USB Joystick Support for OpenVMS V8.3

Joysticks and game pads for USB are standard Human Interface Devices (HID). They provide some number of axes, buttons and perhaps a hat switch. A minimum joystick is simply X and Y axes with perhaps a button or two. But they can be as complex as a "Spaceball" or contain multiple buttons, hat switches, and axis - sliders, wheels, dials, etc. Low end steering wheels also often show up as joysticks or game pads.

The device driver for the joystick/gamepad is AGDRIVER (named for Andy G., who is an aircraft simulator hound and did the original GLUT hack for SKYFLY). AGDRIVER is capable of describing and returning data for pretty much anything that can describe itself as a joystick or gamepad to USB.

> **Note: Force Feedback is not implemented at this time. Neither is the joystick supported by the X Input Extension – and it will not control the X11 pointer.**

It can handle up to 32 (each) of the following axis types:

• X	(Left/Right stick movement)
• Y	(Front/Back stick movement)
• Z	(Up/Down)
• RX	(Rotation around the X axis)
• RY	(Rotation around the Y axis)
• RZ	(Rotation around the Z axis – stick twist)
• VX	(Vector in the X direction)
• VY	(Vector in the Y direction)
• VZ	(Vector in the Z direction)
• VBRX	(Relative vector in the X direction)
• VBRY	(Relative vector in the Y direction)
• VBRZ	(Relative vector in the Z direction)
• VNO	(Undirected vector)
• SLIDER	(A slider control)
• DIAL	(Knob)
• WHEEL	(Like a knob, but more like a thumbwheel)

In addition, up to 32 hat switches can be supported - a hat switch is like a digital joystick that reports one of 8 values depending on which way it is pressed:

8	1	2
7	0	3
6	5	4

> *Think: Atari Joystick.* Nowadays they tend to be little joystick on a PlayStation (in digital mode) or little thumb-switches on cool joysticks.

Up to 64 discrete buttons can be supported in addition to the Hat switch. A typical gaming joystick will have something like 12 to 26 buttons and a hat switch (sometimes hat switches are reported as normal buttons instead of hat switches – I think Windows games or Windows itself has a problem with multiples of an axis or hat).

If you want a simple interface to the joystick, skip to the section VMSJOYSTICK below. It presents a library interface that is used in the OpenVMS port of OpenGLUT, and can be compiled and used as a standalone library interface to the joystick. It provides both polled and asynchronous input delivery, scaling, normalization, calibration files, and joystick definition files for implementing joystick code that is independent of the specific joystick in use.

Connecting a joystick to an OpenVMS system is simple provided there is a USB connection. USB is standard on most 1-4 CPU Integrity Servers, and on EV7 based Alpha Servers. For “unsupported” use (hobbyist) - a number of PCI multi-port boards are available which the OpenVMS USB port driver will handle. You may need a USB HUB if there are not enough USB connections for your keyboard, mouse and joystick.

If Auto-Load is set for USB on your system, plugging in the device will create a VMS device called AGA0 – additional joysticks will get increasing unit numbers. If Auto-Load is not enabled on your system, then you will need to use the UCM command to display (SHOW DEVICE/UNCONFIGURED) and to add (ADD DEVICE AGA0) as needed, followed by a hot-plug of the joystick.

The DECW$OPENGLUTSHR.EXE has been extended to allow the joystick to be used on OpenVMS, and the application SKYFLY from the GLUT 3.7 demo area has been modified to use the joystick when present.

Device Driver Programming

If you plan on directly interfacing to the device driver, you should obtain a copy of the USB HID Specification and USB HID Usage Page Specification from:

http://www.usb.org/developers/hidpage/

In addition, the code example VMSJOYSTICK.C and VMSJOYSTICK.H are also good reference material.

AGDRIVER provides data to the application using two structures, the JOYSTICK_LIMIT_DATA and JOYSTICK_INPUT_FRAME. Both of these structures include ‘version’ and ‘minor’ fields. Code can compare these against the constants defined in AGDEF to determine if the driver and application are in sync for the structure layout. The version field will change when incompatible changes are made to the structure, the minor field will change when things are added (using spare fields) that does not break backwards compatibility.

The constants are:

JOYSTICK_INPUT_FRAME_VERSION
JOYSTICK_INPUT_FRAME_MINOR_REV

JOYSTICK_LIMIT_DATA_VERSION
JOYSTICK_LIMIT_DATA_MINOR_REV

The following descriptions of driver functions do not provide numeric values, but use the symbolic values defined in AGDEF. AGDEF is generated from a SDL file, and is available for multiple languages. The descriptions below use the C language versions of the structures, and include no comments. See the language-specific AGDEF for more information on fields and constants.

Limit Data:

AGDRIVER reports information about a joystick using a SENSEMODE QIO to return a JOYSTICK_LIMIT_DATA structure defined in AGDEF:

```
typedef struct _joystick_limit_data {
    int size;
    int type;
    int version;
    int minor;
    int vendor_id;
    int product_id;
    unsigned __int64 button_valid;
    int button_count;
    int num_axes;
    JOYSTICK_AXIS_INFO hat;
    JOYSTICK_AXIS_INFO x;
    JOYSTICK_AXIS_INFO y;
    JOYSTICK_AXIS_INFO z;
    JOYSTICK_AXIS_INFO rx;
    JOYSTICK_AXIS_INFO ry;
    JOYSTICK_AXIS_INFO rz;
    JOYSTICK_AXIS_INFO vx;
    JOYSTICK_AXIS_INFO vy;
    JOYSTICK_AXIS_INFO vz;
    JOYSTICK_AXIS_INFO vbrx;
    JOYSTICK_AXIS_INFO vbry;
    JOYSTICK_AXIS_INFO vbrz;
    JOYSTICK_AXIS_INFO vno;
    JOYSTICK_AXIS_INFO slider;
    JOYSTICK_AXIS_INFO wheel;
    JOYSTICK_AXIS_INFO dial;
    int extras_longs [32];
    int output_available;
    int output_max_bytes;
    int output_item_usage_page;
    } JOYSTICK_LIMIT_DATA;
```

Each axis is structure within this structure has the format:

```
typedef struct _joystick_axis_instance {
    unsigned int flags;
    int min;
    int max;
    int size;
    int pmin;
    int pmax;
    int unit;
    int unit_e;
    int center_off;
    int cmin;
    int cmax;
    } JOYSTICK_AXIS_INSTANCE;
```

```
typedef struct _joystick_axis_info {
    int count;
    JOYSTICK_AXIS_INSTANCE axis [32];
    } JOYSTICK_AXIS_INFO;
```

This report is important, because it provides you the information needed to interpret the data report for joystick input.

The USB joystick interface reports on buttons, hat switches, and various types of axes. The minimum information that must be reported by every device is the size of the field (which usually corresponds to the resolution of the axis), and a logical minimum and maximum. They also can report a physical minimum and maximum, as well as a unit code and unit exponent – when these are present, it is possible to determine what the logical values represent – degrees, weight, force, temperature, etc. Most devices don't report anything for these values.

Each type of axis can have multiple instances, although in practice this has yet to be seen. Each axis type consists of a count, followed by 32 axis instance records. A non-zero count tells you that there is a valid axis, and the first 'count' indexes in the axis array are available for use.

Buttons are consolidated into a 64-bit unsigned integer. There is no guidance from the USB information on the meaning of a button, or it's ordering. So there is no way to determine just from the raw data which button is, for example the "Trigger" or "Fire" or "Function 1" button. Buttons are reported in the typical way, a bit value of (1) means pressed, (0) means released.

Hat switches return a value as shown above from 0 (center) to 8.

The cmin, cmax and center_off fields are **calibration** data. This data is not provided by the device, but instead is provided to the driver via a SETMODE QIO. When provided, this data overrides the logical min and max, and provides a value (center_off) which is added to the input value to allow "centering" the joystick axis. This data would typically be obtained by a calibration program that asks the user to move the various range of motions, and measures the actual results. This data would then be written to the driver, and subsequent applications would then be able to use this calibration information. The information is ***not*** persistent and will be lost on a reboot, or a device hot-plug. See VMSJOYSTICK for an example of implementing disk based calibration persistence.

The flags field will return one or more of the following:

```
JOYSTICK_AXIS_SIGNED
JOYSTICK_AXIS_PMIN_VALID
JOYSTICK_AXIS_PMAX_VALID
JOYSTICK_AXIS_UNIT_VALID
JOYSTICK_AXIS_UNIT_E_VALID
JOYSTICK_AXIS_CENTER_OFF_VALID
JOYSTICK_AXIS_CMIN_VALID
JOYSTICK_AXIS_CMAX_VALID
```

These bits, as is apparent, indicate which fields are valid.

All values are signed this means that for example when an axis is signed and centered on zero, min and max might contain -1023 and +1023. If the JOYSTICK_AXIS_SIGNED bit is not set – min, pmin, and cmin will be a positive value or zero.

To read this structure, a SETMODE QIO is issued:

```
JOYSTICK_LIMIT_DATA info;

status = sys$qiow (0, channel, IO$_SENSEMODE,
                  &iosb, 0, 0,
                  JOYSTICK_IO_SENSE_LIMIT_DATA, /* P1   */
                  (int) &info,                  /* P2   */
                  sizeof (JOYSTICK_LIMIT_DATA), /* P3   */
                  0,                            /* P4   */
                  0,                            /* P5   */
                  0);                           /* P6   */
```

To write calibration data is a SETMODE QIO with a JOYSTICK_LIMIT structure containing the calibration data for the axes and the calibration flags set for those you wish to store:

```
status = sys$qiow (0, channel, IO$_SETMODE,
                  &iosb, 0, 0,
                  JOYSTICK_IO_SET_CALIBRATION_DATA,   /* P1 */
                  sizeof (JOYSTICK_LIMIT_DATA),       /* P2 */
                  (int) &info,                        /* P3 */
                  0,                                  /* P4 */
                  0,                                  /* P5 */
                  0);                                 /* P6 */
```

Input Frame Data

Actual joystick data (the axes and buttons) is returned in a JOYSTICK_INPUT_FRAME structure:

```
typedef struct _joystick_input_frame {
    int size;
    int type;
    int version;
    int minor;
    int vendor_id;
    int product_id;
    unsigned __int64 timestamp;
    unsigned __int64 button_state;
    int hat [32];
    int x [32];
    int y [32];
    int z [32];
    int rx [32];
    int ry [32];
    int rz [32];
    int vx [32];
    int vy [32];
    int vz [32];
    int vbrx [32];
    int vbry [32];
    int vbrz [32];
    int vno [32];
    int slider [32];
    int dial [32];
    int wheel [32];
    int extras [64];
    } JOYSTICK_INPUT_FRAME;
```

Each axis is returned into a 32-bit signed integer (when present). Each axis type can have as many as 32 instances. The buttons are aggregated into a 64-bit unsigned integer, where a bit value of (1) indicates the button is pressed, and a bit value of (0) indicates the button is released. Hat switches return a value from 0 to 8 in the low-order 3 bits. A timestamp of the OpenVMS system time is also written to the report to indicate the time the data was read from the driver.

The data is read by a READVBLK QIO:

```
JOYSTICK_INPUT_FRAME data;

status = sys$qiow (0, channel, IO$_READVBLK, &read_iosb, 0, 0,
                        (int) &data,                        /* P1 */
                        sizeof (JOYSTICK_INPUT_FRAME),      /* P2 */
                        0,                                  /* P3 */
                        0,                                  /* P4 */
                        0,                                  /* P5 */
                        0);                                 /* P6 */
```

The return will be the most recent joystick data. The driver does not buffer up reports, but instead only saves the last report from the device. The application must do timely reads to obtain the joystick data.

To get notification of new joystick data, the application can issue a SETMODE request for either (or both) the setting of an EFN, or an AST routine to be called when new data arrives.

```
status = sys$qiow (0, channel, IO$_SETMODE, &iosb, 0, 0,
                        JOYSTICK_IO_SET_AST_NOTIFY,   /* P1 */
                        0,                            /* P2 */
                        newJoystickDataRtn,           /* P3 */
                        (int) astprm,                 /* P4 */
                        mode,                         /* P5 */
                        0);                           /* P6 */
```

This illustrates setting an AST routine delivery routine for new joystick data. Each new joystick report will call the routine "newJoystickDataRtn" with "astprm" as the only parameter each time a new report arrives in the driver. This routine can then perform the READVBLK call, or set a flag for non-AST code to do the read.

The full list of SETMODE operations for setting and clearing AST and EFN notification is:

Enable EFN notification:

```
P1      =       JOYSTICK_IO_SET_EFN_NOTIFY
                P2 = EFN number
```

Enable AST notification:

```
P1      =       JOYSTICK_IO_SET_AST_NOTIFY
                P3 = AST routine address
                P4 = AST parameter
                P5 = Mode (PSL$C_USER default)
```

Enable AST and EFN notification:

```
P1      =       JOYSTICK_IO_SET_ASTEFN_NOTIFY
                P2 = EFN number
                P3 = AST routine address
                P4 = AST parameter
                P5 = Mode (PSL$C_USER default)
```

Clear EFN notification:

```
P1      =       JOYSTICK_IO_CLEAR_EFN_NOTIFY
```

Clear AST notification:

```
P1      =       JOYSTICK_IO_CLEAR_AST_NOTIFY
```

Clear both AST and EFN notification:

```
P1      =       JOYSTICK_IO_CLEAR_ALL_NOTIFY
```

VMSJOYSTICK Programming

You have chosen wisely grasshopper. VMSJOYSTICK provides a much simpler way to interface to the joystick than the direct driver interface, or even the X Input Extension.

Using this code can be simple or complex. The simplest way to use this code is to:

```
#include “vmsjoystick.h”

extern void input_callback(unsigned int buttons,
                                    int x,
                                    int y,
                                    int z,
                                    int rz,
                                    int throttle,
                                    JOY_CONTEXT *joy);

JOY_CONTEXT *joy = 0;

status = vmsOpenJoystick (&joy);

vmsStartJoystick (&joy, JOYSTICK_ASYNC, input_callback);

/* Any non-AST mainline code here */

vmsStopJoystick (joy);
```

The input callback routine will be called every time a button or axis changes. By default this will just return the logical, un-altered/un-scaled data from the joystick.

You might also want to call

```
vmsQueryJoystick (&num_buttons, &num_axes, &valid_axes, &joy)
```

This will return the number of buttons, axes, and which axes are valid.

You can pretty much count on X and Y, RZ is a twisting joystick. The throttle is either a slider or sometimes is the "Z" axis. This is really vendor specific (see below for ways to normalize joysticks). VMSJOYSTICK returns five axes – X, Y, Z, RZ, and Throttle – when present. Other axes must be directly pulled out of the raw data (which is available though the JOY_CONTEXT structure in the “data” structure). The JOY_CONTEXT structure contains ***all*** of the data, plus all of the information about each axis. This allows more sophisticated code to mine this data out directly.

Data can be signed or unsigned, and the neutral center position might be at zero, or at 1/2 the full range (unsigned data). This information is available in the per-axis data for each (see VMSJOYSTICK.H or AGDEF.H). But you can explicitly request the type of data you want returned…

You can ask that the data be scaled and normalizes by calling vmsJoySetScale.

```
status = vmsOpenJoystick (&joy);

vmsJoySetScale(joy,     JOYSTICK_SCALE_ENABLE | JOYSTICK_SCALE_CENTER,
                        1000, JOYSTICK_X_AXIS);
vmsJoySetScale(joy,     JOYSTICK_SCALE_ENABLE | JOYSTICK_SCALE_CENTER,
                        1000, JOYSTICK_Y_AXIS);
vmsJoySetScale(joy,     JOYSTICK_SCALE_ENABLE | JOYSTICK_SCALE_CENTER,
                        1000, JOYSTICK_Z_AXIS);
vmsJoySetScale(joy,     JOYSTICK_SCALE_ENABLE | JOYSTICK_SCALE_CENTER,
                        1000, JOYSTICK_RZ_AXIS);
vmsJoySetScale(joy,     JOYSTICK_SCALE_ENABLE,
                        1000, JOYSTICK_THROTTLE_AXIS);
```

This code example says to enable scaling, and to center the data - so the X, Y, Z and RZ axes will return -1000 to 1000 with 0 at the center of travel. The throttle will return 0 to 1000.

You can also invert the data with JOYSTICK_SCALE_INVERT which will flip the positive or negative (or min/max direction).

And that is pretty much all you need to know to do basic programming.

For more in depth understanding VMSJOYSTICK.C and VMSJOYSTICK.H are the places to begin.

Joystick Definition Files:

But wait. There is even more help for you here, vendor/device joystick data can be provided so that code doesn't have to special case things too much. Here's the problem: You've written a GLUT application to fly an airplane, but you don't know what joystick might get plugged in... heck, you are only indirectly using VMSJOYSTICK because that is what GLUT uses on OpenVMS.

The Joystick definition file: This file allows a joystick to be "normalized" against some arbitrary "norm". In this case, the author of the code had a Saitek Cyborg EVO Wireless joystick... so that is the "norm" for our purposes. By placing a definition for specific vendor/model of joystick, an application may not need any "special" logic for joystick differences, or at least can use a standard way of falling back when a feature (like an axis) is missing.

A really good joystick (think: Saitek X45) has at least 5 axes: X, Y, Z, RZ and a throttle. This is great, now you can turn, pitch, roll, yaw, and change speed like a real airplane. However, someone plugs in a really simple joystick - it only has X and Y axes. Well, you can't do all the spiffy things above, but you might want Y to be substituted for the throttle instead of pitch. That is where the joystick definition file comes in. In this file, you specify which "real" axis you want to map to the X, Y, Z, RZ, Throttle data that is returned.

So - for the X45, you would want to map:

- X = X(0)
- Y = Y(0)
- Z = Z(0)
- RZ = RZ(0)
- THROTTLE = SLIDER(0)

(Note the (0) means the first instance of the axis).

For a dumb X/Y only device you might set:

- X = X(0)
- Y = NONE
- Z = NONE
- RZ = NONE
- THROTTLE = Y(0)

It gets even more important for in-between devices. Say the Cyborg EVO, which is a 4 axis joystick. It reports X, Y, RZ, and **Z.** Where, Z is a separate up/down lever on the back of the joystick. You might decide that throttle is more important than Z (after all you can use speed and pitch to change height - unless you are a helicopter), so you might map this:

X	= X(0)
Y	= Y(0)
Z	= NONE
RZ	= RZ(0)
THROTTLE	= Z(0)

VMSJOYSTICK will try to apply some common sense fallbacks for undefined joysticks (like above) but there is nothing like being able to be explicit.

The other problem is buttons. Say you want a button that fires a missile - but each joystick reports their buttons as simply a collection of buttons not as "Fire". The joystick definition file also can cause the buttons to be remapped. So if on one joystick model button 4 is the "Fire" button, and on another button 1 is "Fire", you can have them both mapped to (for example) button 1 - without vendor-specific logic in the code - or your code itself knowing about the remapping.

The joystick definition file is first looked for as a logical JOY$DEFINITION, then in []joystick.dat, then sys$login:joystick.dat, and then sys$manager:joy$definition.dat Failing that, there are a handful of known ones baked into the vmsjoystick.h file:

Joystick Definition File Format

General format:

```
    JOYSTICK_DEFINITION "Vendor Name", "Product Name";
        VENDOR_ID     <usb vendor id>;
        PRODUCT_ID    <usb product id>;
        FLAGS         <axis flags>;
        X_AXIS        <source_axis> [<axis number>];
        Y_AXIS        <source_axis> [<axis number>];
        Z_AXIS        <source_axis> [<axis number>];
        RZ_AXIS       <source_axis> [<axis number>];
        THROTTLE_AXIS <source_axis> [<axis number>];
        BUTTON_MAP  n, n, n, n, n, n, n, n, ...
    END_DEFINITION;
```

The flags can be one or more of the following, and allow the individual axes to be inverted. If there are no flags to set, the keyword can be omitted.

```
    FLAGS X_INVERT
    Y_INVERT
    Z_INVERT
    RZ_INVERT
    THROTTLE_INVERT
```

One additional flag:

```
    FORCE_FULL_SCALE
```

Setting this will cause any axis that does not have calibration data in the driver to be forced to full scale based on the field width (that is, a 12-bit number, or a 10-bit, or 8-bit or whatever is reported from the USB driver. Some joysticks report bogus logical min/max values, and this is a quick way to "fix" the problem.

There are 5 standard axes: X, Y, Z, RZ, THROTTLE - but these may not be present, and in the case of the THROTTLE - there is no standard axis. Each axis can be mapped to a specific USB defined axis: X, Y, Z, RX, RY, RZ, VX, VY, VZ, VBRX, VBRY, VBRZ, VNO, SLIDER, DIAL, or wheel. There can be multiple instances of an axis (although, I think Windows has trouble with it - since I've not seen one yet) - so following the axis type is an instance number. So, for a typical joystick that reports a single instance of SLIDER that you want to map to the THROTTLE:

```
    THROTTLE_AXIS   SLIDER[0];
```

If no USB axis will map to a standard axis, then NONE can be specified (or the axis definition omitted, which will default to NONE).
Buttons:

VMSJOYSTICK returns 28 of a possible 64 buttons, the 28 buttons can be remapped from any bit position to any bit position. The first hat switch (when present) is in the upper 4 bits of the button (which is why the limit is 28 buttons and not 32). The other 28-bits can be routed by using the flag BUTTON_MAP keyword. Each value following the BUTTON_MAP keyword corresponds to the input bit position, the value corresponds to the final button bit position. If button mapping is enabled **only** the buttons defined in the button map will appear in the button data (along with the HAT, which is never masked out when present).

```
BUTTON_MAP 0, 5, 6, 3, 4, 2
```

Will for example, map button 0 to 0, 1 to 5, 3 to 6, 4 to 3, 5 to 4, 6 to 2. So a button input of 1 will set bit 1, but a button input of 0x02 will result in 0x20 (that is, bit position 1 (0x02) will become bit position 5 (0x20), etc.

NORMS:

Joystick norms:

- X Absolute data: Full Left 0, Full Right MAX, Center (MAX+1)/2
  Centered: Full Left +MAX, Full Right -MAX, Center 0
- Y Absolute data: Full Forward 0, Full Back MAX, Center (MAX+1)/2
  Centered: Full Forward +MAX, Full Back -MAX, Center 0
- Z Absolute data: Full Up 0, Full Down MAX, Center (MAX+1)/2
  Centered: Full Up +MAX, Full Down -MAX, Center 0
- RZ Absolute data: Full Twist L 0, Full Twist R MAX, Center (MAX+1)/2
  Centered: Full Twist Left +MAX, Full Twist Right -MAX, Center 0
- THROTTLE Absolute data: Full Forward 0, Full Back MAX, Center (MAX+1)/2
  Centered: Full Forward +MAX, Full Back -MAX, Center 0

The norms for buttons:

• TRIGGER	Bit 0
• FIRE CENTER	Bit 1
• FIRE LEFT	Bit 2
• FIRE RIGHT	Bit 3
• AUX FIRE LEFT	Bit 4
• AUX FIRE RIGHT	Bit 5
• F1	Bit 6
• F2	Bit 7
• F3	Bit 8
• F4	Bit 9

Example:

```
JOYSTICK_DEFINITION "Saitek", "Cyborg EVO Wireless";
  VENDOR_ID     1699;
  PRODUCT_ID    13630;
  X_AXIS        X_AXIS[0];
  Y_AXIS        Y_AXIS[0];
  Z_AXIS        NONE;
  RZ_AXIS       RZ_AXIS[0];
  THROTTLE_AXIS Z_AXIS[0];
END_DEFINITION;
```

Calibration Files:

In addition, this code supports calibration data. This data can be used to compensate for inaccuracy in analog devices - where the neutral position does not read zero, or where you cannot get the full range of data that the logical range indicates (or where your scaling isn't integral, and you want to "fudge" things to make sure you get full sale readings -- this can happen when you try to scale things to non multiples/divisors of the input range).

The code looks for a calibration file (a logical name JOY$CALIB) and then for calib.dat in the current directory, and then in the login directory, and then in sys$manager:joy$calib.dat.

Calibration Data File Format

The general format is:

```
calibration "My Joystick" {

        device_name   = nnn;
        vendor_id       = nnn;
        product_id      = nnn;

           axis NAME [nn] {
              center      = nnn;
              cmin        = nnn;
              cmax        = nnn;
           };

           axis NAME [nn] {
              center  = nnn;
              cmin    = nnn;
              cmax    = nnn;
           };

} end_calibration;
```

A lot of this is just making it pretty. You can use C-like comments in it, and a C-like structure definition (C-like, not valid-C). It can also be written:

```
CALIBRATION;
DEVICE_NAME nnn;
VENDOR_ID nnn;
PRODUCT_ID nnn;
AXIS NAME nn;
CENTER nnn;
CMIN nnn;
CMAX nnn;
AXIS NAME nn;
CENTER nnn;
CMIN nnn;
CMAX  nnn;
END_CALIBRATION;
```

This is the actual simple syntax, the command followed by parameters and closing terminator. The “;” terminator is needed to terminate each command (the "{" can also be used) so that commands can span lines…

Since it is possible to have multiple instances of an axis (like multiple sliders) the [nn] says which one - typically this will just be a single instance, and 0. The valid axis names are: X, Y, Z, RX, RY, RZ, VX, VY, VZ, VXBR, VXBY, VXBZ, VNO, SLIDER, DIAL, and WHEEL.

The vendor/device is the USB ID's as returned by the joystick. The device name is the VMS device - like AGA0. This allows the code to match the vendor/product and device name against the current device - so you can calibrate multiple sticks, or multiple types.

All numbers are decimal.

The values are written to the device driver in a SETMODE call, and will be persistent until the joystick is hot plugged or the system is rebooted, the device is hot-plugged, or they are overwritten by another SETMODE to set calibration.

Example:

```
calibration_data "Saitek Cyborg EVO Wireless" {

  device_name   = aga0;
  vendor_id     = 1699;
  product_id    = 13630;

  axis x [0] {
    center_offset = -1;
    cmin          = 2;
    cmax          = 1022;
  }

  axis y [0] {
    center_offset = -1;
    cmin          = 2;
    cmax          = 1022;
  }

} END_CALIBRATION;
```