows a wiring schematic for the DC power system. The original battery powers:

1. Vehicle ignition - starting power for the engine;
2. Dash panel - all switches on the dash panel;
3. Interior lights - overhead lights in the research area;
4. Control electronics - the input voltage to two power supplies. One converts the 12 V into -12 V and the other converts 12 V to 5 V. Each power supply has a limit of 3 amps.

The second battery provides:

1- starting power for the generators;
2. power for generator compartment lights;
3. power to run the hydraulic fluid cooler fan.

An additional 28V power supply is mounted in the equipment racks. This takes an input from the 110 AC system and produces up to 30 amps of current, most of which is used by an ERIM laser scanner. The inertia! navigation device will operate on the same power supply.

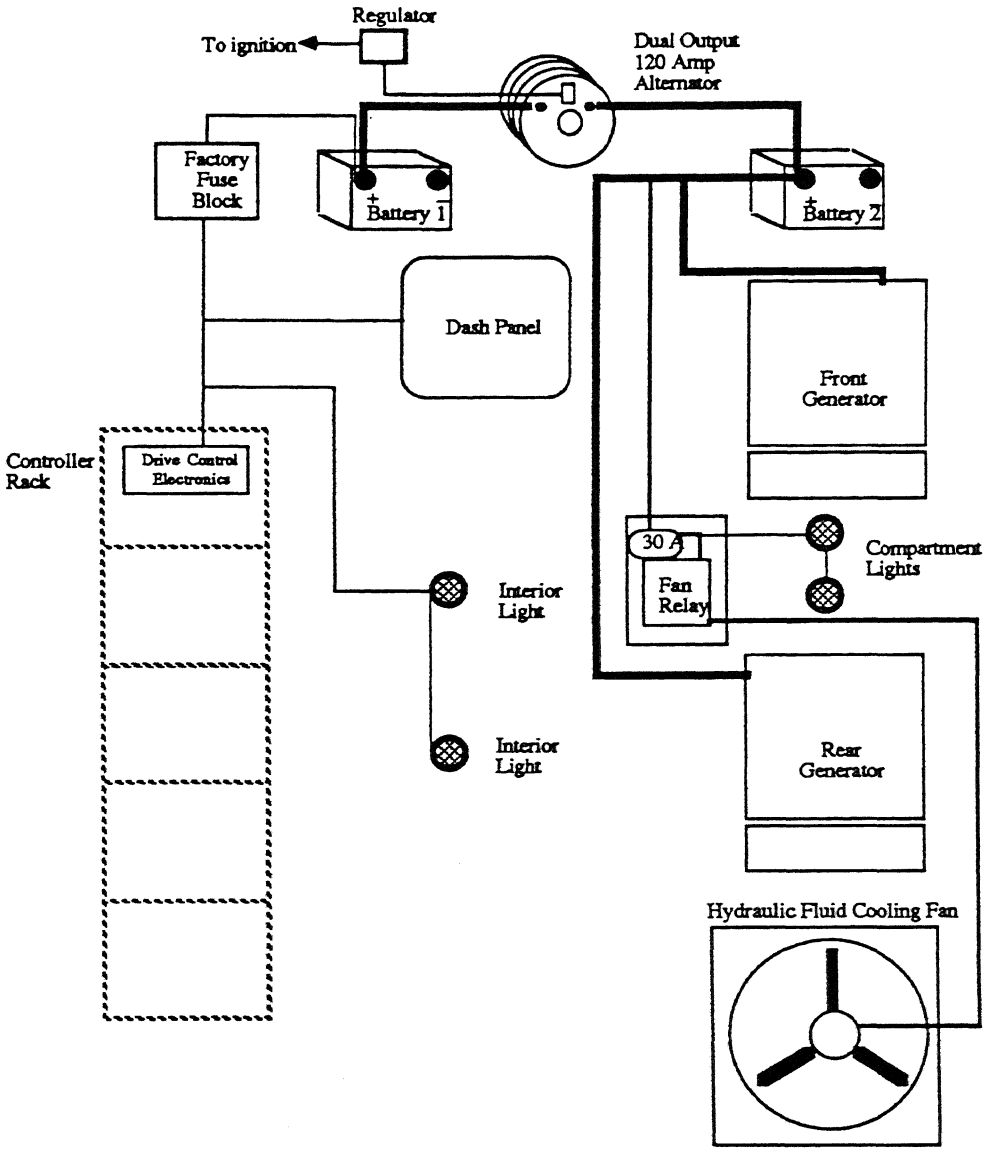


Figure 5-2: Wiring Schematic for DC Power

6. Telemetry

Telemetry to the vehicle was thought to be useful where environmental sensitivity, location, or size of computing equipment precludes installation onboard the NavLab. This feature has not yet found use in practice. NavLab telemetry provides control and monitoring from a remote site, allowing stationary computers to be used in navigation experiments.

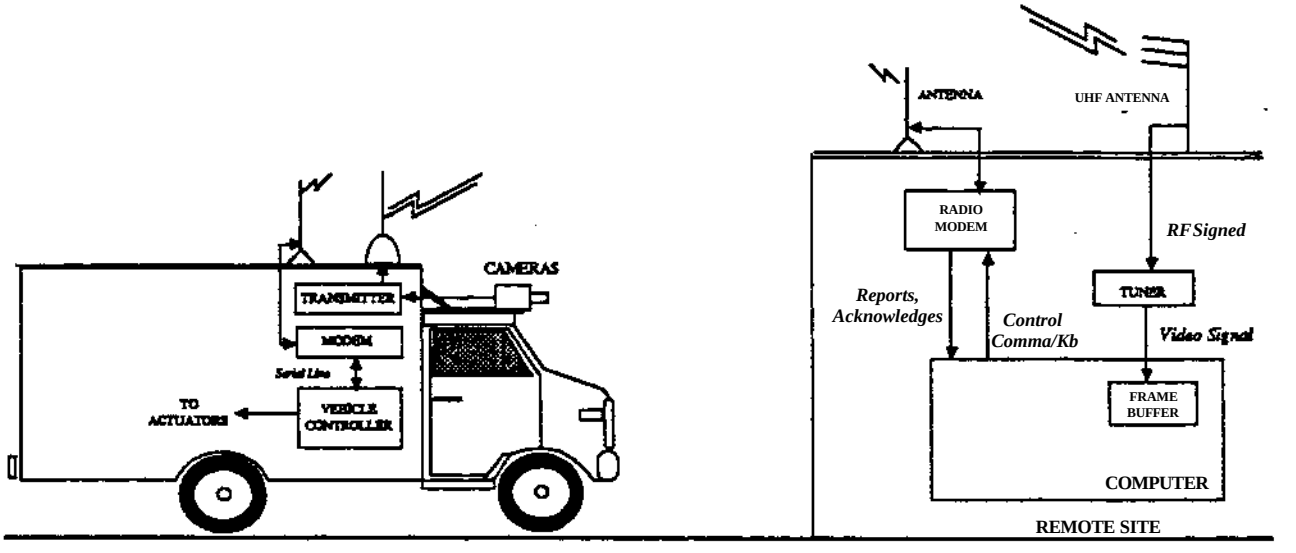


Figure 6-1: Telemetry Configuration of NavLab

The scenario in Figure 6-1 shows the ctosed loop set-up of a vehicle experiment where computing might be distributed offboard. The camera outputs a video signal that is broadcast over a UHF frequency and picked up by an antenna and receiver located on the Carnegie Mellon campus. The receiver provides the video signal to a frame buffer within the computer that processes the image. The signal is digitized and then analyzed. Commands to the vehicle are sent over a serial line to a wireless modem. A radio modem on the vehicle picks up this signal and feeds it to the controller.

6.1 High Bandwidth Transmission

An experimental radio license obtained from the FCC covers several broadcast frequencies. The license covers 2 UHF television channels, a full duplex radio link, and a 2 MHz microwave link.

The video signal is transmitted on the video transmitter while range data are transmitted over an aural sideband of one UHF channel. Because transmission rates can be as high as 56 K baud, the other aural sidebands not currently in use could serve several other data transmission needs.

6.2 Low Bandwidth Transmission

Two sets of 1200 baud radio modems are used for simple, low-bandwidth digital communication. These devices provide a transparent RS-232 connection between computers and facilitate sending commands to and from an offboaid machine and the NavLab controller.

6.3 Cellular Phone

Separation of vehicle from stationary base facilities involves not only communication over distance for the machines but also for humans. For this reason, a cellular phone using existing mobile communications networks within the city has been installed.

7. Perceptive Sensing and Computing

A variety of sensors can be mounted on the NavLab depending on the type of research being conducted. Vision, laser ranging, and sonar ranging are the most popular sensing modes. More details of sensors and computing can be found in articles listed in Appendix V.

7.1 Video

Cameras provide a standard RS-170 video image to the frame buffer. Vision processing transforms this image into a scene description to support navigation.

Typically, a single camera mounted in the front of the vehicle provides a wide-angle view of the scene. Some vision algorithms, however, call for a stereo pair of cameras. Broadcast-quality cameras that provide red-green-blue color signals are used. Remote control units allow control over camera functions like gain, color balance, and iris size. Presently, camera focus and zoom must be controlled manually.

7.2 Laser Ranging

Laser ranging is useful in areas where vision algorithms fail -- in detecting depth discontinuities in scenes where the edges are not obvious and in those scenes that have uneven lighting because of shadows. Whereas the camera is a passive instrument, the laser rangefinder is an active device that emits a beam in a raster fashion and captures the reflection to provide two types of information -- distance and reflectance. The data are then analyzed to provide a scene description. Laser ranging provides a direct 3-D description of the scene while vision requires more expensive computation to extract this information. Range readings are particularly useful because they are not affected by ambient light.

The current laser ranging device, manufactured by ERIM, provides a 256x64x8 bit depth map. The scanner output is processed by a Motorola M68000 processor and sent to a Sun computer dedicated to ranging.

7.3 Pan and Tilt Mechanism

Vision and ranging sensors can be mounted in various configurations. Most configurations call for two independent pan motions -- one for the laser scanner and another for the cameras. Tilt is needed for both the laser scanner and cameras.

Pan and tilt design reflects a need to accurately position sensors over a large viewing range. Less than 4 seconds is required to view 180 degrees.

Cameras are mounted on rigid 5 cm diameter aluminum poles 2 meters long mounted horizontally through a worm drive gearset with a hollow bore. The gearset provides a 50:1 ratio and is driven by a DC brushed servo motor. An 800 line encoder provides feedback for the tilt motion.

7.4 Computing Configuration for Sensing

Figure 7-2 shows configuration of computing for simple, perceptive sensing. Much more complex setups are common. Each sensor commonly requires its own workstation or specialized processor. Another computer runs the blackboard system that integrates perception, modeling, and planning.

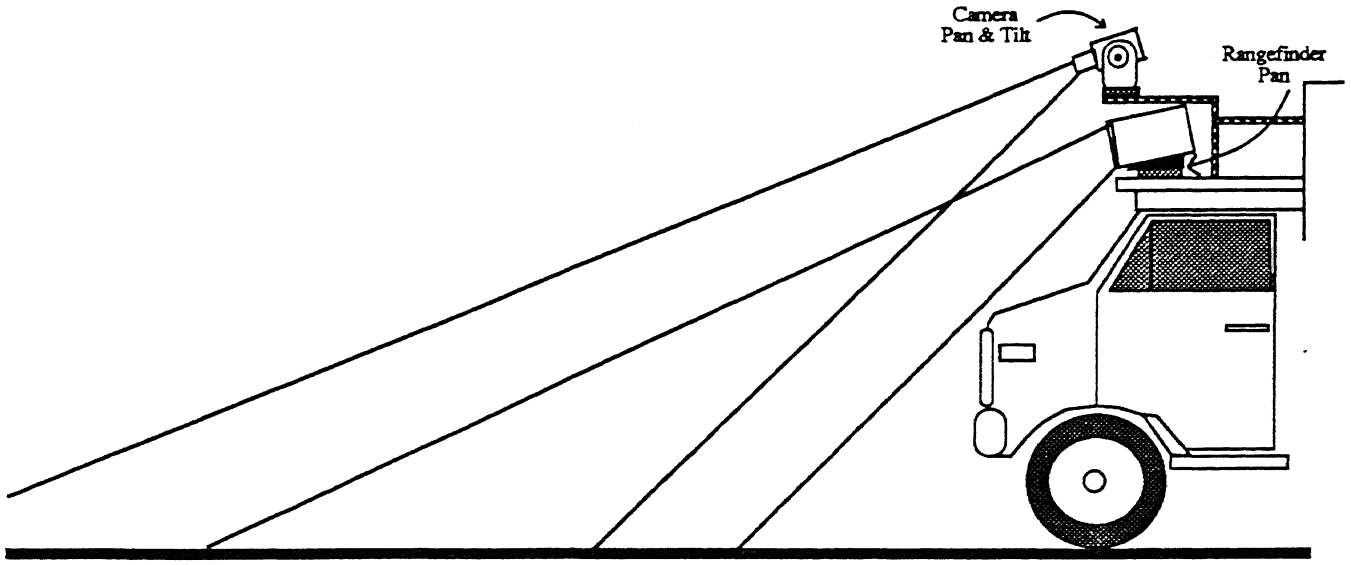


Figure 7-1: Pan and Tilt Mechanism

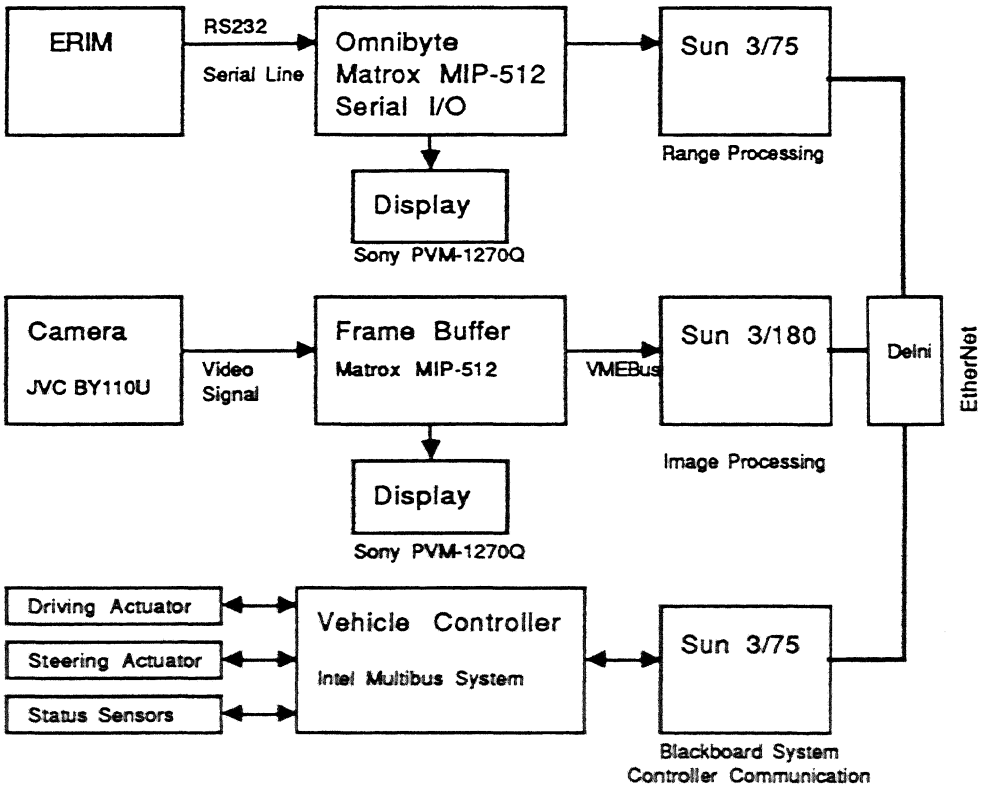


Figure 7-2: Typical Architecture

I. Modifications to Vehicle

The chassis was originally rated for 10,000 lbs. gross vehicle weight. The final vehicle weight was established as 12,000 lbs., thus necessitating a more robust suspension. In order to achieve the necessary load rating the front coil springs were upgraded and two extra leaves were added to the rear springs, increasing the gross vehicle weight by 2400 lbs. Heavy duty gas/oil shock absorbers were installed to minimize a slight tendency to pitch due to the extra weight. In addition, the original equipment tires were exchanged for Goodyear radials with a higher load and all-weather rating.

Modifications to the frame were minimized to preserve strength and stiffness. However, in order to mount some of the larger hydrostatic components alterations had to be made. The main forward crossbeam, transmission rear support beam and surrounding floor were completely removed. The crossbeam was replaced by a box section which bridges the hydrostatic pump. A channel section was added as support for the pump and also provides additional frame strength.

II. Power Budget

NavLab AC power requirements have risen steadily since the first integration to accommodate the WARP computing system, air conditioning, and uninterruptible power supplies (UPS). Power is currently provided by a 20.0 kW generator. The power budget is as follows:

CIRCUIT	PEAK AMP DRAW	PEAK WATTAGE	FUSE
Main No. 1	90	10,000	100
Coleman Air Conditioner	17.5	1,930	20
Lights (4 / 100 w)	3.6	400	5
Monitor Strip (5 / 80 w)	3.6	400	5
PMI Amplifier	18.2	2,000	20
Desk Strip (10 / 60 w)	5.5	600	10
Sola UPS (600 w output)	7.0	750	10
Intel Controller Computer	11.4	1,250	*
Behlman UPS (2000 w output)	27.0	3,000	40
Sun Vision Host Computer	12.0	1,320	*
Disk Drives	6.3	700	*
Development Circuit	15.0	1,650	20
Main No. 2	90	10,000	100
McLean Midwest Air Conditioner	14.0	1,540	20
Behlman UPS (2000 w output)	27.0	3,000	40
Sun WARP Host Computer	12.0	1,320	*
Disk Drives	6.3	700	*
VME WARP Control Cage	9.0	1,000	*
Behlman UPS (2000 w output)	27.0	3,000	40
WARP Power	16.4	1,800	*
Video	4.5	500	5
Erim Laser Scanner	2.7	300	5
Development Circuit	15.0	1,650	20

III. Weight and Center of Gravity Budget

Location in (x,y) is given with left (driver's side) rear corner as the origin. Positive y is toward the cab, positive x toward the passenger side. The shell is 241cm wide (x) and 343cm long (y).

Item:	Location (cm, cm)	Weight (kg)
Driver	63,368	79.4
Passenger	152,368	79.4
Empty Rack & 10kg hardware	51, 34	81.6
	51,103	81.6
	51,171	81.6
	51,240	81.6
	51,309	81.6
Signal Wiring		22.7
Power Wiring		45.4
Insulation		13.6
Passenger side		
Generator	203, 58	117.9
	203,285	117.9
Air Conditioner	117,200	52.2
Power Conditioner	224,171	100.7
Breaker Panel	201, 8	18.1
Table Top	201,171	31.8
Monitor Shelf	216,170	2.3
Sony Monitors	208,279	14.1
	208,165	14.1
Seat	132,157	18.1
Operator	132,157	68.0
Terminals		
Sun	201,216	27.2
Wyse	201,102	6.8
Marlite		
back wall	122, 0	9.5
left	0,171	26.3
right	241,171	11.8
front	122,343	5.0
Gas Tank	135, 97	119.7
Shore Power Switch	198,135	18.1
	198,183	18.1
Shore Power Plug	198,152	2.3
Track Lights	185,178	4.5
Hydraulic Equipment:		
Funk	109,287	124.7
Reservoir	201,272	136.0
Sunstrand Pump	122,373	132.9
Sunstrand Motor	89,330	74.4
Pall Filter	51,333	11.3
Dowty Tandem Pump & Charge Pump Motor	135,203	5.9
Heat Exchanger & Fan	53, 51	24.9

Rack 1:		
" UPS		54.4
2 Amplifiers, 1 Transformer, 3 Power Supplies		52.2
Intel Chassis		24.9
Patch Panel & Control Circuitry		9.1
Tables		11.3
Total	42,287	152.0
Rack 2:		
" Video		22.7
ERIM Power Supply		24.0
Total	42,226	46.7
Rack 3:		
Sun		68.0
UPS		54.4
Total	42,165	122.5
Rack 4:		
Sun		68.0
Total	42,104	68.0
Rack 5:		
Empty		
Total	42, 43	0.0
Pan & Tilt		
2 1135 Motors		8.9
2 Turntables		26.3
1 Tilt Gearing		22.7
1 Aluminum Rod.		4.5
2 JVC's		5.0
ERIM		34.0
PMI motor		4.5
56 C Coupling		4.5
Bracketing		
lower mount		18.1
upper mount		45.4
Total	121,447	174.0
Shell	121,171	1153
Van Bodr		
front axle	101,427	1120
rear axle	101,109	649

Overall:

Carter of Gravity: 112,244 Weight Total: 5449kg

IV. Implementation of the Virtual Vehicle Instruction Set

Protocol

A simple high-level handshaking protocol has been designed for RS-232 communication between the Virtual Vehicle and any host computer issuing commands to the vehicle. It provides a means of exchanging commands and status information with a reasonable level of reliability and optional error recovery. The motivation for this is that errors in communication should be detected and acted upon without interfering with normal operations of host or Virtual Vehicle. By adhering more or less strictly to the protocol, the relative importance and subsequent computational overhead of error-free communication can be chosen at will and may be varied dynamically.

Messages between Virtual Vehicle and host are of the following form:

`<length><packet ID><opcode>[<argument 1>/../<argument n>/]<CR>`

The individual fields of a packet are defined as follows:

`<length>`	2 characters wide. Contains the total number of ASCII characters in the packet, including the length field itself. Length is represented in decimal, so messages are limited to 99 characters.
`<packet ID>`	3 characters wide. Unique identifier to be used as reference to the packet in subsequent protocol transactions. Can be any combination of printable ASCII characters (20H - 7FH), although numeric values (30H through 39H) will be used most frequently. See description below for usage and purpose of this field.
`<opcode>`	2 characters wide. Represents, in decimal, the numeric index of the command to be acted upon by the Virtual Vehicle. This implies a range of 0 through 99 for possible opcodes.
`<arguments>`	Zero or more numeric arguments, of variable width. Arguments are terminated by a slash ("/"). Leading zeroes are allowed. All arguments must be integer values.
`<CR>`	Carriage Return (0DH) character indicates end-of-packet. Must follow immediately after the last argument-terminating slash, or after the opcode in the case of zero arguments. The `<CR>` is NOT considered part of the packet, so it is not included in the `<length>` field.

The following rules define the handshake between two devices. They should be followed closely to achieve maximum communication reliability. However, as is indicated in the appropriate paragraphs, error checking is done at the discretion of the receiving device.

- For every packet to be sent, the sending device generates a unique 3-character packet ID code. This can be done, for example, by incrementally numbering packets or by encoding the current system time. Random generation of codes is discouraged, since this theoretically allows duplicate packet ID's. Using the full range of 96 symbols in each of the 3 character positions yields a range of 884,736 unique ID codes, which is in excess of the anticipated number of messages exchanged during a typical mission of the Virtual Vehicle.
- Packets are prefixed with the length of the packet and terminated by the end-of-packet character <CR>. The receiving device should check the actual length of the received packet against the `<length>` field to ensure integrity of each packet.
- Each packet received may be further validated by ensuring that
  - the opcode is valid,
  - the number of arguments is correct for the given opcode, and
  - the arguments are within allowable limits. These limits may change dynamically as a

function of, for example, vehicle speed or road conditions.

- If none of the above error conditions are detected, the receiving device returns an ACK message to the sender, indicating that the message was received correctly and the appropriate action, if any, is being performed. The ACK message uses the *same* packet ID as that of the message being acknowledged.
- In case of an error, the receiving device must return a NAK message to the sender to indicate that the packet was rejected and no command is being executed. As its only argument, the NAK message contains an error code indicating the reason for rejecting the packet. A NAK message also has the same packet ID code as the message in question.
- Upon receiving a NAK as reply to a message, a device has the option of retransmitting the offending message (with new packet ID), logging the error, ignoring it, or taking any other action that might be appropriate. By the same token, while expecting a NAK or ACK in response to a transmitted message, a device may choose to time out, wait forever, or take other appropriate action. These conventions provide for very flexible operation that allows critical system operations to continue even in case of protocol errors. In dealing with these situations, the Virtual Vehicle will adhere to the following conventions:
  - NAK or ACK messages are *always* generated and sent in response to data messages received by the VVI. At this point, handshake for the current message is considered complete; i.e., no further action is *expected.*
  - Unexpected NAK or ACK messages (i.e., those referring to an unknown or previously acknowledged packet ID) are ignored. However, the error *is* logged and/or announced at the VV system console.
  - If a NAK is received as response to a message originated by the VV, the message may or may not be retransmitted repeatedly (with a new packet ID), depending on the type of message and reason for rejection.
  - If neither NAK nor ACK is received by the VV within a certain timeout period (configurable parameter, typically on the order of seconds), the error condition is logged and/or announced at the VV system console. After this, the VV still expects a response to the packet in question, but no further action is taken and subsequent messages are treated as if no error had occurred.

Commands issued by Host/Console

Mnemonic	Opcode	Meaning	Arguments
NAK	00	Negative Acknowledge	cc/ cc= 00 : Packet length error 01 : Num. of Args error 02 : Not ready for cmd 03 : Illegal Opcode 1X : X'th argument below current minimum limit (1 < X < 9) 2X = X'th argument above current maximum limit (1 < X < 9)
ACK	01	Acknowledge	
ABO	02	Abort Motion	
STO	03	Stop/Suspend Motion	
STA	04	Startup	
TRA	05	Travel	l/r/i/ l :arc length r :radius of curv. i : 1 = immediate

CHP	06	Change Position	x/y/h/ x : Δ x pos. y : Δ y pos. h : Δ heading
STM	11	Set time to zero	
SVL	12	Set Velocity	v/i/ v : velocity i : 1 = immediate
SAC	13	Set Acceleration	a/i/ a : acceleration i : 1 = immediate
SP1	14	Set Pan 1	p/ p : pan angle
ST1	15	Set Tilt 1	t/ t : tilt angle
SP2	16	Set Pan 2	p/ p : pan angle
ST1	17	Set Tilt 2	t/ t : tilt angle
SSR	18	Set Steering Rate	s/ s : steering rate (0-99 %)
INF	21	Get Vehicle Info	
POS	22	Get position	
TIM	23	Get Vehicle Time	
VEL	24	Get Vehicle Velocity	
ACC	25	Get Vehicle Acc.	
PN1	26	Get Pan Angle 1	
TL1	27	Get Tilt Angle 1	
PN2	28	Get Pan Angle 2	
TL1	29	Get Tilt Angle 2	
STR	30	Get Steering Rate	
ROL	31	Get Roll	
RLR	32	Get Roll Rate	
PIT	33	Get Pitch	
PTR	34	Get Pitch Rate	
HDR	35	Get Heading Rate	
REP	36	Get Status	d/ d : device number

Responses from Virtual Vehicle

Mnemonic	Opcode	Meaning	Arguments
NAK	00	Negative Acknowledge	cc/ cc= 00 : Packet length error 01 : Num. of Args error 02 : Not ready for cmd 03 : Illegal Opcode 1X : X'th argument below current minimum limit (1 < X < 9) 2X = X'th argument above current maximum limit (1 < X < 9)
ACK	01	Acknowledge	
RVI	51	Vehicle Info	l/w/h/g/m/x/y/s/

			l : length w : width h : height g : weight m : minimum turning radius x : X of C.G y : Y of C.G s : 0 = wheel steer 1 = skid steer
RVP	52	Vehicle position	x/y/h/t/ x : x position y : y position h : heading t : time
RVT	53	Vehicle Time	t/
RVV	54	Vehicle Vel.	v/t/
RVA	55	Vehicle Acc.	a/t/
RP1	56	Pan angle 1	p/t/
RT1	57	Tilt Angle 1	t/t/
RP2	58	Pan angle 2	p/t/
RT2	59	Tilt Angle 2	t/t/
RSR	60	Steering Rate	s/t/
RRL	61	Roll	r/t/
RRR	62	Roll Rate	r/t/
RPT	63	Pitch	p/t/
RPR	64	Pitch Rate	p/t/
RHR	65	Heading Rate	h/t/
RST	66	Report Status	d/s/t/ d : device number s : status code
ADN	80	Arcdone	id/x/y/h/t/ id : arc ID code x,y : Pos. at end of arc h : heading at end of arc t : time at end of arc

Note: All distance units are cm, time units are msecs, velocity units are cm/s, acceleration units are cm/s^2 and angle units are half degrees. Weight units are kilograms.