AVRIL Technical Reference -- Appendices

Version 2.0
March 28, 1995

Bernie Roehl

Note: These are the appendices for the technical reference manual; the manual itself is kept in a separate file, as is the Tutorial.

Appendix A - REVISION HISTORY

This is the second major release of AVRIL. The changes made since the first pre-release (0.9c) are listed below.

Changes made in version 2.0:

Added support for a number of new devices.

The vrl_DevicePollAll() now returns non-zero if any of the devices had new data.

Added support for stereoscopic rendering.

Added support for Gouraud shading, including the vrl_RepComputeVertexNormals() function for computing vertex normals as the average of polygon normals.

Standardized the display (i.e. scan-converter) and video (frame buffer) interfaces.

Modified vrl_Palette type; it's no longer an array, it's now a struct containing additional information about the palette (specifically, the huemap).

Provided vrl_malloc(), vrl_calloc() and vrl_free() functions as "wrappers" around the system functions.

Defined a "raster" type for bitmaps.

The vrl_SurfaceInit() function no longer takes a *hue* parameter.

The vrl_Surfacemap type is no longer just an array; it's a struct containing additional information. New functions have been added for accessing that information.

Surfaces are now kept in a list.

Surfacemaps are now kept in a list.

Replaced vrl_ObjectGetX() and its sister functions with vrl_ObjectGetWorldX() and vrl_ObjectGetRelativeX() and company; the first returns absolute world coordinates, the second returns coordinates relative to the object's parent. Did the same for vrl_ObjectGetLocation().

Added a vrl_TransformVertexToScreen() function, to convert a vertex from object space to screen space.

All the file-loading functions now use the loadpath.

Renamed some of the reading and loading functions; the "Read" type functions now take a FILE *, the "Load" type functions now take a filename.

Fixed typo that caused the vrl_ShapeRescale() function to be undefined.

Modified joystick driver to not zero out buttons and nbuttons when a buttonmap is being used.

Changes made in version 1.1:

Added support for the Polhemus Isotrak.

Added the functions vrl_ObjectLookAt(), vrl_CameraLookAt() and vrl_LightLookAt().

Simplified the code in cfg.c; as several people pointed out, it was a complicated solution to a simple problem. Devices now have "nicknames" by which they can be referenced by the application, eliminating the need for the cfg.c code to keep track of that information. The configuration file is read *after* (not before) vrl_SystemStartup() is called, and a "display" statement will simply re-initialize the display to use the new driver and/or mode.

Fixed a bug in the horizon routine that had caused sky to be displayed when looking straight down, and ground to be displayed when looking straight up.

Changes made in version 0.9c:

Added support for multi-channel input devices.

Added support for serial communications.

Renamed all the types to begin with the "vrl_" prefix; the types affected are Scalar, Factor, Angle, Vector and Matrix. Also changed UNITY to VRL_UNITY. There are #defines at the end of avril.h to ease the transition; they'll be removed for version 2.00.

Added vrl_Boolean and vrl_Time types.

Modified avril.h to #include <stdio.h> and #include <mem.h> (for memcpy()).

Added #defines for XROT, YROT and ZROT for indexing device channels.

Added various additional vector and matrix functions such as vrl_VectorNegate() and vrl_VectorEqual(). Also added a new global variable, the null vector vrl_VectorNULL.

Added a "leftside" parameter to the vrl_MatrixRotX(), vrl_MatrixRotY(), vrl_MatrixRotZ(), and vrl_MatrixRotVector() functions.

Removed the vrl_List structure and put names into the structs for vrl_Lights, vrl_Cameras, and vrl_Objects. Added functions for accessing those names, and for finding entities based on their name. Also added routines for traversing and iterating over the linked lists of vrl_Lights and vrl_Cameras.

Added vrl_ObjectRotate() and vrl_ObjectTranslate() routines, and modified several older movement and rotation functions to make calls to those two.

Made surface maps into a struct, rather than just an array of vrl_Surface pointers. This allows for additional information about a surface map to be kept, and for the surface maps to be kept in a linked list.

Added the notions of a world having "bounds" and a "radius", and routines for supporting those notions.

The RVD drivers are now searched for along the PATH, as well as in the current directory.

Renamed all the vrl_New* functions to make their naming consistent with everything else. For example, vrl_NewObject() is now vrl_ObjectCreate(). Again, #defines were added to the end of avril.h to ease the transition; these will be removed as of the 2.00 release.

Added a vrl_WorldUpdate() macro.

Added application-specific data to vrl_Objects, vrl_Lights and vrl_Cameras.

Added functions to vrl_Objects, which get called during world updating.

Added ID numbers to facets.

Renamed vrl_SetFigurePartArray() to vrl_SetReadFIGpartArray() to be more consistent.

Switched to using standard the DOS timer (at an accelerated rate, and chaining to the old one periodically) because the RTC timer interfered with Turbo Debugger.

Standardized on a tick rate of 1000 per second.

Eliminated the vrl_TimerAddHandler() routine, since there was too much risk that handler routines would do things that ought not be done inside an interrupt handler; a bug in Borland C 3.1 bug also forced all routines called from within an interrupt to be assembled without 386 instructions.

Added a call to directly query the frames per second rate.

Renamed the vrl_DrawCompass() and vrl_DropText() routines to vrl_UserInterfaceDrawCompass() and vrl_UserInterfaceDropText() to make it clear that they're a part of the user interface family of functions.

Modified the vrl_PrimitiveSphere() routine to accept separate counts of the number of longitudinal and latitudinal sides.

Add support for configuration files.

Added the additional mouse routines vrl_MouseGetUsage(), vrl_MouseSetUsage(), vrl_MouseSetPointer(), vrl_MouseGetPointer() and vrl_MouseGetLimits().

Appendix B - CFG FILE FORMAT

A CFG file is a platform-specific ascii file used to describe the user's preferred configuration. It contains a series of statements, one per line; anything after a '#' character on any given line is taken as a comment, and blank lines are ignored. At the moment, there are only a few statement types defined; this is expected to change.

COMPASS *state*
: If *state* is "on", then the compass should be displayed on the screen.

FRAMERATE *state*
: If *state* is "on", then the frame rate should be displayed on the screen.

POSITION *state*
: If *state* is "on", then the current X,Z position should be displayed on the screen.

CURSOR *state*
: If *state* is "on", then the cursor should be visible; this is the default state.

DEVICE *name type mode address irq buffsize*
: Sets up a device driver, and assigns it a symbolic name that can be referenced by the application. The *type* field is the name of a device type, like "Cyberman" or "Spaceball". The *mode* is passed to the device using vrl_DeviceSetMode(), and the *address*, *irq* and *buffsize* fields are just passed to the call to vrl_SerialOpen(); see the section on Serial Ports for details. All parameters except the *name* and *type* are optional; devices that are not interfaced over a serial port do not need the *address*, *irq* or *buffsize* parameters, and devices all have a default *mode*. The *name* can be anything you like; however, it should be something that's referenced by the application. For example,

```
device headtracker Redbaron 0 0x2F8 3 2000
```

would set up the Logitech Red Baron ultrasonic tracking device; the device would be hooked up to COM2 (address = 0x2F8, irq = 3) with a 2000 byte buffer. The application would simply look up the device called "headtracker" and use the values it returns without having to worry about what kind of device it actually is.

DEVCONFIG *name channel scale deadzone*
: Sets the scale and deadzone parameters for a particular channel of a particular device. The *name* must match the name of a device already specified with a DEVICE statement. The *channel* can be either a number or one of the special values X, Y, Z, XROT, YROT or ZROT. The *scale* can be either a number (in which case it's taken to be a scalar distance) or a number with an 'a' or 'A' in front of it (in which case it's

taken to be an angle). The *deadzone* value is a number, specified in device coordinates. The *deadzone* is optional, and defaults to zero.

For example,

devconfig headtracker 2 15

would cause channel 2 (the Z channel) to have a scale factor of 15 and a deadzone of zero, while

devconfig headtracker YROT a45 10

would cause channel 4 (the Y rotation channel) to have an angular scale factor of 45 degrees and a deadzone of 10 device units.

DISPLAYDRIVER *name*
Specifies which display driver to use. Currently the only values are "ModeY" or "default". See also the description for VIDEODRIVER, and the technical reference manual.

INCLUDE *file*
Includes the contents of the specified file as if they appeared in the current file in place of the INCLUDE.

LOADPATH *path*
Specifies the path from which all subsequent files should be loaded. The path is effectively prepended to any filename that does not begin with a '/' or '\'.

STEREOTYPE *type*
Specifies which type of stereoscopic viewing to use. The *type* can be any of the following:
NONE, SEQUENTIAL, ANAGLYPH_SEQUENTIAL, ANAGLYPH_WIRE_ALTERNATE, ENIGMA, FRESNEL, CYBERSCOPE, CRYSTALEYES, CHROMADEPTH, SIRDS, TWOCARDS, ANAGLYPH_SOLID_ALTERNATE
See Appendix I for descriptions of each of the types; note that not all of them are implemented yet in version 2.0 of AVRIL.

STEREOPARAMS *params*
Sets parameters for the stereo viewing mode. The *params* will differ from one type of stereo viewing to another. For CHROMADEPTH, the parameters are the ChromaNear and ChromaFar distances, in world units. For all other currently supported modes, the parameters are the eyespacing and (optionally) the convergence distance, both as floating-point numbers. They must be in the same units (e.g. millimeters).

STEREOLEFT *shift* [*rotation*]
STEREORIGHT *shift* [*rotation*]

Specifies the shift and (optionally) the rotation for the left or right eye. See the section on stereoscopic viewing in the technical reference manual for details.

VERSION *nn*

Indicates which version of this specification the configuration conforms to. This can be omitted; if present, it should be set to 1.

VIDEODRIVER *name* [*mode*]

Specifies the name of the low-level video driver to use; currently the only values are "Mode13", "ModeY" and "7thSense". The *mode* indicates the graphics sub-mode to use; it's passed directly to the driver function, in the initialization call. See also the description for DISPLAYDRIVER, and the technical reference manual.

Note the difference between the VIDEODRIVER and the DISPLAYDRIVER. The VIDEODRIVER deals with the hardware, the DISPLAYDRIVER with the scan-conversion module; see the technical reference manual for details. You can, for example, have the following:

```
videodriver mode13
displaydriver default
```

in which case mode 0x13 will be used throughout, or

```
videodriver modeY
displaydriver modeY
```

in which case mode Y will be used throughout. Alternatively, you can mix them:

```
videodriver modeY
displaydriver default
```

in which case rendering will be done to an offscreen buffer, and the completed screen image copied to the current draw page in mode Y.

Appendix C - PLG FILE FORMAT

I originally designed PLG files for use with REND386; for better or worse, they seem to have become something of a standard. REND386, AVRIL, VR386 and Jon Blossom's Gossamer all use them for object geometry description; there are also translators that turn other formats into PLG, and the NorthCAD-3D program can generate PLG files as output. The PLG in the name stands for "polygon".

There will soon be a file format for the interchange of virtual objects and virtual worlds between VR systems; at that point, support for the PLG file format will diminish. Conversion programs will be made available to convert PLG files to the new format.

A PLG file basically has three parts: a header line, a list of vertices and a list of facets.

The header line has the object name, the number of vertices, and the number of facets; for example:

```
kitchen_table 100 35
```

which would mean that the object "kitchen_table" has 100 vertices and 35 facets.

The number of facets can optinally be followed by two numbers; the first is ignored, and should be set to zero (it's for compatability with VR386), and the second is the sorting type. It should be set to zero if (and only if) you know the object is convex (i.e. no part of the object can hide any other part). Anything after those two numbers should be ignored, since it may be used for future expansion.

Following this line are the vertices, one x,y,z triplet per line (each value is a floating-point number, and they're separated by spaces). For example:

```
18027 23025 98703
```

Vertex normals may optionally be specified after the vertex coordinates. Anything after the vertex normals is ignored (for future expansion).

This is followed by the facet information, one line per facet; each of these lines is of the form

```
surfacedesc n v1 v2 v3 ...
```

The *surfacedesc* is described below. The *n* is the number of vertices in the facet. The *v1*, *v2*, *v3* and so on are indices into the array of vertices; the vertices are listed in a counterclockwise order as seen from the "front" (i.e. visible side) of the facet. Note that the vertices are counted "origin zero", i.e. the first vertex is vertex number 0, not vertex number 1.

For example:

```
0x8002 4 8 27 5 12
```

would mean a four-sided facet bounded by vertices 8, 27, 5 and 12. This facet has a surface descriptor of 0x8002.

Anything after the list of vertex indices should be ignored.

The PLG format supports comments. Anything after a # should be ignored by any program that parses PLG files. In addition, lines beginning with a '*' should be ignored.

PLG files can have multiple copies of an object at different resolutions. PLG files containing such multiple-resolution versions of objects must have "#MULTI" as their first line.

For each object defined in such a file, the object name includes a number specifying the pixel size of the object on the screen. The object names for each representation must be

```
<name>_####
```

where #### is the smallest pixel width to use this representation for. For example, TABLE_15 would be a valid name.

If the smallest rep size is zero, then that representation will be used no matter how small the object gets. If the smallest rep size is 1 or greater, then the object will vanish if it gets too small.

The surface descriptor can either be a decimal integer or a 0x or 0X followed by a hexadecimal integer value. The surface descriptor is a 16-bit value which is interpreted as follows:

```
H SSS CCCC BBBBBBBB
```

If the H bit is set, it indicates that this is a "mapped" surface descriptor; the bottom 14 bits are taken to be an index into a surface map.

If the H bit is clear, the SSS bits are interpreted as follows:

000 -- This facet is "solid shaded"; i.e. it should be drawn in a fixed color, with no special effects. If the CCCC bits are zero, then the BBBBBBBB bits directly specify one of the 256 available colors; if the CCCC bits are non-zero, then

they specify one of sixteen hues and the top four bits of BBBBBBBB specify which shade of that hue to use.

001 -- This facet is "flat shaded"; i.e. it should be drawn with a constant shading that is determined by the angle at which light is striking it; thus, as the facet moves around, its apparent brightness will change. The CCCC bits specify one of sixteen hues, and the bottom 8 bits BBBBBBBB represent the "brightness" of the color. This brightness value is multiplied by the cosine of the angle between the facet's normal vector and the vector from the facet to the light source; the result is used to specify an offset into the given color's array of shades. Note that if the CCCC value is 0, the color will always be black.

010 -- This facet should be treated as being "metallic"; the CCCC bits (which should be non-zero) specify one of the 16 hues, and the top 5 bits of the BBBBBBBB value are used as an offset into a range of shades to cycle through to give the metallic effect, i.e. a starting offset into the color cycle.

011 -- This facet should be treated as being "transparent"; it is just like surface type 10, except that alternating rows of dots are used instead of solid colors, allowing you to "see through" the facet.

100 -- This facet should be Gouraud shaded; the meaning of the other bits is the same as for flat shading.

Other values of SSS are reserved and should not be used.

Appendix D - FIG FILE FORMAT

FIG files are a way of representing multi-segmented, hierarchical entities.

This format will soon be considered obsolete.

There will soon be a file format for the interchange of virtual objects and virtual worlds between VR systems; at that point, support for the FIG file format will diminish. Conversion programs will be made available to convert FIG files to the new format.

The syntax of a figure file is simple, and very C-like. It consists of a series of segments, each of which can possess a set of attributes, as well as child segments. Each segment is bounded by braces. Attributes are arbitrary text strings ending in a semicolon.

The list of possible attributes is open-ended and extensible; programs that read figure files should ignore any attributes they don't recognize.

An example will make all this clearer.

```
{
comment = a human body;
name = pelvis;  comment = this is the name of the root segment;
        {
        name = chest;
                { name = left upper arm; { name = left lower arm; } }
                { name = right upper arm; { name = right lower arm; } }
                { name = head; }
        }
        { name = left upper leg; { name = right lower leg; } }
        { name = right upper leg; { name = right lower leg; } }
        }
}
```

In general, attributes are of the form "keyword = value;", though this is not a requirement. The attributes used above are *name* and *comment*. Note that no program ever has to recognize a comment attribute, since by definition comments should be ignored.

The attributes currently defined are as follows:

```
name = somestring;
pos = x,y,z;
rot = x,y,z;
plgfile = filename scale x,y,z shift X,Y,Z sort type map filename;
segnum = someinteger;
```

The *pos* is the x,y,z displacement of the origin of this segment relative to the parent's coordinate system. The *rot* is the rotation of this segment relative to the parent. For root objects (which have no parent) these values are the absolute location and rotation of the entire figure in world coordinates.

The *plgfile* gives the name of a .plg file containing a geometric representation of the segment. Note that the figure file format does not strictly depend on .plg files; the reason the syntax is "plgfile = " rather than just "file =" is because a segment may have a large number of different representations and an application can choose whichever one it likes.

The *scale*, *shift*, *sort* and *map* values are all optional, but in order to specify any of them you must specify all the preceding ones (i.e. you cannot simply omit the *scale* parameter). The *scale* values represent the amount by which the object should be scaled along each of its axes when it's loaded. The *shift* value is the amount by which to shift the object's origin at load time. The *sort* value is the type of depth-sorting to use for this segment's representation (the default is zero). The *map* value is the name of a file containing a list of unsigned values that are to be used in surface remapping for this segment. If the top bit of a color value is set in a plg file, the bottom fourteen bits are used as an index into this map.

The difference between *shift* and *pos* is important. The *shift* value is used to shift an object relative to its "native" origin, while the *pos* value is the amount by which the new origin should be displaced from the parent node's origin.

For example, suppose you want to represent the head of a human figure with a cube. The cube may, in the .plg file, be defined with its (0,0,0) point at one corner. Clearly, this origin is inconvenient for the head, since if the origin is centered over the neck of the figure then the head will be displaced to one side.

Alternatively, the cube might be defined with its (0,0,0) point at its geometric center. However, this is also impractical; your head should not rotate freely about its center. If it does, stop reading this document immediately and seek medical attention.

What you do is shift the cube so that its origin lies below the center of the cube, where your "neck joint" is. That's what the *shift* value in the *plgfile* attribute specifies.

Important note: objects rotate about their [0,0,0] point as loaded.

The *pos* attribute specifies where this neck joint is in relation to the origin of the chest segment. If your chest were longer vertically, then the *pos* attribute of the head segment should be increased in the Y direction (for example).

The *segnum* attribute associates a simple integer value with a segment, which can subsequently be used to refer to the segment when manipulating it.

Note that a figure file can in fact contain a series of segments; each of these is a root segment, so a figure file can in effect store a complete scene description (excluding lights and cameras).

Appendix E - WLD FILE FORMAT

WLD files were designed to store information about the layout of objects in a virtual world.

This format will soon be considered obsolete.

There will soon be a file format for the interchange of virtual objects and virtual worlds between VR systems; at that point, support for the WLD file format will diminish. Conversion programs will be made available to convert WLD files to the new format.

A WLD file is entirely ascii. Each statement is one line; anything after the first '#' is treated as a comment and ignored. Blank lines are also ignored. The format is intended to be highly extensible; any line which cannot be recognized should simply be ignored. Each statement contains some information about the scene; the possible types of statements are listed below. Everything is case-insensitive; keywords are shown below in uppercase, but are generally entered in lowercase.

LOADPATH *path*

> Specifies a path prefix for loading files. Any files (whether specified in the world file itself, subsequent world files, or in referenced FIG files) will be loaded from the specified directory. However, if a filename begins with the '\' or '/' characters, it is used verbatim (i.e. the LOADPATH setting is ignored).

PALETTE *filename*

> Loads a 256-entry binary palette file (3 bytes (R,G,B) for each entry). Note that alternate palettes may not handle shading as well as the default one does. If there are more that 768 bytes (256 times 3) in the file, the remaining values are interpreted as a hue map; pairs of bytes are starting indices (origin 0) and number of shades.

SKYCOLOR *index*

> Specifies which of the 256 available colors should be used for the "sky".

GROUNDCOLOR *index*

> Specifies which of the 256 available colors should be used for the "ground". If the sky and ground color are identical, a solid screen clear is used; this is a bit faster.

SCREENCLEAR *value*

> If the specified value is non-zero, then the screen will be cleared before each frame; if it's zero, the screen clearing is not done (this is useful if you know that the entire window will be covered by the image, and that no background will show through; in such a situation, specifying this option will improve performance).

AMBIENT *value*

Specifies the level of the ambient light; 76 is the default, and a good value to use.

LIGHT *x,y,z*

Specifies the location of a light source in world coordinates.

CAMERA *x,y,z tilt,pan,roll zoom*

Specifies your starting location, viewing direction and zoom factor. The *x,y,z* values are floating-point numbers giving coordinates, the *tilt,pan,roll* values are floating-point angles, and the *zoom* is a floating-point number giving the zoom factor. Remember that the order of rotations is pan, tilt, roll.

HITHER *value*

Specifies the near clipping distance in world coordinates. The value should typically be 10 or more.

YON *value*

Specifies the far clipping distance in world coordinates. The value should typically be 1000000 or more.

OBJECT [*objname*=]*filename sx,sy,sz rx,ry,rz tx,ty,tz depthtype mappings parent*

Loads an object from a .plg file with the given *filename*. If the *objname*= is present, it assigns the newly-loaded object that name for future reference. The *sx,sy,sz* values are floating-point scale factors to increase or decrease the size of the object as it's loaded. The *rx,ry,rz* values are the angles to rotate the object around in each of the three axes; *ry* is done first, then *rx* and finally *rz*. The *tx,ty,tz* values translate (move) the object to a new location; this is done after the scaling and rotation. The *depthtype* field is not currently used. The *mappings* feature is explained below. The *parent* field is the name of the object that this object is attached to; if omitted, the child moves independently. If *parent* is the word "fixed", then the object is fixed in space and cannot be moved. All fields are optional, but you must include all the fields prior to the last one you wish to use (i.e. you can only leave things off the end, not off the beginning or out of the middle).

FIGURE [*figname*=]*filename sx,sy,sz rx,ry,rz tx,ty,tz parent*

Loads a segmented figure from a FIG file with the given filename. All the parameters have the same meaning as for the OBJECT statement described above.

POLYOBJ *npts surface x1,y1,z1 x2,y2,z2* [...] *x8,y8,z8*

Directly specifies a facet to be placed in the scene. The value *npts* is the number of points (maximum 8), the *surface* is a surface name (see below on surfaces) and the vertices are given in world coordinates.

POLYOBJ2 *npts surface1,surface2 x1,y1,z1 x2,y2,z2* [...] *x8,y8,z8*

Directly specifies a double-sided facet to be placed in the scene. The value *npts* is the number of points (maximum 8), *surface1* and *surface2* are surface names (see below on surfaces) and the vertices are given in world coordinates.

INCLUDE *filename*

Includes the specified file as if its contents appeared at the current point in the current file.

POSITION *objname x,y,z*

Moves (i.e. translates) the specified object to the given *x,y,z* location.

ROTATE *objname rx,ry,rz*

Rotates the specified object to the given angles about each of the axes. The angles are specified in floating point, and are measured in degrees. The rotation order is Y then X then Z.

VERSION *number*

Allows you to define a version number. Not currently used for anything; can be omitted.

TITLE *text*

Allows you to define a title for your world.

About Mapping

A PLG file can contain indexed color values (such as 0x8002) which are used to index a surface map. Entries in surface maps refer to surfaces. Surfaces are created using the SURFACEDEF statement, surface maps are created with the SURFACEMAP statement, and entries are placed in them with the SURFACE statement. The statement formats are as follows:

SURFACEDEF *name value*

Defines a new surface; maps a surface name (such as "wood") to a numeric surface descriptor (*value*) of the type described in Appendix C.

SURFACEMAP *name maxentries*

Marks the start of a new surface map. All subsequent SURFACE entries will be placed in this map. The *maxentries* field gives the maximum number of entries this surface map will have; if omitted, it defaults to 10.

SURFACE *index name*

Defines an entry in the current surface map, which takes an *index* value (the bottom 14 bits of the value in the .plg file) and maps it into a surface *name* (which is in turn mapped to a 16-bit color value).

USEMAP *mapname*

Causes all subsequently loaded objects that don't have a mapname on their OBJECT statements to use the specified *mapname*.

Appendix F - WRITING DEVICE DRIVERS

Writing device drivers for AVRIL is easy. You basically create a single function with a unique name; for example, if you want to support a (mythical) RealTronics Atomic Tracking System, your function might be

```
int vrl_ATSDevice(vrl_DeviceCommand cmd, vrl_Device *device)
        {
        [...]
        }
```

You should add an entry for your new function to the list in avrildrv.h, and possibly to the cfg.c file.

Your driver routine will get called periodically by the application. The vrl_Device struct is pre-allocated by AVRIL, so you just have to fill in the various fields. The *cmd* is one of VRL_DEVICE_INIT, VRL_DEVICE_RESET, VRL_DEVICE_SET_RANGE, VRL_DEVICE_POLL, or VRL_DEVICE_QUIT.

When a device is first opened, AVRIL will set all the fields in the vrl_Device struct to reasonable values. The VRL_DEVICE_INIT call should fill in the following fields with driver-specific information:

```
char *desc;                  /* user-readable device description */
int nchannels;               /* number of input channels the device has */
vrl_DeviceChannel *channels;   /* pointer to array of channels */
```

The *desc* is a string describing the device, the *nchannels* value is the number of input channels the device has (should be at least 6) and the *channels* field is set to point to an array of vrl_DeviceChannel structs, one per channel. These channels should be dynamically allocated, rather than using a static struct; this is to allow for multiple instances of the same type of device (for example, a Cyberman on each of COM1 and COM2, each with its own channel-specific data). For this same reason, your driver shouldn't use any global variables; you should instead dynamically allocate memory for any additional per-device-instance data and store the pointer to that data in the *localdata* field of the vrl_Device struct. The VRL_DEVICE_INIT call should also fill in the appropriate values for all the channels.

The VRL_DEVICE_INIT call may also choose to fill in some or all of the following:

```
int nbuttons;                /* number of buttons the device has */
int noutput_channels;        /* number of output channels */
vrl_DeviceOutputFunction *outfunc;  /* function to call to generate output */
vrl_DeviceMotionMode rotation_mode;     /* rotation mode for this device */
vrl_DeviceMotionMode translation_mode;  /* translation mode for this device */
vrl_Buttonmap *buttonmap;    /* button mapping table for 2D devices */
int version;                 /* device struct version number */
int mode;                    /* mode of operation */
vrl_Time period;             /* milliseconds between reads */
```

The number of buttons the device has is assumed to be zero unless you set it otherwise, as is the number of output channels. The *outfunc* field is a pointer to a function (probably declared static in the same source file as your driver function) that handles output to the device; this is described in more detail below. If your device doesn't do output, leave this field at its default value of NULL.

The meaning of the two vrl_DeviceMotionMode type fields is described in the main part of the technical reference manual, in the section on Devices. They both default to VRL_MOTION_RELATIVE. The *mode* is driver-specific, and can be initialized to whatever value you like (since the value is only interpreted by your driver). The *version* field should be left at its default value of zero by drivers following this version of the driver specification; as the driver specification evolves, this value will increase.

The *period* defaults to zero, meaning that a call to vrl_DevicePoll() will always result in your driver function being called with a *cmd* of VRL_DEVICE_POLL. If you don't want to be polled every cycle, set the *period* to the minimum number of ticks (usually milliseconds) that should elapse between polls. Note that this is a *minimum* value; the delay between polls may be even longer if the system is busy doing other things.

The VRL_DEVICE_RESET command is very similar to VRL_DEVICE_INIT, and may in fact be the same for some devices. The difference is that VRL_DEVICE_INIT does one-time initializations (such as taking over interrupt vectors).

On receiving the VRL_DEVICE_RESET command, the driver should note the current values of each channel and make them the centerpoint for that channel. On receiving a VRL_DEVICE_SET_RANGE command, it should make the current values the range for that channel. See the notes below about centerpoint and range.

The VRL_DEVICE_QUIT command should "shut down" the device, putting it in a quiescent state and undoing anything that was done in VRL_DEVICE_INIT and VRL_DEVICE_RESET (for example, restoring interrupt vectors). It's also responsible for releasing any memory that was dynamically allocated by VRL_DEVICE_INIT, including that pointed to by the *channels* field and the *localdata* field if it was used.

Serial devices can assume that when VRL_DEVICE_INIT is called, the port they'll be using is already open, and that the *port* field is set; the driver also does not need to (and should not) close the port. However, devices that actually use the *port* should check that it's not NULL, and return -4 if it is.

The VRL_DEVICE_POLL command should read the raw data from the hardware (for example, by calling vrl_DeviceGetPacket()) and decode the values into the *rawdata* fields of the appropriate channels. You should be sure to set the *changed* field for any channels that you update.

There are a number of values associated with each channel; they are as follows:

```
struct _vrl_device_channel
	{
	vrl_32bit centerpoint;        /* value of center point in raw device coords */
	vrl_32bit deadzone;           /* minimum acceptable value in device coords */
	vrl_32bit range;              /* maximum absolute value relative to zero */
	vrl_Scalar scale;             /* maximum returned value */
	vrl_Boolean accumulate : 1;   /* if set, accumulate values */
	vrl_Boolean changed : 1;      /* set if rawvalue has changed */
	vrl_32bit rawvalue;           /* current value in raw device coordinates */
	vrl_32bit oldvalue;           /* previous value in raw device coordinates */
	vrl_Scalar value;             /* current value (processed) */
	};
```

The only fields you must set are *centerpoint*, *deadzone*, *range*, *scale* and *accumulate*. The *centerpoint* is the current "zero" value of the device; for example, the value an analog joystick on a PC-compatible reports when the stick is at rest can be considered its centerpoint.

The *deadzone* has two different interpretations. If the *accumulate* flag is set, then the deadzone is the minimum displacement from the centerpoint that will be recognized. If the *accumulate* flag is clear, then the value is the minimum change from the previous value (as stored in *oldvalue* by the higher-level routines) that will be recognized.

The *scale* and *range* values are used to convert the *rawvalue* into units more suitable for the application. The *scale* is the number of world-space units corresponding to the *range* in device units. The *scale* and *deadzone* should both be positive values, and can be changed by the application. The *range* value can be negative; this is useful for reversing the direction of a device axis. The *range* value is only ever set by your driver.

The *accumulate* flag, in addition to controlling how the *deadzone* is interpreted, causes the value to be scaled by the elapsed time in seconds since the last poll.

The best way to understand all this is to consider what happens when you read new values from the device in response to a VRL_DEVICE_POLL command. First, you store the data for each channel in the corresponding channel's *rawvalue* field. You can re-map axes at this point (device coordinate Y goes into the Z channel, for example); you may want to use the buttonmap (see below) for this purpose. You should set the *changed* flag, to indicate there's a new value there.

Next, the higher-level code takes your *rawvalue* and subtracts the *centerpoint*. If the channel is in *accumulate* mode, it checks if the absolute value of the data is less than *deadzone*; if it is, it truncates it to zero. If the channel is not in *accumulate* mode, the data is compared to the *oldvalue* field; if it's within plus or minus *deadzone* of it, the data is ignored.

Once the value has been centered and deadzoned, it is multiplied by the *scale* and divided by the *range*. If *accumulate* is set, the resulting value is multiplied by the elapsed time in milliseconds and then divided by 1000 to convert to seconds.

Buttonmaps

Some 2D devices (such as mice and joysticks) can be used as 6D devices, by using their buttons to map their input axes to the 6 possible degrees of freedom. Such devices should set their *nbuttons* field to zero (or at least to the number of buttons that will not be used for mapping). They should also set their *buttonmap* field to point to a default set of axis mappings.

The *buttonmap* field is a pointer to a two-dimensional array. Each row of the array corresponds to a button combination; on a two-button device, row 0 is for no buttons down, row 1 is for the first button down, row 2 is for the second button down and row three is for both buttons down. There are two columns in the array, the first of which contains the index of the channel that the input device's X value should be stored in, and the second of which contains the index of the Y channel.

For example,

```
static vrl_Buttonmap default_map =
        { { YROT, Z }, { X, Y }, { ZROT, XROT }, { X, Y }};
```

Would mean that when no buttons are down, the device's X axis corresponds to a Y rotation, and its Y channel to a Z translation. When the first button is down, the device's X axis corresponds to an X translation, and its Y axis to a Y translation, and so on.

The application can change the *buttonmap* field (using vrl_DeviceSetButtonmap()) to point to a different set of mappings.

One thing to watch out for: since only two channels at a time are active, the others should have their *rawvalue* set equal to their *centerpoint*, and their *changed* flags set; otherwise, they'll retain whatever values they had the last time a particular button combination was active.

Output

Some devices are capable of tactile or auditory feedback; those that are should set the *outfunc* field in the vrl_Device struct to point to a function that does the actual work, and set the *noutput_channels* field to the number of output channels the device has. Such a function for our mythical ATS device might look like this:

```
int vrl_ATSOutput(vrl_Device *dev, int parm1, vrl_Scalar parm2)
        {
        [...]
        }
```

The *parm1* parameter is the output channel number, and *parm2* is the value to output (in the range 0 to 255). The routine should return 0 on success and non-zero on failure, although those values are not currently used or reported.

Appendix G - WRITING VIDEO DRIVERS

The low-level interface to the video hardware is handled by video drivers. Each video driver is simply a function of the form

```
vrl_32bit vrl_VideoDriverFunction(vrl_VideoCommand cmd, vrl_32bit lparm, void *pparm1);
```

The *lparm* is a 32 bit value, the *pparm* is a pointer to somewhere in system memory, and the return value of the function is a 32-bit value.

You can implement your own video driver by writing a function of the form above (giving it a unique name) and setting it as the driver function using

```
vrl_VideoSetDriver(YourDriver);
```

The application (and AVRIL itself) will call your function, passing it a *cmd* to tell it what to do. Any commands you don't want to deal with, you can ignore; however, be sure to return zero as the value of the function. For example, if you only have one physical page in your display adapter, you could ignore the VRL_VIDEO_SET_DRAW_PAGE and VRL_VIDEO_SET_VIEW_PAGE calls.

The values for *cmd*, and the behaviour your function should exhibit for each, are as follows:

VRL_VIDEO_GET_VERSION

Return the version number; for this specification, return 1.

VRL_VIDEO_GET_DESCRIPTION

The *pparm* parameter points to a buffer in system memory; fill that buffer with anything you like (preferably, something that describes your driver). The *lparm* parameter gives the size of the buffer; don't write past the end.

VRL_VIDEO_SETUP

Enter graphics mode; the *lparm* parameter indicates which submode to use. Return zero, unless for some reason you can't enter graphics mode (in which case, return a non-zero value). You should, if possible, save the current mode before entering graphics mode. You should also set the internal cursor flag to -1, and position the cursor at the center of the screen.

VRL_VIDEO_SHUTDOWN

Exit graphics mode, and if possible, return to the mode that was in effect before the last call to VRL_VIDEO_SETUP.

VRL_VIDEO_GET_MODE

Return the current graphics mode (zero is an acceptable value).

VRL_VIDEO_SET_DRAW_PAGE
Set the current drawing page to the value of *lparm*.

VRL_VIDEO_SET_VIEW_PAGE
Set the currently visible page to the value of *lparm*.

VRL_VIDEO_GET_NPAGES
Return the number of pages. If you only have one page, return 1.

VRL_VIDEO_HAS_PALETTE
Return non-zero if your hardware uses a palette, or zero if it doesn't (i.e., 15, 16 or 24 bit color).

VRL_VIDEO_SET_PALETTE
The top 16 bits of *lparm* are the starting index, the bottom 16 bits are the ending index, and *pparm* points to the entire 256-entry palette. Values from the starting index through the ending index (inclusive) should be copied from the palette in system memory to the physical palette. The starting and ending indexes are "origin zero", i.e. the first entry in the palette is zero, not one.

VRL_VIDEO_CHECK_RETRACE
Return non-zero if a vertical retrace is taking place.

VRL_VIDEO_GET_RASTER
Return a pointer to a vrl_Raster describing your display hardware. It's important that the height, width and depth fields are correct.

VRL_VIDEO_BLIT
Copy the contents of the vrl_Raster pointed to by *pparm* into the current draw page. If *pparm* points to our raster (i.e., the one that references the physical framebuffer) don't bother doing the copy.

VRL_VIDEO_CURSOR_HIDE
If the internal cursor flag is greater than or equal to zero, erase the cursor (by restoring what was under it). In any case, decrement the internal cursor flag.

VRL_VIDEO_CURSOR_SHOW
Increment the internal cursor flag; if it's equal to zero, draw the cursor (saving what was under it).

VRL_VIDEO_CURSOR_RESET
Center the cursor on the screen, and set the internal cursor flag to -1.

VRL_VIDEO_CURSOR_MOVE

The top 16 bits of *lparm* contain the new X coordinate, and the bottom 16 bits contain the new Y coordinate. Move the cursor to that location (restoring what was under it, and saving what's under the new location).

VRL_VIDEO_CURSOR_SET_APPEARANCE

Set the cursor appearance to the data pointed to by *pparm*.

You should add an entry for your new function to the list in avrildrv.h, and possibly to the cfg.c file.

If you wind up writing video drivers to support additional modes, drop me some email (broehl@sunee.uwaterloo.ca). I'll happily include your driver in the main AVRIL release, and give you full credit for having written your driver.

Appendix H - WRITING DISPLAY DRIVERS

The interface to the display driver (i.e. scan-converter) is a function of the form

```
vrl_32bit vrl_DisplayDriverFunction(vrl_DisplayCommand cmd, vrl_32bit lparm,
        void *pparm1);
```

The *lparm* is a 32 bit value, the *pparm* is a pointer to somewhere in system memory, and the return value of the function is a 32-bit value.

You can implement your own display driver by writing a function of the form above (giving it a unique name) and setting it as the driver function using

```
vrl_DisplaySetDriver(YourDriver);
```

The application (and AVRIL itself) will call your function, passing it a *cmd* to tell it what to do. Any commands you don't want to deal with, you can ignore; however, be sure to return zero as the value of the function. For example, if your driver doesn't do Z-buffering, you could ignore the VRL_DISPLAY_CLEAR_Z_BUFFER call. (Note that version 2.0 of AVRIL does not yet support Z-buffering, so for now you can just ignore all those calls).

The values for *cmd*, and the behaviour your function should exhibit for each, are as follows:

VRL_DISPLAY_GET_VERSION
: Return the version number; for this specification, return 1.

VRL_DISPLAY_GET_DESCRIPTION
: The *pparm* parameter points to a buffer in system memory; fill that buffer with anything you like (preferably, something that describes your driver). The *lparm* parameter gives the length of the buffer; don't write past the end.

VRL_DISPLAY_INIT
: Initialize the display subsystem. The *pparm* parameter points to a raster you should use.

VRL_DISPLAY_QUIT
: De-initialize the display subsystem.

VRL_DISPLAY_CLEAR
: Clear the display; *lparm* is the color to use.

VRL_DISPLAY_POINT
: The *pparm* parameter points to a linked list of output vertices; set those points on the current draw page.

VRL_DISPLAY_LINE

The *pparm* parameter points to a linked list of output vertices; they should be connected sequentially by a series of lines. The color of each line should be set from the starting vertex of the line.

VRL_DISPLAY_CLOSED_LINE

The *pparm* parameter points to a linked list of output vertices; they should be connected sequentially by a series of lines. The last point should be connected back to the first point. The color of each line should be set from the starting vertex of the line.

VRL_DISPLAY_BOX

The *pparm* parameter points to a two-element linked list of output vertices; the first output vertex is the top-left corner of the box, and the second is the bottom-right corner. The box should be filled with the color of the first output vertex.

VRL_DISPLAY_TEXT

Display the text pointed to by *pparm* in the color specified by *lparm*.

VRL_DISPLAY_TEXT_POSITION

The top 16 bits of *lparm* give the new X position for drawing text, and the bottom 16 bits give the new Y position. The text position marks the location where the top left corner of the first character of a string will appear.

VRL_DISPLAY_GET_TEXTWIDTH

Return the width in pixels of the string pointed to by *pparm*.

VRL_DISPLAY_GET_TEXTHEIGHT

Return the height in pixels of the string pointed to by *pparm*.

VRL_DISPLAY_CAN_GOURAUD

Return non-zero if you support Gouraud shading.

VRL_DISPLAY_CAN_XY_CLIP

Return non-zero if you are willing to do all the X-Y clipping.

VRL_DISPLAY_UPDATE_PALETTE

Do whatever you need to do when the palette changes; *pparm* points to the new palette.

VRL_DISPLAY_BEGIN_FRAME

Do whatever you need to do at the beginning of a frame.

VRL_DISPLAY_END_FRAME

Do whatever you need to do at the end of a frame.

VRL_DISPLAY_SET_RASTER

The *pparm* parameter points to a vrl_Raster, which should be used for all subsequent drawing.

VRL_DISPLAY_GET_RASTER

The *pparm* parameter is a pointer to a pointer to a vrl_Raster; in other words, you should do the equivalent of:

*((vrl_Raster **) pparm1) = our_raster;

VRL_DISPLAY_SET_Z_BUFFER

The *pparm* parameter points to a vrl_Raster that you should use as the new Z-buffer.

VRL_DISPLAY_GET_Z_BUFFER

The *pparm* parameter is a pointer to a pointer to a vrl_Raster; in other words, you should do the equivalent of:

*((vrl_Raster **) pparm1) = our_z_raster;

VRL_DISPLAY_USE_Z_BUFFER

If *lparm* is non-zero, enable use of the Z-buffer; if *lparm* is zero, disable the Z-buffer. Return 0 if you can't Z-buffer, 1 if you do it in software, or 2 if you do it in hardware.

VRL_DISPLAY_CLEAR_Z_BUFFER

Clear the Z-buffer, if you have one. Fill it with the value of *lparm* (if possible).

VRL_DISPLAY_SET_SHADING

The *lparm* parameter is a hint from the user; the higher the number, the more time you should spend on shading.

VRL_DISPLAY_POLY

This is where the action is. The *pparm* parameter points to an output facet; draw it into the current framebuffer. Easy, right?

The two data types used by the display driver are vrl_OutputVertex and vrl_OutputFacet. A vrl_OutputVertex looks like this:

```
struct _vrl_outvertex
        {
        vrl_ScreenCoord x,y,z;           /* X, Y screen coordinates and Z-depth */
        vrl_16bit red, green, blue;      /* components of the color */
        vrl_OutputVertex *next, *prev;   /* doubly-linked circular list */
        };
```

The vrl_ScreenCoord values are fractional; the number of bits to the right of the binary point is specified by VRL_SCREEN_FRACT_BITS.

The colors are in 8.8 format (i.e. 8 bits of color, 8 bits of fraction). For paletted systems, only the red value is used; it's interpreted as a palette index.
Note that future versions of AVRIL may support additional information per vertex, such as texture map coordinates.

A vrl_OutputFacet looks like this:

```
struct _vrl_outfacet
        {
        vrl_OutputVertex *points;     /* doubly-linked list of vertices for this facet */
        vrl_Surface *surface;         /* surface properties */
        vrl_Color color;              /* color of this facet (flat shading only) */
        };
```

The list of points is doubly linked, with the *next* field pointing at the next (clockwise) vertex and the *prev* field pointing back to the previous (counterclockwise) vertex. The vrl_Surface struct is the same one used throughout AVRIL; see the technical reference manual for information about accessing the various fields, especially the type (flat, Gouraud, etc).

For 8-bit systems, the vrl_Color value is a palette index (i.e. only the bottom 8 bits of the 32-bit *color* field are used). For 15-, 16- and 32-bit systems, the actual color is stored in the 32 bit word; the bottom byte is red, the next one up is green, and the next one up is blue. The top byte may be used as an "alpha" channel; if not, it should be zero. Current versions of AVRIL are 8-bit (paletted) only.

Note that future versions of AVRIL may provide additional information for output vertices and output facets.

If you write display drivers, especially ones which support the new 3D graphics accelerators coming on the market, please drop me some email (broehl@sunee.uwaterloo.ca). Since I'm giving AVRIL away for free, I can't afford to buy every card that comes out, so I'm counting on other people to create drivers. I'll be happy to include your drivers with the main AVRIL release, with full credit.

Appendix I - STEREOSCOPIC VIEWING TYPES

A number of stereoscopic viewing types are supported in AVRIL. The current version doesn't implement all of them, but all of them are described here.

VRL_STEREOTYPE_NONE
Monoscopic; no stereo.

VRL_STEREOTYPE_ANAGLYPH_SEQUENTIAL
Field sequential, with alternate frames using red and blue palettes. View with anaglyph (red-blue) glasses. Not yet implemented.

VRL_STEREOTYPE_ANAGLYPH_WIRE_ALTERNATE
Wireframe anaglyph, with left-eye image on even scanlines and right-eye image on odd scanlines. View with anaglyph (red-blue) glasses.

VRL_STEREOTYPE_ANAGLYPH_SOLID_ALTERNATE
Similar to wireframe anaglyph, but solid instead of wireframe.

VRL_STEREOTYPE_ENIGMA
Alternate scanline encoding, for use with CyberMaxx HMD.

VRL_STEREOTYPE_FRESNEL
Left-right split screen, for use with Fresnel viewer such as the one described in Virtual Reality Creations.

VRL_STEREOTYPE_CYBERSCOPE
Left-eye image on left half of screen, right-eye image on right half of screen, both rotated 90 degrees; should work with Cyberscope from Simsalabim Systems Inc, but so far untested (since I don't have one!).

VRL_STEREOTYPE_CRYSTALEYES
Puts left-eye image on top of screen, right-eye image on bottom of screen, for eventual support of CrystalEyes shutter glasses from StereoGraphics (can't do any real development on this, since I don't have them!).

VRL_STEREOTYPE_CHROMADEPTH
Single-image system, mapping depth into red through blue shades; view with Chromadepth glasses (which you can get at most laser light shows, or with certain Valiant comic books).

VRL_STEREOTYPE_SIRDS
Single-Image Random Dot Stereogram. Not yet implemented.

VRL_STEREOTYPE_TWOCARDS

Assumes two video cards, separate outputs going to the two displays of an HMD.

VRL_STEREOTYPE_SEQUENTIAL

Field sequential, with shutter glasses; not yet supported in AVRIL (as of version 2.0).

If you know of some way to produce stereoscopic images that I haven't mentioned here, please drop me some email (broehl@sunee.uwaterloo.ca).

Appendix J - INPUT DEVICES

The following devices are supported in AVRIL, as of version 2.0:

vrl_MouseDevice

Standard mouse; buttons are used to map two axes into six. It works like this:

No buttons down:	left-right causes Y rotation, forward-back moves in Z
Left button down:	left-right moves in X, forward-back moves in Y
Right button down:	left-right rotates around Z, forward-back rotates around X

vrl_GlobalDevice

Global Devices Controller; not currently being manufactured. Has tactile stimulation.

vrl_CybermanDevice

Logitech Cyberman. Has cheesy tactile feedback.

vrl_RedbaronDevice

Logitech ultrasonic mouse. Was originally code-named the "Red Baron".

vrl_CTMDevice

CyberMaxx Tracking Module, in the CyberMaxx HMD.

vrl_VIODevice

Head tracker in the i-glasses! HMD from Virtual i/o.

vrl_SpaceballDevice

Original Spatial Systems Spaceball.

vrl_IsotrakDevice

Original Polhemus Isotrak. I'd add support for newer versions, but I don't have them.

The ones below are PC-specific:

vrl_KeypadDevice

Uses keypad arrows, the PgUp and PgDn keys, and shifted versions of each. It works like this:

no shift down:	left-right arrows rotate in Y, up-down arrows move in Z
left shift down:	left-right arrows move in X, up-down arrows move in Y
right shift down:	left-right arrows rotate in Z, up-down arrows rotate in X
PGUP and PGDN:	move in Y (same as up and down arrows with left shift)

vrl_JoystickDevice

Ordinary analog joystick; buttons are used to map two axes into six. Also supports joystick-compatable devices such as the popular PC gamepads (some of which are available in a wireless version, which is good if you're in an HMD and don't want to trip over the wire!). The buttons map like this:

no buttons down:	tilt left-right rotates in Y, forward-back moves in Z
trigger button down:	tilt left-right moves in X, forward-back moves in Y
thumb down button:	tilt left-right rotates around Z, forward-back around X

vrl_FifthDevice

The FifthGlove, from Fifth Dimension Technologies (5DT). The finger positions are mapped into channels 6 through 10 (channels 0 through 5 are not used, since the glove has no built-in tracker).

vrl_CyberwandDevice

The CyberWand, from InWorld VR. The "hat" controller's axes are mapped using the "pinky" and "grip" buttons (the grip button being the one on the side of the stick, halfway up).

no buttons down:	hat moves forward/backward, sideways
pinky button down:	hat rotates around X, Y
grip button down:	hat moves up/down, rotates around Z

vrl_7thSenseDevice

Head tracker in the 7th Sense HMD from All-Pro.

vrl_PadDevice

A Nintendo-style gamepad, using the cable from the July 1990 Byte magazine. Works well with the Grip-It free-flying joystick (which uses mercury switches for two-axis tilt information).

Sorry, no PowerGlove driver yet. The timing stuff is hairy, and I've got lots of other things to add to AVRIL. They're also not manufacturing PowerGloves anymore. If someone else would like to write one, go for it! I'll help if I can.

In fact, if you write a driver for *any* device, drop me some email (broehl@sunee.uwaterloo.ca); since I'm giving AVRIL away for free, I can't afford to buy every different input device on the market, so adding support for them is difficult.

If you write drivers for AVRIL, I'll be happy to include them in the main release and give you full credit for having written them.