ORAL ORCS

A Framework for Roll20

(Not affiliated with Roll20)

ORAL ORCS is a framework for Roll20, comprised of two pieces: the "OnyxRing API Library" (ORAL) and the "OnyxRing Client Script" (ORCS). Like most frameworks, much of ORAL ORCS is an abstraction of underlying services. It adds support for modern features of the JavaScript language and simplifies Roll20 development. With it, code is easier to write, easier to read, and easier to understand. You can do more, with less.

A tenet of the framework is passivity, which is another way of saying that adding it to an existing project introduces no behavioral changes until you intentionally choose to use its features. The exception to this is client-side support for ES6 Promises and other asynchronous language elements like `async/await`, `setTimeout()`, and `setInterval()`. Simply enabling ORCS in your Sheetworker will "light these up." The risk of enabling these by default, rather than by opt-in, is low since these features do not work in vanilla Roll20 so should have a low prevalence in the wild.

Me, OnyxRing, and ORCS

Let's get introductions out of the way: I'm Jim, and I like to make stuff.

I've been working in software development for more than half my life now and have a dozen or so languages under my belt. JavaScript is, of course, numbered among these as is the One True Language (C++.) I'm a casual writer and a casual artist. No surprise, I game: card, table-top, video, and roll playing games. I design these as well as play them because, as I said, I like to make stuff.

I've been floating around the internet brandishing the OnyxRing username, a reference to a piece of Interactive Fiction I wrote years ago. It's unlikely you've seen my work on the ORLibrary for the Inform 6 language, but if you have, you'll recognize that I make liberal use of the "OR" prefix. In part, I do this to avoid naming collisions with other scripts, but it's also a sort of mini signature.

The obligatory Dad joke:

> *"You might say I put the 'OR' in 'ORCS.'"*

Assumptions of This Guide

While it is possible to "hack" the framework and pull out some its non-pro features, this guide assumes developers are using it *ut totem*. Doing so requires access to the Roll20 API, something restricted to Pro subscribers.

It also assumes readers are well-versed with Roll20 and its interfaces, are acquainted with object-oriented programming generally, and have a solid familiarity with JavaScript specifically.

Installation and Setup

We install ORAL ORCS as we would any other API script:

1. Go to the appropriate GitHub project and grab the latest version of the script:

   https://github.com/onyxring/Roll20OralOrcs

2. Paste the script into our Roll20 Game's "API Scripts" settings page. For simplicity, this would occupy the first script slot, but as long as it precedes any use of framework features, this is not strictly necessary.

At this point, the API Library portion of the framework (ORAL) is ready for use in our personal API scripts. One additional step will enable the Client Script part of the framework (ORCS) for our Sheetworkers:

3. Cut and paste the following single line into our Sheetworker, immediately after our worker's script tag:

```
ç=[],µ=on;µ('sheet:opened',i=>getAttrs(['Ω'],v=>{(0,eval)(v.Ω);}));on=(e,c)=>ç.push([e,c]);
```

To verify that things are working as expected, check the browser's JavaScript console after a character sheet is opened for this text (note: the version may differ):

```
ORCS (version 0.1) enabled
```

ORCS Gibberish

The line of code which includes ORCS is cryptic and could be mistaken as being obfuscated. That is not the intent.

No secrets here.

The justification behind this enigmatic line is simply that it reduces the number of characters and provides an easy, paste-this-one-liner-and-get-all-the-framework-goodness approach to enabling client-side functionality.

For the curious, a detailed examination of this line, and what it does, appears in Appendix A.

Walkthrough

In the following sections, we cover the highlights of both ORAL and ORCS in tutorial fashion. These sections are intended to guide the readers through select examples and get them up and running, without being distracted by the more nuanced portions of the framework.

A more specific, technical review of the framework and all methods it contains is housed in the "Dry Details" section of this guide.

ORAL

ORAL is the API Library portion of the framework and it runs squarely on the Roll20 servers. It embraces object-oriented methodologies, so using it often involves creating an instance of an object, then calling methods on said object. Using ORAL doesn't remove the underlying API commands you're familiar with; they are all still there, and you are free to mix and match API functions with ORAL functions as you see fit.

Processing Commands with orProcessor

One core mechanism for communication between API scripts and the Sheetworkers is via Roll20's chat services. Players type commands in chat, macros place commands in chat, and roll buttons inject commands in chat. To handle these, API developers traditionally hook into events and search chat messages for patterns that represent commands they recognize and know how to react to.

`orProcessor` wraps this ability into a reusable class that developers inherit from. The following code, placed in a script tab following the ORAL ORCS script, creates a processor that parses all chat commands beginning with `!API`. It also registers a chat command named "time."

```
class api extends orProcessor{
    constructor(){
        super("!API");
    }
    time(){
        orSend.all(`Current Time:${new Date().toUTCString()}`);
    }
}
new api();
```

The last line, which "news up" an instance of our processor, is all that is required to turn it on. Don't get hung up on the `orSend.all()` reference above. The framework's `orSend` object has a number of features that we'll cover later, but for now, think of this as equivalent to the traditional `sendChat("text", msg.who)` call.

We can verify that things are working by typing the following command in the chat window:

```
!API time
```

Which will output the current GMT date and time, à la:

OnyxRing (GM): Current Time:Tue, 01 Oct 2020 00:13:38 GMT

Note:

Of course, the username shown here is mine, acting as GM. Other GMs will have their usernames displayed.

Note how the framework automatically tied the command text `time` to the function with the same name. Declaring the function is all it takes

to wire the chat command to the handler. Change one without the other and the framework will complain:

```
!API timeGmt
```

Returns:

(From System): Error: No handler for API message: timeGmt

orProcessor Parameters

The API handler functions take two distinctly new parameters, `pc` and `params`, plus the traditional `msg` argument tacked on as a third if we really need it. Most often we don't:

```
time(pc, params){ orSend.all(params);
```

For rudimentary usage, the `params` argument contains the remaining command text as a string. The above version of our handler produces the following:

```
!API time keeps on slipping, slipping, slipping
```

OnyxRing (GM): keeps on slipping, slipping, slipping

With this approach, the onus is on the developer to parse the string for usefulness. In the above, we've done nothing of the sort and instead just dump it all to the chat pane. Alternatively, and more usefully, we could supply the command with JSON. The framework will detect if the string is valid JSON and, if so, will parse it, passing the resulting object to `params` rather than the string:

```
!API song {"title":"Fly Like an Eagle", "author":"Steve Miller"}
```

A handler for such a song command might look like this:

```
song(pc, params){
    orSend.all(`Song:${params.title}; Band:${params.author}`);
}
```

Which would produce this response:

OnyxRing (GM): Song:Fly Like an Eagle; Band: Steve Miller

It is a best practice to adopt one of these two approaches and stick with it, making all macros pass either JSON or string text; however, if there is a need to support both, either form of the parameter provides the `isObj` property to help you determine how to interpret it. For example:

```
song(pc, params){
    if(params.isObj)
        orSend.all('Song:${params.title}; Band:${params.author}');
    else
        orSend.all(params);
}
```

Readers may have noticed that we skipped over the first `pc` parameter. This is the highly useful, `orCharacter` object, representing the acting character. We cover `orCharacter` later in this guide.

Attribute Events

In addition to chat messages, `orProcessor` also supports attribute "change" handlers. Like the API handlers described above, change event handlers are bound by the framework to the name in the form of: "onChanged_AttributeName."

Consider the following handler, which is triggered when an attribute named "HP" changes:

```
onChanged_HP(pc, obj){
    if(pc.HP<0)
        orSend.all(`${pc.character_name} takes the damage, then dies.`);
}
```

The first parameter passed to the attribute handlers is the framework's `orCharacter` object, which we just saw. We'll cover it in detail next, but here we use it to check the value of the `HP` attribute.

The second parameter, `obj`, is the same one provided to the traditional `on("change:attribute"...)` Roll20 API handler.

Manipulating Characters with orCharacter

The `orCharacter` class simplifies working with characters. Reading and writing character attributes is done against the object itself which simplifies code substantially. For example, a game's "healing" action might add a player-character's Recovery to his or her Hit Points. The following handler, using the framework-provided `orCharacter` object (`pc`) does exactly this:

```
heal(p, pc){
    pc.HP=Number(pc.HP)+Number(pc.REC);
 }
```

As an added benefit, assigning an attribute in this fashion will create it on the character if it doesn't already exist.

`orCharacter` also supports attribute access using strings. The following is equivalent to the above:

```
heal(p, pc){
    pc.["HP"]=Number(pc["HP"])+Number(pc["REC"]);
 }
```

Compare

The single-line"heal" handler shown above is roughly equivalent to the following raw API code:

```
heal(pcId){
    var rec=Number(getAttrByName(pcId,"REC"));
    var hpAttrib=findObjs(JSON.parse({"type":"attribute", "characterid":"${pcId}",
            "name":"HP"}))[0];
    if(hpAttrib==null){
        hpAttrib=createObj("attribute", {name: "HP", current: "0", characterid: pcId });
    }
    var newTotal=Number(hpAttrib.get('current')) + rec;
    hpAttrib.setWithWorker({current:newTotal});
}
```

Repeating Sections

Repeating Sections on a character's sheet are represented as collections off `orCharacter`'s reserved property `repeating`. Members of these collections provide the same object-bound access to repeating attributes that we saw on the `orCharacter` object proper. To demonstrate, assume the following repeating section is defined on the character sheet:

```
<fieldset class="repeating_skills">
    <input type="checkbox" name="attr_isOn" />
    <input type="text" name="attr_name" />
</fieldset>
```

The below contrived example iterate through all character skills, logs the ones that are turned on, modifies their names, and deletes those that are not turned on. Assume the `pc` variable is an `orCharacter` instance:

```
pc.repeating.skills.forEach(skill=>{
    if(skill.isOn=="on") {
        log(skill.name);
        skill.name=skill.name+" (active)";
    }
    else skill.delRow();
});
```

We can also use the `addNew()` method to create a new entry in the Repeating Section:

```
var skill=pc.repeating.skills.addNew();
    skill.isOn="on";
    skill.name="New Skill";
```

Or, more concisely:

```
pc.repeating.skills.addNew({isOn:"on", name:"New Skill"}});
```

Creating orCharacters

Although `orCharacter` objects are automatically passed into many handlers by the framework; there are cases where we need to create `orCharacter` objects explicitly. Rather than "`new`ing" these up, we instead use the factory pattern built into the class definition:

```
var pc=orCharacter.fromId(charId);
```

For convenience, a few other options are available, notably:

```
var pc=orCharacter.fromToken(tokenId);
```

Crafting the Narrative with orTemplates

Unlike most of the ORAL ORCS framework, `orTemplates` doesn't simplify the Roll20 API, rather it provides a tool to simplify what many developers end up doing by hand: creating reusable text, typically HTML, to pass to the chat window.

At their core, templates are about text-substitution. `orTemplates` provides a mechanism to define templates, easily reference them, create default substitution patterns, and override these as needed. The system is recursive, allowing templates to contain other templates, and it is surprisingly powerful.

Registering a new template is quite simple and uses the global `orTemplates` object. We call the `add()` function to register templates, declared as a simple object of property-name/template-value pairs.

```
orTemplates.add({hello:"Hello, {player}!"});
```

Notice the convention for defining text placeholders in a template by wrapping them in braces. Above, we defined one with the name "player".

Getting the final text from our template is straightforward:

```
var text= orTemplates.hello({player:"wizard"});
log(text);
```

Which logs the expected output:

```
"Hello, wizard!"
```

Above we can see that defined templates are turned into functions, taking parameters for text substitution purposes. If a placeholder is not passed as a parameter, `orTemplate` will look at other known templates to see if they can resolve it. If all attempts fail, the substitution text remains unchanged in the return value, braces included.

To better appreciate the power behind `orTemplates`, consider a moderately sophisticated substitution pattern, which demonstrates how it might be used in practice. Since templates are defined with plain-old-JSON-objects, we can define several at once. In this example, we call `add()` a single time, passing in several templates as one object;

however, we could just as easily call `add()` multiple times, passing in templates individually. The `add()` function aggregates all calls together, replacing those already defined if new versions are supplied:

```
orTemplates.add({
    playerWhisper: `<div style="{_whisperStyle}">
        <div style="{_messageStyle}">{message}</div></div>`,
    _whisperStyle: `width:120px;
        background-color:{_backgroundColor};padding:5px;`,
    _messageStyle: `text-align:center;color:{_foregroundColor};`,
    _backgroundColor: `{_blue}`,
    _foregroundColor: `{_white}`,
    _blue: `#005`,
    _gray: `#ccc`,
    _white: `#eee`
});
```

Look closely at the `playerWhisper` template:

```
playerWhisper: `<div style="{_whisperStyle}">
    <div style="{_messageStyle}">{message}</div></div>`,
```

Notice it contains three placeholders, two of which are references to other templates, `_whisperStyle` and `_messageStyle`. If we look those templates up our original declaration, we see they too reference other templates, which resolve to still more templates, finally resolving to color values. That's four levels of text substitution!

Let's create a simple `!API` command to test this:

```
whisper(){
    var msg=orTemplates.playerWhisper({message:"Pssst! Got you."});
    orSend.actor(msg);
}
```

Again, we see that our `playerWhisper` template is presented as a function to which we pass substitution text as parameters. In chat, `!API whisper` produces the following output, demonstrating the style and colors are correctly resolved from other templates.

In our call, we can instead supply other text substitutions, overriding the defaults these templates to resolve to. These may be values, or alternative templates:

```
var msg=orTemplates.playerWhisper({message:"Pssst! Got you.",
        _backgroundColor:"{_gray}", _foregroundColor:"#f00"});
```

With this parameter change, our `playerWhisper` template renders as follows:

Sending Templates to the Client

Using ORCS, `orTemplates` is also available for Sheetworkers and acts in the same way: we create templates using calls to `add()` and resolve templates as needed. In some cases, it may be worthwhile to share all server-defined templates with or worker. This can be done easily with the following assignment on the server:

```
orTemplates.includeOnClient=true;
```

Saying It With orSend

As we've seen in previous examples, `orSend` abstracts the vanilla `sendChat()` API function. The `orSend` object provides four, root level entry points for sending messages through chat. These correspond to the target audience:

```
orSend.all(text);
orSend.character(char, text);
orSend.actor(text);
orSend.gm(text);
```

The first of these, `orSend.all()` pushes a traditional chat, visible to everyone. The second, `orSend.character()` is sent to the player controlling a specific character. This function is like passing the `/w` flag to `sendChat()` and the `char` parameter takes either an `orCharacter` instance or a character ID. The remaining two, `orSend.actor()` and `orSend.gm()`, are both short-hand versions of `orSend.character()` which target the currently acting character or the GM, respectively.

In each of these, the traditional "sendAs" parameter is optional. If not supplied, it is inferred using the most likely option; however, it can be explicitly supplied as a final parameter to each of these functions. The sendAs parameter can be an orCharacter, or a simple string:

```
orSend.actor("All your base are belong to us.", "Grammar-challenged invaders");
```

Sending Templates

As we saw in the orTemplates example, orSend can divvy out HTML, and while it is fine to resolve a template to a string and pass it to orSend as a parameter, as we did in previous examples, orSend is orTemplates-aware and offers a shorthand to make our code more concise:

```
orSend.actor.playerWhisper({message:"This message is for your eyes only."});
```

This is equivalent to:

```
orSend.actor(orTemplate.playerWhisper({message:"This message is for your eyes only."}));
```

Rolling with orDice

Roll20 offers a robust dice rolling system when initiated in chat; however, there are a few scenarios where rolling directly in code is more appropriate. orDice supports a declarative syntax for rolling dice, allows us to interpret rolls in various ways, and feed output into templates. For most rolls, using orDice directly is sufficient, but the class can also be extended to implement more advanced roll combinations.

Expression-Based Rolls

The most straightforward means of rolling dice is by passing in a textual expression:

```
var total=orDice.roll("2.5d12+2").total();
```

Here, we roll 2½ 12-sided dice and add 2 to the total. Note the orDice object supports whole dice and half dice, only. Expressions such as "2.3d12" are not supported.

Roll Result

We can see above, the roll() function returns an object. This "result object" contains all the information about our roll and tools for interpreting it. We leveraged the built-in total() function to get the sum of the individual dice, but if we choose, we can scrutinize the roll in more detail by capturing the result:

```
var result=orDice.roll("2.5d12+2");
```

An array of the individual dice is provided in the `dice` property. Each element in the array includes the number of sides rolled, the actual roll of the die, and any tags that may have been assigned to the die roll. We can run through the list of dice and log their result individually without much trouble:

```
result.dice.forEach(die=>log(die.roll));
```

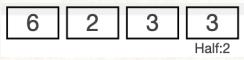
The `total()` function saves us the time of iterating through the array and adding them up ourselves. By default, `total()` just totals the face value of all dice, but more sophisticated calculations are also supported. Consider the HERO system which takes a "damage roll," adds up the pips to determine STUN damage, but determines BODY damage by interpreting rolls of 2-5 as one, rolls of 6 as two, and discarding rolls of 1 entirely:

```
var result=orDice.roll("5d6");
var stun=result.total();
var body=result.total(die=>{
       if(die.roll==6) return 2;
       if(die.roll==1) return 0;
       return 1;
   });
```

The roll's result object also exposes the `diceLineHtml()` function which renders individual dice as HTML for display in the chat window:

```
var html=orDice.roll("3.5d6").diceLineHtml();
orSend.all(html);
```

Depending on the roll, the above will render something like the following in the chat window:



`orDice` uses the "singleDie" template to generate this HTML. We can modify it by replacing it in `orTemplate`:

```
orTemplates.add({singleDie:
    `<div style="float:left;width:35px;">
       <div style="text-align:center; margin:2px; border:#b15b14 solid 1px;
                   border-radius:10px; background-color:#4863B; color:#ccffcc;">
              <div>{faceValue}</div>
       </div>
       {subLabel}
    </div>`});
```

Now our roll produces something more akin to:

ORCS

The OnyxRing Client Script portion of the framework is meant for use in Roll20 Sheetworkers. Like ORAL, it simplifies underlying Roll20 services. ORCS provides a number of features to creators, including client versions of ORAL functionality. API features not commonly available to Sheetworkers. Additionally, it provides improved support for the asynchronous ES6 enhancements to the JavaScript language.

Client Use of orCharacter

The ORCS version of `orCharacter` differs slightly from the ORAL version. By default, the client-side version does not support arbitrary creation. Instead, ORCS provides a single, pre-created instance, to the Sheetworker which represents the active character and is provided in the global variable `pc`. Manipulating characters other than the active one is unsupported on Sheetworkers.

Unleash the ORCS!

Remember to enable ORCS on the Sheetworker by pasting the requisite line into your character sheet, as described in the Installation and Setup portion of this guide.

Another difference is the client-side Roll20 attribute calls `orCharacter` abstracts are asynchronous. Each time an attribute is referenced, the Sheetworker reaches out to the server and retrieves or updates its value. Vanilla Roll20 code relies heavily on simple callbacks, but the ORCS version of `orCharacter` uses modern JavaScript Promises. This allows us to apply the ES6 `Async/Await` constructs to avoid declaring callbacks, especially nested callbacks, contributing to the code complexity commonly referred to as "callback hell."

Duplicating the "heal" example given in the ORAL version of `orCharacter`, but using client-side `Async/Await` syntax, looks like this:

```
async heal(p, pc){ pc.HP = Number(await pc.HP) + Number(await pc.REC); }
```

Take note that each time we call the server to lookup an attribute value, we "`await`" for the response before continuing.

Repeating Sections

The ORCS version of `orCharacter` also has a `repeating` property. As you might expect, this version of `repeating` also makes asynchronous calls to the server and the `Async/Await` pattern serves us in the same way. Here's the example from the ORAL flavor of the Repeating Sections tutorial, revised slightly for asynchronous use in a browser:

```
pc.repeating.skills.forEach(async skill=>{
    if(await skill.isOn=="on") {
          log(await skill.name);
          skill.name=await skill.name+" (active)";
    }
    else skill.delRow();
});
```

Browser Templates with orTemplate

With only a couple points of note, the ORCS version of `orTemplate` is identical to the ORAL version. Keep in mind:

- ORCS, including `orTemplates`, does not exist outside of event handlers, so on the Sheetworker, this alone...

```
orTemplates.add({temp:"hey, {name}"});
```

...will fail, while this will succeed:

```
on("sheet:opened", ()=>{
    orTemplates.add({temp:"hey, {name}"});
});
```

- As mentioned previously, in the ORAL version of `orTemplates` includes a property named `includeOnClient`. Setting this to `true` on the server, will automatically make all templates defined in the API, also available in the Sheetworker.

Of course, it may not be immediately obvious why you would want to use templates in the browser since you can't send them to the Chat window. Except, you can...

orSend too...

`orSend` also works in the browser, with no obvious differences from its server-based analog.
This includes using `orSend` to deliver `orTemplates`.

sendChat() for Sheetworkers

The vanilla Roll20 API function, `sendChat()`, is limited to server-side API code. ORCS brings this to the client. What follows is a call that will be familiar to any API developer; the difference here is that it executes client-side from the Sheetworker:

```
sendChat("Do your homework, yo.", "Your mother");
```

The `callback` and `options` parameters of `sendChat()` are not currently supported by ORCS.

Although `sendChat()` is available, the framework's `orSend` object remains the preferred method for sending chat messages, even from a Sheetworker.

Support for Asynchronous Code

Some readers will have already noticed from previous examples that ORCS seems to "fix" problems with asynchronous code. A current gap in the vanilla Roll20 system is its lack of support for generic callbacks on Sheetworkers. For example, the following code snippet *should* log a character's `STR` every ten seconds:

```
setInterval(()=>{
    getAttrs(["str"],(vals)=>{
        log(vals.str);
    });
},10000);
```

Instead, without the framework, it produces the following error message every ten seconds:

```
Character Sheet Error: Trying to do getAttrs when no character is active in sandbox.
```

As the JavaScript language has advanced, this gap in asynchronous support has translated into an obstacle for ES6 features, such as `async/await`. For example, the following is perfectly valid from a language perspective:

```
function getSingleAttrAsync(prop){ return new Promise((resolve,reject)=>{
    try{getAttrs([prop],(values)=>{ resolve(values[prop]); }); }
    catch{ reject(); }
});}
on("sheet:opened", async ()=>{
    log("str..."+ await getSingleAttrAsync("str"));
    log("dex..."+ await getSingleAttrAsync("dex"));
    log("int..."+ await getSingleAttrAsync("int"));
});
```

Yet, it produces a similar error after the first successful call without ORCS:

```
str...10
Character Sheet Error: Trying to do getAttrs when no character is active
in sandbox.
```

The framework enables `Async/Await`, Promises, `setTimeout()`, and `getInterval()` for use within Sheetworkers. Simply enabling ORCS will turn this support on:

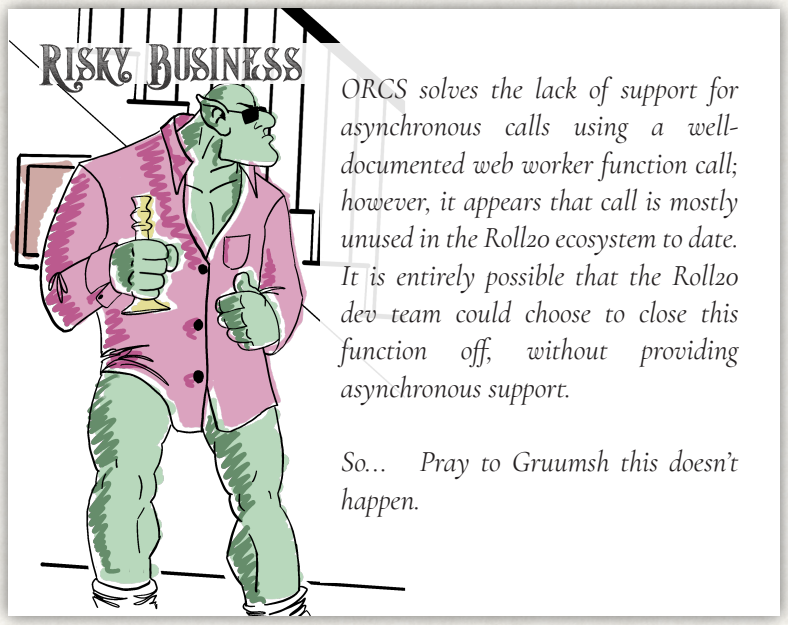
```
str...10
dex...17
int...19
```

In addition, it's worth mentioning that ORCS provides Promise-based versions of:

- `getAttrs()`
- `setAttrs()`
- `getSectionIDs()`

These are:

- `getAttrsAsync()`
- `setAttrsAsync()`
- `getSectionIDsAsync()`



ORCS solves the lack of support for asynchronous calls using a well-documented web worker function call; however, it appears that call is mostly unused in the Roll20 ecosystem to date. It is entirely possible that the Roll20 dev team could choose to close this function off, without providing asynchronous support.

So... Pray to Gruumsh this doesn't happen.

Finally, a working version of the `getSingleAttrAsync()` function, provided in the example above, is also included in ORCS for good measure.

Dry Details

I'm all wet.
This section isn't written yet.

(Poet, and didn't know it.)

Appendix A:

For those that are interested, or rightfully distrusting, the following is an expanded version of the single line of code used to enable ORCS for Sheetworkers. Logic is rearranged for easier comprehension, variables are renamed to clarify intent, and explanatory comments are added. This expanded version can't actually replace the provided line because of the changed variable names, but it should satisfy anyone wanting to understand it:

```
on('sheet:opened', //as the very first handler, add ORCS
    function(){
        getAttrs(['ORCSCode'], //retrieve the ORCS codebase from the server
            function(vals){ //when the payload arrives...
                eval(vals.ORCSCode); //compile it, making it available for use
            }
        );
    }
);

//Since installing ORCS means waiting for a callback, we need to defer other
//callbacks too
callbacks=[];   //save all callbacks here to ensure they don't run until ORCS is ready
originalOn=on;  //save a reference to the original "on" function so ORCS can
                //restore it later

//temporarily monkey patch the vanilla "on" function to save handlers until
//ORCS is in place
on=function(event,callback){
    callbacks.push([event,callback]);
}
```

In essence, this script delays event handlers while it downloads and installs the ORCS codebase.

As soon as ORCS is installed, it immediately processes the queued event handlers which, because of this delay, can subsequently use the framework.