**********

The VEOS project
Human Interface Technology Lab
University of Washington
Seattle, WA

# VEOS 2.0

# Tool Builders Manual

Geoffrey P. Coco
May 4, 1992

**********

**********

The VEOS Project Team:

William Bricken
Geoff Coco
Dav Lion
Andy MacDonald

**********

**********

CONTENTS:



**********

**********

## What is VEOS?

**********

VEOS is an extendable environment for prototyping distributed applications for unix. The VEOS application programmer's interface is provided by XLISP 2.1 (by David Betz). XLISP provides familiar program control; VEOS provides interprocess message passing and content addressable database access.

VEOS (The Virtual Environment Operating Shell) was developed for distributed Virtual Reality applications at The Human Interface Technology Lab in Seattle and has been in use for two years. However, VEOS is by no means limited to these types of applications.

VEOS is ideal for applications where hardware resources are not physically proximal or where machine-dependent resources (e.g. software packages, interface devices, etc..) are isolated because of their platform.

VEOS is also ideal for prototyping programs which employ coarse grain parallellism. That is, VEOS uses heavyweight sequential processes, corresponding roughly to unix processes. In this way, VEOS can be used to utilize a network of workstations as a virtual multiprocessor.

C programmers can build custom VEOS tools which are accessable from XLISP and thus are immediately compatible with other VEOS tools. Lisp programmers can quickly design and run distributed programs utilizing diverse hardware and software resources.

VEOS is not an operating system. VEOS is a user-level framework for prototyping distributed applications. Its primary focus is flexibility and ease of use. This design comes somewhat at the expense of real-time performance. This is not to say, however, that VEOS cannot achieve good performance with proper application structuring and tuning.

Relying on only the most common unix interface, VEOS is platform independent. VEOS 2.0 has been extensively tested on platforms such as DEC 5000, Sun 4, and Silicon Graphics VGX and Indigo.

VEOS, the Virtual Environment Operating Shell, is a software environment for prototyping distributed virtual world applications.

**********

**********

Fundamentals of VEOS

**********

With VEOS, a programmer can specify how to accomplish the many computational tasks of a proposed virtual environment. VEOS allows a programmer to clump these tasks into computational nodes of a distributed system. These clumps of computational tasks are known as 'entities'.

Presently in VEOS, each entity is implemented as a distinct Unix process with the following native capabilities: interpretation of a coded task description (presently written in lisp), inter-entity communication, generalized data management (akin to tuple architecture), and generic pattern matching over local data space.

With VEOS, a designer can implement a virtual environment using a set of entities, each residing on the same or different Unix based machines (presumimg network accessibility). Furthermore, because VEOS relies only on the most common Unix interface, one virtual environment application may utilize multiple hardware platforms.

The best method for utilizing VEOS entities as nodes in a distributed application greatly depends on the parameters of the applcation (e.g. the computational resources available, the topology of the network, the location and compatibility of interface devices, etc..). Consequently, the VEOS design, whereever possible, foregoes making policy decisions so that the programmer enjoys maximum flexibility in designing a distributed application.

VEOS programmers have had mixed results with various distributed models (e.g. The Fern System). It should prove useful to study these prototypical models for subtleties before attempting a full-scale distributed application with VEOS.

## Entities

...

Since VEOS is intended as a software platform for distributed virtual environment applications, a primary design focus was the mechanism for specifying the tasks and assigning computational resources in a distributed design. The resulting model is the following.

A VEOS application is broken down into distinct processes which are as self-reliant as possible (the more self-reliant, the more efficient the whole system

will be). Each process can be coded separately with provisions for communicating with the other processes in the system. Each process is implemented by a single VEOS entity on any network accessible Unix workstation.

A VEOS entity, then, is a stand-alone executable program that is equipped with the same VEOS native capabilities: data management, process management, and inter-entity communication.

## The Grouple

...

One of VEOS's aims is to support a consistent and general format for program specification, inter-entity communication and database management.

Independently, these needs can be solved by some existing methods. For example, Lisp provides generalized data abstraction (i.e. lists), and useful program control. Mathematica provides a generalized specification format and consistent treatment of data and function. Linda provides generalized data management.

But these tools do not share a common form. In addition, large components of these systems are extraneous for most situations. Thus, intergrating such packages would prove costly and wasteful. Moreover, tying together such varied systems emphasizes the differences between their respective forms and expressions rather than their commonalities.

Consequently, we chose an enhancement to the tuple as the VEOS general format. This format is called the 'grouple'. Grouples can be seen as a nestable tuples. Grouples are extrordinarily general for flexibility, yet the data they contain can be accessed specifically for debugging and performance efficiency. The subfields of a grouple are called 'elements'. As grouples can be nested, an element can be a grouple.

Nancy, the VEOS data manager uses the grouple for it's standard data format. Talk, the VEOS communications module, transports linearized grouple stuctures.

VEOS now has a lisp interface. The Lisp language was chosen because there is a clear mapping from lisp lists to VEOS grouples. Furthermore, because lisp boasts program-data equivalence, programmers can store and pass fragments of entity specification using the VEOS kernel mechanisms.

## Grouplespaces

...

Technically, a grouplespace is an ordinary grouple. There are two grouplespaces associated with an entity which partition the entity's data according to the following semantics.

The grouplespaces are: the workspace, the programmer's runtime database. And the inspace, where incoming inter-entity messages are placed. In VEOS 2.0, the inspace has been bypassed as an efficiency concern.

## Pattern Matching

...

The regime of pure algebraic reduction is well known to provide the ability to perform many kinds of computation (see prolog). This scheme, also known as the Match/Substitute/Execute paradigm (ex. in expert systems) can be implemented using a pattern matching mechanism.

Nancy supports pattern matching over its grouplespaces. A pattern is a specific form that data can take. Patterns can contain specific data to match or wildcard symbols which can be used to bind variables to values. For specifics of the Nancy Pattern Matching Language, see the section Using VEOS from Lisp.

**********

Components of the VEOS Kernel

**********

## Nancy: The VEOS grouple manager

...

Nancy is a homebrew database manager designed specifically for the VEOS project. Much like Linda systems manage data as tuples, Nancy mananges data as grouples.

Nancy performs all grouple manipulation within VEOS. This involves creation, destruction, insertion, copying, etc.. Since Nancy grouplespaces are merely named grouples, these are also an entity's fundamental database operations.

In addition, Nancy provides a powerful semantic for data searching, inserting and replaceing within a grouplespace. This is the pattern matcher. It is a complete set of primitives to implement a match/substitute/execute engine as in an expert system.

From lisp, programmers have access to three Nancy operations, vput, vcopy and vget. These are sufficient for all grouplespace operations. As lisp expressions are passed to these nancy primitives, they are converted to grouple format and then handled by nancy's standard C interface.

## Talk: The VEOS communications module

...

Talk is the only supported mechanism for entity communication. As VEOS entities provide coarse grain parallelism though Unix processes, process (entity) synchronization and shared memory is not practical. Thus VEOS supports message passing as the only means of entity communication.

Talk provides VEOS with two simple message passing primitives, vthrow and vcatch. From lisp, programmers can transmit a message *asynchronously* to another entity. The receiving entity is then at leisure to receive the message, also *asynchronously*.

Asynchronous message passing means this: When an entity tranmits a message, the operation succeeds *reliably*, whether the receiver is waiting for a message or not. Likewise, an entity can always check for incoming messages without waiting indefinitely for a message to arrive. This is also called non-blocking communication.

Talk passes messages between entities with a flat linearized grouple format. The lisp/veos interface module performs this translation.

## Shell: The VEOS kernel control module

...

The shell is the administrative workings of the VEOS kernel. It dispatches initializations, interrupt handling, manages kernel memory, etc..

**********

Using VEOS from LISP

**********

## veos methodology

the veos kernel provides:

1. fast and powerful local database access.
   - built-in pattern matching.
   - internal grouple form converts to lisp lists.

2. clean and reliable message passing.
   - utilizes Berkeley sockets.
   - symbolic entity addressing.

3. well defined C interface.
   - can be retrofit to any high-level language (like lisp).

## getting your feet wet

* Get an account on a Unix machine that supports VEOS (Suns, DECs, SGs).

* Make sure /home/veos/bin/ is in your path.

* You are ready to run a basic VEOS entity. The basic VEOS entity with is built with an xlisp interface. What you get is complete xlisp plus the VEOS native primitives described in the next section.

You can run entities at least two ways:

1. Type 'entity' at the Unix command line.

2. Alternatively (and recommended), you can run entities from inside emacs.

   Within emacs, do meta-x eval-expression:
   (setq inferior-lisp-program "/home/veos/bin/entity")

or put this line in your .emacs file.

Then, do meta-x run-lisp.  Once in this emacs mode, you can do meta-x describe-mode to learn about fancy emacs features with respect to lisp interaction.

*  To load a program into lisp, type (load "filename.lsp") to lisp.

---

## veos kernel interface

Currently xlisp provides the abstract access to veos kernel capabilities.  The following describes the native priitives as of February 8, 1992.

### startup veos kernel from lisp

```
(vinit <port number> )
; initialize veos kernel,
; establish network port for incoming messages.
; if port number is not given, veos chooses an available port.
; vinit returns uid of this entity upon success
```

### takedown veos kernel from lisp

```
(vclose)
; takedown veos kernel,
; cleanup network connections.
; xlisp (exit) function will also perform necessary veos takedown.
```

### kernel system tasking

```
(vtask)
; pass control to veos kernel to perform network activity.
; for best performance, call vtask every time through
; the entity's main loop, calling (vcatch) immediately
; afterward.
```

```
local grouplespace access
''''''''''''''''''''''''''
       (vput <data element> <nancy pattern> :freq <"all"> )
       ; add data to local grouplespace.
       ; data element can be a list.
       ; pattern for vput may contain ^.
       ; optional frequency key-argument ("all" is only value allowed)
       ; requests exaustive matching.

       (vcopy <nancy pattern> :test-time <time-stamp> :freq <"all"> )
       ; non-destructive local grouplespace query.
       ; pattern for vcopy may not contain ^.
       ; optional timestamp key-argument must be created by (vmintime)
       ; given timestamp is modified in place.

       (vget <nancy pattern> :freq <"all"> )
       ; destructive local grouplespace query.
       ; pattern for vget may not contain ^.

       (vmintime)
       ; returns a guaranteed oldest time stamp.

       ; see below for nancy pattern language specification.


message passing
'''''''''''''''
       (vthrow (destination1 destination2 ...) (lisp expression) )
       ; send to a list of desinations.
       ; list may contain zero or one destination, and
       ; the same destination may appear more than once.
       ; destination is lisp vector of hostname and port number,
       ; ex.  '#("vorpal" 7800)
       ; message is appended to each desination's incoming grouplespace.

       (vcatch <nancy pattern> )
       ; perform standard nancy pattern match from incoming grouplespace.
       ; acts like vget in all other ways.
       ; since new messages are appended, oldest message is at top.
       ; to simply retrieve oldest message, use: (vcatch '(> @ @@)).

       ; NOTE: currently, VEOS 2.0 does not support
       ; the pattern argument for efficiency reasons.
       ; (vcatch) simply returns the oldest available message.
```

## caveats of xlisp for veos

There are many types of VEOS data elements. The most common and useful element types to programmers are available from lisp. The VEOS types that are supported by lisp are:

| VEOS | equivalent in Lisp |
|---|---|
| grouple | list - NIL or () is the empty grouple |
| integer | integer - use a regular number |
| float | float - use a decimal point |
| string | string - use "" to make a string |

Likewise, lisp supports many data element types. There is an equivalent data element in VEOS for some of these. The data elements that VEOS supports are those listed above and the following:

| Lisp | equivalent in VEOS |
|---|---|
| symbol * | prim type. |
| vector | special grouple |

* A symbol in lisp can be used for a variable, function, or be unbound. When a lisp symbol is passed to VEOS (for grouplespace storage of message passing), VEOS retains the name of the symbol but not the bindings. Lisp normally retains these symbol bindings.

All other lisp data types are unsupported by VEOS. This means that they cannot be passed to Nancy for grouplespace storage. And they cannot be used within inter-entity messages.

## nancy pattern matching language

Nancy controls access to the grouplespaces through a pattern matching language. The goals of the language are to provide a succinct format to express the locale in the grouplespace you would like to work and what you would like to do there.

There are three fundamental grouplespace operations that Nancy provides:

- find some location (specified by pattern), insert data there.
- find some data (specified by pattern), retrieve it.
- find some data (specified by pattern), retrieve it and replace it with other data.

Thus, these are the rules of nancy pattern matching:

To find anything (called a site), specify a pattern. Patterns consist of information about where the site is and/or specific data that the site contains.

> The ^ (void) symbol specifies a location within a grouple for inserting. It points to the void between data. Technically, the ^ always matches.
>
> The > (mark) symbol points to a piece of data within a grouple retrieval or replacement operation. It designates the immediately following element of the pattern as the site of action. The > does not itself match data.

For a pattern to be meaningful, one of these symbols (^ or >) must appear somewhere in the pattern. In other words, the pattern must always specify a site of action.

To match data within a pattern, you can either specify actual data to compare, or wildcard symbols which are content-blind.

> The @ (this) symbol specifies a single element at a specific location within a grouple. This will match any single element including a grouple.
>
> The @n (these) symbol specifies exactly n sequential elements within a grouple. This will match the next n elements. @1 is equivalent to @.
>
> The @@ (these all) symbol specifies zero or more elements within a grouple. This will match all the remaining elements in a grouple. This special form is allowed only at the end of a pattern grouple.

The ** (any) symbol specifies zero or more elements *anywhere* within a grouple. This will match all the remaining unmatched elements in a grouple. This special form is allowed only at the end of a pattern grouple.

The ~ (touch) symbol specifies that the immediately following element be 'touched' during a (vput ...) operation. That is, the data that matches the pattern symbol following the ~ is marked as having been recently modified. There can be any number of ~ in a pattern.

Anything else is taken literally and compared with the actual data in the grouplespace.

Note that nancy patterns are recursive. That is, a pattern may contain a grouple which may contain wildcards and data, some of which are more grouples, etc..

**Examples.**

Here is an actual session with lisp. My comments are added.

```
;; we begin with an empty grouplespace, ie ().
;; to vput, we match the grouplespace and point to the void within it.
;; thus, the (^) pattern.
;; this is literally where the given data is put.

> (vput "first" '(^))
T

;; here, we reqeusted a simple insert operation.  so vput vput return T or NIL
;; depending on the success of the match.  during a replace vput operation, vput
;; returns what your action replaced in the grouplespace.  in a moment, we'll
;; see when this comes into play.



;; to see the entire contents of the grouplespace, we need to do a
;; nondestructive query.  the most succinct way to do this is to match the
;; grouplespace and copy all of its elements:

> (vcopy '(> @@))
("first")

;; note that the @@ can match a single element if there is only one, or it can
;; match many elements if there are more than one.  in the latter case, the
;; results are conveniently returned in a list.  in the interest of consistency,
;; all nancy results are returned in lists.
```

```
;; let's now insert a list after the first element of the grouplespace:

> (vput '("third") '(@ ^))
NIL

;; again, no data was removed in the process.


;; the entire contents...

> (vcopy '(> @@))
("first" ("third"))


;; let's now insert an element between the first element and all the rest of the
;; elements in the grouplespace.

> (vput "second" '(@ ^ @@))
NIL

;; note that (@ ^ @) would have also worked anbd would have been more precise.


;; inspect the entire contents

> (vcopy '(> @@))
("first" "second" ("third"))


;; here, we are inserting a vector into the grouplespace.
;; this demonstrates two other features.  the @2, in the given position in
;; the pattern matches the first two elements of the grouplespace.  the "third"
in
;; the pattern matches the actual data in the grouplespace.  this, of course, is
;; unlike how an @ matches any element where no data is compared.

> (vput '#(1.4 3.9 9.0) '(@2 ("third" ^)))
NIL


;; the data now resides where the ^ was in the previous pattern.

> (vcopy '(> @@))
("first" "second" ("third" #(1.4 3.9 9)))


;; sometimes it is useful to replace existing data in the grouplespace.
;; this can be done by removing, then inserting correct data.
```

```
;; or it can be done with a replacing vput.
;; here, we replace the first element of the grouplespace with the given data.
;; we must also match the remaining elements in the grouplespace to achieve a
;; successful match.

> (vput "uno" '(> @ @@))
("first")

;; vput returns the removed data.


;; and the first element has been replaced.

> (vcopy '(> @@))
("uno" "second" ("third" #(1.4 3.9 9)))


;; now, let's look at the more subtle features of pattern matching.  begin by
;; emptying the current contents of the grouplespace to begin a new session.

> (vget '(> @@))
("uno" "second" ("third" #(1.4 3.9 9)))


;; confirm that the grouplespace is empty

> (vcopy '(> @@))
NIL


;; here, we'll insert some new data

> (vput '("animal" "giraffe") '(^))
T
> (vput '("plant" "fern") '(^ @))
T
> (vput '("animal" "lion") '(^ @2))
T
> (vcopy '(> @@))
(("animal" "lion") ("plant" "fern") ("animal" "giraffe"))


;; now we can perform useful matches on the grouplespace.  suppose we want to
;; find the tag associated with the "fern" data.  we simply ask for the element
;; immediately preceding the "fern" element.  note the use of the ** wildcard.
;; the ** pattern element has two functions in this pattern.  first, like @@ it
;; matches all the reminaing elements in the grouple.  second, it explicitly
;; makes the containing grouple an order-independent pattern.  in other words,
```

```
;; when nancy sees a ** in a pattern grouple, it ignores the order of the source
;; (grouplespace) elements when matching; it matches purely by content.

> (vcopy '((> @ "fern") **))
("plant")

;; note that although the marked element is "plant", but the result is contained
;; within an extra grouple.  as explained above, all nancy results are contained
;; within a grouple.


;; if we want to find the data associated with the tag "animal", we can do a
;; similar match.

> (vcopy '(("animal" > @) **))
("lion")


;; but this is only a partial answer.  because there is more than one instance
;; of the tag "animal" at that level of the grouplespace.  we may want all the
;; possible matches of this form.

> (vcopy '(("animal" > @) **) :freq "all")
("lion" "giraffe")


;; we can also use this feature with vput to do an exaustive replace.  here,
;; xlisp allows the arguments to a function to appear on consecutive lines.

> (vput "mammal"
         '((> "animal" @) **)
        :freq "all")
("animal" "animal")

;; vput returns exactly what it replaced.
;; check that the replace was successful.

> (vcopy '(> @@))
(("mammal" "lion") ("plant" "fern") ("mammal" "giraffe"))


now let's look at the pattern matching features that correspond to the
dynamic issues of maintaining the grouplespace.  specifically, matching that
is sensitive to the relative ages of the data can be used for so-called
'delta matching'.  we can use the previous contents to illustrate.

first, we'll make a nancy time stamp.
```

```
> (setq ts (vmintime))
0


;; we'll use this time stamp with vcopy.  if you pass a time stamp to vcopy, it
;; perform the given match but only returns matched data that is *younger* than
;; the time stamp.  vmintime returns a guaranteed oldest time-stamp.  this means
;; that using it will guarantee seeing everything in the grouplespace.

> (vcopy '(> @@) :test-time ts)
(("mammal" "lion") ("plant" "fern") ("mammal" "giraffe"))

;; vcopy compared all the matched data against the time stamp, all of it had
;; been modified (with vput) after the time given by the time stamp.


;; vcopy *modifies* your time stamp in place to reflect that you've matched with
;; that pattern.  and as you can see, the time stamp has now been modified.

> ts
1.4013e-44


;; if we make the same match again with the same time stamp, we see no data.
;; this corresponds to there being no change (or delta) in the data since we
;; last matched.

> (vcopy '(> @@) :test-time ts)
NIL


;; let's insert some new data to see how the delta will be shown.

> (vput '("mammal" "fox") '(> (@ "lion") **))
(("mammal" "lion"))


;; when we match the entire grouplespace with our time stamp, we only get the
;; new data, since it was added since our last match.

>  (vcopy '(> @@) :test-time ts)
(("mammal" "fox"))


;; again the time stamp has been modified in place to reflect a new matching.
;; here we inspect the time stamp only to show that is has been changed.  the
;; actual value in the time stamp is irrelevant - it used internally by nancy.
```

```
> ts
1.82169e-44
```

notice that above we replaced both the data and the tag ("mammal" "fox") in the grouplespace.  suppose that we want to replace only the data in the grouplespace.

```
> (vput "carrot" '((@ > "fern") **))
("fern")

> (vcopy '(> @@) :test-time ts)
(("carrot"))

;; as expected, vcopy returns only the changed data.


;; above, the changed data is ambiguous without its tag.  but the tab was not
;; modified and so is not returned.  we can use the 'touch' feature during a
;; vput to mark elements in the grouplespace as having also been modified.

> (vput "palm" '((~ @ > "carrot") **))
("carrot")

;; note the ~ pattern modifier.  if the pattern successfully matches, nancy
;; 'touches' the element following the ~ in the pattern as having been modified.


> (vcopy '(> @@) :test-time ts)
(("plant" "palm"))

;; and this is exactly what we wanted.
;; unlike the > or ^ symbols, you can any number of ~ in a vput pattern.


;; it is important to remember that a single time stamp represents the most
;; recent match of a *particluar* pattern from a *particular* point of view.
;; for example, if you're writing program for a server entity to which many
;; entities make matching requests, the server should maintain a time stamp for
;; each entity and each pattern type.  see the fx.lsp file of the Fern System
;; for examples of this strategy.
```

## how to use the kernel

- since all behavior description is specified in lisp, (and thus all kernel primitive composition), lisp's native evaluation mechanism is best suited to direct flow control.

- it is an explicit goal of veos to provide program distribution services to generic programming frameworks. that is, the language we supplant above veos must be independent of the kernel implementation. were process control a kernel resposibility, the kernel would require language-specific maintenance and would directly oppose the stated fundamental premise.

- the mechanism in lisp for handling entity process management is available. it consists of a main processing loop as described below. each pass through the loop can be thought of as a 'frame'. that is, one complete cycle through all an entity's responsibilities constitutes a quanta of entity activity.

  1. invoke kernel maintenance.

     this entails polling network, garbage collection, etc..

  2. gather waiting inter-entity messages.

     this process can be augmented by filtering. In VEOS 2.0, this filtering feature has been disabled for performance reasons.

  3. evaluate messages from (2).

     implemented with lisp's eval.

     eval directs program control to the 'program stubs' contained within the messages. thus, messages are single lisp expressions. for example, the message:

     ```
          (add-to-database (frog x1 y1) (bird x2 y2))
     ```

     is a program stub that when evaluated by lisp will invoke the function 'add-to-database' with the arguments (frog x1 y1) and (bird x2 y2).

4. give processing burst to background tasks.

remember that each entity is a single unix process. and in general, a process is the smallest grain of parallelism that unix supports. consequently, there is no explicit provision in veos for concurrency (e.g. lightweight threads) *within* an entity.

but sometimes it is conceptually easier to design a program with concurrency in mind. this desire for parallelism can be fulfilled with virtual processes.

given this regime of processing 'frames', programmers may simulate concurrent processes with discrete background tasks. these tasks can be implemented as functions (program stubs) which perform a frame of computation and quickly return.

the process table is a lisp association list of program stubs. program stubs are small fragments of lisp code that can compose an entity's behavior program stubs should never block.

- an entity's main processing loop can be expressed in lisp by:

```
(defun go ()
      (loop
        (do-frame)))

      (defun do-frame ()
(progn

        ;; kernel maintenance
  (vtask)

        ;; gather and evaluate waiting messages
        (do ((msg (vcatch)))
            ((null msg))

            (eval msg)
            (setq msg (vcatch))
            )

        ;; give processing burst to background processes
        (dolist (proc background-procs)
            (eval (cadr proc)))
        ))
```

- since VEOS is intended as a prototyping system, the design is based less on performance considerations and more on reliable functionality. you will notice that the above scheme ignores many real-time issues.

  for example, messages may arrive while an the entity is servicing background processes. thus, messages must wait an average of half the time of a processing frame before being evaluated. thus, when an entity is heavily loaded, there may be significant latency in message passing. this can be troublesome when attempting to sychronize entities though message passing.

  likewise, important work may be waiting in the backgound processes while the entity is servicing messages. this can be troublesome when attempting to ensure reliable interface device update rates.

  these problems can be lessened through judicious use of inter-entity messages (i.e. as few and small as possible) and through efficient and sparing use of background tasks.

- The Fern System is a good example of how to use VEOS from lisp. It is included with the VEOS distribution.

**********

Writing Software Tools in C

**********

## == Philosophy ==

Any software tool can be integrated with VEOS. However, certain features of software tools make them easier to integrate. The process of VEOSifying a software tool is writing custom lisp primitives (usually in C) which decode lisp arguments and pass control to the software tool.

Given that all control will be passed to a tool via lisp primitives, the VEOSified tool will perform best if it is:

non-blocking:
    That is, primitives should always return after a reasonable slice of time (depends on application).

interrupt-driven is ok:
    As long as primitive entry points appear to accomplish their tasks quickly (use static caches, etc..). In fact, for polling type software tools (input device drivers, etc..), interrupts are preferrable to timeout reads in some situations. Especially when data arrival is infrequent.

modular:
    The tool should perform distinct, well-defined tasks. This assists the lisp programmer with task specification. And it makes writing custom primitives easy.

robust:
    Eg, can handle lags in data flow, can handle out-of-band data, etc..
    Remember, world designers will want to rely on these distributed services.

## == Building a Custom Entity with Lisp Interface ==

The xlisp that is bound with with VEOS is a software library that anybody can use. You can compile the xlisp library once, then link in user-defined lisp primitives at runtime. Or, you can recompile xlisp by declaring user-defined primitives in advance as the xlisp documentation describes. This section only describes the dynamic scheme for binding xlisp primitives.

To use this version of xlisp, you need to provide two functions that xlisp expects.

The first function ( xlinclude_hybrid_prims() ) supplies xlisp with the function pointers of your user-defined primitives (also called hybrid primitives). If you're not binding any user-defined primitives (ie. making basic xlisp w/o VEOS), this function should have an empty body. See file /home/xlisp/xexec/c/main.c for an example of the bare-bones main.

The second function ( xlhybrid_shutdown() ) gives user defined modules a chance to clean their respective environments become xlisp exits. xlisp will call this function just before a graceful exit. Again, if no special actions are to be taken, give this function an empty body.

Lastly to use this version of xlisp, you need to provide a main() function. In your main(), do any setup that is required, then call xlisp's main ( xmain() ). xmain() will never return. Notice that VEOS does not do any setup in main() (see example file below). Instead, kernel setup is a lisp primitive. This practice is encouraged since it allows users to decide when to use particular modules.

What follows are the step-by-step instructions to build a VEOS entity with xlisp and and other custom software tools:

1. Create a main control file that minimally looks like this:

```
#include "world.h"
extern xmain();

/*****************************************************************************
 * main
 * launchpad of any stand-alone program
 *****************************************************************************/
main(argc, argv)
    int        argc;
    char *argv[];
{
    /** call the xlisp controller, never returns **/
    xmain(argc, argv);
    }

/*****************************************************************************
 * xlinclude_hybrid_prims
 * lisp calls this function to load user-defined lisp primitives
 *****************************************************************************/
xlinclude_hybrid_prims()
{
    /** load veos native lisp primitive entries.
     ** this function lives in the xvnative_glue library
     **/
```

```
    Shell_LoadNativePrims();


    /** here, make calls to other software tool setup functions **/
    MyModule_LoadMyPrims();
    }


/************************************************************************
 * xlshutdown_hybrid
 * lisp calls this function before graceful exit
 ************************************************************************/
xlshutdown_hybrid()
{
    /** let the kernel unwind
     ** this function lives in the kernel library
     **/
    Kernel_Shutdown();

    /** here, make calls to other software tools shutdown code **/
    }
```

2. Create a file that contains your primitives. Write C functions which handle lisp arguments. These functions are the glue between your software and lisp (ie. your primitives). See the kernel interface files, xv_native.c and xv_glutils.c, for model code. You may use any of the utilities found in these files. The functions themselves are compiled into the xvnative_glue library.

   In this file, provide xlisp with the function pointers to your primitives. In step 1, the function xlinclude_hybrid() loads all the user-defined prims. In addition to loading the native kernel primitives, it should call a function like this to load your primitives into lisp:

```
#include "xlisp.h"

/************************************************************************
 * MyModule_LoadMyPrims
 * load user-defined lisp primitives
 ************************************************************************/
```

```
TVeosErr MyModule_LoadMyPrims()
{
    DEFINE_SUBR( "ROLLOVER", Tricks1_RollOver )
    DEFINE_SUBR( "PLAYDEAD", Tricks2_PlayDead )
    DEFINE_SUBR( "FETCH", Tricks3_Fetch )
    DEFINE_SUBR( "GONUTS", Tricks4_GoNuts )

    DEFINE_FSUBR( "REPEAT", Tricks5_Repeat )

    /** NOTES:
     **
     ** No semicolon is necessary after these macros.
     ** This is to preserve compatibility with another xlisp
     ** mechanism for including user-defined functions.
     **
     ** First arg is name you want prim to be in lisp, use all CAPS.
     ** Second arg is actual C function, must return an LVAL.
     ** Use DEFINE_FSUBR for lisp special forms.
     ** Most likely you wont need them.
     **/

    return(VEOS_SUCCESS);
    }
/*****************************************************************************/
```

3. Compile your main and primitives and link with the options:

   ```
   -L/home/veos/lib/
   -lxlisp -lxvnative_glue -lkernel
   ```

   See VEOS kernel makefile for example build sequence.
   Don't forget to link to the software packages which contain included
   primitives.  Notice that the kernel and the kernel interface to lisp are
   separate libraries.

4. Now you have a stand-alone entity.  This entity should behave just like the
   base-level entity for application-nonspecific tasks, ie. standard lisp.