Joystick Mapper for Linux – Programming Guide

Alexandre Hardy

February 20, 2016

This document describes programming language used by the joystick mapping software. The driver creates a new joystick device that can be mapped with the normal joystick mapping software. To use the programming language effectively the virtual joystick device must be mapped to some joystick button or axis. The virtual joystick device is indicated by a vendor id of `0x00ff` and a product id of `0x0000`.

At each execution cycle, the program is executed from the beginning until termination (in sequence) or a maximum number of instructions has been executed (to prevent the driver from looping indefinitely). The exception to this rule is `threads` described in section 3.8.

Comments are indicated by a hash (#) and continue to the end of the line.
We now describe the basic components of a program.

1 Variables

All variables in the programming language are 32-bit integers. These variables can be used to store results from calculations and can represent button values or axis values. A new variable is declared with the `var` statement:

`var` *variable_name*;

where *variable_name* is any identifier that begins with a letter followed by letters, digits or the underscore. Variable names are case-sensitive. These variables can be used for any purpose. Variables are allocated to registers in the virtual machine. Since there are 256 such registers, only 256 variables can be declared in addition to the predefined variables.

Arrays can be declared as follows

`var` *variable_name*`[`*constant*`]`;

The size of the array must be constant. Each element of the array is assigned to one register. Registers can be used up very quickly by declaring arrays! Arrays

are 0 based, that is the first index of any array is 0.

A variable can be declared with the `global` keyword instead, in which case the variable is *shared* between all threads. This variable can then be used for inter thread communication.
There are a few predefined variables that have a special meaning. These variables are described in the following sections.

1.1 `firstscan`

The `firstscan` variable is set to 1 if this is the first time the program is executed, and 0 otherwise. This variable can be used to decide when to initialize program variables.

1.2 `clocktick`

The `clocktick` variable is set to 1 if the program is executed as a result of a timer event, or 0 if the program is executed due to some other event (such as a joystick button being pressed).

1.3 `timestamp`

The `timestamp` variable indicates the time since first execution of the script measured in milliseconds. This variable can be used to queue statements based on time. Examples of use include the `delay` statement (section 3.9).

1.4 `currentmode`

The `currentmode` variable is defined by the program to provide state referring to a mode of operation. The variable is completely controlled by the user and may be used for any purpose. Unlike general variables, this variable is accessible from all `threads` (see section 3.8).

1.5 `js`

The `js` variables give access to the current state of joysticks attached to the system. There are 16 `js` variables, namely `js0` through `js15` that provide access to a maximum of 16 joysticks. The joysticks are numbered according to the specification in the current mapping configuration file. That is, the joystick numbers are explicitly assigned when the joystick is mapped. Each `js` variable has two fields:

- An `a` field that is an array of integers. Each element provides the current position of that axis of that joystick. For example `js0.a[1]`, refers to the second axis of the joystick designated as 0 in the configuration file.

- A `b` field that is an array of integers. Each element provides the current button status of that button of that joystick. For example `js1.b[3]`, refers to the fourth button of the joystick designated as 1 in the configuration file. The value is 1 for a depressed button and 0 otherwise.

1.6 `a`

The variable `a` is an array of integers specifying the position of the virtual axis of the virtual joystick created by the program. For example the statement

```
a[1]=10;
```

reports that the second axis of the virtual program joystick is in position 10. The variable may also be read.

1.7 `b`

The `b` variable is an array of integers indicating whether a virtual button on the virtual program joystick is depressed or not. For example, to indicate that the first button on the virtual joystick is depressed, we write

```
b[0]=1;
```

and to indicate (at a later time) that the button is released we write

```
b[0]=0;
```

This variable may also be read.

2 Operators

Operators are used to build expressions to perform some calculation. The operators are very similar to C operators with some minor differences. Operators take constants, variables, array elements or other expressions as parameters.

2.1 Arithmetic operators

The arithmetic operators are as follows:

- Unary +, eg. $+v$.
- Unary -, eg. $-v$.
- Binary + eg. $v_1 + v_2$.
- Binary - eg. $v_1 + v_2$.
- Binary $*$ eg. $v_1 * v_2$.
- Binary / – integer division. eg. v_1/v_2.

- Binary % – integer remainder. eg. $v_1 \% v_2$. This function is not implemented with primitive instructions and is implemented as $a\%b = a - (a/b) * b$.

2.2 Boolean operators

Boolean operators allow certain conditions to be tested. The result is an integer indicating the truth of the statement. 0 is taken to mean *false*, and anything else is taken to mean *true*. Thus integer values (variables, results from a calculation) are also boolean values.

- `a==b`: true if integer `a` is equal to integer `b`, false otherwise.
- `a!=b`: true if integer `a` is NOT equal to integer `b`, false otherwise.
- `a<b`: true if integer `a` is less than integer `b`, false otherwise.
- `a>b`: true if integer `a` is greater than integer `b`, false otherwise.
- `a<=b`: true if integer `a` is less than or equal to integer `b`, false otherwise.
- `a>=b`: true if integer `a` is greater than or equal to integer `b`, false otherwise.
- `a&&b`: true if `a` is true and `b` is true, false otherwise.
- `a||b`: true if `a` is true or `b` is true, false otherwise.
- `!a`: true if `a` is false, false otherwise.

2.3 Precedence

Unary operators have the highest precedence. The precedence of binary operators is similar to C. The precedence from highest to lowest is:

- && and ||
- $*$, / and %
- + and -
- `==`, `!=`, `<=`, `>=`, `<` and `>`

3 Program statements

All statements are terminated by a semi-colon, except the statement block.

3.1 Assignment

Assignment is the most commonly used statement. An example of assignment is

```
currentmode=1;
```

This sets the variable currentmode to have the value 1. Be careful not to confuse the assignment operator = with the boolean equality test ==. A few shorthand notations exist for some common assignment operations:

- `a++` → `a=a+1;`
- `a--` → `a=a-1;`
- `a+=b` → `a=a+b;`
- `a-=b` → `a=a-b;`
- `a*=b` → `a=a*b;`
- `a/=b` → `a=a/b;`

3.2 Statement blocks

If several statements need to be combined into a unit, a statement block can be created as follows:

{
    statement$_1$;
    statement$_2$;
    statement$_3$;
    ⋮
    statement$_n$;
}

3.3 `if`

An `if` statement has the form

`if (`*condition*`)` *statement*

If the *statement* is a simple statement, then it is terminated by a semi-colon. If it is a block statement, then no semi-colon is necessary.

If the expression *condition* evaluates to 0 then the *statement* is not executed, otherwise the *statement* is executed. It is also possible to specify to alternatives as in

`if (`*condition*`)` *statement*$_1$ else *statement*$_2$

where *statement*$_2$ is executed if *condition* evaluates to 0, otherwise *statement*$_1$ is executed.

3.4 `while`

A `while` loop has the form

`while (`*condition*`)` *statement*`;`

As long as *condition* evaluates to a non-zero value *statement* will be executed. The *condition* is evaluated before the *statement* is executed and is tested directly before the first (possible) execution, and directly after execution of *statement*. If *condition* evaluates to 0, then the loop is terminated.

3.5 `signal`

The `signal` statement has the form

`signal(`*expression*`);`

This statement sends the result of expression to the mapper device (the device used to program the joystick) to be relayed to a client program. This can be used to trigger events outside of the kernel space. For example, the joystick could be reprogrammed by the client program based on the signal sent, or a particular program such as an e-mail client or multimedia player could be executed.

3.6 `press`

The `press` statement allows keypresses to be sent directly to the driver. `press` has the form

`press("`*key*`");`

where *key* is one of the constants listed in `keys.txt`. `press` sends a key pressed event to the driver.

3.7 `release`

The `release` statement allows key release events to be sent directly to the driver. `release` has the form

`release("`*key*`");`

where *key* is one of the constants listed in `keys.txt`. `release` sends a key released event to the driver.

3.8 `thread`

`Threads` are threads of execution in the sense that each thread will remember which statement was executing during the last execution of this thread. Thus threads maintain state information in terms of an instruction pointer and a collection of registers. `Threads` do not imply concurrent execution in any way whatsoever! The `thread` statement declares a thread with independent state, as

well as indicating that the thread must be executed at this point. The main program will stop executing until execution of this thread completes or the thread yields. If the thread halts, then the state information of the thread is reset. If the thread yields, then the thread will stop execution and save the instruction pointer so that the thread can be resumed later. The first time the thread is executed (or after the last halt), the current registers are stored in the state of the thread and the instruction pointer is set to the first instruction (statement) of the thread. Thereafter (until the next halt), the thread will use its own copy of the registers (except for special registers such as `timestamp` and `currentmode`). If the thread was not halted, then the instruction pointer will be used to resume execution of the thread.

The thread statement has the form

`thread statements;`

The statement declares a new thread to the compiler. The compiler will also generate instructions to begin execution of the thread (suspending execution of the main program until the thread has halted or yielded). Each thread is allocated a unique thread number. The maximum number of threads in a program is 8. However, an alternative declaration

`thread` *name* `statements;`

can be used to provide a specific name to a thread. All threads in the program with the same name will share the same thread number, thus increasing the potential number of threads. If there are two or more such threads, then the programmer is declaring to the compiler that only one such thread will NOT be in the halted state at any time.

3.9 `delay`

The `delay` statement is only valid within a `thread` statement. The `delay` statement has the form

`delay(`*expression*`);`

This statement will delay the executing `thread` *expression* milliseconds (see `timestamp` in section 1.3). The mechanism used to delay this period of time is a check of the amount of time delayed so far, followed by a yield if the required time has not elapsed. The thread will resume execution at this statement the next time it is executed if the required time has not elapsed. Note that the expression is reevaluated every time the delay is checked.

3.10 `wait`

The `wait` statement is only valid within a `thread` statement. The `wait` statement has the form

`wait(`*condition*`);`

This statement halts the executing thread until *condition* becomes true (that is non-zero). The `wait` statement is implemented by a yield if *condition* is false, which returns control to the calling thread. The thread will resume execution at this statement the next time it is executed.

3.11 `halt`

The halt statement has the form

`halt;`

and halts execution of the current thread or the main program. Every program must halt. The compiler automatically adds a `halt` statement to the end of any program to ensure that the program will halt.

It is also possible to halt a specific thread (possibly different to the current one) with the statement

`halt` *name*`;`

4 Examples

A few examples of programs are presented in this section. In most of the examples it is necessary to provide a mapping from the virtual joystick to a real joystick, or keyboard events.

4.1 Toe Brakes

Some flight simulators support toe brakes in the form of buttons or keypresses, but do not support an axis for toe brakes. If this is the case, we can use the toe brake axis on a set of rudder pedals to simulate joystick button or key pressed. Assume that the rudder pedals have been designated as joystick 2, and that the toe brake axes are axes 0 and 1. The program to convert the axis positions to button presses is:

```
b[0]=(js2.a[0]>128);
b[1]=(js2.a[1]>128);
```

Remember that `b` is the array of buttons for the virtual joystick, and that boolean and integer expressions are interchangeable. The typical range for any axis is 0–255, hence the choice of comparison to 128. The program's virtual joystick buttons should then be mapped to real joystick buttons, or to key presses.

4.2 Car Accelerator and Brakes

Most rudder pedals have to separate axes for the left and right toe brake, but car simulators tend to use only one joystick axis for both acceleration and braking. Assume the same setup as above with axis 0 to be used for the brake, and axis 1 to be used for the accelerator. The two axes can be mapped to one axis with the program

```
var val;
#get a positive value for acceleration
# >128 indicates acceleration
val=js2.a[1]/2+128;
#produce a braking value
# <128 indicates brakes
# we need to reverse the sense of the axis
val-=js2.a[0]/2;
a[0]=val;
#note that accelerating and braking at
# the same time results in no action
# but cannot be resolved here
```

4.3 Delayed Release of Countermeasures

In modern combat flight simulators it is necessary to drop countermeasures such as flares or chaff to confuse the seeker heads of missiles launched at the aircraft. In such a situation it is standard practice to release several countermeasures with a delay between the release of each one. A thread should be used to achieve this, so that other conditions may be checked and other actions followed despite the presence of countermeasures.

```
var i;
thread {
     if (js0.b[5]) {
          i=5;
          while (i>0) {
               b[0]=1;
               delay(2);
               b[0]=0;
               delay(2000);
               i--;
          }
     }
}
```

This code will trigger the release of 5 countermeasures with a delay of 2 seconds between each countermeasure. Countermeasures will be launched as soon as button number 5 of joystick 0 is pressed. They will continue to be launched even if the button is immediately released.

4.4 Trimming

It is sometimes necessary to trim the controls of an aircraft so that straight and level flight can be maintained with the joystick in a central position. The trim position may change due to changes in air speed or other factors. The program below trims the joystick according to the current joystick position.

```
var trimx;
var trimy;
# the original values of trimx and trimy
var ox, oy;
if (firstscan) {
     #center position is zero
     trimx=128;
     trimy=128;
     ox=128;
     oy=128;
}
#check for trimming button pressed
if (js0.b[5]) {
     trimx=128-js0.a[0]+ox;
     trimy=128-js0.a[1]+oy;
} else {
     ox=trimx;
     oy=trimy;
}
#check for reset button pressed
if (js0.b[6]) {
     #reset center position is zero
     trimx=128;
     trimy=128;
     ox=128;
     oy=128;
}
a[0]=js0.a[0]-trimx+128;
a[1]=js0.a[1]-trimy+128;
```

4.5 Waiting for Release of a Button

Assume you want to launch exactly one missile with a button press, and the simulation software launches missiles in sequence according to whether the button is depressed or not.

```
thread {
     #wait for first press of the button
     wait(js0.b[1]);
     #wait for release
```

```
        wait(!js0.b[1]);
        #press virtual button
        b[0]=1;
        #and release after 1 second
        delay(1000);
        b[0]=0;
    }
```

Due to the input driver system it should be possible to omit the delay, both the press and release should be reported. However, the simulation software may not work in entirely the same way.