Programming gameport drivers
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

1. A basic classic gameport
~~~~~~~~~~~~~~~~~~~~~~~~~~~

If the gameport doesn't provide more than the inb()/outb() functionality, the code needed to register it with the joystick drivers is simple:

```
	struct gameport gameport;

	gameport.io = MY_IO_ADDRESS;
	gameport_register_port(&gameport);
```

Make sure struct gameport is initialized to 0 in all other fields. The gameport generic code will take care of the rest.

If your hardware supports more than one io address, and your driver can choose which one to program the hardware to, starting from the more exotic addresses is preferred, because the likelihood of clashing with the standard 0x201 address is smaller.

Eg. if your driver supports addresses 0x200, 0x208, 0x210 and 0x218, then 0x218 would be the address of first choice.

If your hardware supports a gameport address that is not mapped to ISA io space (is above 0x1000), use that one, and don't map the ISA mirror.

Also, always request_region() on the whole io space occupied by the gameport. Although only one ioport is really used, the gameport usually occupies from one to sixteen addresses in the io space.

Please also consider enabling the gameport on the card in the ->open() callback if the io is mapped to ISA space - this way it'll occupy the io space only when something really is using it. Disable it again in the ->close() callback. You also can select the io address in the ->open() callback, so that it doesn't fail if some of the possible addresses are already occupied by other gameports.

2. Memory mapped gameport
~~~~~~~~~~~~~~~~~~~~~~~~~

When a gameport can be accessed through MMIO, this way is preferred, because it is faster, allowing more reads per second. Registering such a gameport isn't as easy as a basic IO one, but not so much complex:

```
	struct gameport gameport;

	void my_trigger(struct gameport *gameport)
	{
		my_mmio = 0xff;
	}

	unsigned char my_read(struct gameport *gameport)
	{
		return my_mmio;
```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

```
        }

        gameport.read = my_read;
        gameport.trigger = my_trigger;
        gameport_register_port(&gameport);
```

3. Cooked mode gameport
~~~~~~~~~~~~~~~~~~~~~~~~

There are gameports that can report the axis values as numbers, that means
the driver doesn't have to measure them the old way - an ADC is built into
the gameport. To register a cooked gameport:

```
        struct gameport gameport;

        int my_cooked_read(struct gameport *gameport, int *axes, int *buttons)
        {
                int i;

                for (i = 0; i < 4; i++)
                        axes[i] = my_mmio[i];
                buttons[i] = my_mmio[4];
        }

        int my_open(struct gameport *gameport, int mode)
        {
                return -(mode != GAMEPORT_MODE_COOKED);
        }

        gameport.cooked_read = my_cooked_read;
        gameport.open = my_open;
        gameport.fuzz = 8;
        gameport_register_port(&gameport);
```

The only confusing thing here is the fuzz value. Best determined by
experimentation, it is the amount of noise in the ADC data. Perfect
gameports can set this to zero, most common have fuzz between 8 and 32.
See analog.c and input.c for handling of fuzz - the fuzz value determines
the size of a gaussian filter window that is used to eliminate the noise
in the data.

4. More complex gameports
~~~~~~~~~~~~~~~~~~~~~~~~~~

Gameports can support both raw and cooked modes. In that case combine either
examples 1+2 or 1+3. Gameports can support internal calibration - see below,
and also lightning.c and analog.c on how that works. If your driver supports
more than one gameport instance simultaneously, use the ->private member of
the gameport struct to point to your data.

5. Unregistering a gameport
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Simple:

```
gameport_unregister_port(&gameport);
```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

6. The gameport structure
~~~~~~~~~~~~~~~~~~~~~~~~~~

```
struct gameport {

	void *private;
```

A private pointer for free use in the gameport driver. (Not the joystick driver!)

```
	int number;
```

Number assigned to the gameport when registered. Informational purpose only.

```
	int io;
```

I/O address for use with raw mode. You have to either set this, or ->read() to some value if your gameport supports raw mode.

```
	int speed;
```

Raw mode speed of the gameport reads in thousands of reads per second.

```
	int fuzz;
```

If the gameport supports cooked mode, this should be set to a value that represents the amount of noise in the data. See section 3.

```
	void (*trigger)(struct gameport *);
```

Trigger. This function should trigger the ns558 oneshots. If set to NULL, outb(0xff, io) will be used.

```
	unsigned char (*read)(struct gameport *);
```

Read the buttons and ns558 oneshot bits. If set to NULL, inb(io) will be used instead.

```
	int (*cooked_read)(struct gameport *, int *axes, int *buttons);
```

If the gameport supports cooked mode, it should point this to its cooked read function. It should fill axes[0..3] with four values of the joystick axes and buttons[0] with four bits representing the buttons.

```
	int (*calibrate)(struct gameport *, int *axes, int *max);
```

Function for calibrating the ADC hardware. When called, axes[0..3] should be pre-filled by cooked data by the caller, max[0..3] should be pre-filled with expected maximums for each axis. The calibrate() function should set the sensitivity of the ADC hardware so that the maximums fit in its range and recompute the axes[] values to match the new sensitivity or re-read them from the hardware so that they give valid values.

```
	int (*open)(struct gameport *, int mode);
```
~~~~~~~~~~~~~~~~~~~~~~~~~~

Open() serves two purposes. First a driver either opens the port in raw or in cooked mode, the open() callback can decide which modes are supported. Second, resource allocation can happen here. The port can also be enabled here. Prior to this call, other fields of the gameport struct (namely the io member) need not to be valid.

```
	void (*close)(struct gameport *);
```

Close() should free the resources allocated by open, possibly disabling the gameport.

```
	struct gameport_dev *dev;
	struct gameport *next;
```

For internal use by the gameport layer.

```
};
```

Enjoy!