Chapter 20

Struct examples

This section is just a bunch of examples with mostly the `Vector3` and `Color` structs. Nothing we'll really need later – feel free to skim and skip parts you hate.

20.1 More Vector3, Color code

This first thing is a rewrite of the old movement code, to show off `Vector3`s. It doesn't run any better, but it might look a little nicer.

Since the Unity system thinks position is a `Vector3`, it might make sense for us to use one ourselves. Instead of declaring `x`, `y` and `z` for our movement variables, like we did before, we can declare `Vector3 pos;`.

This is the move&wrap code, rewritten with a Vector3 replacing `float x` we had before:

```
Vector3 pos;

void Start() {
  pos.x=-7; pos.y=2; pos.z=0;
  transform.position=pos;
}

void Update() {
  pos.x+=0.1f;
  if(pos.x>7) pos.x=-7;

  transform.position = pos;
}
```

Because `pos` is a `Vector3`, it's easy to think it is our position. But we still need to assign it to `transform.position`, in the last line.

A neater example of this is rewriting the "each lap is a random y" version. We needed global x and y for that before. Now declaring `Vector3 pos;` gives us both. But otherwise the code is the same:

```
Vector3 pos;

void Start() { // no changes in start
  pos.x=-7; pos.y=2; pos.z=0;
  transform.position=pos;
}

void Update() {
  pos.x+=0.1f;
  if(pos.x>7) { // wrap-around
    pos.x=-7;
    pos.y = Random.Range(-3.0f, 3.0f); // <-- new line
  }

  transform.position = pos; // copies the x and y we changed
}
```

A neat thing is it's like we assign `y` for free. The last line copies all of `pos`, including our each-lap changes to `pos.y`.

This next example is a little different. I'd like to cycle through three different colors. To get the colors, I'll declare three `Color` variables. That's the main point of this example – we could do this by declaring 9 floats (rgb for each color,) but it's nicer to use 3 Colors, now that we have structs.

We could set them in the program, but we may as well just do it through the Inspector. Here's the first part, with a function to set us to the correct color:

```
public Color c0, c1, c2; // set these using Inspector
int curColNum=0; // 0, 1 or 2. Which Color we're on now
int delay=40;

void Start() {
  setToCurCol();
}

void setToCurCol() {
  Color cc;
  if(curColNum==0) cc=c0;
  else if(curColNum==1) cc=c1;
  else cc=c2;
  GetComponent<Renderer>().material.color=cc;
}
```

`setToCurCol` is one of those sloppy global-using functions for just this program. It uses the compute-then-use idea: the cascading `if` gets the correct color into temporary Color variable `cc`, then the last line uses `cc` to set our color.

I think it's pretty cool how we can assign color structs around like we would normal variables.

The Update part isn't special. It just moves `curColNum` through 0,1,2,0 . . . , with a delay:

```
void Update() {
  delay--;
  if(delay<=0) {
    delay=40;
    curColNum++;
    if(curColNum>2) curColNum=0;
    setToCurCol();
  }
}
```

This part doesn't even use colors. It just slow spins a simple int through the 0,1,2 color numbers. But that's part of the point – most of the work is just moving numbers around.

I want to grow the example a little, to show that `Color` variables are still just variables, and everything is under our control. I'll change the last color to be random, so it shows `c0`, `c1` then a random color. I also want it so pressing the space-bar picks new random `c0` and `c1` colors (we'll see it spin through those exact new colors, until we press space again).

This is the whole thing redone, including my random color function from before:

```
public Color c0, c1; // 2 colors in the sequence (may change)
int curColNum=0; // which Color we're on now: 0, 1 or 2
int delay=40;

void Start() {
  setToCurCol();
}

float rand01() { return Random.Range(0,1.0f); }

Color randCol() {
  Color cc; cc.r=rand01(); cc.g=rand01(); cc.b=rand01(); cc.a=1;
  return cc;
}
```

```
void setToCurCol() {
  Color cc;
  if(curColNum==0) cc=c0;
  else if(curColNum==1) cc=c1;
  else cc=randCol(); // <- last color is random
  GetComponent<Renderer>().material.color=cc;
}

void Update() { // no change in first part:
  delay--;
  if(delay<=0) {
    delay=40;
    curColNum++;
    if(curColNum>2) curColNum=0;
    setToCurCol();
  }

  // pressing space rerolls the first two colors:
  if(Input.GetKeyDown(KeyCode.Space)) {
    c0=randCol();
    c1=randCol();
  }
}
```

I like this because it's mostly just old tricks. Because `c0` and `c1` are variables, we can change them (the last part, using the space bar.) Variables are better than constants.

`rand01()` is a typical "helper" function – it makes `randCol` easier to write.

20.2 More built-in functions

Most built-in structs also have built-in functions using them. This section uses two fun ones for `Vector3's`: `Distance` and `MoveTowards`

There's one very funny thing about them: `Distance` is really `Vector3.Distance`, same with `MoveTowards` really being `Vector3.MoveTowards`. They're in the namespace `Vector3`.

We've seen functions in namespaces, like `Random.Range`. But this is funny because `Vector3` is already a struct. It so happens you can double-use them. Many struct names are re-used as namespace names.

The idea is, suppose you want to find built-in functions using `Vector3`s. You can type `Vector3`-dot, using it as a namespace, and look at the options in the pop-up.

20.2.1 Distance

`Distance` takes two `Vector3` points and tell you how far apart they are:

```
void Start() {
  Vector3 aa = new Vector3(0,2,0);
  Vector3 bb = new Vector3(0,5,0);
  float d = Vector3.Distance(aa,bb);
  print( d ); // 3
}
```

(0,2,0) is 3 away from (0,5,0). It also uses proper Pythagorean theorem math for diagonals:

```
  Vector3 aa = new Vector3(10,0,10);
  Vector3 bb = new Vector3(13,4,10);
  float d = Vector3.Distance(aa,bb);
  print( d ); // 5 (3 right and 4 up -- pythagorean says 5)
}
```

It's a pure math function. The only funny part is the way it takes two Vector3 structs, and returns one float (which is exactly what it should do, but can feel funny.)

20.2.2 MoveTowards

`MoveTowards` is named for what people usually do with it – move themselves towards some target. It's useful because it will even move the correct distance diagonally. I like it because it's really a pure function that looks like it changes you, but doesn't, but does something really nice once you figure it out.

We already know it's inside `Vector3`, so the full name is `Vector3.MoveTowards`. The heading is:

```
Vector3 MoveTowards(Vector3 current, Vector3 target, float maxDistanceDelta)
```

Remember, the parameter names are just made up: `current` and `target` are good descriptive names, but don't let `maxDistanceDelta` scare you. It's one number (a `float`,) for how far we want to move.

A sample call: `Vector3 p3 = Vector3.MoveTowards(p1, p2, 0.1f);`. It sets `p3` to a point 0.1 away from `p1`, towards `p2`. That's very math-y – three inputs, changes nothing; gives you one answer, which you can use however you want.

Moving yourself looks like this: `pos = MoveTowards(pos, targ, 0.1f);`. Notice `pos` is on both sides. It computes a point from `pos` which is 0.1 closer to `targ`. Then copies it to `pos`. It's the same "change-me" logic as `n=abs(n);` to make `n` positive.

If you run that over and over, `pos` will eventually hit `targ` (it won't even overshoot.)

Here's a short program using it. It moves the Cube from a random spot, to the center of the screen, over and over. The first line does the movement. The `if` is for resetting. I'm assuming the Camera is in a front-view:

```
public Vector3 pos, targ; // targ starts at 000

void Update() {
  pos = Vector3.MoveTowards(pos, targ, 0.1f);

  // random teleport when we get there:
  if(pos.x==targ.x && pos.y==targ.y) {
    pos.x = Random.Range(-7.0f, 7.0f);
    pos.y = Random.Range(-5.0f, 5.0f);
  }
  transform.position=pos;
}
```

If we set `targ` in the Inspector to (7,0,0) this will move diagonally to the center of the right edge, random teleport, and repeat.

The `if` is just a cheap way of checking when `pos` and `targ` are equal. We could also check `z`'s, but they're always 0, so there's no need.

The teleport inside the `if` might randomly put us in the same spot, or very close. But it's fine for now.

Another fun thing we could do is change the `if` to check for close-enough, now that we have `Distance`:

```
if(Vector3.Distance(pos, targ)<0.3f) { // teleport reset
```

Most people would read that as "if the distance between `pos` and `targ` is small enough." It's a function inside of an `if`, but I think it looks fine that way. We could rewrite it to compute distance ahead of time. This might be good if we want to use `dist` for something else later:

```
float dist=Vector3.Distance(pos, targ);
if(dist<0.3f) { // reset
```

Just for fun, we could occasionally change the target position. This adds a lap-counter and randomizes the target when it hits 5:

```
public Vector3 pos, targ;
int laps=0;
```

```
void Update() {
  pos = Vector3.MoveTowards(pos, targ, 0.1f);

  // random teleport when we get there:
  if(Vector3.Distance(pos, targ)<0.3f) {
    pos.x = Random.Range(-7.0f, 7.0f);
    pos.y = Random.Range(-5.0f, 5.0f);

    // change target every 5 trips:
    laps++;
    if(laps>=5) {
      laps=0;
      targ.x = Random.Range(-7.0f, 7.0f);
      targ.y = Random.Range(-7.0f, 7.0f);
    }
  }
  transform.position=pos;
}
```

The last thing I like about `MoveTowards` is you have to think about the units of the third input. 0.1 looks a little like it means 10% – like it moves you 10% of the way from `p1` to `p2`. There's actually a different function which does that. For `MoveTowards`, the third input is the distance. We've been moving `x` sideways by 0.1 each update, and that looked fine, so distance 0.1 here seems right.

20.2.3 Color namespace

`Color` is also re-used as a namespace. `Color`-dot brings up a list of pre-defined color globals: black, blue, cyan .... `Color.blue` is just a short, descriptive way to get (0,0,1). Serious programs don't use them – they custom set all Color values, but the Color globals are nice for quick, simple colors.

A really confusing one is `Color red = Color.red;`. That declares a variable named `red`, and sets it to (1,0,0) using the global for `red` in the `Color` namespace. As usual, `red` is just a variable. We can change it afterwards, and it doesn't have to be red.

20.3 Transform struct

You may have realized this: `transform` is a struct variable. It looks like this:

```
public struct Transform {
  public Vector3 position;
  public Vector3 localScale;
  ...
  public string name;
```

```
}

Transform transform; // <- our transform variable being declared
```

Like `Cow cow1;`, the variable name is `transform`, and the type is `Transform`. That seems confusing, to have the variable be just lower-case the type, but it's a style people understand.

This explains why we're been using `transform.position` and `transform.localScale`. Those two are just fields in our `transform` variable. This also explains when I used `transform.position.x` – it's just using dots to look at a nested field.

There are obviously some extra magic things going on: the system automatically declares `transform` for us, and keeps `transform.position` and `transform.localScale` in synch with the object we're on. But other than than, they are just regular Vector3's.

Like any other struct, typing just transform-dot will show us the fields. Most are pretty technical, but a nice one is `name`. It's the string, and the actual name that we see. We can change it:

```
void Start() {
  print("Old name: " + transform.name);
  transform.name = "abc" + Random.Range(1,1000);
  // look in Inspector/Hierarchy - name has changed
}
```

This is pretty much the same thing as changing `c1.name` for a `Cow`. The difference, the system tracks `transform`, and updates the name we see in Hierarchy.

20.3.1 Special errors; pull-out trick

This is a horrible, horrible error you can get with some of the built-ins. Anytime you "reach-through" a field in a built-in, you can read, but not write. An example:

```
float x1 = transform.position.x; // legal
transform.position.x = x1; // error
```

The error will be: *Cannot modify a value type return value of 'UnityEngine.Transform.position'. Consider storing the value in a temporary variable.*

Normal structs wouldn't have this error. The second line would be 100% legal if we were just using them. You get the error because `transform` isn't exactly a regular struct. Unity is using a trick, which we'll see later.

But this is a common C# error. A lot of people use that trick, so it's sort-of good we at least see it now.

The safe way to play with these is to pull out the struct, change it, then put it back:

```
Vector3 p = transform.position;
p.x+=0.1f;
transform.position = p;

Color cc = GetComponent<Renderer>().material.color;
cc.b+=0.1f; if(cc.b>1) cc.b=1;
GetComponent<Renderer>().material.color = cc;
```

You're allowed to copy the entire thing at once. You just can't change the subparts one-by-one. That's just the crazy rule.

This is partly why I've been making `Vector3 pos;` as a global, filling it in, and copying it into `transform.position`. I also think that's a nice way to do things, but it's also because `transform.position.x += 0.1f;` should be legal, but isn't.

20.4 Rigidbodies

Most game engines have something they call *physics*. It means you can set up an object to automatically act like a real ball. You can push it once, and it rolls, falls, bounces, and eventually stops. It does all of that automatically, with no script needed.

The reason we care is that we can jump in with programming to change it: the speed is a `Vector3`, and there's a standard programming way to check when we hit something.

Some fun background: the whole name is rigidbody physics, but no one uses it. A rigidbody is like a pool ball – something that bounces and rolls, but never changes shape. Standard physics equations predict those pretty well, and the computer can run them pretty quickly.

The other type is soft body physics – things which can smush and bend. For real everything is like that – even steel balls flex a bit when they hit something. But the math is too hard, so games assume most things are rigid bodies.

20.4.1 Simple physics set-up

To start, let's just get a Cube that falls and bounces around. We'll need a Cube with no script (can remove the script, or make a fresh cube) and a front view Camera (the one showing x and y movement,) since gravity makes you fall on `y`. We'll also eventually need a floor and walls (or it won't stay on the screen.)

To let the Unity system know it should auto-move the Cube, select it, find `Component` on the top bar, and select `Component->Physics->Rigidbody`. The Cube's Inspector should now have a mini-panel named Rigidbody. Just in case,

make sure the ball is somewhere the camera can see it, maybe a little near the top, like (0,4,0), and Play. It should fall, hit the floor, and stop.

To see it bounce more, we can tilt the Cube so it doesn't hit flat. Giving it a z-rotation of 10-30 degrees should make it fall and bounce sideways (just type 20 into the z part of Rotation in the Inspector panel.) It should rock just a little, then stop.

The system has a way to set an object's slipperiness and bounciness. As you can see, the default setting is like a bean-bag – not bouncy at all. To give ourselves more to work with, we can make it very bouncy. It takes two steps. We have to make "Bouncy," then apply it to the Cube.

Down in the Assets panel, select `Create->PhysicMaterial`. That should create something with a green pictures of a Bounce, named `NewPhysicMaterial`, asking you to rename it. If you like, name it Bouncy (but the name won't matter.)

Select the new PhysicMaterial and look at its Inspector. The Bounciness setting is a 0-1 percent for how much it will bounce back. Set `Bounciness` to around 0.9 or so. Then set the `BounceCombine` dropdown to Maximum.

We can use that to make anything be bouncy. Drag that PhysicMaterial onto the Cube. Play should now have the cube really bounce and roll around for a while. It will fall off the edge unless you put some walls there (back in the section about setting up a nicer Scene.) Just in case, the actual location of the PhysicMaterial is in the Cube, under BoxCollider (pop it open) in the Material slot (starts with `[none(Physic Material)]`). It will have your new PhysicMaterial if it worked.

That's a lot. I'll sum up, but this stuff is also easy enough to look up, if you know the terms:

- Add a Rigidbody component to a Cube, with no scripts. Play should have it fall.
- Add at least a floor, if you don't have one (a wide, deep, short Cube,) so we can see it bounce. Rotate the Cube slightly, so the bottom is tilted and it bounces to one side.
- Create a PhysicMaterial. Set bounciness=0.9; BounceCombine=Maximum. Drag the PhysicMaterial onto the Cube. Should now bounce more.

Plus, if you have trouble with this, there are lots of Unity places to read about these, and what some of the other settings do.

20.4.2 Playing with velocity

The "physics" system automatically moves us, storing our current speed in a `Vector3`. We can't see it in the Inspector, but we can in code, and we can

change it. It's `GetComponent<Rigidbody>().velocity;`.

It takes a while to get used to code plus the automatic system. Normally, when you press Play, `velocity` is (0,0,0), but gravity decreases the `y`. Suppose we hit the floor and bounce right, `velocity.y` flips to positive and `velocity.x` changes from 0 to positive. If we bounce off the right-side wall, `velocity.x` flips to negative.

The built-in system automatically does the math, and uses `velocity` to remember how fast it's going.

If we set `velocity` once ourselves, at the start, the automatic system will still move and change it. So it's like we're firing the object. We can fire it straight up, but the system will slow it down, then make it fall.

To set it, we need to know what the numbers mean. x, y and z are the same directions as before, but the numbers are in units/second. Setting `velocity` to `(7,0,0)` will have us move 7 units right each second – about 2 seconds to cross our -7 to 7 screen. Except there's friction, gravity and bouncing that can slow or stop it.

This will launch us up and a little right. Positive y is up, and 10 is enough go up for about a second before gravity takes over:

```
void Start() {
  Vector3 vel;
  vel.x=2; vel.y=10; vel.z=0; // good push up, a little to the right
  GetComponent<Rigidbody>().velocity = vel;
}
```

We can make that a little nicer by having the space bar give us a random extra pop:

```
void Update() {
  if(Input.GetKeyDown(KeyCode.Space)) {
    Vector3 vel;
    vel.x=Random.Range(-5.0f, 5.0f); // random left/right
    vel.y=Random.Range(5.0f, 15.0f); // up 5-15
    vel.z=0;
    GetComponent<Rigidbody>().velocity = vel;
  }
}
```

This replaces the old `velocity` with a new random one. It's like we magically change direction.

A fun effect is to give us a fake bouncy floor. I'll say when anything gets below y=-3, it gets that same random upward pop (make sure your real floor is below this, or it won't work. Mine is at -5):

```
void Update() {
  if(transform.position.y<-3) {
    Vector3 vel;
    vel.x=Random.Range(-5.0f, 5.0f);
    vel.y=Random.Range(5.0f, 15.0f);
    vel.z=0;
    GetComponent<Rigidbody>().velocity = vel;
  }
}
```

This gives completely unrealistic bounces. If you were falling really slowly, it might roll the maximum up speed, and +5 or -5 for sideways. It's because of `=` vs. `+=`. The same as usual, using `=` doesn't respect the old value.

A variation is to give an upwards push, using `+=` to add a little each update when it gets low. That will gradually slow us, then push us up, giving an effect like bobbing in water:

```
void Update() {
  if(transform.position.y<0) {
    Vector3 vel = GetComponent<Rigidbody>().velocity;
    vel.y+=0.3f; // number is trial&error
    GetComponent<Rigidbody>().velocity=vel;
  }
}
```

Since it takes a while to reverse direction, I moved it up to y=0. Adding 0.3 was trial&error. Any slower and it doesn't beat gravity, and just falls really slowly.

There are a lot of similar things like this we can do. I like them because the physics system is a really strange beast, but `velocity` is just a regular `Vector3`, which we can change the in usual ways.

20.4.3 Collision callback

The usual term for something that could happen at any time is an **event**. A keypress is a typical one.

There are two basic ways of getting events. One way is a command that you have to write, which checks for it. `if(Input.GetKeyDown(KeyCode.A))` is an example of that. That's offiicially called **polling**. I like that name, because it sounds like what you do – you ask every Update, like you're taking a poll.

The other way is having a callback function. You write a function and register it somehow so that when the event happens, it automatically runs that function.

The physics system tells you about hits using a callback. If you write a function with the exact name and heading `void OnCollisionEnter()`, it's automatically called when you hit something.

Here's a very short collision callback. It makes you a little smaller after each hit. There's no Start or Update (you can also have them, but don't need them):

```
float sz=1; // normal size

void OnCollisionEnter() {
  sz-=0.1f; if(sz<0.1f) sz=0.1f;
  transform.localScale = new Vector3(sz,sz,sz);
}
```

The neat thing about it is how ordinary it is. The heading looks the same as any other do-something function. It uses a global, and sets size, both in the regular way. The only thing special is how it gets automatically called. If it was in Update, it would very quickly be called enough to shrink all the way. Since it's in `OnCollisionEnter`, it take 9 bounces for it to completely shrink.

Here's another one that uses a counter to redrop you after the 4th bounce. It brings you back to the upper middle, with a random sideways speed:

```
int bounces=0;

void Start() { resetDrop(); }

void OnCollisionEnter() {
  bounces++;
  if(bounces>=4) {
    bounces=0;
    resetDrop();
  }
}

void resetDrop() {
    transform.position = new Vector3(0,5,0);

    Vector3 toss; toss.z=0; toss.y=0;
    toss.x=Random.Range(-4.0f, 4.0f);
    GetComponent<Rigidbody>().velocity=toss;
}
```

I used a separate `resetDrop` function mostly to show `OnCollisionEnter` can call other functions.

20.4.4 The Collision struct

An obvious problem with `OnCollisionEnter()` is it doesn't tell you what you hit. Along with that, it might also be nice to know what part of us hit, and other stuff. There's a better version of `OnCollisionEnter` which tells you that.

To make it look nice, Unity created a struct who's only purpose is to hold collision data, named `Collision`. Here's a partial listing for it:

```
struct Collision {
  public Transform transform; // what we hit
  public Vector3 relativeVelocity; // combined hit speed
  ...
}
```

The idea for making this struct is simple – we've got a bunch of related variables, so put them in a struct. The struct itself is full of complicated physics math stuff, but we'll just ignore the ones we don't understand.

The name of the first field seems confusing. The trick is that names inside structs never have anything to do with regular variable names. They just reused the word `transform` since they felt like it. If `A` was a Collision variable, `A.transform` is what we hit, and regular `transform` is us.

To get collision data, write the collision callback with a `Collision` parameter:
`void OnCollisionEnter(Collision col) { ...`

This tells the system you want it to send that extra data. It's a bit of Unity/C# magic (obviously, it involves overloaded functions.) The key thing is that it's still automatic. You will never have to create a `Collision` variable. You just read it and use whichever parts you want.

Here's a simple use, which just prints the name of whatever we hit:

```
void OnCollisionEnter(Collision col) {
  string myName = transform.name;
  string hitName = col.transform.name;
  print( myName + " ran into " + hitName);
}
```

The first two lines look up our name, and the name of what we hit. `col.transform.name` is just using the struct-in-a-struct rule – use as many dots as you need to get to what you want.

This might print only "Cube ran into Cube" a few times. If the floor and walls were renamed, and you added the "fling" in Start, you could get things like "Cube hit floor" and "Cube hit Rwall".

Not important to know, but: this only gets called on fresh hits. It won't print if you're just sitting on the floor, or even if you rock on a corner and fall

back. You have to break contact, then hit again.

Here's something more useful, that tries to climb the walls. It bounces normally off the floor, but hitting anything else shoots it upwards. It assumes the floor is named "floor," and works better if you use Start to give a sideways push:

```
void OnCollisionEnter(Collision col) {
  if(col.transform.name != "floor") {
    Vector3 vel = GetComponent<Rigidbody>().velocity;
    vel.y=Random.Range(5.0f, 10.0f);
    GetComponent<Rigidbody>().velocity = vel;
  }
}
```

The middle of the `if` is using the pull-out trick to change just our y-speed. We when bounce, it will be the proper sideways amount, since the code leaves x and z alone. We'll just magically launch into the air.

The only field I used from `Collision` was what we hit (`transform.`) The rest either have tricky rules, or tricky math. But just for fun, we can test another. Looking inside, using col-dot, we can see there's a `Vector3` field named `impulse`.

I don't know what impulse stands for, but I know what a Vector3 is, so we can print it:

```
void OnCollisionEnter(Collision C) {
  Vector3 imp = C.impulse;
  print( imp.x + ", " + imp.y + ", " + imp.z );
}
```

Those numbers didn't help me understand any better, but they were legal and ran. The code also could have used `C.impulse.x`. I just felt like pulling it into a different Vector3, first.

20.4.5 Misc event notes

In general, any event can be handled either way – polling or a callback – and different systems mix&match. For example, some C# systems handle keys using a callback. It looks a bit like this (I'm making up an example where A and D move us):

```
void OnKeyPress(KeyCode key) { // not legal in Unity
  if(key == KeyCode.A) moveDir=-1;
  else if(key == KeyCode.D) moveDir=+1;
  else moveDir=0;
}
```

When you press `A`, the system would automatically call `OnKeyPress(KeyCode.A)`. This would replace checking for key-presses using an `if`. Update wouldn't have anything about key presses.

Most systems give you a command to check, or a callback, but not both. In a system like that, there wouldn't even be an `Input.GetKeyDown` function. You'd have to write it like above.

Unity knows about callbacks through the function names. Most other systems have you register the function, so you can name it whatever you want. In a system like that, you might name your function `void bounceAction(Collision col)`. Then register it in Start with a magic line like `collision.enter += bounceAction;`.

Whenever you want to check for something happening, you have to figure out, is it done with polling, or a callback? If it's with polling, what's the command? If it's a callback, you have to look up how they want you to write the function, and how to register it as the callback.

But then the rest is just programming.