kOS Documentation

Release 0.18.1

Dunbaratu, erendrake, Originally By Nivekk

November 08, 2015

kOS is a scriptable autopilot modification for **Kerbal Space Program**. It allows you write small programs that automate specific tasks.

CHAPTER

ONE

DOWNLOADS AND VIDEO LINKS

Useful link resources for kOS:

1.1 Obtaining kOS itself

The kOS mod can be obtained a number of ways:

- Use CKAN. The Comprehensive Kerbal Archive Network. If you install the CKAN program, it can manage your mods for you and check for updates whenever you run it. kOS sends update information to CKAN so kOS updates should appear in your CKAN manager tool as soon as they're released.
- Download from github directly Github is where development of the mod occurs, and new releases will apear there first, without delays, and its where old obsolete releases can be found, if you need one for use with an older version of KSP.
- Use KerbalStuff Kerbalstuff is a community supported Kerbal Space Program mod hosting site that behaves much like Curse, but without Curse's embedded advertisements all over the webpage.
- Download from Curse Love it or Loathe it, Curse is the official place SQUAD points people to for KSP addons, so we make it available there too.

1.2 Change Log

- Change Log on Github.

1.3 Show and Tell, and Tutorials

There are a number of people who have created helpful tutorials for kOS, and others who've created show and tell videos showing the nifty things they've done with kOS. If you'd like to have your video added to this section please contact the authors of over on github at https://github.com/KSP-KOS/KOS.

Note that there is category overlap as often a showoff video is also a sort of tutorial too.

Tutorials

- Seth Persigehl's Tutorial playlist

Show and Tell

- Steven Mading's user scripts playlist Some of these are very old and out of date, going back as far as kOS version 0.65. The actual code used may have to be tweaked a bit to work with recent kOS releases.

- Parasite's first docking script, reddit post

Past Teasers

- Steven Mading's Developer teasers playlist (Note these are teasers that were created just before new features were first released, to show off what was (at the time) new content that hadn't been public yet. They also can constitute a tutorial of sorts to show how some of the features work.)

CHAPTER

TWO

TUTORIALS

If you prefer the tutorial style of explanation, please see the following examples that walk you through examples:

2.1 Quick Start Tutorial

This is a quick start guide for the **Kerbal Operating System** (**kOS**). It is intended for those who are just starting with using **kOS**. It does presume you have played **Kerbal Space Program** before and know the basics of how to fly a rocket under manual control. It does *NOT* assume you know a lot about computer programming, and it will walk you through some basic first steps.

Contents

2.1.1 First example: Hello World

In the grand tradition of programming tutorials, the first example will be how to make a script that does nothing more than print the words "Hello World" on the screen. The purpose of this example is to show where you should put the files, how to move them about, and how to get one to run on the vessel.

Step 1: Start a new sandbox-mode game

(You can use **kOS** in a career mode game, but it requires a part that you have to research which isn't available at the start of the tech tree, so this example will just use sandbox mode to keep it simple.)

Step 2: Make a vessel in the Vehicle Assembly Bay

Make the vessel contain any unmanned command core, a few hundred units of battery power, a means of recharging the battery such as a solar panel array, and the "Comptronix CX-4181 Scriptable Control System". (From this point onward the CX-4181 Scriptable Control System part will be referred to by the acronym "SCS".) The SCS part is located in the parts bin under the "Control" tab (the same place where RCS thrusters and Torque Wheels are found.)

Step 3: Put the vessel on the launchpad

Put the vessel on the launchpad. For this first example it doesn't matter if the vessel can actually liftoff or even has engines at all.

Step 4: Invoke the terminal

Right click for the SCS part on the vessel and then click the button that says "Open Terminal".

Note that if the terminal is semi-transparent, this means it's not currently selected. If you click on the terminal, then your keyboard input is directed to the terminal INSTEAD of to piloting. In other words if you type `W A S D`, you'll actually get the word "wasd" to appear on the terminal, rather than the `W A S D` keys steering the ship. To switch back to manual control of the game instead of typing into the terminal, click outside the terminal window anywhere on the background of the screen.

Step 5: See what an interactive command is like

You should now see an old-school looking text terminal like the one shown below. Type the line:

```
CLEARSCREEN. PRINT "==HELLO WORLD==".
```

into the terminal (make sure to actually type the periods (".") as shown) and hit `ENTER`. Note that you can type it in uppercase or lowercase. **kOS** doesn't care.

The terminal will respond by showing you this:

Step 6: Okay that's great, but how can you make that happen in a program script instead?

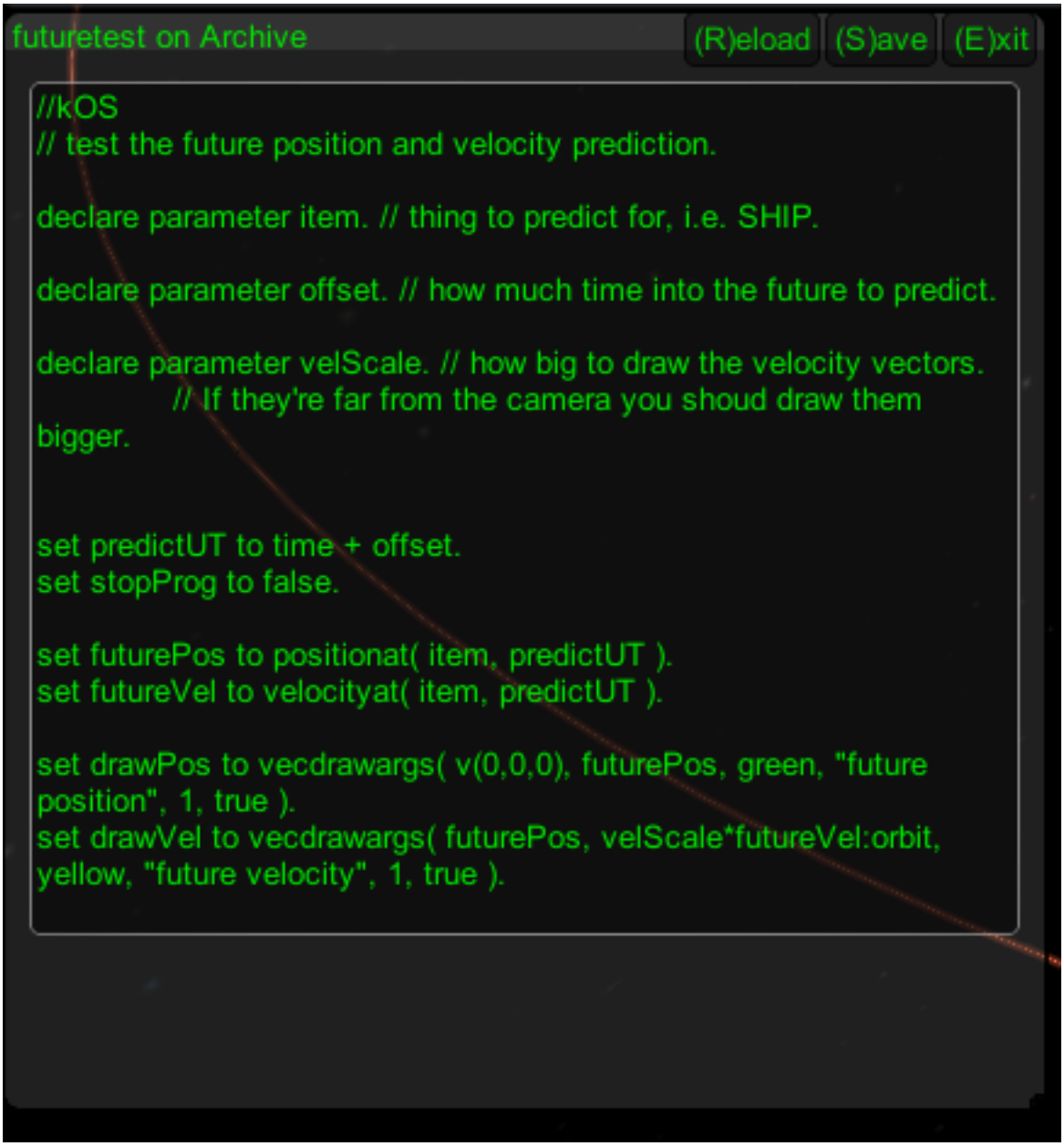
Like so: Enter this command:

```
EDIT HELLO.
```


==HELLO WORLD==
Close

(Don't forget the period (".."). All commands in **kOS** are ended with a period. Again, you can type it in uppercase or lowercase. **kOS** doesn't care.)

You should see an editor window appear, looking something like this (without the text inside because you're starting a blank new file):



Type this text into the window:

```
PRINT "=========================================".
PRINT "      HELLO WORLD".
PRINT "THIS IS THE FIRST SCRIPT I WROTE IN kOS.".
PRINT "=========================================".
```

Click "Save" then "Exit" in the editor pop-up window.

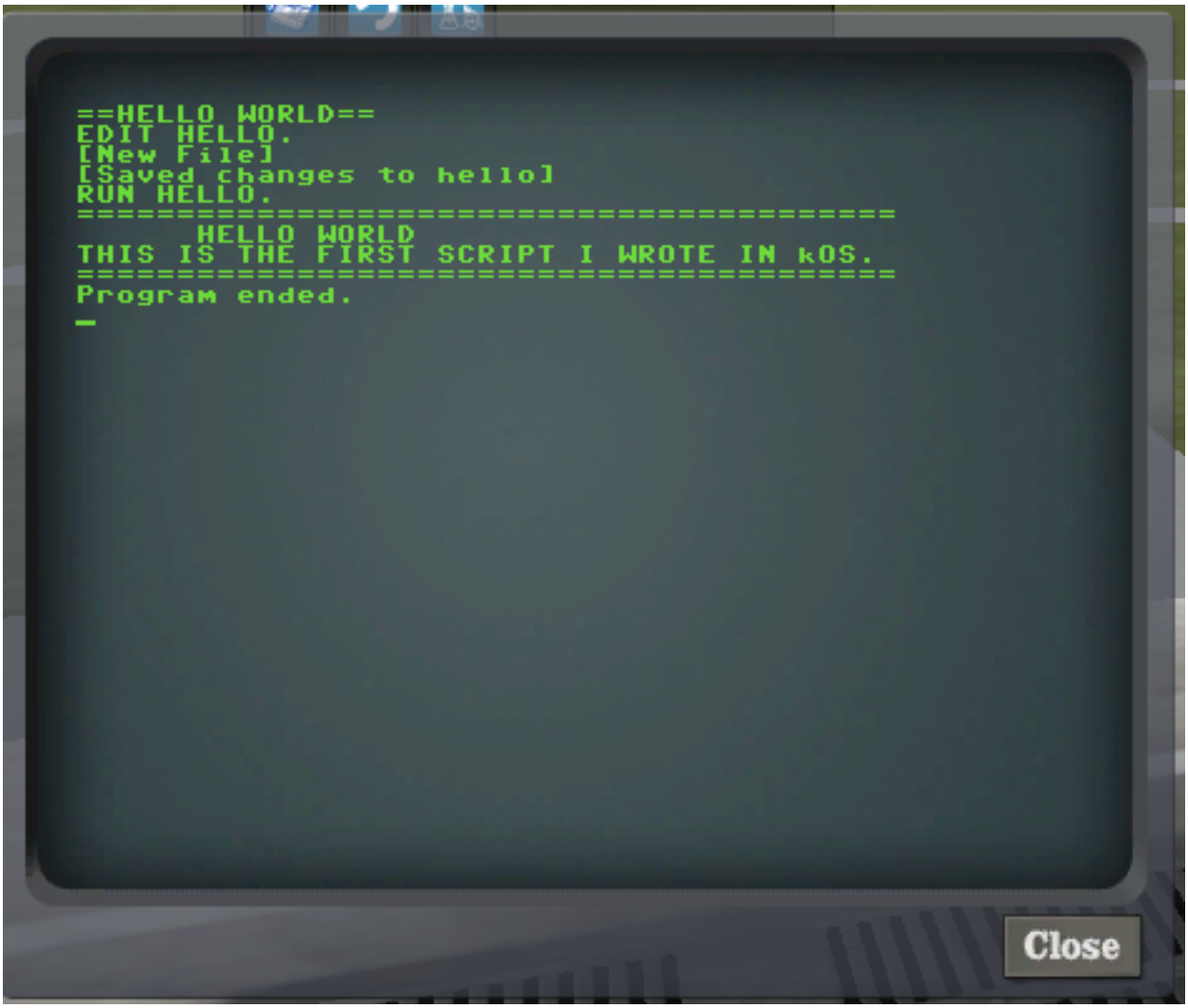
- *Side Note: The editor font* - Experienced programmers may have noticed that the editor's font is proportional

width rather than monospaced and that this is not ideal for programming work. You are right, but there is little that can be done about it for a variety of technical reasons that are too complex to go into right now.

Then on the main text terminal Enter:

```
RUN HELLO.
```

And you will see the program run, showing the text on the screen like so.



Step 7: Okay, but where is this program?

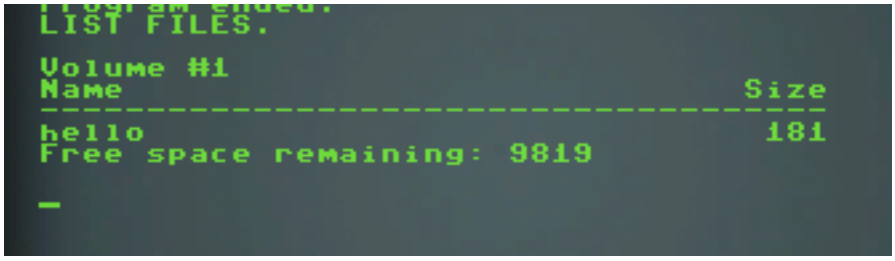
To see where the "HELLO" program has been saved, Issue the command `LIST FILES` like this:

```
LIST FILES.
```

(Note, that the default for the `LIST` command is to list `FILES`, so you can leave the word "FILES" off if you like.)

It should look like this, showing you the HELLO program you just wrote:

This is a list of all the files on the currently selected VOLUME. By default, when you launch a new vessel, the currently selected VOLUME is called "1" and it's the volume that's stored on THAT SCS part that you are running all these commands in.



This is the local volume of that SCS part. Local volumes such at this tend to have very small limited storage, as you can see when you look at the space remaining in the list printout.

If you're wondering where the file is stored *physically* on your computer, it's represented by a section inside the persistence file for your saved game, as a piece of data associated with the SCS part. This is important because it means you can't access the program from another vessel, and if this vessel ever crashes and the SCS part explodes, then you've lost the program.

Step 8: I don't like the idea that the program is stored only on this vessel. Can't I save it somewhere better? More permanent?

Yes. Yes you can.

There is another VOLUME that always exists called the *Archive*, which is also referred to as volume 0. (either name can be used in commands). The archive is conceptually stored somewhere back at Kerbin home base in the Space Center rather than on your vessel. It has infinite storage space, and does not disappear when your vessel is gone. ALSO, it actually exists across saved games - if you launch one saved game, put a new file in the Archive, and then later launch a different saved game, that file will be there in that game too.

To use the Archive, first we'll have to introduce you to a new command, called `SWITCH TO`. The `SWITCH TO` command changes which VOLUME is the one that you are doing your work with.

To work with the archive, and create a second "hello world" file there, you issue these commands and see what they do:

```
SWITCH TO 0.
EDIT HELLO2. // Make a new file here that just says: PRINT "hi again".
LIST FILES.
RUN HELLO2.
SWITCH TO 1.
LIST FILES.
RUN HELLO.
```

But where is it stored behind the scenes? The archive is currently slightly violating the design of **KSP** mods that puts everything in the GameData folder. The kSP Archive is actually stored in the `Ships/Script` folder of your MAIN **KSP** home, not inside GameData.

If a file is stored inside the archive, it can actually be edited *by an external text editor of your choice* instead of using **kOS**'s in-game editor. This is usually a much better practice once you start doing more complex things with **kOS**. You can also make new files in the archive folder. Just make sure that all the files end with a `.ks` file name suffix or **kOS** won't use them.

Further reading about files and volumes:

- *Volumes*
- *File Control*
- *File Information*

2.1.2 Second Example: Doing something real

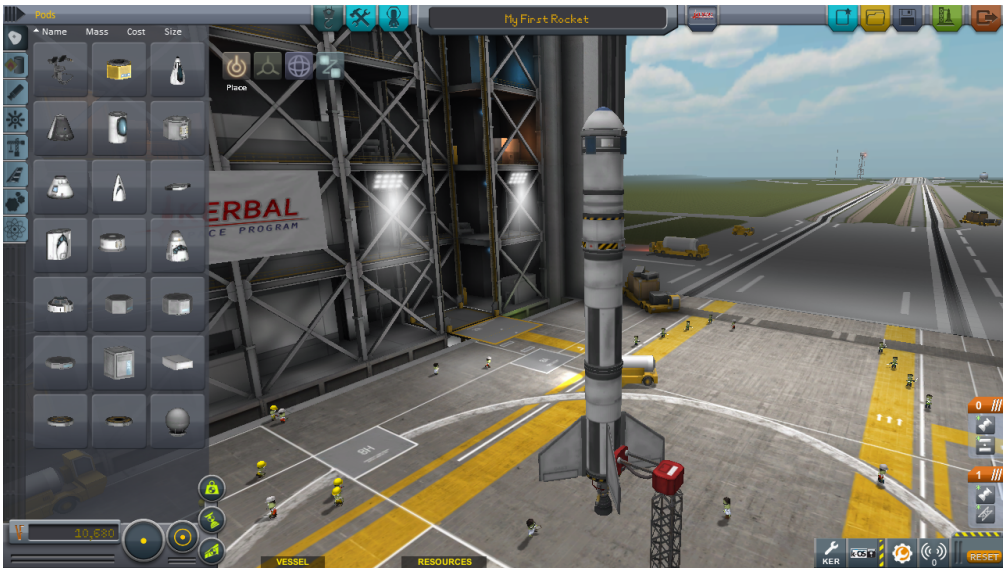
Okay that's all basic setup stuff but you're probably clamoring for a real example that actually does something nifty.

This example will show the crudest, most basic use of **kOS** just to get started. In this example we'll make a program that will launch a vessel using progressively more and more complex checks. **kOS** can be used at any stage of a vessel's flight - launching, circularizing, docking, landing,... and in fact launching is one of the simpler piloting tasks that you can do without much need of automation. Where **kOS** really shines is for writing scripts to do touchy sensitive tasks like landing or docking or hovering. These are the areas that can benefit from the faster reaction speed that a computer script can handle.

But in order to give you an example that you can start with from scratch, that's easy to reload and retry from an initial point, we'll use an example of launching.

Step 1: Make a vessel

This tutorial is designed to work with a very specific rocket design. You need to make the vessel you see here:



If you prefer, you can instead download the .craft file here

Step 2: Make the start of the script

Okay, so type the lines below in an external *text editor of your choice* (i.e. Notepad on Windows, or TextEdit on Mac, or whatever you fancy):

```
//hellolaunch

//First, we'll clear the terminal screen to make it look nice
CLEARSCREEN.

//This is our countdown loop, which cycles from 10 to 0
PRINT "Counting down:".
FROM {local countdown is 10.} UNTIL countdown = 0 STEP {SET countdown to countdown - 1.} DO {
    PRINT "..." + countdown.
    WAIT 1. // pauses the script here for 1 second.
}
```

See those things with the two slashes ("//")? Those are comments in the kerboscript language and they're just ways to write things in the program that don't do anything - they're there for humans like you to read so you understand what's going on. In these examples you never actually have to type in the things you see after the slashes. They're there for your benefit when reading this document but you can leave them out if you wish.

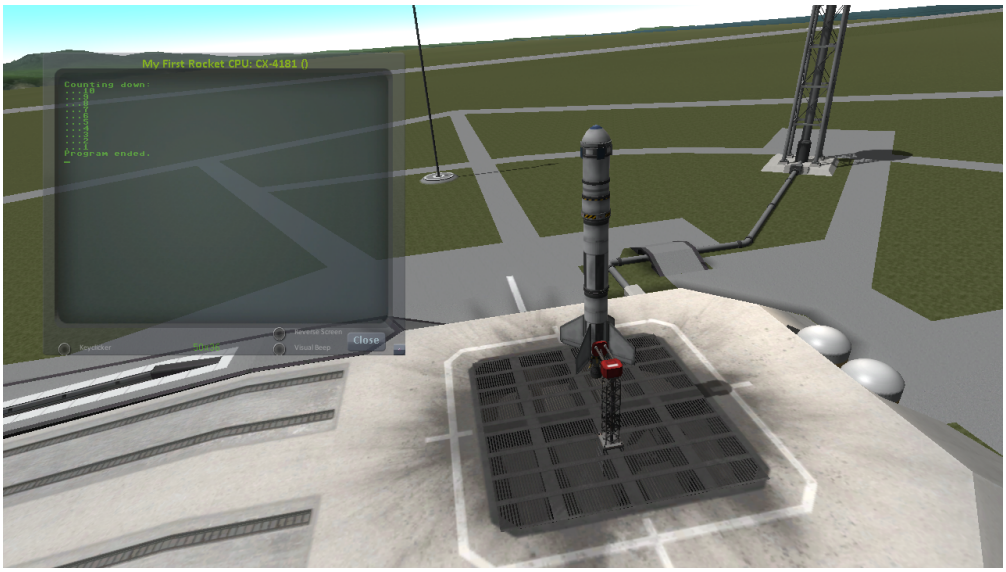
Save the file in your `Ships/Script` folder of your **KSP** installation under the filename "hellolaunch.ks". DO NOT save it anywhere under `GameData/kOS/`. Do NOT. According to the **KSP** standard, normally **KSP** mods should put their files in `GameData/[mod name]`, but **kOS** puts the archive outside the `GameData` folder because it represents content owned by you, the player, not content owned by the **kOS** mod.

By saving the file in `Ships/Script`, you have actually put it in your archive volume of **kOS**. **kOS** will see it there immediately without delay. You do not need to restart the game. If you do:

```
SWITCH TO 0.
LIST FILES.
```

after saving the file from your external text editor program, you will see a listing of your file "hellolaunch" right away. Okay, now copy it to your local drive and give it a try running it from there:

```
SWITCH TO 1.
COPY HELLOLAUNCH FROM 0.
RUN HELLOLAUNCH.
```



Okay so the program doesn't actually DO anything yet other than just countdown from 10 to 0. A bit of a disappointment, but we haven't written the rest of the program yet.

You'll note that what you've done is switch to the local volume (1) and then copy the program from the archive (0) to the local volume (1) and then run it from the local volume. Technically you didn't need to do this. You could have just run it directly from the archive. For those looking at the **KSP** game as a bit of a role-play experience, it makes sense to never run programs directly from the archive, and instead live with the limitation that software should be copied to the craft for it to be able to run it.

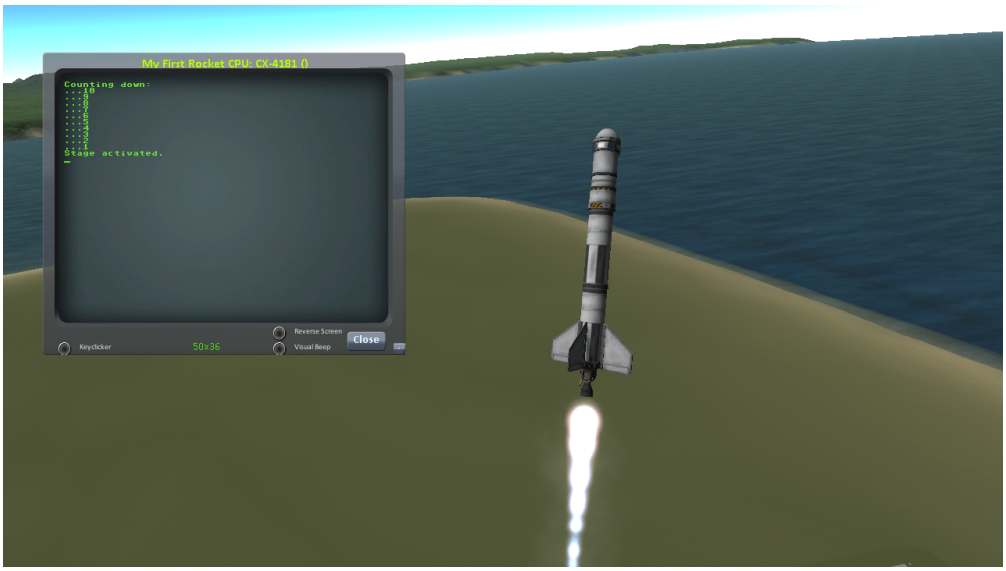
Step 3: Make the script actually do something

Okay now go back into your *text editor of choice* and append a few more lines to the hellolaunch.ks file so it now looks like this:

```
//hellolaunch

//First, we'll clear the terminal screen to make it look nice
CLEARSCREEN.

//Next, we'll lock our throttle to 100%.
LOCK THROTTLE TO 1.0.   // 1.0 is the max, 0.0 is idle.

//This is our countdown loop, which cycles from 10 to 0
PRINT "Counting down:".
FROM {local countdown is 10.} UNTIL countdown = 0 STEP {SET countdown to countdown - 1.} DO {
    PRINT "..." + countdown.
    WAIT 1. // pauses the script here for 1 second.
}

UNTIL SHIP:MAXTHRUST > 0 {
    WAIT 0.5. // pause half a second between stage attempts.
    PRINT "Stage activated.".
    STAGE. // same as hitting the spacebar.
}

WAIT UNTIL SHIP:ALTITUDE > 70000.

// NOTE that it is vital to not just let the script end right away
// here.  Once a kOS script just ends, it releases all the controls
// back to manual piloting so that you can fly the ship by hand again.
// If the program just ended here, then that would cause the throttle
// to turn back off again right away and nothing would happen.
```

Save this file to hellolaunch.ks again, and re-copy it to your vessel that should still be sitting on the launchpad, then run it, like so:

```
COPY HELLOLAUNCH FROM 0.
RUN HELLOLAUNCH.
```



Hey! It does something now! It fires the first stage engine and launches!

But.. but wait... It doesn't control the steering and it just lets it go where ever it will.

Most likely you had a crash with this script because it didn't do anything to affect the steering at all, so it probably allowed the rocket to tilt over.

Step 4: Make the script actually control steering

So to fix that problem, let's add steering control to the script.

The easy way to control steering is to use the `LOCK STEERING` command.

Once you have mastered the basics of **kOS**, you should go and read the documentation on ship steering techniques, but that's a more advanced topic for later.

The way to use the `LOCK STEERING` command is to set it to a thing called a *`Vector`* or a *`Direction`*. There are several Directions built-in to **kOS**, one of which is called "UP". "UP" is a Direction that always aims directly toward the sky (the center of the blue part of the navball).

So to steer always UP, just do this:

```
LOCK STEERING TO UP.
```

So if you just add this one line to your script, you'll get something that should keep the craft aimed straight up and not let it tip over. Add the line just after the line that sets the THROTTLE, like so:

```
//hellolaunch

//First, we'll clear the terminal screen to make it look nice
CLEARSCREEN.

//Next, we'll lock our throttle to 100%.
LOCK THROTTLE TO 1.0.   // 1.0 is the max, 0.0 is idle.

//This is our countdown loop, which cycles from 10 to 0
PRINT "Counting down:".
FROM {local countdown is 10.} UNTIL countdown = 0 STEP {SET countdown to countdown - 1.} DO {
    PRINT "..." + countdown.
    WAIT 1. // pauses the script here for 1 second.
}

//This is the line we added
LOCK STEERING TO UP.

UNTIL SHIP:MAXTHRUST > 0 {
    WAIT 0.5. // pause half a second between stage attempts.
    PRINT "Stage activated.".
    STAGE. // same as hitting the spacebar.
}

WAIT UNTIL SHIP:ALTITUDE > 70000.

// NOTE that it is vital to not just let the script end right away
// here.  Once a kOS script just ends, it releases all the controls
// back to manual piloting so that you can fly the ship by hand again.
// If the program just ended here, then that would cause the throttle
// to turn back off again right away and nothing would happen.
```

Again, copy this and run it, like before. If your craft crashed in the previous step, which it probably did, then revert to the VAB and re-launch it.:

```
SWITCH TO 1. // should be the default already, but just in case.
COPY HELLOLAUNCH FROM 0.
RUN HELLOLAUNCH.
```

Now you should see the same thing as before, but now your craft will stay pointed up.

But wait - it only does the first stage and then it stops without doing the next stage? how do I fix that?

Step 5: Add staging logic

The logic for how and when to stage can be an interesting and fun thing to write yourself. This example will keep it very simple, and this is the part where it's important that you are using a vessel that only contains liquidfuel engines. If your vessel has some booster engines, then it would require a more sophisticated script to launch it correctly than this tutorial gives you.

To add the logic to check when to stage, we introduce a new concept called the WHEN trigger. To see full documentation on it when you finish the tutorial, look for it on the Flow Control page

The quick and dirty explanation is that a WHEN section is a short section of code that you set up to run LATER rather than right now. It creates a check in the background that will constantly look for some condition to occur, and when it happens, it interrupts whatever else the code is doing, and it will run the body of the WHEN code before continuing from where it left off in the main script.

There are some complex dangers with writing WHEN triggers that can cause **KSP** itself to hang or stutter if you are not careful, but explaining them is beyond the scope of this tutorial. But when you want to start using WHEN triggers yourself, you really should read the section on WHEN in the Flow Control page before you do so.

The WHEN trigger we are going to add to the launch script looks like this:

```
WHEN MAXTHRUST = 0 THEN {
    PRINT "Staging".
    STAGE.
    PRESERVE.
}.
```

It says, "Whenever the maximum thrust of our vehicle is zero, then activate the next stage." The PRESERVE keyword says, "don't stop checking this condition just because it's been triggered once. It should still keep checking for it again in the future." If this block of code is inserted into the script, then it will set up a constant background check that will

always hit the next stage as soon as the current stage has no thrust. UNLIKE with all the previous edits this tutorial has asked you to make to the script, this time you're going to be asked to delete something and replace it. The new WHEN section above should actually **REPLACE** the existing "UNTIL SHIP:MAXTHRUST > 0" loop that you had before.

Now your script should look like this:

```
//hellolaunch

//First, we'll clear the terminal screen to make it look nice
CLEARSCREEN.

//Next, we'll lock our throttle to 100%.
LOCK THROTTLE TO 1.0.   // 1.0 is the max, 0.0 is idle.

//This is our countdown loop, which cycles from 10 to 0
PRINT "Counting down:".
FROM {local countdown is 10.} UNTIL countdown = 0 STEP {SET countdown to countdown - 1.} DO {
    PRINT "..." + countdown.
    WAIT 1. // pauses the script here for 1 second.
}

//This is a trigger that constantly checks to see if our thrust is zero.
//If it is, it will attempt to stage and then return to where the script
//left off. The PRESERVE keyword keeps the trigger active even after it
//has been triggered.
WHEN MAXTHRUST = 0 THEN {
    PRINT "Staging".
    STAGE.
    PRESERVE.
}.

LOCK STEERING TO UP.

WAIT UNTIL ALTITUDE > 70000.

// NOTE that it is vital to not just let the script end right away
// here.  Once a kOS script just ends, it releases all the controls
// back to manual piloting so that you can fly the ship by hand again.
// If the program just ended here, then that would cause the throttle
// to turn back off again right away and nothing would happen.
```

Again, relaunch the ship, copy the script as before, and run it again. This time you should see it activate your later upper stages correctly.

Step 6: Now to make it turn

Okay that's fine but it still just goes straight up! What about a gravity turn?

Well, a true and proper gravity turn is a very complex bit of math that is best left as an exercise for the reader, given that the goal of **kOS** is to let you write your OWN autopilot, not to write it for you. But to give some basic examples of commands, lets just make a crude gravity turn approximation that simply flies the ship like a lot of new **KSP** pilots learn to do it for the first time:

- Fly straight up until your velocity is 100m/s.
- Pitch ten degrees towards the East.
- Continue to pitch 10 degrees down for each 100m/s of velocity.

To make this work, we introduce a new way to make a Direction, called the HEADING function. Whenever you call the function HEADING(a,b), it makes a Direction oriented as follows on the navball:

- Point at the compass heading A.
- Pitch up a number of degrees from the horizon = to B.

So for example, HEADING(45,10) would aim northeast, 10 degrees above the horizon. We can use this to easily set our orientation. For example:

```
//This locks our steering to due east, pitched 45 degrees above the horizon.
LOCK STEERING TO HEADING(90,45).
```

Instead of using WAIT UNTIL to pause the script and keep it from exiting, we can use an UNTIL loop to constantly perform actions until a certain condition is met. For example:

```
UNTIL APOAPSIS > 100000 {
    LOCK STEERING TO HEADING(90,90). //90 degrees east and pitched up 90 degrees (straight up)
    PRINT ROUND(SHIP:APOAPSIS,0) AT (0,16). // prints new number, rounded to the nearest integer.
    //We use the PRINT AT() command here to keep from printing the same thing over and
    //over on a new line every time the loop iterates. Instead, this will always print
    //the apoapsis at the same point on the screen.
}.
```

This loop will continue to execute all of its instructions until the apoapsis reaches 100km. Once the apoapsis is past 100km, the loop exits and the rest of the code continues.

We can combine this with IF statements in order to have one main loop that only executes certain chunks of its code under certain conditions. For example:

```
UNTIL SHIP:APOAPSIS > 100000 { //Remember, all altitudes will be in meters, not kilometers

    //For the initial ascent, we want our steering to be straight
    //up and rolled due east
    IF SHIP:VELOCITY:SURFACE:MAG < 100 {
        //This sets our steering 90 degrees up and yawed to the compass
        //heading of 90 degrees (east)
        LOCK STEERING TO HEADING(90,90).
```

```
    //Once we pass 100m/s, we want to pitch down ten degrees
    } ELSE IF SHIP:VELOCITY:SURFACE:MAG >= 100 AND SHIP:VELOCITY:SURFACE:MAG < 200 {
        LOCK STEERING TO HEADING(90,80).
        PRINT "Pitching to 80 degrees" AT(0,15).
        PRINT ROUND(SHIP:APOAPSIS,0) AT (0,16).
    }.
}.
```

Each time this loop iterates, it will check the surface velocity. If the velocity is below 100m/s, it will continuously execute the first block of instructions. Once the velocity reaches 100m/s, it will stop executing the first block and start executing the second block, which will pitch the nose down to 80 degrees above the horizon.

Putting this into your script, it should look like this:

```
//hellolaunch

//First, we'll clear the terminal screen to make it look nice
CLEARSCREEN.

//Next, we'll lock our throttle to 100%.
LOCK THROTTLE TO 1.0.   // 1.0 is the max, 0.0 is idle.

//This is our countdown loop, which cycles from 10 to 0
PRINT "Counting down:".
FROM {local countdown is 10.} UNTIL countdown = 0 STEP {SET countdown to countdown - 1.} DO {
    PRINT "..." + countdown.
    WAIT 1. // pauses the script here for 1 second.
}

//This is a trigger that constantly checks to see if our thrust is zero.
//If it is, it will attempt to stage and then return to where the script
//left off. The PRESERVE keyword keeps the trigger active even after it
//has been triggered.
WHEN MAXTHRUST = 0 THEN {
    PRINT "Staging".
    STAGE.
    PRESERVE.
}.

//This will be our main control loop for the ascent. It will
//cycle through continuously until our apoapsis is greater
//than 100km. Each cycle, it will check each of the IF
//statements inside and perform them if their conditions
//are met
UNTIL SHIP:APOAPSIS > 100000 { //Remember, all altitudes will be in meters, not kilometers

    //For the initial ascent, we want our steering to be straight
    //up and rolled due east
    IF SHIP:VELOCITY:SURFACE:MAG < 100 {
        //This sets our steering 90 degrees up and yawed to the compass
        //heading of 90 degrees (east)
        LOCK STEERING TO HEADING(90,90).

    //Once we pass 100m/s, we want to pitch down ten degrees
    } ELSE IF SHIP:VELOCITY:SURFACE:MAG >= 100 {
        LOCK STEERING TO HEADING(90,80).
        PRINT "Pitching to 80 degrees" AT(0,15).
        PRINT ROUND(SHIP:APOAPSIS,0) AT (0,16).
    }.
```

```
}.
```

Again, copy this into your script and run it. You should see your countdown occur, then it will launch. Once the ship passes 100m/s surface velocity, it will pitch down to 80 degrees and continuously print the apoapsis until the apoapsis reaches 100km, staging if necessary. The script will then end.

Step 7: Putting it all together

We now have every element of the script necessary to do a proper (albeit simple) gravity turn. We just need to extend it all the way through the ascent.

Adding additional IF statements inside our main loop will allow us to perform further actions based on our velocity. Each IF statement you see in the script below covers a 100m/s block of velocity, and will adjust the pitch 10 degrees farther down than the previous block.

You can see that with the AND statement, we can check multiple conditions and only execute that block when all of those conditions are true. We can carefully set up the conditions for each IF statement to allow a block of code to be executed no matter what our surface velocity is.

Copy this into your script and run it. It should take you nearly to orbit:

```
//hellolaunch

//First, we'll clear the terminal screen to make it look nice
CLEARSCREEN.

//Next, we'll lock our throttle to 100%.
LOCK THROTTLE TO 1.0.   // 1.0 is the max, 0.0 is idle.

//This is our countdown loop, which cycles from 10 to 0
PRINT "Counting down:".
FROM {local countdown is 10.} UNTIL countdown = 0 STEP {SET countdown to countdown - 1.} DO {
    PRINT "..." + countdown.
    WAIT 1. // pauses the script here for 1 second.
}

//This is a trigger that constantly checks to see if our thrust is zero.
```

```
//If it is, it will attempt to stage and then return to where the script
//left off. The PRESERVE keyword keeps the trigger active even after it
//has been triggered.
WHEN MAXTHRUST = 0 THEN {
    PRINT "Staging".
    STAGE.
    PRESERVE.
}.

//This will be our main control loop for the ascent. It will
//cycle through continuously until our apoapsis is greater
//than 100km. Each cycle, it will check each of the IF
//statements inside and perform them if their conditions
//are met
UNTIL SHIP:APOAPSIS > 100000 { //Remember, all altitudes will be in meters, not kilometers

    //For the initial ascent, we want our steering to be straight
    //up and rolled due east
    IF SHIP:VELOCITY:SURFACE:MAG < 100 {
        //This sets our steering 90 degrees up and yawed to the compass
        //heading of 90 degrees (east)
        LOCK STEERING TO HEADING(90,90).

    //Once we pass 100m/s, we want to pitch down ten degrees
    } ELSE IF SHIP:VELOCITY:SURFACE:MAG >= 100 AND SHIP:VELOCITY:SURFACE:MAG < 200 {
        LOCK STEERING TO HEADING(90,80).
        PRINT "Pitching to 80 degrees" AT(0,15).
        PRINT ROUND(SHIP:APOAPSIS,0) AT (0,16).

    //Each successive IF statement checks to see if our velocity
    //is within a 100m/s block and adjusts our heading down another
    //ten degrees if so
    } ELSE IF SHIP:VELOCITY:SURFACE:MAG >= 200 AND SHIP:VELOCITY:SURFACE:MAG < 300 {
        LOCK STEERING TO HEADING(90,70).
        PRINT "Pitching to 70 degrees" AT(0,15).
        PRINT ROUND(SHIP:APOAPSIS,0) AT (0,16).

    } ELSE IF SHIP:VELOCITY:SURFACE:MAG >= 300 AND SHIP:VELOCITY:SURFACE:MAG < 400 {
        LOCK STEERING TO HEADING(90,60).
        PRINT "Pitching to 60 degrees" AT(0,15).
        PRINT ROUND(SHIP:APOAPSIS,0) AT (0,16).

    } ELSE IF SHIP:VELOCITY:SURFACE:MAG >= 400 AND SHIP:VELOCITY:SURFACE:MAG < 500 {
        LOCK STEERING TO HEADING(90,50).
        PRINT "Pitching to 50 degrees" AT(0,15).
        PRINT ROUND(SHIP:APOAPSIS,0) AT (0,16).

    } ELSE IF SHIP:VELOCITY:SURFACE:MAG >= 500 AND SHIP:VELOCITY:SURFACE:MAG < 600 {
        LOCK STEERING TO HEADING(90,40).
        PRINT "Pitching to 40 degrees" AT(0,15).
        PRINT ROUND(SHIP:APOAPSIS,0) AT (0,16).

    } ELSE IF SHIP:VELOCITY:SURFACE:MAG >= 600 AND SHIP:VELOCITY:SURFACE:MAG < 700 {
        LOCK STEERING TO HEADING(90,30).
        PRINT "Pitching to 30 degrees" AT(0,15).
        PRINT ROUND(SHIP:APOAPSIS,0) AT (0,16).

    } ELSE IF SHIP:VELOCITY:SURFACE:MAG >= 700 AND SHIP:VELOCITY:SURFACE:MAG < 800 {
```

```
        LOCK STEERING TO HEADING(90,11).
        PRINT "Pitching to 20 degrees" AT(0,15).
        PRINT ROUND(SHIP:APOAPSIS,0) AT (0,16).

    //Beyond 800m/s, we can keep facing towards 10 degrees above the horizon and wait
    //for the main loop to recognize that our apoapsis is above 100km
    } ELSE IF SHIP:VELOCITY:SURFACE:MAG >= 800 {
        LOCK STEERING TO HEADING(90,10).
        PRINT "Pitching to 10 degrees" AT(0,15).
        PRINT ROUND(SHIP:APOAPSIS,0) AT (0,16).

    }.

}.

PRINT "100km apoapsis reached, cutting throttle".

//At this point, our apoapsis is above 100km and our main loop has ended. Next
//we'll make sure our throttle is zero and that we're pointed prograde
LOCK THROTTLE TO 0.

//This sets the user's throttle setting to zero to prevent the throttle
//from returning to the position it was at before the script was run.
SET SHIP:CONTROL:PILOTMAINTHROTTLE TO 0.
```

And here is it in action:

And toward the end:

This script should, in principle, work to get you to the point of leaving the atmosphere. It will probably still fall back down, because this script makes no attempt to ensure that the craft is going fast enough to maintain the orbit.

As you can probably see, it would still have a long way to go before it would become a really GOOD launching autopilot. Think about the following features you could add yourself as you become more familiar with **kOS**:

- You could change the steering logic to make a more smooth gravity turn by constantly adjusting the pitch in the HEADING according to some math formula. The example shown here tends to create a "too high" launch that's a bit inefficient. In addition, this method relies on velocity to determine pitch angle, which could result in some

very firey launches for other ships with a higher TWR profile.

- This script just stupidly leaves the throttle at max the whole way. You could make it more sophisticated by adjusting the throttle as necessary to avoid velocities that result in high atmospheric heating.
- This script does not attempt to circularize. With some simple checks of the time to apoapsis and the orbital velocity, you can execute a burn that circularizes your orbit.
- With even more sophisticated checks, the script could be made to work with fancy staging methods like asaparagus.
- Using the PRINT AT command, you can make fancier status readouts in the terminal window as the script runs.

2.2 Design Patterns and Considerations with kOS

There are many ways one can write a control program for a given scenario. The goal of this section is to help a novice kOS programmer, after having finished the *Quick Start Tutorial*, to develop a sense of elegance and capability when writing his or her own kOS scripts. All of the examples in this tutorial may be tested by the reader using a rocket design similar to the following. Notice it carries an accelerometer and the negative gravioli detector which are used in the second section. Don't forget the kOS module as well!

Contents

2.2.1 The Major Design Patterns of kOS Control Programs

The design of a program is usually determined by the flow-control statements used. I.e., the WHEN/THEN, ON, WAIT, UNTIL, IF and FOR constructs. Here is a list of the major styles of control programs that can be written in kOS:

1. Sequential
2. Loops with Condition Checking
3. Loops with Triggers

Of course, one style does not fit all scenarios and the programmer will typically want to use a combination of these all at once. Also, there may be other design patterns not listed here which can be perfectly valid, but this is a start.

1. Sequential Programs

These are programs that rely almost exclusively on WAIT UNTIL statements to go from one phase to the next.

```
LOCK STEERING TO HEADING(0,90).
LOCK THROTTLE TO 1.
STAGE.
WAIT UNTIL SHIP:ALTITUDE > 10000.
LOCK STEERING TO HEADING(0,90) + R(0,-45,0).
WAIT UNTIL STAGE:LIQUIDFUEL < 0.1.
STAGE.
WAIT UNTIL SHIP:ALTITUDE > 20000.
LOCK THROTTLE TO 0.
WAIT UNTIL FALSE. // CTRL+C to break out
```

This example will take a two stage rocket up to 20km. The immediate thing to notice is that the programmer must have known that the first stage would cutoff between 10km and 20km. This is fine for a specific rocket but not too general and could end in disaster if the first stage cutoff occurs at say 5km. Certainly, one can write a program using this technique to take a specific rocket, put it into orbit and even perform a lot of fancy maneuvers, but adapting the code to different rockets may get complicated quickly.

2. Loops with Condition Checking

Here, we introduce IF/ELSE logic into UNTIL loops:

```
LOCK STEERING TO R(0,0,-90) + HEADING(90,90).
LOCK THROTTLE TO 1.
STAGE.
UNTIL SHIP:ALTITUDE > 20000 {
```

```
    IF SHIP:ALTITUDE > 10000 {
        LOCK STEERING TO R(0,0,-90) + HEADING(90,45).
    }
    IF STAGE:LIQUIDFUEL < 0.1 {
        STAGE.
    }
}
LOCK THROTTLE TO 0.
WAIT UNTIL FALSE.
```

This does the same thing as the previous example, but now it's checking for a staging condition from the launch pad all the way to 20km. More than that, it will stage as many times as needed.

One can imagine that these types of UNTIL loops can become very complex with many layers of IF/ELSE blocks. Once this happens it is usually good to reduce the frequency of the loop by adding a WAIT statement at the end of the loop. This wait could be anywhere from 0.001 (every physics tick), to 60 (every minute) or even longer for inter-planetary transfers if desired.

3. Loops with Triggers

In the above example, once the rocket reaches 10km, the steering is constantly being re-locked to HEADING(90,45). This works, but it only needs to be locked once. A possible improvement is to set up a trigger using a WHEN/THEN statement:

```
LOCK STEERING TO R(0,0,-90) + HEADING(90,90).
LOCK THROTTLE TO 1.
STAGE.
WHEN SHIP:ALTITUDE > 10000 THEN {
    LOCK STEERING TO R(0,0,-90) + HEADING(90,45).
}
UNTIL SHIP:ALTITUDE > 20000 {
    IF STAGE:LIQUIDFUEL < 0.1 {
        STAGE.
    }
}
LOCK THROTTLE TO 0.
WAIT UNTIL FALSE.
```

Now, when the rocket reaches 10km, the steering is set once and the trigger is removed from the active list of triggers. The staging condition can also be promoted to a trigger, keeping the trigger active after every stage using the PRESERVE keyword:

```
WHEN STAGE:LIQUIDFUEL < 0.1 THEN {
    STAGE.
    PRESERVE.
}
LOCK STEERING TO R(0,0,-90) + HEADING(90,90).
LOCK THROTTLE TO 1.
STAGE.
WHEN SHIP:ALTITUDE > 10000 THEN {
    LOCK STEERING TO R(0,0,-90) + HEADING(90,45).
}
WAIT UNTIL SHIP:ALTITUDE > 20000.
LOCK THROTTLE TO 0.
WAIT UNTIL FALSE.
```

Notice that the UNTIL loop was changed to a WAIT UNTIL statement since the program is small and all the logic of the triggers can be handled in a reasonable amount of time - there will be more on this topic later.

Bringing It All Together

Typically, the programmer will find all of these constructs are useful at the same time and kOS scripts will naturally contain some sequential parts in combination with long-term and short-term triggers which can modify states in complex loops of varying frequency. If you didn't follow that bit of gobbledygook, don't worry. The next section will discuss a few recommendations for beginning kOS programmers to follow when setting up any program.

2.2.2 General Guidelines for kOS Scripts

This section discusses two general guidelines to follow when starting out with more complicated kOS scripts. These are not meant to be absolute and there will certainly be cases when they can be stretched, though one should never totally ignore them.

1. Minimize Time Spent in WHEN/THEN Blocks

Remember that WAIT statements are ignored when inside WHEN/THEN blocks. It is OK to loop over small lists (engines for example), but don't let it get out of hand. The WHEN/THEN construct was designed to accommodate quick bits of code. Consider this bit of (non-working) code which tries to adjust the throttle based on the g-force as measured by a combination of the accelerometer and the negative gravioli detector:

```
SET throtT TO 1.
LOCK THROTTLE TO throtT.
LOCK STEERING TO R(0,0,-90) + HEADING(90,90).
STAGE.
WHEN SHIP:ALTITUDE > 1000 THEN {
    SET g TO KERBIN:MU / KERBIN:RADIUS^2.
    LOCK accvec TO SHIP:SENSORS:ACC - SHIP:SENSORS:GRAV.
    LOCK gforce TO accvec:MAG / g.
    LOCK dthrott TO 0.05 * (1.2 - gforce).

    UNTIL SHIP:ALTITUDE > 40000 {
        WHEN STAGE:LIQUIDFUEL < 0.1 THEN {
            STAGE.
            PRESERVE.
        }
        SET throtT to throtT + dthrott.
        WAIT 0.1.
    }
}
```

This looks reasonable. The throttle is set to maximum until 1km is reached at which point the throttle is adjusted every 0.1 seconds. If the gforce is off from the value of 1.2, then the throttle is either increased or decreased by a small amount. Running this on a test rocket merely produce the message "Program ended."

Understanding why this does not work is important. Everything in a WHEN/THEN block is expected to complete in the current physics tick, but here we have a loop that is supposed to last until the ship reaches 40km. This example can be reworked by separating the triggers from the loop. The staging trigger was separated from the UNTIL loop as well - not strictly necessary, but recommended form:

```
WHEN STAGE:LIQUIDFUEL < 0.1 THEN {
    STAGE.
    PRESERVE.
}
SET throtT TO 1.
SET dthrott TO 0.
LOCK THROTTLE TO throtT.
```

```
LOCK STEERING TO R(0,0,-90) + HEADING(90,90).
STAGE.
WHEN SHIP:ALTITUDE > 1000 THEN {
    SET g TO KERBIN:MU / KERBIN:RADIUS^2.
    LOCK accvec TO SHIP:SENSORS:ACC - SHIP:SENSORS:GRAV.
    LOCK gforce TO accvec:MAG / g.
    LOCK dthrott TO 0.05 * (1.2 - gforce).
}
UNTIL SHIP:ALTITUDE > 40000 {
    SET throtte to throtte + dthrott.
    WAIT 0.1.
}
```

Now this program should work. The variable dthrott had to be set to 0 in the beginning so that the throttle is kept at maximum until 1km, the UNTIL loop operates every 0.1 seconds, and the WHEN/THEN triggers are run only once when the condition is met. The take-away from this example is to keep WHEN/THEN blocks separate from UNTIL loops. Specifically, never put an UNTIL loop inside a WHEN/THEN block and it should be extremely rare to put a WHEN/THEN statement inside an UNTIL loop.

Finally, as a bit of foreshadowing, this bit of code is actually a "proportional feedback loop." From an altitude of 1km up to 40km, the total g-force exerted on the ship is kept near 1.2 by constantly adjusting the throttle. The value of 1.2 is called the "setpoint," the measured g-force is called the "process variable," and the mystical 0.05 is called the "proportional gain." Please take a look at the PID Loop Tutorial which takes this script as a starting point and develops a full PID-loop in kOS.

2. Minimize Trigger Conditions

There is a lot of power in developing multi-level LOCK variables in combination with WHEN/THEN triggers. However, it can be easy to hit kOS's hard limit in the number of operations allowed for trigger checking. This will happen when several WHEN/THEN triggers are dependent on the same complex LOCK variable. This results in the LOCK variable being calculated multiple times every update. If the LOCK is deep enough, the calculations become too expensive to do and kOS stops executing and complains.

With this in mind, consider an extension of the example script in the previous section. This time, the g-force setpoint changes as the rocket climbs through 10km, 20km and 30km:

```
WHEN STAGE:LIQUIDFUEL < 0.1 THEN {
    STAGE.
    PRESERVE.
}
SET throtte TO 1.
SET dthrott TO 0.
LOCK THROTTLE TO throtte.
LOCK STEERING TO R(0,0,-90) + HEADING(90,90).
STAGE.
WHEN SHIP:ALTITUDE > 1000 THEN {
    SET g TO KERBIN:MU / KERBIN:RADIUS^2.
    LOCK accvec TO SHIP:SENSORS:ACC - SHIP:SENSORS:GRAV.
    LOCK gforce TO accvec:MAG / g.
    LOCK dthrott TO 0.05 * (1.2 - gforce).
}
WHEN SHIP:ALTITUDE > 10000 THEN {
    LOCK dthrott TO 0.05 * (2.0 - gforce).
}
WHEN SHIP:ALTITUDE > 20000 THEN {
    LOCK dthrott TO 0.05 * (4.0 - gforce).
}
```

```
WHEN SHIP:ALTITUDE > 30000 THEN {
    LOCK dthrott TO 0.05 * (5.0 - gforce).
}
UNTIL SHIP:ALTITUDE > 40000 {
    SET throtte to throtte + dthrott.
    WAIT 0.1.
}
```

This example does what is expected of it without problems. But the ship's altitude is being checked at least five times for every update, including the UNTIL loop check. Certainly, the kOS CPU can keep up with this, however, one can imagine a whole series of WHEN/THEN statements which make use of complicated calculations based on atmospheric data or orbital mechanics. One way to minimize the trigger condition checking is to take strictly-sequential triggers and nest them:

```
WHEN STAGE:LIQUIDFUEL < 0.1 THEN {
    STAGE.
    PRESERVE.
}
SET throtte TO 1.
SET dthrott TO 0.
LOCK THROTTLE TO throtte.
LOCK STEERING TO R(0,0,-90) + HEADING(90,90).
STAGE.
WHEN SHIP:ALTITUDE > 1000 THEN {
    SET g TO KERBIN:MU / KERBIN:RADIUS^2.
    LOCK accvec TO SHIP:SENSORS:ACC - SHIP:SENSORS:GRAV.
    LOCK gforce TO accvec:MAG / g.
    LOCK dthrott TO 0.05 * (1.2 - gforce).

    WHEN SHIP:ALTITUDE > 10000 THEN {
        LOCK dthrott TO 0.05 * (2.0 - gforce).

        WHEN SHIP:ALTITUDE > 20000 THEN {
            LOCK dthrott TO 0.05 * (4.0 - gforce).

            WHEN SHIP:ALTITUDE > 30000 THEN {
                LOCK dthrott TO 0.05 * (5.0 - gforce).
            }
        }
    }
}
UNTIL SHIP:ALTITUDE > 40000 {
    SET throtte to throtte + dthrott.
    WAIT 0.1.
}
```

Now this is quite elegant! The number of triggers have been reduced to two per update for the entire running of this script. The trigger at 1km sets up the next trigger which will happen at 10km which sets up then next at 20km and so on. This can save a lot of processing time for triggers that will happen sequentially. As a general rule, one should try to nest WHEN/THEN statements whenever possible. Again, both examples above will work, but when scripts start to have deep and complicated triggers, this nested construct can save it from the dreaded kOS trigger limit.

2.3 PID Loops in kOS

This tutorial covers how one can implement a PID loop using kOS. A P-loop, or "proportional feedback loop" was already introduced in the second section of the *Design Patterns Tutorial*, and that will serve as our starting point.

After some code rearrangement, the integral and derivative terms will be added and discussed in turn. Next, a couple extra features will be added to the full PID-loop. Lastly, we'll show a case-study in tuning a full PID loop using the Ziegler-Nichols method. We'll use the LOG method to dump telemetry from KSP into a file and our favorite graphing software to visualize the data.

The code examples in this tutorial can be tested with a similar rocket design as shown. Do not forget the accelerometer, gravioli detector or the kOS CPU module. The engine is purposefully overpowered to demonstrate the feedback in action.

Those fuel-tank adapters are from the Modular Rocket Systems (MRS) addon, but stock tanks will work just fine. The design goal of this rocket is to have a TWR of 8 on the launchpad and enough fuel to make it past 30km when throttled for optimal atmospheric efficiency.

Contents

2.3.1 Proportional Feedback Loop (P-loop)

The example code from the *Design Patterns Tutorial*, with some slight modifications looks like the following:

```
// staging, throttle, steering, go
WHEN STAGE:LIQUIDFUEL < 0.1 THEN {
    STAGE.
    PRESERVE.
}
LOCK THROTTLE TO 1.
LOCK STEERING TO R(0,0,-90) + HEADING(90,90).
STAGE.
WAIT UNTIL SHIP:ALTITUDE > 1000.

// P-loop setup
SET g TO KERBIN:MU / KERBIN:RADIUS^2.
LOCK accvec TO SHIP:SENSORS:ACC - SHIP:SENSORS:GRAV.
LOCK gforce TO accvec:MAG / g.
LOCK dthrott TO 0.05 * (1.2 - gforce).

SET throtт TO 1.
LOCK THROTTLE to throtт.

UNTIL SHIP:ALTITUDE > 40000 {
    SET throtт to throtт + dthrott.
    WAIT 0.1.
}
```

The first several lines sets up a simple staging condition, puts the throttle to maximum, steers the rocket straight up and launches. The rocket is assumed to use only liquid fuel engines. After the rocket hits 1km, the script sets up the LOCK used in the P-loop which is updated every 0.1 seconds in the UNTIL loop. The use of LOCK variables makes this code fairly clean. When the script comes up to the first line in the UNTIL loop, i.e. "SET throtт TO throtт + dthrott.", the variable dthrott is evaluated which causes the LOCK on gforce to be evaluated which in-turn causes accvec to be evaluated.

The input to this feedback loop is the acceleration experienced by the ship (gforce) in terms of Kerbin's gravitational acceleration at sea level (g). The variable accvec is the total acceleration vector and is obtained by the accelerometer and gravioli detectors, both of which must be on the ship for this to work. The variable dthrott is the change in throttle that should be applied in a single iteration of the feedback loop.

In terms of a PID loop, the factor 1.2 is called the setpoint, gforce is the process variable and 0.05 is called the proportional gain. The setpoint and gain factors can be promoted to their own variables with names. Also, the code up to and including the "WAIT UNTIL SHIP:ALTITUDE > 1000." will be implied for the next few examples of code:

```
// P-loop
SET g TO KERBIN:MU / KERBIN:RADIUS^2.
LOCK accvec TO SHIP:SENSORS:ACC - SHIP:SENSORS:GRAV.
```

```
LOCK gforce TO accvec:MAG / g.

SET gforce_setpoint TO 1.2.
SET Kp TO 0.05.
LOCK dthrott TO Kp * (gforce_setpoint - gforce).

SET throtт TO 1.
LOCK THROTTLE to throtт.

UNTIL SHIP:ALTITUDE > 40000 {
    SET throtт to throtт + dthrott.
    WAIT 0.1.
}
```

This is not a big change, but it will set us up to include the integral and derivative terms in the next section.

2.3.2 Proportional-Integral Feedback Loop (PI-loop)

Adding the integral term requires us to keep track of time. This is done by introducing a variable (t0) to store the time of the last iteration. Now, the throttle is changed only on iterations where some time has elapsed so the WAIT time in the UNTIL can be brought to 0.001. The offset of the gforce has been set to the variable P, and the integral gain to Ki.

```
// PI-loop
SET g TO KERBIN:MU / KERBIN:RADIUS^2.
LOCK accvec TO SHIP:SENSORS:ACC - SHIP:SENSORS:GRAV.
LOCK gforce TO accvec:MAG / g.

SET gforce_setpoint TO 1.2.

LOCK P TO gforce_setpoint - gforce.
SET I TO 0.

SET Kp TO 0.01.
SET Ki TO 0.006.

LOCK dthrott TO Kp * P + Ki * I.

SET throtт TO 1.
LOCK THROTTLE to throtт.

SET t0 TO TIME:SECONDS.
UNTIL SHIP:ALTITUDE > 40000 {
    SET dt TO TIME:SECONDS - t0.
    IF dt > 0 {
        SET I TO I + P * dt.
        SET throtт to throtт + dthrott.
        SET t0 TO TIME:SECONDS.
    }
    WAIT 0.001.
}
```

Adding the integral term has the general effect of stabilizing the feedback loop, making it less prone to oscillating due to rapid changes in the process variable (gforce, in this case). This is usually at the expense of a longer settling time.

2.3.3 Proportional-Integral-Derivative Feedback Loop (PID-loop)

Incorporating the derivative term (D) and derivative gain (Kd) requires an additional variable (P0) to keep track of the previous value of the proportional term (P).

```
// PID-loop
SET g TO KERBIN:MU / KERBIN:RADIUS^2.
LOCK accvec TO SHIP:SENSORS:ACC - SHIP:SENSORS:GRAV.
LOCK gforce TO accvec:MAG / g.

SET gforce_setpoint TO 1.2.

LOCK P TO gforce_setpoint - gforce.
SET I TO 0.
SET D TO 0.
SET P0 TO P.

SET Kp TO 0.01.
SET Ki TO 0.006.
SET Kd TO 0.006.

LOCK dthrott TO Kp * P + Ki * I + Kd * D.

SET thrott TO 1.
LOCK THROTTLE to thrott.

SET t0 TO TIME:SECONDS.
UNTIL SHIP:ALTITUDE > 40000 {
    SET dt TO TIME:SECONDS - t0.
    IF dt > 0 {
        SET I TO I + P * dt.
        SET D TO (P - P0) / dt.
        SET thrott to thrott + dthrott.
        SET P0 TO P.
        SET t0 TO TIME:SECONDS.
    }
    WAIT 0.001.
}
```

When tuned properly, the derivative term will cause the PID-loop to act quickly without causing problematic oscillations. Later in this tutorial, we will cover a way to tune a PID-loop using only the proportional term called the Zieger-Nichols method.

2.3.4 Final Touches

There are a few modifications that can make PID loops very robust. The following code example adds three range limits:

1. bounds on the Integral term which addresses possible integral windup
2. bounds on the throttle since it must stay in the range 0 to 1
3. a deadband to avoid changing the throttle due to small fluctuations

Of course, KSP is a simulator and small fluctuations are not observed in this particular loop. Indeed, the P-loop is sufficient in this example, but all these features are included here for illustration purposes and they could become useful for unstable aircraft or untested scenarios.

```
// PID-loop
SET g TO KERBIN:MU / KERBIN:RADIUS^2.
LOCK accvec TO SHIP:SENSORS:ACC - SHIP:SENSORS:GRAV.
LOCK gforce TO accvec:MAG / g.

SET gforce_setpoint TO 1.2.

LOCK P TO gforce_setpoint - gforce.
SET I TO 0.
SET D TO 0.
SET P0 TO P.

LOCK in_deadband TO ABS(P) < 0.01.

SET Kp TO 0.01.
SET Ki TO 0.006.
SET Kd TO 0.006.

LOCK dthrott TO Kp * P + Ki * I + Kd * D.

SET throtT TO 1.
LOCK THROTTLE to throtT.

SET t0 TO TIME:SECONDS.
UNTIL SHIP:ALTITUDE > 40000 {
    SET dt TO TIME:SECONDS - t0.
    IF dt > 0 {
        IF NOT in_deadband {
            SET I TO I + P * dt.
            SET D TO (P - P0) / dt.

            // If Ki is non-zero, then limit Ki*I to [-1,1]
            IF Ki > 0 {
                SET I TO MIN(1.0/Ki, MAX(-1.0/Ki, I)).
            }

            // set throttle but keep in range [0,1]
            SET throtT to MIN(1, MAX(0, throtT + dthrott)).

            SET P0 TO P.
            SET t0 TO TIME:SECONDS.
        }
    }
    WAIT 0.001.
}
```

2.3.5 Tuning a PID-loop

We are going to start with the same rocket design we have been using so far and actually tune the PID-loop using the Ziegler-Nichols method. This is where we turn off the integral and derivative terms in the loop and bring the proportional gain (Kp) up from zero to the point where the loop causes a steady oscillation with a measured period (Tu). At this point, the proportional gain is called the "ultimate gain" (Ku) and the actual gains (Kp, Ki and Kd) are set according to this table taken from wikipedia:

Control Type	Kp	Ki	Kd
P	0.5 Ku		
PI	0.45 Ku	1.2 Kp / Tu	
PD	0.8 Ku		Kp Tu / 8
classic PID	0.6 Ku	2 Kp / Tu	Kp Tu / 8
Pessen Integral Rule	0.7 Ku	0.4 Kp / Tu	0.15 Kp Tu
some overshoot	0.33 Ku	2 Kp / Tu	Kp Tu / 3
no overshoot	0.2 Ku	2 Kp / Tu	Kp Tu / 3

An immediate problem to overcome with this method is that it assumes a steady state can be achieved. With rockets, there is never a steady state: fuel is being consumed, altitude and therefore gravity and atmosphere is changing, staging can cause major upsets in the feedback loop. So, this tuning method will be some approximation which should come as no surprise since it will come from experimental observation. All we need is enough of a steady state that we can measure the oscillations - both the change in amplitude and the period.

The script we'll use to tune the highly overpowered rocket shown will launch the rocket straight up (using SAS) and will log data to an output file until it reaches 30km at which point the log file will be copied to the archive and the program will terminate. Also, this time the feedback loop will be based on the more realistic "atmospheric efficiency." The log file will contain three columns: time since launch, offset of atmospheric efficiency from the ideal (in this case, 1.0) and the ship's maximum thrust. The maximum thrust will increase monotonically with time (this rocket has only one stage) and we'll use both as the x-axis when plotting the offset on the y-axis.

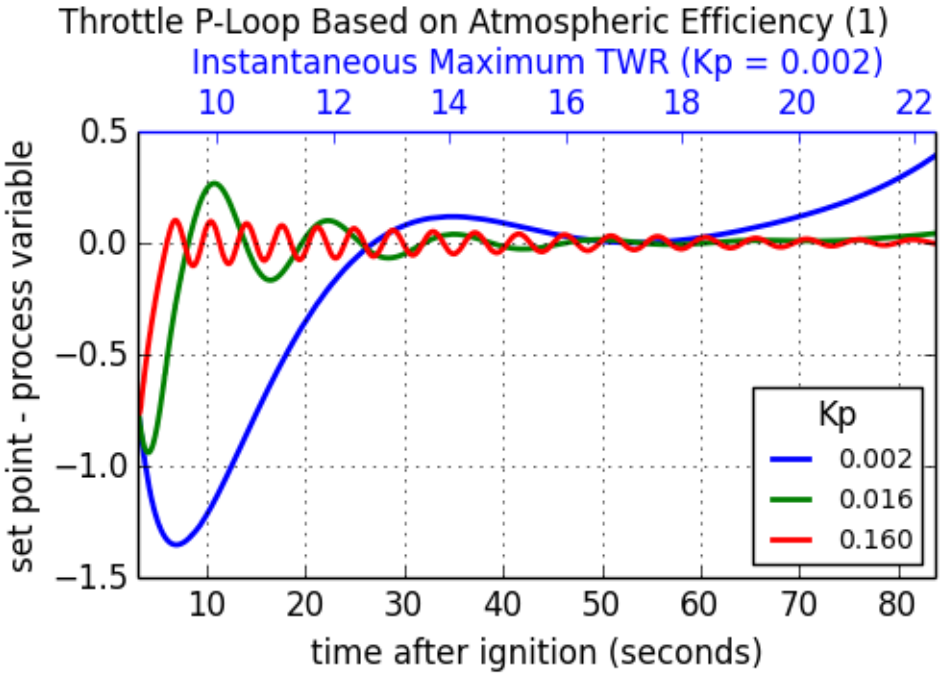
```
DECLARE PARAMETER Kp.

LOCK g TO SHIP:BODY:MU / (SHIP:BODY:RADIUS + SHIP:ALTITUDE)^2.
LOCK maxtwr TO SHIP:MAXTHRUST / (g * SHIP:MASS).

// feedback based on atmospheric efficiency
LOCK surfspeed TO SHIP:VELOCITY:SURFACE:MAG.
LOCK atmoeff TO surfspeed / SHIP:TERMVELOCITY.
LOCK P TO 1.0 - atmoeff.

SET t0 TO TIME:SECONDS.
LOCK dthrott TO Kp*P.
SET start_time TO t0.

LOG "# Throttle PID Tuning" TO throttle_log.
LOG "# Kp: " + Kp TO throttle_log.
LOG "# t P maxtwr" TO throttle_log.

LOCK logline TO (TIME:SECONDS - start_time)
        + " " + P
        + " " + maxtwr.

SET throtT TO 1.
LOCK THROTTLE TO throtT.
SAS ON.
STAGE.
WAIT 3.

UNTIL SHIP:ALTITUDE > 30000 {
    SET dt TO TIME:SECONDS - t0.
    IF dt > 0 {
        SET throtT TO MIN(1,MAX(0,throtT + dthrott)).
        SET t0 TO TIME:SECONDS.
        LOG logline TO throttle_log.
    }
    WAIT 0.001.
```

```
}
COPY throttle_log TO 0.
```

Give this script a short name, something like "tune.txt" so that running is simple:

```
copy tune from 0.
run tune(0.5).
```

After every launch completes, you'll have to go into the archive directory and rename the output logfile. Something like "throttle_log.txt" –> "throttle.01.log" will help if you increment the index number each time. To analyze the data, plot the offset (P) as a function of time (t). Here, we show the results for three values of Kp: 0.002, 0.016 and 0.160, including the maximum TWR when Kp = 0.002 as the top x-axis. The maximum TWR dependence on time is different for the three values of Kp, but not by a lot.

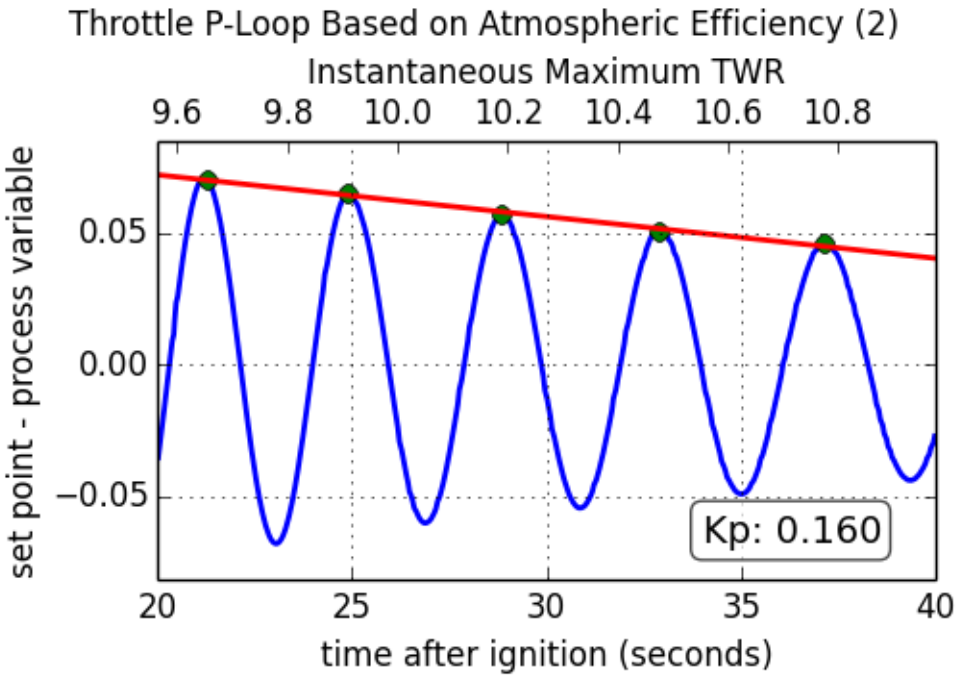


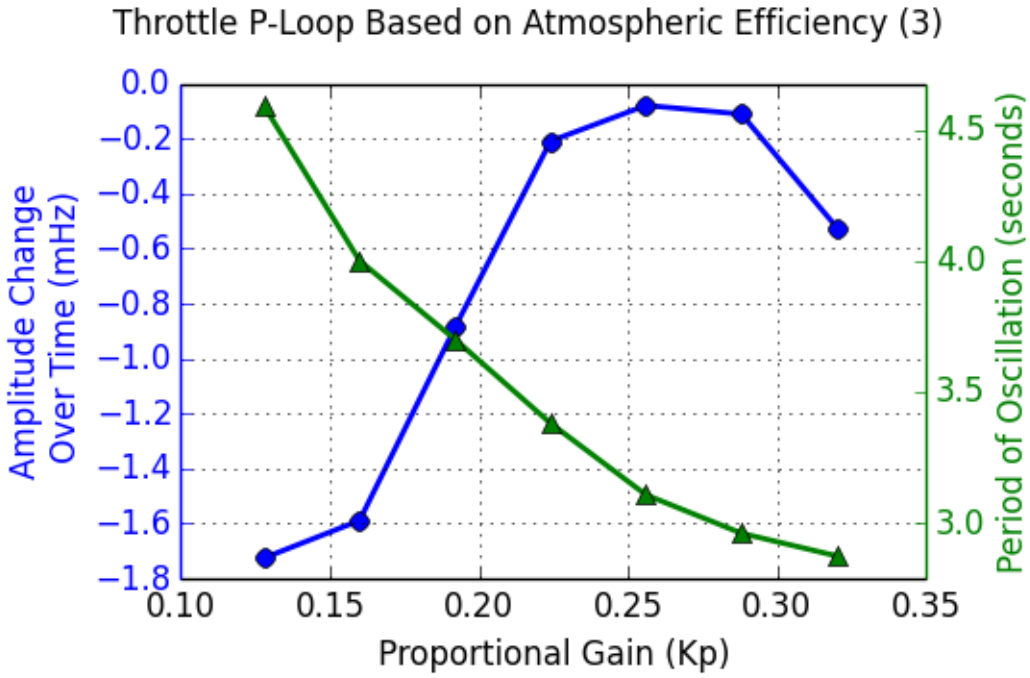
The value of 0.002 is obviously too low. The settling time is well over 20 seconds and the loop can't keep up with the increase in terminal velocity at the higher altitudes reached after one minute. When Kp = 0.016, the behavior is far more well behaved, and though some oscillation exists, it's damped and slow with a period of about 10 seconds. At Kp = 0.160, the oscillations are prominent and we can start to measure the change in amplitude along with the period of the oscillations. This plot shows the data for Kp = 0.160 from 20 to 40 seconds after ignition. The peaks are found and are fit to a line.

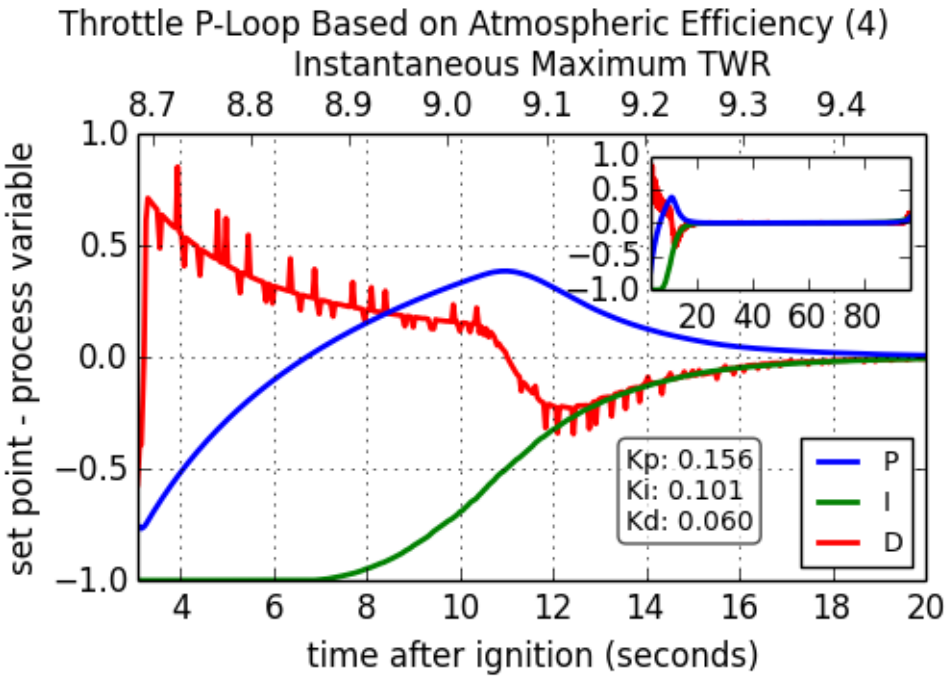
This is done for each value of Kp and the slopes of the fitted lines are plotted as a function of Kp in the following plot:

The period of oscillation was averaged over the interval and plotted on top of the amplitude change over time. Notice the turn over that occurs when Kp reaches approximately 0.26. This will mark the "ultimate gain" and 3.1 seconds will be used as the associated period of oscillation. It is left as an exercise for the reader to implement a full PID-loop using the classic PID values (see table above): Kp = 0.156, Ki = 0.101, Kd = 0.060, producing this behavior:

As soon as the PID-loop was activated at 3 seconds after ignition, the throttle was cut. At approximately 7 seconds, the atmospheric efficiency dropped below 100% and the integral term started to climb back to zero. At 11 seconds, the engine was reignited and the feedback loop settled after about 20 seconds. The inset plot has the same axes as the parent and shows the long-term stability of the final PID-loop.


Throttle P-Loop Based on Atmospheric Efficiency (2)
Instantaneous Maximum TWR
9.6
9.8
10.0
10.2
10.4
10.6
10.8
set point - process variable
0.05
0.00
−0.05
Kp: 0.160
20
25
30
35
40
time after ignition (seconds)


Throttle P-Loop Based on Atmospheric Efficiency (3)
Amplitude Change
Over Time (mHz)
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
−1.2
−1.4
−1.6
−1.8
Period of Oscillation (seconds)
4.5
4.0
3.5
3.0
0.10
0.15
0.20
0.25
0.30
0.35
Proportional Gain (Kp)



2.3.6 Final Thoughts

The classic PID values used above are fairly aggressive and there is some overshoot at the beginning. This can be dealt with in many ways and is discussed on the wikipedia page about PID controllers. For example, one might consider trying to implement a switch to a PD-loop when the integral term hits some limit, switching back once P crosses zero. The PID behavior should look like the following:

Finally, Controlling the throttle of a rocket is perhaps the easiest thing to implement as a PID loop in KSP using kOS. The steering was largely ignored and the orientation was always up. When writing an autopilot for horizontal atmospheric flight, one will have to deal with the direction the ship is traveling using SHIP:HEADING as well as it's orientation with SHIP:FACING. Additionally, there are the SHIP:ROTATION and SHIP:TRANSLATION vectors which can tell you the rate of change of the ship's facing and heading respectively. The controls in this case are six-dimensional using SHIP:CONTROL with YAW, PITCH, ROLL, FORE, STARBOARD, TOP and MAINTHROTTLE.

The PID gain parameters are dependent on the characteristics of the ship being controlled. The size, shape, turning capability and maximum TWR should be considered when tuning a PID loop. Turning RCS on can also have an effect and you might consider changing the PID loop's gain parameters every time to switch them on or off.

2.4 Advanced Tutorial

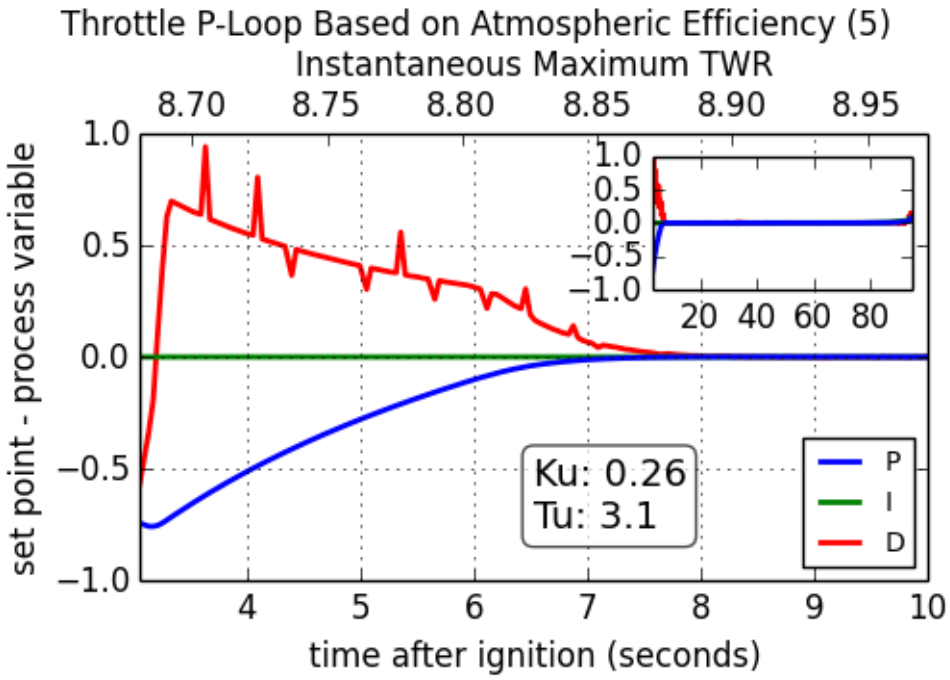
Let's try to automate one of the most common tasks in orbital maneuvering - execution of the maneuver node. In this tutorial I'll try to show you how to write a script for precise maneuver node execution.

So to start our script we need to get the next available *maneuver node*:

```
set nd to nextnode().
```

Our next step is to calculate how much time our vessel needs to burn at full throttle to execute the node:

```
//print out node's basic parameters - ETA and deltaV
print "Node in: " + round(nd:eta) + ", DeltaV: " + round(nd:deltav:mag).
```




```
//calculate ship's max acceleration
set max_acc to ship:maxthrust/ship:mass.

//now we just need to divide deltav:mag by our ship's max acceleration
set burn_duration to nd:deltav:mag/max_acc.
print "Estimated burn duration: " + round(burn_duration) + "s".
```

So now we have our node's deltav vector, ETA to the node and we calculated our burn duration. All that is left for us to do is wait until we are close to node's ETA less half of our burn duration. But we want to write a universal script, and some of our current and/or future ships can be quite slow to turn, so let's give us some time, 60 seconds, to prepare for the maneuver burn:

```
wait until node:eta <= (burn_duration/2 + 60).
```

This wait can be tedious and you'll most likely end up warping some time, but we'll leave kOS automation of warping for a given period of time to our readers.

The wait has finished, and now we need to start turning our ship in the direction of the burn:

```
set np to lookdirup(nd:deltav, ship:facing:topvector). //points to node, keeping roll the same.
lock steering to np.

//now we need to wait until the burn vector and ship's facing are aligned
wait until abs(np:pitch - facing:pitch) < 0.15 and abs(np:yaw - facing:yaw) < 0.15.

//the ship is facing the right direction, let's wait for our burn time
wait until node:eta <= (burn_duration/2)
```

Now we are ready to burn. It is usually done in the *until* loop, checking main parameters of the burn every iteration until the burn is complete:

```
//we only need to lock throttle once to a certain variable in the beginning of the loop, and adjust
set tset to 0.
```

```
lock throttle to tset.

set done to False.
//initial deltav
set dv0 to nd:deltav.
until done
{
    //recalculate current max_acceleration, as it changes while we burn through fuel
    set max_acc to ship:maxthrust/ship:mass.

    //throttle is 100% until there is less than 1 second of time left to burn
    //when there is less than 1 second - decrease the throttle linearly
    set tset to min(nd:deltav:mag/max_acc, 1).

    //here's the tricky part, we need to cut the throttle as soon as our nd:deltav and initial deltav
    //this check is done via checking the dot product of those 2 vectors
    if vdot(dv0, nd:deltav) < 0
    {
        print "End burn, remain dv " + round(nd:deltav:mag,1) + "m/s, vdot: " + round(vdot(dv0, nd:de
        lock throttle to 0.
        break.
    }

    //we have very little left to burn, less then 0.1m/s
    if nd:deltav:mag < 0.1
    {
        print "Finalizing burn, remain dv " + round(nd:deltav:mag,1) + "m/s, vdot: " + round(vdot(dv0
        //we burn slowly until our node vector starts to drift significantly from initial vector
        //this usually means we are on point
        wait until vdot(dv0, nd:deltav) < 0.5.

        lock throttle to 0.
        print "End burn, remain dv " + round(nd:deltav:mag,1) + "m/s, vdot: " + round(vdot(dv0, nd:de
        set done to True.
    }
}
unlock steering.
unlock throttle.
wait 1.

//we no longer need the maneuver node
remove nd.

//set throttle to 0 just in case.
SET SHIP:CONTROL:PILOTMAINTHROTTLE TO 0.
```

That is all, this short script can execute any maneuver node with 0.1 m/s dv precision or even better.

2.5 Introductory

Quick Start Tutorial Walks you through the beginnings of making a beginner's ship launcher script.

2.6 Intermediate

Design Patterns Tutorial Discusses some general aspects of kOS flow control and optimizations.

PID Loop Tutorial Starts with a basic proportional feedback loop and develops, in stages, a complete PID-loop to control the throttle of a simple rocket design.

Execute Node script ZiwKerman describes a generic "execute manuever node" script to be a one-size-fits-all solution to many situations in KSP. If you can make a manuever node for something, exenode will execute it.

CHAPTER

THREE

COMMUNITY EXAMPLES LIBRARY

Starting with version 0.17.0 of kOS, we have decided to support a separate repository of examples and libraries that "live" entirely in kerboscript code only. This is a useful place to find helpful code written by other users of kOS, some of whom may be members of the main kOS development team, and some of whom might ber users in the community.

The separate repository is found here:

https://github.com/KSP-KOS/KSLib

Some examples of useful things you can find there are:

- A generic all-purpose **PID controller** function:
  - library script: https://github.com/KSP-KOS/KSLib/blob/master/library/lib_pid.ks
  - documentation: https://github.com/KSP-KOS/KSLib/blob/master/doc/lib_pid.md
  - example https://github.com/KSP-KOS/KSLib/blob/master/examples/example_lib_pid.ks
- A library for getting **navball orientation** information:
  - library script https://github.com/KSP-KOS/KSLib/blob/master/library/lib_navball.ks
  - documentation https://github.com/KSP-KOS/KSLib/blob/master/doc/lib_navball.md
  - example https://github.com/KSP-KOS/KSLib/blob/master/examples/example_lib_navball.ks
- An example of how to use the *sasmode* feature:
  - example https://github.com/KSP-KOS/KSLib/blob/master/examples/example_testsasmode.ks

CHAPTER

FOUR

GENERAL TOPICS

These topics discuss the interfacing between **kOS** and **Kerbal Space Program**.

4.1 Catalog of Bound Variable Names

This is the list of special reserved keyword variable names that kOS will interpret to mean something special. If they are used as normal variable names by your kOS script program they may not work. Understanding them and their meaning is crucial to creating effective kOS scripts.

4.1.1 NAMED VESSELS AND BODIES

SHIP:

Variable name: SHIP
Gettable: yes
Settable: no
Type: Vessel
Description: Whichever vessel happens to be the one containing the CPU part that is running this Kerboscript code at the moment. This is the CPU Vessel.

TARGET:

Variable Name: TARGET
Gettable: yes
Settable: yes
Type: Vessel or Body
Description: Whichever Orbitable object happens to be the one selected as the current KSP target. If set to a string, it will assume the string is the name of a vessel being targetted and set it to a vessel by that name. For best results set it to Body("some name") or Vessel("some name") explicitly.

4.1.2 Alias shortcuts for SHIP fields

The following are all alias shortcuts for accessing the fields of the SHIP vessel. To see their definition, please consult the Vessel page, as they are all just instances of the standard vessel suffixes.

Variable	Same as
HEADING	Same as SHIP:HEADING
PROGRADE	Same as SHIP:PROGRADE
RETROGRADE	Same as SHIP:RETROGRADE
FACING	Same as SHIP:FACING
MAXTHRUST	Same as SHIP:MAXTHRUST
VELOCITY	Same as SHIP:VELOCITY
GEOPOSITION	Same as SHIP:GEOPOSITION
LATITUDE	Same as SHIP:LATITUDE
LONGITUDE	Same as SHIP:LONGITUDE
UP	Same as SHIP:UP
NORTH	Same as SHIP:NORTH
BODY	Same as SHIP:BODY
ANGULARMOMENTUM	Same as SHIP:ANGULARMOMENTUM
ANGULARVEL	Same as SHIP:ANGULARVEL
ANGULARVELOCITY	Same as SHIP:ANGULARVEL
COMMRANGE	Same as SHIP:COMMRANGE
MASS	Same as SHIP:MASS
VERTICALSPEED	Same as SHIP:VERTICALSPEED
GROUNDSPEED	Same as SHIP:GROUNDSPEED
SURFACESPEED	This has been obsoleted as of kOS 0.18.0. Replace it with GROUNDSPEED.
AIRSPEED	Same as SHIP:AIRSPEED
VESSELNAME	Same as SHIP:VESSELNAME
ALTITUDE	Same as SHIP:ALTITUDE
APOAPSIS	Same as SHIP:APOAPSIS
PERIAPSIS	Same as SHIP:PERIAPSIS
SENSORS	Same as SHIP:SENSORS
SRFPROGRADE	Same as SHIP:SRFPROGRADE
SRFREROGRADE	Same as SHIP:SRFREROGRADE
OBT	Same as SHIP:OBT
STATUS	Same as SHIP:STATUS
SHIPNAME	Same as SHIP:NAME

4.1.3 Constants (pi, e, etc)

Get-only.

The variable `constant` provides a way to access a few *basic math and physics constants*, such as Pi, Euler's number, and so on.

Example:

```
print "Kerbin's circumference: " + (2*constant:pi*Kerbin:radius) + "meters.".
```

The full list is here: *constants page*.

4.1.4 Terminal

Get-only. `terminal` returns a *terminal* structure describing the attributes of the current terminal screen associated with the CPU this script is running on.

4.1.5 Core

Get-only. `core` returns a *`core`* structure referring to the CPU you are running on.

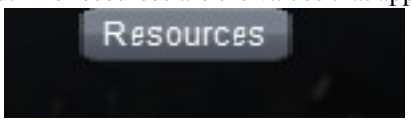
4.1.6 Stage

Get-only. `stage` returns a *`stage`* structure used to count resources in the current stage. Not to be confused with the COMMAND stage which triggers the next stage.

4.1.7 NextNode

Get-only. `nextnode` returns the next planned manuever `node` in the SHIP's flight plan. Bombs out if no such node exists.

4.1.8 Resource Types

Any time there is a resource on the ship it can be queried. The resources are the values that appear when you click on the upper-right corner of the screen in the KSP window. 


```
LIQUIDFUEL
OXIDIZER
ELECTRICCHARGE
MONOPROPELLANT
INTAKEAIR
SOLIDFUEL
```

All of the above resources can be queried using either the prefix SHIP or STAGE, depending on whether you are trying to query how much is left in the current stage or the entire ship:

How much liquid fuel is left in the entire ship:

```
PRINT "There is " + SHIP:LIQUIDFUEL + " liquid fuel on the ship.".
```

How much liquid fuel is left in just the current stage:

```
PRINT "There is " + STAGE:LIQUIDFUEL + " liquid fuel in this stage.".
```

How much liquid fuel is left in the target vessel:

```
PRINT "There is " + TARGET:LIQUIDFUEL + " liquid fuel in the target ship.".
```

Any other resources that you have added using other mods should be query-able this way, provided that you spell the term exactly as it appears in the resources window.

You can also get a list of all resources, either in SHIP: or STAGE: with the :RESOURCES suffix.

4.1.9 ALT ALIAS

The special variable ALT gives you access to a few altitude predictions:

ALT:APOAPSIS

ALT:PERIAPSIS

ALT:RADAR

Further details are found on the ALT page .

4.1.10 ETA ALIAS

The special variable ETA gives you access to a few time predictions:

ETA:APOAPSIS

ETA:PERIAPSIS

ETA:TRANSITION

Further details are found on the ETA page .

4.1.11 ENCOUNTER

The orbit patch describing the next encounter with a body the current vessel will enter. If there is no such encounter coming, it will return the special string "None". If there is an encounter coming, it will return an object *of type Orbit*. (i.e. to obtain the name of the planet the encounter is with, you can do: `print ENCOUNTER:BODY:NAME.`, for example.).

4.1.12 BOOLEAN TOGGLE FIELDS:

These are variables that behave like boolean flags. They can be True or False, and can be set or toggled using the "ON" and "OFF" and "TOGGLE" commands. Many of these are for action group flags. **NOTE ABOUT ACTION GROUP FLAGS:** If the boolean flag is for an action group, be aware that each time the user presses the action group keypress, it *toggles* the action group, so you might need to check for both the change in state from false to true AND the change in state from true to false to see if the key was hit.

Variable Name	Can Read	Can Set	Description
SAS	yes	yes	(Same as "SAS" indicator on the navball.)
RCS	yes	yes	(Same as "RCS" indicator on the navball.)
GEAR	yes	yes	Is the GEAR enabled right now? (Note, KSP does some strange things with this flag, like needing to hit it twice the first time).
LEGS	yes	yes	Are the landing LEGS extended? (as opposed to GEAR which is for the wheels of a plane.)
CHUTES	yes	yes	Are the parachutes extended? (Treats all parachutes as one single unit. Does not activate them individually.)
LIGHTS	yes	yes	Are the lights on? (like the "U" key in manual flight.)
PANELS	yes	yes	Are the solar panels extended? (Treats all solar panels as one single unit. Does not activate them individually.)
BRAKES	yes	yes	Are the brakes on?
ABORT	yes	yes	Abort Action Group.
AG1	yes	yes	Action Group 1.
AG2	yes	yes	Action Group 2.
AG3	yes	yes	Action Group 3.
AG4	yes	yes	Action Group 4.
AG5	yes	yes	Action Group 5.
AG6	yes	yes	Action Group 6.
AG7	yes	yes	Action Group 7.
AG8	yes	yes	Action Group 8.
AG9	yes	yes	Action Group 9.
AG10	yes	yes	Action Group 10.
AGn	yes	yes	If you have the Action Groups Extended mod installed, you can access its groups the same way, i.e. AG11, AG12, AG13, etc.

4.1.13 Flight Control

There are bound variables used in controlling the flight of a ship, which can be found at the following links:

If you want to let kOS do a lot of the work of aligning to a desired heading for you, use Cooked Control.

If you want your script to manipulate the controls directly (as in "set yaw axis halfway left for a few seconds (using the 'A' key)", then use Raw Control.

If you want to be able to READ what the player is attempting to do while your script is running, and perhaps respond to it, then use Reading the Pilot's Control settings (i.e reading what the manual input is attempting) (By default your script will override manual piloting attempts, but you can read what the pilot's controls are set at and make your autopilot take them under advisement - sort of like how a fly-by-wire plane works.)

Controls that must be used with LOCK

```
THROTTLE            // Lock to a decimal value between 0 and 1.
STEERING            // Lock to a direction, either a Vector or a Direction.
WHEELTHROTTLE       // Separate throttle for wheels
WHEELSTEERING       // Separate steering system for wheels
```

4.1.14 Time

Time is the simulated amount of time that passed since the beginning of the game's universe epoch. (A brand new campaign that just started begins at TIME zero.)

TIME is a useful system variable for calculating the passage of time between taking physical measurements (i.e. to calculate how fast a phenomenon is changing in a loop). It returns the KSP *simulated* time, rather than the actual realtime sitting in the chair playing the game. If everything is running smoothly on a fast computer, one second of simulated time will match one second of real time, but if anything is causing the game to stutter or lag a bit, then the simulated time will be a bit slower than the real time. For any script program trying to calculate physical properties of the KSP universe, the time that matters is the simulated time, which is what TIME returns.

It's important to be aware of the frozen update nature of the kOS computer when reading TIME.

4.1.15 System Variables

This section is about variables that describe the things that are slightly outside the simulated universe of the game and are more about the game's user interface or the kOS mod itself. They represent things that slightly "break the fourth wall" and let your script access something entirely outside the in-character experience.

```
PRINT VERSION.            // Returns operating system version number. e.g. 0.8.6
PRINT VERSION:MAJOR.      // Returns major version number. e.g. 0
PRINT VERSION:MINOR.      // Returns minor version number. e.g. 8
PRINT VERSION:BUILD.      // Returns build version number. e.g. 6
PRINT SESSIONTIME.        // Returns amount of time, in seconds, from vessel load.
```

NOTE the following important difference:

SESSIONTIME is the time since the last time this vessel was loaded from on-rails into full physics.

TIME is the time since the entire saved game campaign started, in the kerbal universe's time. i.e. TIME = 0 means a brand new campaign was just started.

Config

CONFIG is a special variable name that refers to the configuration settings for the kOS mod, and can be used to set or get various options.

CONFIG has its own page for further details.

WARP and WARPMODE

Time warp can be controlled with the variables WARP and WARPMODE. See *WARP*

MAPVIEW

A boolean that is both gettable and settable.

If you query MAPVIEW, it's true if on the map screen, and false if on the flight view screen. If you SET MAPVIEW, you can cause the game to switch between mapview and flight view or visa versa.

LOADDISTANCE

LOADDISTANCE sets the distance from the active vessel at which vessels get removed from the full physics engine and put on-rails, or visa versa. Note that as of KSP 1.0 the stock game supports multiple different load distance settings for different situations such that the value changes depending on where you are. But kOS does not support this at the moment so in kOS if you set the LOADDISTANCE, you are setting it to the same value universally for all situations.

4.1.16 SOLARPRIMEVECTOR

Gives the Prime Meridian *Vector* for the Solar System itself, in current Ship-Raw XYZ coordinates.

Both the *Orbit:LONGITUDEOFASCENDINGNODE* orbit suffix and the *Body:ROTATIONANGLE* body suffix are expressed in terms of degree offsets from this *Prime Meridian Reference Vector*.

What is the Solar Prime Reference Vector?

The solar prime vector is an arbitrary vector in space used to measure some orbital parameters that are supposed to remain fixed to space regardless of how the planets underneath the orbit rotate, or where the Sun is. In a sense it can be thought of as the celestial "prime meridian" of the entire solar system, rather than the "prime meridian" of any one particular rotating planet or moon.

In a hypothetical Earthling's solar system our Kerbal scientists have hypothesized may exist in a galaxy far away, Earthbound astronomers use a reference they called the First Point of Aries, for this purpose.

For Kerbals, it refers to a more arbitrary line in space, pointing at a fixed point in the firmament, also known as the "skybox".

4.1.17 Addons

Get-only. `addons` is a special variable used to access various extensions to kOS that are designed to support the features introduced by some other mods. More info can be found on the *addons* page.

4.1.18 Colors

There are several bound variables associated with *hardcoded colors* such as WHITE, BLACK, RED, etc. See the linked page for the full list.

4.2 CPU Vessel (SHIP)

Note: When kOS documentation refers to the "CPU vessel", it has the following definition:

- The "CPU Vessel" is whichever vessel happens to currently contain the CPU in which the executing code is running.

It's important to distinguish this from "active vessel", which is a KSP term referring to whichever vessel the camera is centered on, and therefore the vessel that will receive the keyboard controls for `W A S D` and so on.

The two terms can differ when you are in a situation where there are two vessels near each other, both of them within full physics range (i.e. 2.5 km), such as would happen during a docking operation. In such a situation it is possible for kOS programs to be running on one, both, or neither of the two vessels. The vessel on which a program is executing is not necessarily the vessel the KSP game is currently considering the "active" one.

Note: The built-in variable called `SHIP` is always set to the current CPU vessel. Whenever you see the documentation refer to CPU vessel, you can think of that as being "the `SHIP` variable".

For all places where a kOS program needs to do something with *this vessel*, for the sake of centering *SHIP-RAW* coordinates, for the sake of deciding which ship is having maneuver nodes added to it and for the sake of deciding which vessel is being controlled by the autopilot. The vessel it is referring to is itself the **CPU vessel** and not necessarily what KSP thinks of as the "active vessel".

4.3 The kOS CPU hardware

While it's possible to write some software without knowing anything about the underlying computer hardware, and there are good design principles that state one should never make assumptions about the computer hardware when writing software, there are still some basic things about how computers work in general that a good programmer needs to be aware of to write good code. Along those lines, the KSP player writing a Kerboscript program needs to know a few basic things about how the simulated kOS CPU operates in order to be able to write more advanced scripts. This page contains that type of information.

4.3.1 Update Ticks and Physics Ticks

Note: New in version 0.17: Previous versions of kOS used to execute program code during the Update phase, rather than the more correct Physics Update phase.

Kerbal Space Program simulates the universe by running the universe in small incremental time intervals that for the purpose of this document, we will call "**physics ticks**". The exact length of time for a physics tick varies as the program runs. One physics tick might take 0.09 seconds while the next one might take 0.085 seconds. (The default setting for the rate of physics ticks is 25 ticks per second, just to give a ballpark figure, but you **must not** write any scripts that depend on this assumption because it's a setting the user can change, and it can also vary a bit during play depending on system load. The setting is a target goal for the game to try to achieve, not a guarantee. If it's a fast computer with a speedy animation frame rate, it will try to run physics ticks less often than it runs animation frame updates, to try to make the physics tick rate match this setting. On the other hand, If it's a slow computer, it will try to sacrifice animation frame rate to archive this number (meaning physics get calculated faster than you can see the effects.)

When calculating physics formulas, you need to actually measure elapsed time in the TIME:SECONDS variable in your scripts.

The entire simulated universe is utterly frozen during the duration of a physics tick. For example, if one physics tick occurs at timestamp 10.51 seconds, and the next physics tick occurs 0.08 seconds later at timestamp 10.59 seconds, then during the entire intervening time, at timestamp 10.52 seconds, 10.53 seconds, and so on, nothing moves. The clock is frozen at 10.51 seconds, and the fuel isn't being consumed, and the vessel is at the same position. On the next physics tick at 10.59 seconds, then all the numbers are updated. The full details of the physics ticks system are more

complex than that, but that quick description is enough to describe what you need to know about how kOS's CPU works.

There is another kind of time tick called an **Update tick**. It is similar to, but different from, a **physics tick**. *Update ticks* often occur a bit more often than *physics ticks*. Update ticks are exactly the same thing as your game's Frame Rate. Each time your game renders another animation frame, it performs another Update tick. On a good gaming computer with fast speed and a good graphics card, It is typical to have about 2 or even 3 *Update ticks* happen within the time it takes to have one *physics tick* happen. On a slower computer, it is also possible to go the other way and have *Update ticks* happening *less* frequently than *physics tics*. Basically, look at your frame rate. Is it higher than 25 fps? If so, then your *update ticks* happen faster than your *physics ticks*, otherwise its the other way around.

Note: As of version 0.17.0, The kOS CPU runs every *physics tick*.

On each physics tick, each kOS CPU that's within physics range (i.e. 2.5 km), wakes up and performs the following steps, in this order:

1. Run the conditional checks of all TRIGGERS (see below)
2. For any TRIGGERS who's conditional checks are true, execute the entire body of the trigger.
3. If there's a pending WAIT statement, check if it's done. If so wake up.
4. If awake, then execute the next `Config:IPU` number of instructions of the main program.

Note that the number of instructions being executed (CONFIG:IPU) are NOT lines of code or kerboscript statements, but rather the smaller instruction opcodes that they are compiled into behind the scenes. A single kerboscript statement might become anywhere from one to ten or so instructions when compiled.

4.3.2 Triggers

There are multiple things within kerboscript that run "in the background" always updating, while the main script continues on. The way these work is a bit like a real computer's multithreading, but not *quite*. Collectively all of these things are called "triggers".

Triggers are all of the following:

- LOCKS which are attached to flight controls (THROTTLE, STEERING, etc), but not other LOCKS.
- ON condition { some commands }.
- WHEN condition THEN { some commands }.

Note: The *WAIT* command only causes mainline code to be suspended. Trigger code such as WHEN, ON, LOCK STEERING, and LOCK THROTTLE, will continue executing while your program is sitting still on the WAIT command.

The way these work is that once per **physics tick**, all the LOCK expressions which directly affect flight control are re-executed, and then each conditional trigger's condition is checked, and if true, then the entire body of the trigger is executed all the way to the bottom *before any more instructions of the main body are executed*. This means that execution of a trigger never gets interleaved with the main code. Once a trigger happens, the entire trigger occurs all in one go before the rest of the main body continues.

Do Not Loop a Long Time in a Trigger Body!

Because the entire body of a trigger will execute all the way to the bottom on *within a single* **physics tick**, *before* any other code continues, it is vital that you not write code in a trigger body that takes a long time to execute. The body of

a trigger must be kept quick. An infinite loop in a trigger body could literally freeze all of KSP, because the kOS mod will never finish executing its update.

As of kOS version 0.14 and higher, this condition is now being checked for and the script will be **terminated with a runtime error** if the triggers like WHEN/THEN and ON take more than `Config:IPU` instructions to execute. The sum total of all the code within your WHEN/THEN and ON code blocks MUST be designed to complete within one physicd tick.

This may seem harsh. Ideally, kOS would only generate a runtime error if it thought your script was stuck in an **infinite loop**, and allow it to exceed the `Config:IPU` number of instructions if it was going to finish and just needed a little longer to to finish its work. But, because of a well known problem in computer science called the halting problem, it's literally impossible for kOS, or any other software for that matter, to detect the difference between another program's infinite loop versus another program's loop that will end soon. kOS only knows how long your triggers have taken so far, not how long they're going to take before they're done, or even if they'll be done.

If you suspect that your trigger body would have ended if it was allowed to run a little longer, try setting your `Config:IPU` setting a bit higher and see if that makes the error go away.

If it does not make the error go away, then you will need to redesign your script to not depend on running a long-lasting amount of code inside triggers.

But I Want a Loop!!

If you want a trigger body that is meant to loop, the only acceptable way to do it is to design it to execute just once, but then use the PRESERVE keyword to keep the trigger around for the next physics update. Thus your trigger becomes a sort of "loop" that executes one iteration per **physics tick**.

It is also important to consider the way triggers execute for performance reasons too. Every time you write an expression for a trigger, you are creating a bit of code that gets executed fully to the end before your main body will continue, once each **physics tick**. A complex expression in a trigger condition, which in turn calls other complex LOCK expressions, which call other complex LOCK expressions, and so on, may cause kOS to bog itself down during each physics tick. (And as of version 0.14, it may cause kOS to stop your program and issue a runtime error if it's taking too long.)

Because of how WAIT works, you cannot put a WAIT statement inside a trigger. If you try, it will have no effect. This is because WAIT requires the ability of the program to go to sleep and then in a later physics tick, continue from where it left off. Because triggers run to the bottom entirely within one physics tick, they can't do that.

Wait!!!

Any WAIT statement causes the kerboscript program to immediately stop executing the main program where it is, even if far fewer than `Config:IPU` instructions have been executed in this **physics tick**. It will not continue the execution until at least the next **physics tick**, when it will check to see if the WAIT condition is satisfied and it's time to wake up and continue.

Therefore ANY WAIT of any kind will guarantee that your program will allow at least one **physics tick** to have happened before continuing. If you attempt to:

```
WAIT 0.001.
```

But the duration of the next physics tick is actually 0.09 seconds, then you will actually end up waiting at least 0.09 seconds. It is impossible to wait a unit of time smaller than one physics tick. Using a very small unit of time in a WAIT statement is an effective way to force the CPU to allow a physics tick to occur before continuing to the next line of code. Similarly, if you just say:

```
WAIT UNTIL TRUE.
```

Then even though the condition is immediately true, it will still wait one physics tick to discover this fact and continue.

Note: The *WAIT* command only causes mainline code to be suspended. Trigger code such as WHEN, ON, LOCK STEERING, and LOCK THROTTLE, will continue executing while your program is sitting still on the WAIT command.

4.3.3 The Frozen Universe

Each **physics** *tick*, the kOS mod wakes up and runs through all the currently loaded CPU parts that are in "physics range" (i.e. 2.5 km), and executes a batch of instructions from your script code that's on them. It is important to note that during the running of this batch of instructions, because no **physics ticks** are happening during it, none of the values that you might query from the KSP system will change. The clock time returned from the TIME variable will keep the same value throughout. The amount of fuel left will remain fixed throughout. The position and velocity of the vessel will remaining fixed throughout. It's not until the next physics tick occurs that those values will change to new numbers. It's typical that several lines of your kerboscript code will run during a single physics tick.

Effectively, as far as the *simulated* universe can tell, it's as if your script runs several instructions in literally zero amount of time, and then pauses for a fraction of a second, and then runs more instructions in literally zero amount of time, then pauses for a fraction of a second, and so on, rather than running the program in a smoothed out continuous way.

This is a vital difference between how a kOS CPU behaves versus how a real world computer behaves. In a real world computer, you would know for certain that time will pass, even if it's just a few picoseconds, between the execution of one statement and the next.

4.4 kOS Control Panel

As of kOS v0.15, kOS now makes use of Kerbal Space Program's built-in Application Launcher toolbar, to open a config/status panel for kOS.

This panel behaves like the other display panels the launcher creates, and operates mutually exclusively with them. (For example you can't make the kOS App Control Panel appear at the same time as the stock Resource display panel. Opening one toggles the other one off, and visa versa.)

Here is an annotated image of the control panel and what it does:

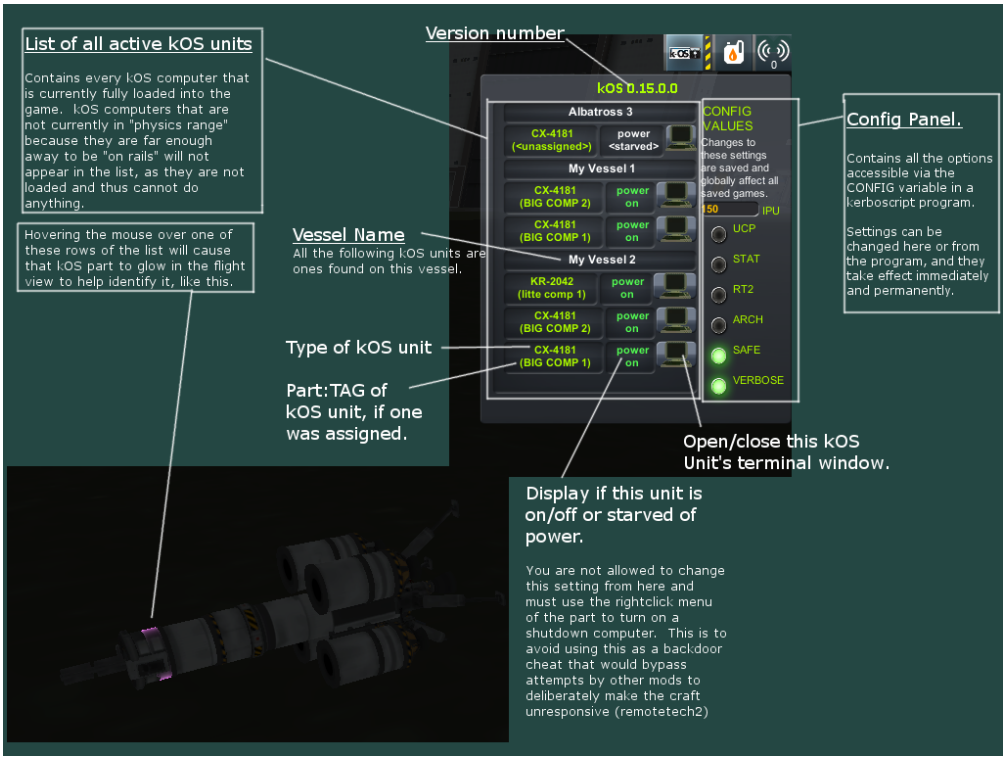
4.5 The kOS Telnet Server

kOS now supports the ability to enable a telnet server inside Kerbal Space Program.

Telnet is an old network protocol designed in the early days of the Internet, long before World Wide Web. Its purpose was (is) to allow you to get access to the remote command line interfaces of distant server computers, acting as if the keyboard and computer screen in front of you was a terminal hooked up to a distant computer. kOS uses this protocol to let you **access the kOS terminal from a program outside of Kerbal Space Program**.

There are freely available programs you can use as the telnet client to behave like terminal windows outside of the KSP window. A list of them appears below in the section called *Telnet clients*.

There are some *security implications* of enabling the kOS telnet server, and the first time you turn it on you will see some warning messages to this effect. If you want to read further about these concerns before deciding to turn it on, see the section called "*Security*" at the bottom of this page.



4.5.1 Telnet clients

The telnet server for kOS requires the use of a telnet client program. We recommend the following programs, although you can use others:

For Windows We recommend Putty, the free terminal emulator for Windows, although any good terminal emulator should do the job, provided it is capable of operating in an "XTERM - compatible" mode.

For Mac You shouldn't have to install anything. There should be a telnet client already installed, which you can access by opening up your command terminal, and then running it as a command-line tool. To see how to use it, read below in the section titled "*HOWTO: Command-line client*". The built-in Terminal.app for OSX understands the XTERM command sequences that kOS uses and in fact identifies itself as a type of XTERM when used with a telnet client.

For Linux You shouldn't have to install anything. There should be a telnet client already installed, and an xterm program already installed in most any Linux distribution. Open an xterm window, and in that window type the telnet command, as described by the section titled "*HOWTO: Command-line client*"

4.5.2 Using it

1. Turn on the telnet server by going into the app control panel and clicking on the green circle next to the word "Telnet". Alternatively, you can issue the command:

```
SET CONFIG:TELNET TO TRUE.
```

   from any terminal window in kOS.

2. The very first time you do this, you will get a warning message, as per SQUAD's rule number 5 about mods that run network services. After accepting and clicking "yes", the server will be running on loopback 127.0.0.1 (if you want to make it run on the non-loopback address, you will get a secondary warning message about that too.

3. Launch your telnet client (there is a list of telnet clients that are known to work listed below.

4. When you first log in to the server you should see the "Welcome menu", which is a screen looking like this:

```
Terminal: type = XTERM, size = 80x24
-------------------------------------------------------------------------------
                Menu GUI    Other
                Pick Open Telnets  Vessel Name (CPU tagname)
                ---- ---- --------  ------------------------------
                  [1]   no     0     Randy Viewer (CX-4181())
                  [2]   no     0     Randy Viewer (CX-4181())
                  [3]   no     0     Randy Viewer (CX-4181())
                  [4]   no     0     Randy Viewer (CX-4181())
                  [5]   no     0     Randy Viewer (CX-4181())
                  [6]   no     0     Randy Viewer (CX-4181())
-------------------------------------------------------------------------------
Choose a CPU to attach to by typing a selection number and pressing
return/enter. Or enter [Q] to quit terminal server.
```

```
(After attaching, you can (D)etach and return to this menu by pressing Control-D
as the first character on a new command line.)
--------------------------------------------------------------------------------
>_
```

Or, if there are no CPU's within range, it will look like this:

```
Terminal: type = XTERM, size = 80x24
________________________________________________________________________________

              Menu GUI   Other
              Pick Open Telnets  Vessel Name (CPU tagname)
              ---- ---- -------  --------------------------------
                                          <NONE>
```

At any moment you can force a redraw of the menu by entering any gibberish non- numeric data and hitting enter.

This menu should match 1:1 with the list of CPU's you see on the kOS applauncher control panel.

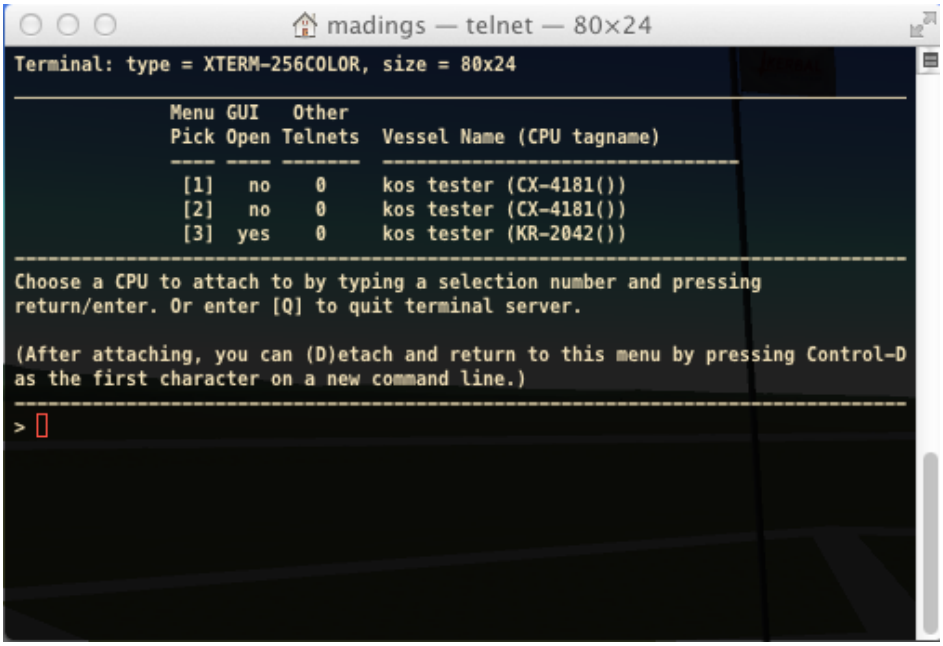


Fig. 4.1: The welcome menu, shown here in a Mac OSX terminal.

5. **Pick a CPU.** Pick one of the CPU's listed by typing its number and hitting enter.
6. Your telnet is now connected to the server and should behave as the terminal for that CPU. You can type commands and do what you like, the same as if you had been directly on its window.
7. See the section labeled *Special Keys* to see how to use the keyboard from your telnet client.
8. It is possible to have multiple terminals hooked up to the same in-game CPU. They will all behave as clones of each other, each being an equal "first citizen". (For example a pair of people could execute the "stage." command by having one of them type "st", then the other types "age", followed by the first person typing "." and the return key.) All the keyboards and all the screens are slaved together to be equal. You can view the in-game gui terminal while somebody is typing on a telnet temrinal.
9. In order to make the terminals act as clones of each other, the game will attempt to keep them all the same size. If you resize your telnet client window, it should cause the in-game window to change size to match. (If your terminal type is XTERM, then the same thing works in reverse. If it's VT100 then it doesn't.)

Warning: Certain implementations of the xterm terminal emulation and the telnet client have created a strange unending cascade of terminal resizes when you have two different telnet clients connected to the same GUI terminal and one of them is dragged to a new size. Because some implementations don't wait until they're done resizing to report their new size through telnet and instead report their intermediate sizes as they are being stretched, the attempt to keep them the same size causes them to effectively "argue" back and forth with each other, constantly changing each other's size. If you experience this problem (your terminal window will be flipping back and forth between two different sizes, resizing itself over and over again in a neverending loop), you can try to get out of it by issuing a hardcoded command to set the terminal size, such as:

```
SET TERMINAL:WIDTH TO 50.
```

Doing this should force all the connected telnet XTERM windows to stop arguing with each other about what the size is, and get them synced up again.

10. At any time you may disconnect your telnet client from the terminal by hitting control-D as the first character of a new line. This will bring you back to the telnet welcome menu again.

4.5.3 Special Keys

The following keys have special meaning in the telnet session:

Control-L Force refresh Pressing Control-L forces the kOS telnet server to redraw your whole screen for you from scratch. This is useful if you encounter strange line noise, interrupted messages, or for just any occasion where you suspect the screen isn't being drawn correctly. Pressing control-L will ensure your display gets fully re-synced with what's in the buffer in memory for the terminal.

Control-C interrupt process This is the same meaning as control-C in the normal GUI terminal - it breaks the program execution. The reason it gets a special mention here is that it also causes a flush of all the pending input you may have typed ahead in the queue. If you've been typing blindly ahead, and then hit Control-C, it will erase your typed-ahead keys as it sends the interrupt to the server. This is deliberate, and typical practice for an interrupt character sent over a remote shell setting.

Control-D detach If you hit control-D as the first character of a new line, it will detach your telnet session from the CPU and return you to the welcome menu.

Cursor Keys should be mapped If your terminal has identified itself as one of the known types that kOS supports, it should understand your arrow keys as arrow keys. If you see the text "[A" when you type up-arrow, or "[C" when you type right-arrow, this is a clue that kOS didn't recognize your terminal type properly.

Other Keys might be mapped Some keys like the Del (to the right), Home, and End keys are often not mapped correctly in some terminal emulator programs. If you have trouble using HOME and END, you can try Control-A and Control-E as alternates for Home and End.

Control-A home This is an alternate way to press the "home" key, just in case your terminal emulation isn't sending the officially understood terminal code for it.

Control-E end This is an alternate way to press the "end" key, just in case your terminal emulation isn't sending the officially understood terminal code for it.

Control-H backspace This is an alternate way to press the "backspace" key, just in case your terminal emulation isn't sending the officially understood terminal code for it.

Control-M Return This is an alternate way to press the "enter" or "return" key, just in case your terminal emulation isn't sending the officially understood terminal code for it.

4.5.4 HOWTO: Putty client

(These instructions assume you use the default kOS Telnet server settings, of the loopback address 127.0.0.1, and port number 5410. If you've changed those settings then alter the numbers you see here accordingly.)

1. Run KSP, and get it into a scene where there exists a vessel with at least one kOS CPU loaded into it.
2. Run Putty.
3. On the first dialog you see, click the *Telnet* radio-button selection.
4. Type in the number 127.0.0.1 in the large blank above the radio buttons that is labeled *"Host Name (or IP address)"*.
5. Type in the number 5410 in the smaller blank to the right of it that is labeled *"Port"*.
6. At the bottom of the screen, select the radio button labeled *"Never"* under *"Close window on exit"*.
7. Click the *Open* button to connect to the server.

(You can also save these settings under a name for later re-use.)

Step 6 is important. Without it, Putty would just make the window disappear any time there's a problem, making it very hard to diagnose because you can't see what message the server was sending back to you just before the window went away.

4.5.5 HOWTO: Command-line client

Fig. 4.2: Showing the use of telnet in an x-term window.

(These instructions assume you use the default kOS Telnet server settings, of the loopback address 127.0.0.1, and port number 5410. If you've changed those settings then alter the numbers you see here accordingly.)

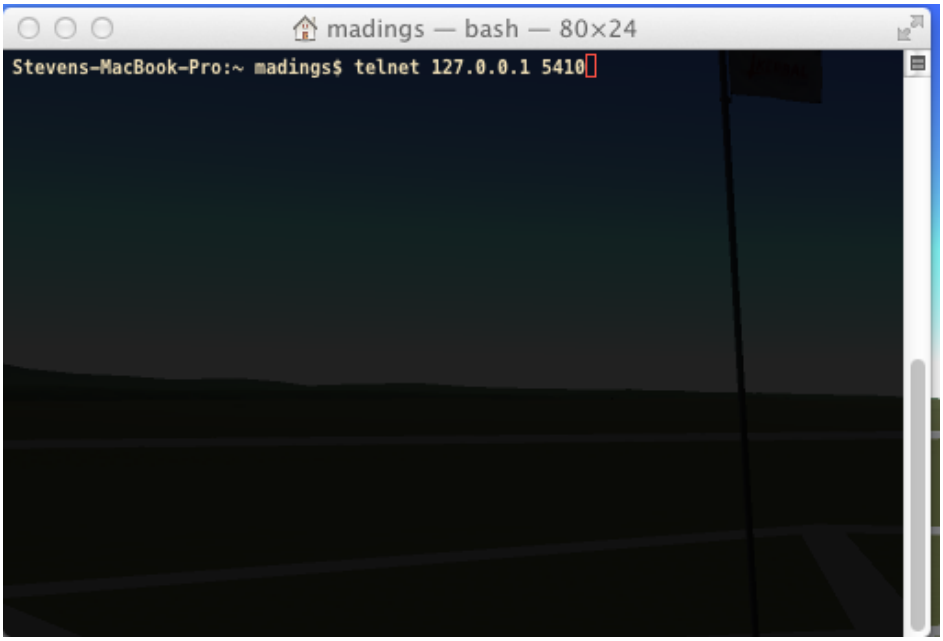


Fig. 4.3: Showing the use of telnet in a Mac OSX terminal.

1. Run KSP, and get it into a scene where there exists a vessel with at least one kOS CPU loaded into it.
2. Open a command shell window that either *IS* xterm, or emulates xterm. For OSX, the default command shell should work fine. For Linux, you should actually have the xterm program itself installed that you can use.
3. At the shell prompt in that window, enter the command:

```
telnet 127.0.0.1 5410
```

4.5.6 HOWTO: Other client

1. Set the IP address to 127.0.0.1 using whatever means the program has for it.
2. Set the port number to 5410 using whatever means the program has for it.
3. Set the terminal to XTERM emulation mode if it has it, or VT100 mode as a less good, but still perhaps workable option.
4. Run the terminal.

4.5.7 Security

The telnet protocol performs no encryption of its data, and as such any attempt at securing the system using a name/password combination would have been utterly pointless. Rather than provide a false sense of security that's not really there, we decided to make it obvious that there's no security by not even implementing a name and password for connecting to the kOS telnet server.

The purpose is to make it clear that if you want to open up your kOS telnet server, you need to be careful about how you do it.

The default settings that kOS ships with restricts your kOS telnet server to only operating on the loopback address (127.0.0.1) so that you won't accidentally open anything up to the public without thinking about it and making a

conscious decision to do so. If you don't know what that means, it means this: Any server that runs on the magic special address 127.0.0.1, known as "loopback", is incapable of taking connections from other computers besides itself.

In order to allow your kOS telnet server to take connections from other computers, you will typically need to do one of two things:

Either turn off the CONFIG:LOOPBACK option in your kOS install and then restart your telnet server (turn it off and on again using the button on the control panel), or (much better), set up a remote ssh tunnel that will map from your current machine's loopback address on the port number of your server to some remote other computer you want to connect from, to a port on it. The ssh tunnel is the preferred method, but describing how to set one up is beyond the scope of this document. You can read more For windows or For UNIX (both Mac and Linux).

Example: Let's say you have a remote Unix machine you'd like to enable logins from, from there and nowhere else. You can forward from your own machine's 127.0.0.1, port 5410, to the remote machine's, oh let's say 127.0.0.1, port 54100. Then anyone on the remote machine could telnet to ITS 127.0.0.1, port 54100 and end up talking to your machine's port 5410 on its loopback address.

Port forwarding

If you opt to turn off the loopback-only mode on your kOS telnet server, then you will probably also, if you have a typical home network setup, need to enable port forwarding on your router if you want people from outside your house to connect to it. (Again, think about the implications of doing so before you do it). This is a topic beyond the scope of this document, but help can be found out on the web for it. Search for "port forward home router". (It is probably also a good idea to include the make & model number of your router device in your search terms, to get a nicely narrowed result that's exactly what you need.)

Why not ssh?

The original plan for kOS was to include an ssh server instead of a telnet server. However this proved problematic as open source solutions in C# for the server-side of ssh were hard to come by (there's several for the client side only, and plenty of server-side code that's not in C#), and implementing the entire server side of the ssh protocol from scratch is a daunting task that would have taken too much time away from other development of kOS. (While implementing from scratch the server side of the older, simpler telnet protocol, while still work, was more doable).

4.5.8 Homemade telnet clients

This section is only of interest to hobbyists making Kerbal console hardware rigs and software developers trying to make interface mods that pretend to be kOS terminals. If you are neither of those two, then don't worry if this section looks like gibberish to you. It can be skipped.

TELNET PROTOCOL

If you wish to make your own homemade telnet client and connect it up to the kOS telnet server, the following is the required subset of the telnet protocol that your telnet client must speak, and the terminal requirements it must fulfill:

1. It must suppress local character echoing, and enter character-at-a-time mode, by implementing both the ECHO negotiation described by RFC857, and the SUPPRESS GO AHEAD negotiation described by RFC858. These are used in the following way: Your client must NOT ECHo (letting the server do it), and your client must suppress go-ahead messages (allowing real-time back-and-forth).
2. It must implement the underlying DO/DONT and WILL/WONT, and SB/SE infrastructure of the main telnet RFC854. It must send break (ctrl-C) as the IP interrupt process command (byte 255 followed by 244). kOS does not use much of the negotiations of the protocol mentioned on RFC854, other than those that are necessary to enable the other ones mentioned here.
3. It must implement the Terminal-Type option described by RFC1091. Furthermore, as of this writing, kOS only knows how to understand two terminal types, "XTERM", and "VT100". If your terminal type is identified as anything else, kOS may deny your connection, or at the very least just not work right. Even terminals that

are capable of emulating XTERM or VT100 commands won't work right if they don't identify themselves as XTERM or VT100. kOS does not know how to guess what emulation mode to enter if it doesn't recognize your terminal type string.

4. It must implement the NAWS, Negotiate About Terminal Size option, as described by RFC1073. kOS uses this to decide how to size its mental image of your terminal to match your terminal's real size. Note that this negotiation is one-way. Your client can use it to tell the server about its size, but the server can't use it to tell your client to change its size. Instead if your client can respond to changing sizes at the behest of the server, it must do so through terminal escape code characters sent back to it on the stream, above the telnet protocol layer itself. (For example, if you identify as XTERM, you will be sent the XTERM escape code pattern ESC [8 ; *height* ; *width* t, which is the XTERM escape code for setting the terminal size.) This is because the telnet protocol was never written to accommodate the concept of server-initiated resizes.

Making a telnet client from scratch that actually follows protocol may be a complex enough task that the smarter solution is to just use an existing telnet program, if you are trying to create some sort of hardware rig. These days a small cheap mini-hardware implementation of Linux should be doable, and could include the telnet client installed in it for very little storage cost.

TERMINAL EMULATION

As of right now, the terminal emulation of kOS only really supports XTERM or VT100 well, however the infrastructure is in place to support modifications to map to other terminal types. If you want to try a hand at adding the terminal emulation for a currently unsupported terminal, you'd do it by subclassing the kOS.UserIO.TerminalUnicodeMapper class. You can look at kOS.UserIO.TerminalXtermMapper as a sample to see what you need to do.

If you have a project where you want to just work with the terminal codes already supported, then these are the subset you need to support:

ASCII The following terms should have their normal ASCII meaning:

0x08 (control-H) backspace key

0x0d (control-M) Return key. On output it means go to left edge but don't go down a line. A typical eoln needs to occur using its ASCII standard of both a return character 0x0d AND a linefeed character 0x0a

0x0a (control-J) On output it means go to go down a line but don't go to the left edge A typical eoln needs to occur using its ASCII standard of both a return character 0x0d AND a linefeed character 0x0a

Terminal codes: *The following terms should have their VT100/XTERM meaning*

Left-Arrow ESC [D – *both on input and on output*

Right-Arrow ESC [C – *both on input and on output*

Up-Arrow ESC [A – *both on input and on output*

Down-Arrow ESC [B – *both on input and on output*

Home-key ESC [1 ~ – *input only*

End-key ESC [4 ~ – *input only*

Delete-to-the-right-key ESC [3 ~ – *input only*

PageUp-key ESC [5 ~ – *input only*

PageDown-key ESC [6 ~ – *input only*

Move-to-home-of-screen-upper-left ESC [H – *output only*

Move-to-end-of-line ESC [F – *output only*

Teleport-cursor-to-coordinate ESC [*row* ; *col* H – *output only: rows and cols start counting at 1, not 0*

Clearscreen ESC [2 J – *output only*

Scroll-screen-up-one-line-keeping-cursor-where-it-is ESC [S – *output only*

Scroll-screen-down-one-line-keeping-cursor-where-it-is ESC [T – *output only*

Delete-to-the-left-of-cursor-ie-backspace ESC [K – *output only*

Delete-at-the-cursor-toward-the-right ESC [1 K – *output only*

XTERM codes: *The following codes are for the XTERM emulation only*

Server-telling-client-to-resize-screen ESC [8 ; *newheight* ; *newwidth* t – *The height/width are in chars*

Server-telling-client-to-change-window-title ESC] 2 ; *title string* BEL – *where BEL is the character normally used to mean beep: control-G or 0x07. But in this context it just marks the end of the title and shouldn't cause a beep.* Note this is NOT a typo that it uses a right-square-bracket ("]") here where all the other codes used a left-square-bracket ("["). That's actually how the xterm control sequence for this really looks.

Any value not mentioned in the list above might still get sent, but you should be able to capture and ignore it.

4.6 Files and Volumes

Using the COPY, SWITCH, DELETE, and RENAME commands, you can manipulate the archive and the volumes as described in the *File I/O page*. But before you do that, it's useful to know how kOS manages the archive and the volumes, and what they mean.

Warning: Changed in version 0.15: **Archive location and file extension change**
The Archive where KerboScript files are kept has been changed from `Plugins/PluginData/Archive` to `Ships/Script`, but still under the top-level **KSP** installation directory. The file name extensions have also changes from `.txt` to `.ks`.

4.6.1 Script Files

There is one file per program. You can use the words "file" and "program" interchangeably in your thinking for the most part. Files are stored in volumes and there can be more than one file in a volume provided there's enough room. Volumes have small storage and there's no way to span a file across two volumes, so the limit to the size of a volume is also effectively a limit to the size of a program.

4.6.2 File Storage Behind the Scenes

In the Archive:

- If a file is stored on the volume called "Archive" (or volume number zero to put it another way), then behind the scenes it's really stored as an actual file, with the extension `.ks`, on your computer (As of right now it's located in `Ships/Script` but that location is likely to change to somewhere in GameData in a future version.) Each program is a simple text file you can edit with any text editor, and your edits will be seen immediately by KOS the next time it tries running that program from the archive volume.

- Historical note: older versions of kOS (0.14 and earlier) used the directory `Plugins/PluginData/Archive` for the archive.

In any other volume besides Archive:

- If a file is stored on any volume other than archive, then behind the scenes it's stored actually inside the saved game's persistence file in the section for the KOS part on the vessel. What's a Volume

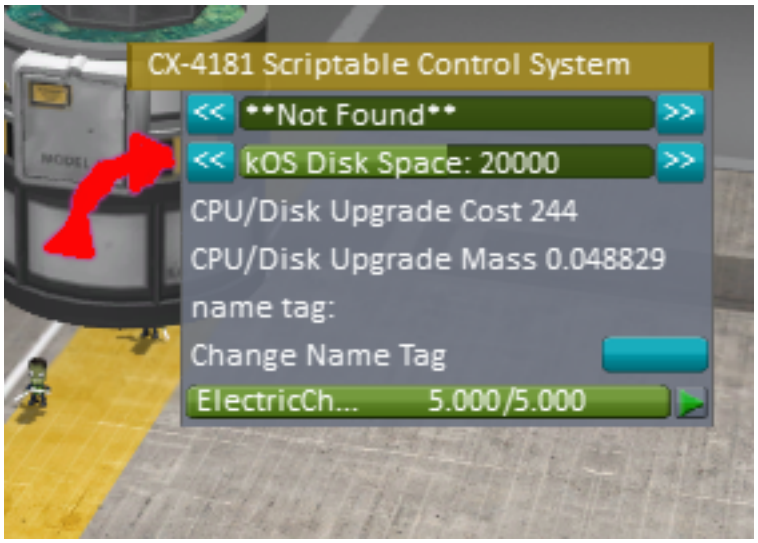
A Volume is a small unit of disk storage that contains a single hard drive with very limited storage capacity. It can store more than one program on it. To simulate the sense that this game takes place at the dawn of the space race with 1960's and 1970's technology, the storage capacity of a volume is very limited.

For example, the CX-4181 Scriptable Control System part defaults to only allowing 1000 bytes of storage.

The byte count of a program is just the count of the characters in the source code text. Writing programs with short cryptic variable names instead of long descriptive ones does save space, although you can also save space by compiling your programs to KSM files where the variable names are only stored once in the file, but that's another topic for another page.

Each of the computer parts that kOS supports have their own different default storage capacity limits for their local volume. As you get better parts higher up the tech tree, they come with bigger default size limits.

4.6.3 You can get more space by paying extra cost in money and mass



If you wish to have more disk space on your local volume, and are willing to pay a little extra cost in money and in mass, you can use the disk space slider in the vehicle assembly building to increase the limit.

Every part comes with 3 different multiplier options:

- 1x default size,
- 2x default size,
- 4x default size

The higher the multiplier the more mass it will cost you, to represent that you're using old storage technology, so it costs a lot of mass to have more storage.

The disk size is only settable like this in the assembly building. Once you launch a vessel, its volume size is stuck the way it was when you launched it.

4.6.4 Multiple Volumes on One Vessel

Each kOS CX-4181 Scriptable Control System part contains ''one'' such volume inside it.

If you have multiple CX-4181 parts on the same craft, they are assumed to be networked together on the same system, and capable of reading each other's hard drives. Their disk drives each have a different Volume, and by default they are simply numbered 1,2,3, unless you rename them with the RENAME command.

For example, if you have two CX-4181's on the same craft, called 1 and 2, with volumes on them called 1 and 2, respectively, it is possible for CPU 1 to run code stored on volume 2 by just having CPU number 1 issue the command ''SWITCH TO 2.''

4.6.5 Naming Volumes

It's important to note that if you have multiple volumes on the same vessel, the numbering conventions for the volumes will differ on different CPUs. The same volume which was called '2' when one CPU was looking at it might instead be called '1' when a different CPU is looking at it. Each CPU thinks of its OWN volume as number '1'.

Therefore using the RENAME command on the volumes is useful when dealing with multiple CX-4181's on the same vessel, so they all will refer to the volumes using the same names.

4.6.6 Archive

The "archive" is a special volume that behaves much like any other volume but with the following exceptions:

- It is globally the same even across save games.
- The archive represents the large bank of disk storage back at mission control's mainframe, rather than the storage on an indivdual craft. While "Volume 1" on one vessel might be a different disk than "Volume 1" on another vessel, there is only volume called "archive" in the entire solar system. Also, there's only one "archive" across all saved universes. If you play a new campaign from scratch, your archive in that new game will still have all the files in it from your previous saved game. This is because behid the scenes it's stored in the plugin's directory, not the save game directory.
- It is infinitely large.
- Unlike the other volumes, the archive volume does not have a byte limit. This is because the mainframe back at home base can store a lot more than the special drives sent on the vessels - so much so that it may as well be infinite by comparison.
- Files saved there do not revert when you "revert flight".
- When you revert a flight, you are going back to a previous saved state of the game, and therefore any disk data on the vessels themselves also reverts to what it was at the time of the saved game. Because the archive is saved outside the normal game save, changes made there are retained even when reverting a flight.
- It's not always reachable if you are out in space, unless you have antennae.
- Once a vessel is more than 100,000 meters away from mission control, by default it cannot access the files on the archive. Commands such as SWITCH TO , and COPY FROM will fail to work when trying to access the archive volume while out of range. This can be changed by putting antennae on the vessel. With enough antennae it becomes possible to reach the archive drive from farther away. Using this method it is possible to alter the software stored on a vessel after launch.
- Files in Archive are editable with a text editor directly and they will have the `.ks` extension.
- Files in the Archive are stored on your computer in the subdirectory: `Ships/Script`. You can pull them up in a text editor of your choice and edit them directly, and the KOS Mod will see those changes in its archive immediately. Files stored in other volumes, on the other hand, are stored inside the vessel's data in the persistence

file of the saved game and are quite a bit bit harder to edit there. Editing the files in the Archive directory is allowed and in fact is an officially accepted way to use the plugin. Editing the section in a persistence file, on the other hand, is a bad idea and probably constitutes a form of cheating similar to any other edit of the persistence file.

4.7 KerboScript Machine Code

4.7.1 Compiling to a KSM File

When you run your Kerboscript programs, behind the scenes they get compiled into a form in memory that runs much smoother but at the same time is quite hard for a Kerbal to read and understand. The actual computer hardware built by your friends at Compotronix incorporated actually run the program using these tiny instructions called "opcodes". In the early days of room-sized computers before we were able to get them down to the compact size of just a meter or so across, all programmers had to use this difficult system, referred to as "machine language". Ah, those were heady days, but they were hard.

The commands you actually write when you say something like `SET X TO 1.0.` are really a euphemism for these "machine language" opcodes under the surface.

When you try to "RUN" your script, the first thing that kOS does is transform your script into this ancient and arcane "machine language" form, storing it in its memory, and from then on it runs using that.

This process of transforming your script into Machine Language, or "ML" is called "Compiling".

The "RUN" command does this silently, without telling you. This is why you may have noticed the universe slightly stutter for a moment when you first run your program. Compiling is hard work for the universe to do.

Why Do I Care?

> **Warning:** This is an experimental feature.

The reason it matters is this: Although once it's loaded into memory and running, these opcodes actually have a lot of baggage and take up a lot of space, when they're stored passively on the disk not doing anything, they can be smaller than your script programs are. For one thing, they don't care about your comments (only other Kerbals reading your script do), and they don't care about your indenting (only other Kerbals reading your script do).

So, given that the compiled "ML" codes are the only thing your program really needs to be run, why not just store THAT instead of storing the entire script, and then you can put the ML files on your remote probes instead of putting the larger script files on them.

And THAT is the purpose of the COMPILE command.

It does some, but not all, of the compiling work that the RUN command does, and then stores the results in a file that you can run instead of running the original script.

The output of the COMPILE command is a file in what we call KSM format.

KSM stands for "KerboScript Machine code", and it has nearly the same information the program will have when it's loaded and running, minus a few extra steps about relocating it in memory.

4.7.2 How to Use KSM Files

Let's say that you have 3 programs your probe needs, called:

- myprog1.ks
- myprog2.ks
- myprog3.ks

And that myprog1 calls myprog2 and myprog3, and you normall would call the progam this way:

```
SWITCH TO 1.
COPY myprog1 from ARCHIVE.
COPY myprog2 from ARCHIVE.
COPY myprog3 from ARCHIVE.
RUN myprog1(1,2,"hello").
```

Then you can put just the compiled KSM versions of them on your vessel and run it this way:

```
SWITCH TO ARCHIVE.

COMPILE myprog1.ks to myprog1.ksm.
COPY myprog1.ksm to 1.

COMPILE myprog2. // If you leave the arguments off, it assumes you are going from .ks to .ksm
COPY myprog2.ksm to 1.

COMPILE myprog3. // If you leave the arguments off, it assumes you are going from .ks to .ksm
COPY myprog2.ksm to 1.

SWITCH TO 1.
RUN myprog1(1,2,"hello").
```

4.7.3 Default File Naming Conventions

When you have both a .ks and a .ksm file, the RUN command allows you to specify which one you meant explicitly, like so:

```
RUN myprog1.ks.
RUM myprog1.ksm.
```

But if you just leave the file extension off, and do this:

```
RUN myprog1.
```

Then the RUN command will first try to run a file called "myprog1.ksm" and if it cannot find such a file, then it will try to run one called "myprog1.ks".

In this way, if you decide to take the plunge and attempt the use of KSM files, you shouldn't have to change the way any of your scripts call each other, provided you just used versions of the filenames without mentioning the file extensions.

4.7.4 Downsides to Using KSM Files

1. Be aware that if you use this feature, you do lose the ability to have the line of code printed out for you when the kOS computer finds an error in your program. It will still tell you what line number the error happened on, but it cannot show you the line of code. Just the number.
2. Know that you cannot view the program inside the in-game editor anymore when you do this. A KSM file will not appear right in the editor. It requires a magic tool called a "hex editor" to properly see what's happening inside the file.
3. **The file isn't always smaller**. There's a threshold at which the KSM file is actually bigger than the source KS file. For large KS files, the KSM file will be smaller, but for short KS files, the KSM file will be bigger, because there's a small amount of overhead they have to store that is only efficient if the data was large enough.

4.7.5 More Reading and Fiddling with Your Bits

So, if you are intrigued by all this and want to see how it all *REALLY* works under the hood, Computronix has deciced to make internal document MLfile-zx1/a on the basic plan of the ML file system open for public viewing, if you are one of those rare Kerbals that enjoys fiddling with your bits. No, not THOSE kind of bits, the computery kind!

4.8 The Name Tag System

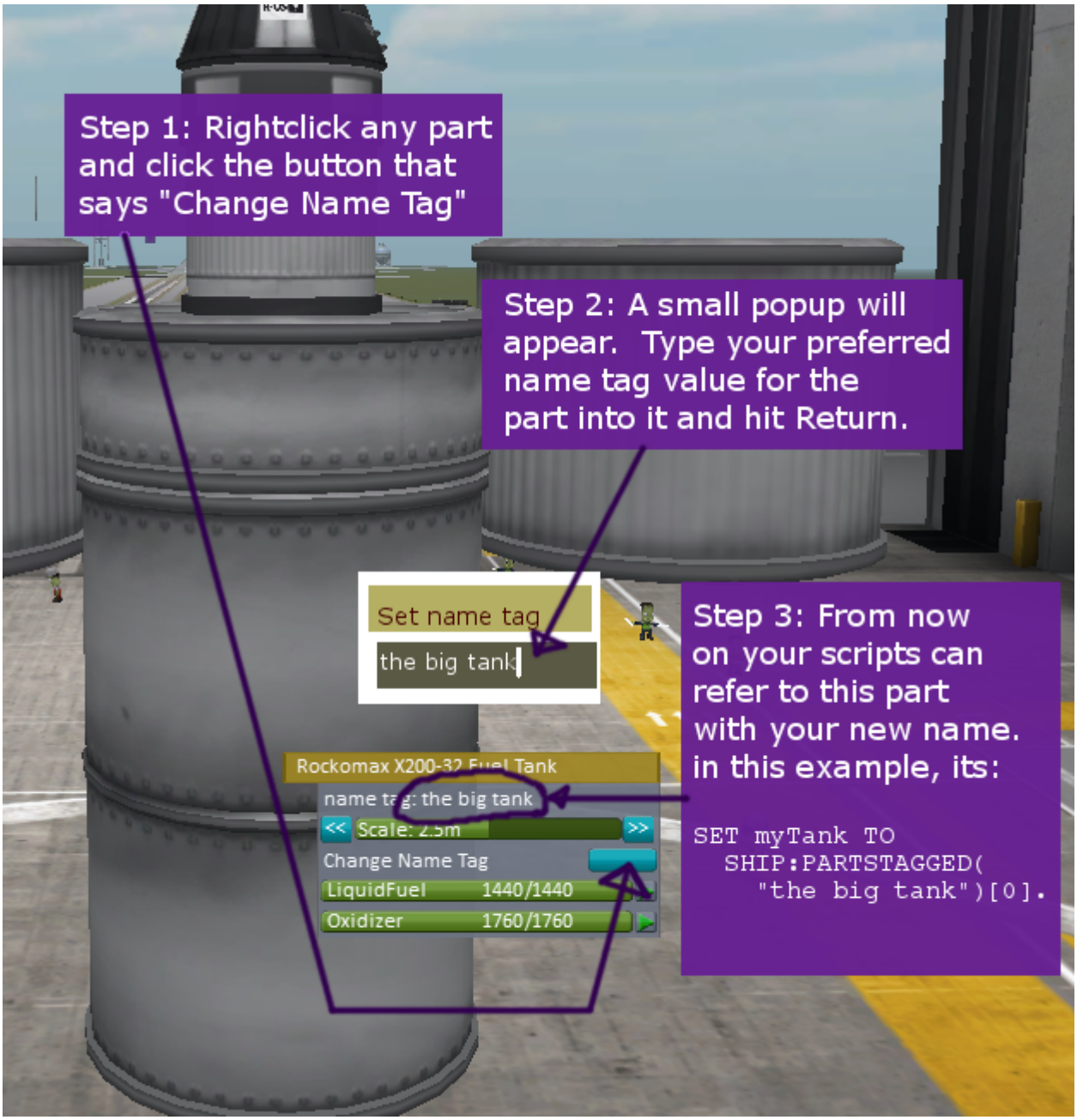
One useful thing to do is to be able to give parts on a craft your own chosen names that you can use to get a reference to the parts, utterly bypassing the naming schemes used by KSP, and ignoring the complex part tree system. The Name Tag system was designed to make it possible for you to just give any part on a craft whatever arbitrary name you feel like.

4.8.1 Giving a Part a Name

1. Right click on any part. **You can do this in either the editor VAB/SPH, or during flight.** It is more useful to do beforehand when making the craft, but you can do it in the midst of a flight, which is useful for experimenting and testing.
2. Click the "change name tag" button. A small text pop-up will appear with the ability to type a name for the part. Type any value you like. In the example picture here, the value "the big tank" was typed in.
3. From now on this part can be retrieved using either the :PARTSDUBBED or the :PARTSTAGGED methods of the vessel, as described on the Vessel Page

4.8.2 Cloning with Symmetry

If you name a part in the editor *first*, and then remove it from the vessel and place it again with symmetry, all the symmetry copies of the part will get the same nametag cloned to them. This is not an error. It is allowed to give the same nametag to more than one part. If you do this, it just means that the :PARTSTAGGED or :PARTSDUBBED methods will return lists containing more than one part.


Step 1: Rightclick any part and click the button that says "Change Name Tag"
Step 2: A small popup will appear. Type your preferred name tag value for the part into it and hit Return.
Set name tag
the big tank
Step 3: From now on your scripts can refer to this part with your new name. in this example, its:
SET myTank TO
SHIP:PARTSTAGGED(
"the big tank")[0].
Rockomax X200-32 Fuel Tank
name tag: the big tank
<<
Scale: 2.5m
>>
Change Name Tag
LiquidFuel 1440/1440
Oxidizer 1760/1760

If you want each part in a symmetrical group to get a unique nametag, you can change their names one at a time after you've placed the parts.

4.8.3 Where is it saved?

If you add a nametag to a part in the editor (VAB or SPH), then that nametag will get saved inside the craft file for that vessel design. All new instances of that craft design that you launch will get the same nametag configuration on them.

On the other hand, if you added a nametag to a part after the vessel was launched, during flight or on the launchpad, then that nametag will only be attached to that part on *that one instance* of the vessel. Other copies of the same design won't have the name. In this case the name is saved inside the saved game's persistence file, but not in the editor's craft design file.

4.8.4 Examples of Using the Name

```
// Only if you expected to get
// exactly 1 such part, no more, no less.
SET myPart TO SHIP:PARTSDUBBED("my nametag here")[0].

// OR

// Only if you expected to get
// exactly 1 such part, no more, no less.
SET myPart TO SHIP:PARTSTAGGED("my nametag here")[0].

// Handling the case of more than
// one part with the same nametag,
// or the case of zero parts with
// the name:
SET allSuchParts TO SHIP:PARTSDUBBED("my nametag here").

// OR

SET allSuchParts TO SHIP:PARTSTAGGED("my nametag here").

// Followed by using the list returned:
FOR onePart IN allSuchParts {
  // do something with onePart here.
}
```

For more details on how this system works see the *Parts and PartModules Section*.

4.9 Ship Parts and PartModules

As of v0.15, kOS has introduced the ability to access the functionality of Parts' actions, and right-click context menus. However, to understand how to use them it is necessary to first understand a little bit about the structure of parts in a vessel in Kerbal Space Program.

In Kerbal Space Program, a Vessel is a collection of Parts arranged in a tree Structure. In kOS, you can access the parts one of two ways:

4.9.1 Tutorial

If you prefer the tutorial approach to learning, the following links will walk you through a specific pair of tasks to introduce you to this system. If you prefer a methodical reference approach, skip the tutorials and read on, then come back to the tutorials afterward.

TODO: PUT LINKS POINTING TO A TUTORIAL WALKING THROUGH WHAT THIS IS TEACHING.

THERE NEEDS TO BE TWO TUTORIALS MINIMUM:

1. A Tuturial made on an ONLY STOCK (other than kOS of course) install, using only STOCK parts to do something interesting.
2. A Tutorial more complex showing that this can be used with mods too, probably the Leg Leveller example from the teaser video, but with the code updated to use the newer part query methods.

4.9.2 Parts

Note: The short and quick thing to remember

If you only remember one technique, it should be using the `Part:PARTSDUBBED` method described down below. It's the most useful one that covers all the other cases.

Accessing the parts by various naming systems

Any time you have a vessel variable, you can use these suffixes to get lists of parts on it by their names using several different naming schemes:

Part Tag: A part's *tag* is whatever custom name you have given it using the nametag system described here. This is probably the best naming convention to use because it lets you make up whatever name you like for the part and use it to pick the parts you want to deal with in your script.

Part Title: A part's *title* is the name it has inside the GUI interface on the screen that you see as the user.

Part Name: A part's *name* is the name it is given behind the scenes in KSP. It never appears in the normal GUI for the user to see, but it is used in places like Part.cfg files, the saved game persistence file, the ModuleManager mod, and so on.

Assuming you've done one of these:

```
SET somevessel to SHIP.
//    Or this:
SET somevessel to VESSEL("some vessel's name").
//    Or this:
SET somevessel to TARGET. // assuming TARGET is a vessel and not a body or docking port.
```

Then you can do one of these to query based on any of the above schemes:

```
// ------ :PARTSTAGGED -------
// Finds all parts that have a nametag (Part:Tag suffix) matching the value given:
SET partlist to somevessel:PARTSTAGGED(nametag_of_part)

// ------ :PARTSTITLED -------
// Finds all parts that have a title (Part:Title suffix) matching the value given:
SET partlist to somevessel:PARTSTITLED(title_of_part)

// ------ :PARTSNAMED -------
// Finds all parts that have a name (Part:Name suffix) matching the value given:
```

```
SET partlist to somevessel:PARTSNAMED(name_of_part)

// ------ :PARTSDUBBED -------
// Finds all parts matching the string in any naming scheme, without caring what kind of naming schem
// This is essentially the combination of all the above three searches.
SET partlist to somevessel:PARTSDUBBED(any_of_the_above)
```

In all cases the checks are performed case-insensitively.

These are different styles of naming parts, all slightly different, and you can use any of them you like to get access to the part or parts you're interested in.

They all return a List of Parts rather than just one single part. This is because any name could have more than one hit. If you expect to get just one single hit, you can just look at the zero-th value of the list, like so:

```
SET onePart TO somevessel:PARTSDUBBED("my favorite engine")[0].
```

If the name does not exist, you can tell by seeing if the list returned has a length of zero:

```
IF somevessel:PARTSDUBBED("my favorite engine"):LENGTH = 0 {
  PRINT "There is no part named 'my favorite engine'.".
}.
```

Examples:

```
// Change the altitude at which all the drouge chutes will deploy:
FOR somechute IN somevessel:PARTSNAMED("parachuteDrogue") {
  somechute:GETMODULE("ModuleParachute"):SETFIELD("DEPLOYALTITUDE", 1500).
}.
```

Accessing the parts list as a tree

Starting from the root part, Vessel:ROOTPART (SHIP:ROOTPART, TARGET:ROOTPART, or Vessel("some ship name"):ROOTPART). You can get all its children parts with the Part:CHILDREN suffix. Given any Part, you can access its Parent part with Part:PARENT, and detect if it doesn't have a parent with Part:HASPARENT. By walking this tree you can see how the parts are connected together.

The diagram here shows an example of a small vessel and how it might get represented as a tree of parts in KSP.

Accessing the parts list as a list

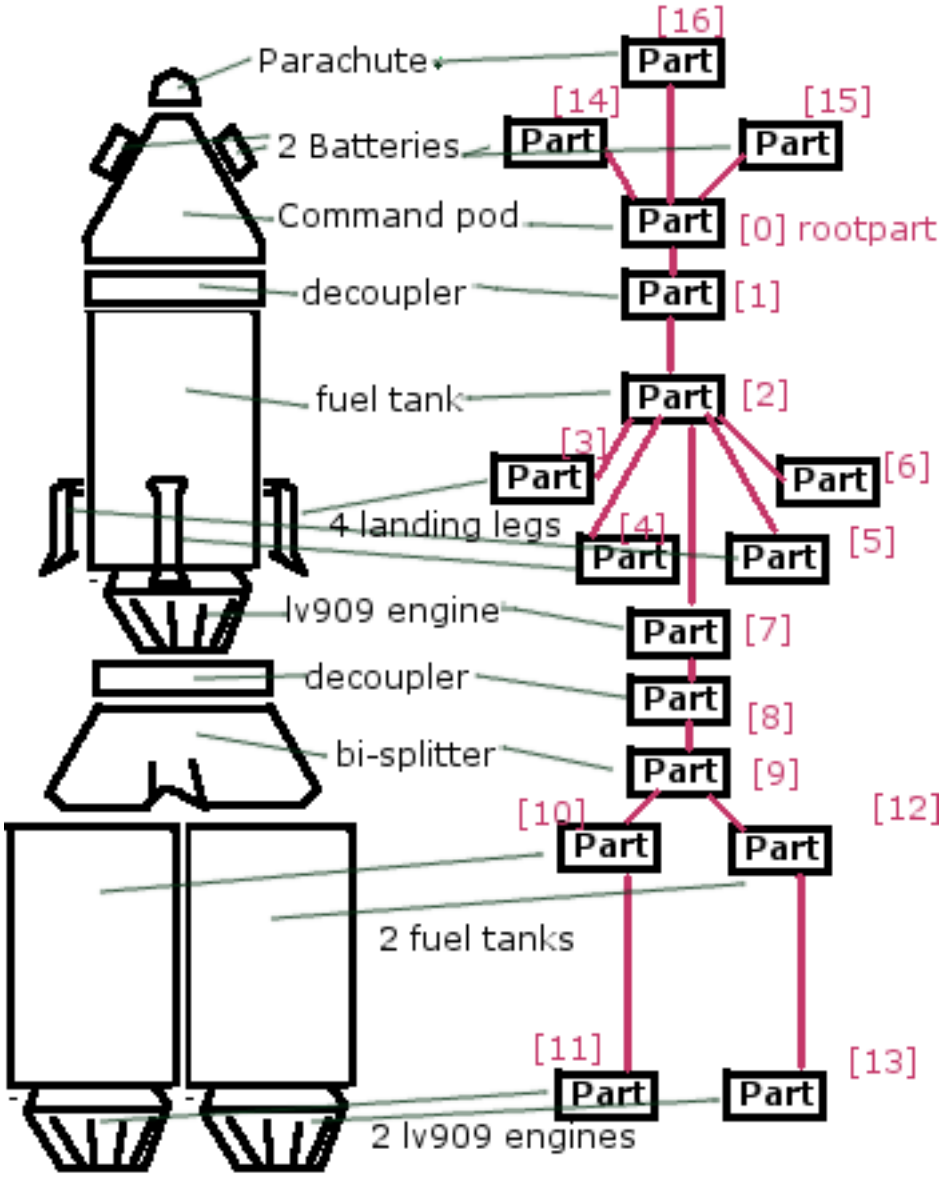
You can get a list of all the parts on a vessel using the suffix :PARTS, or by using the LIST PARTS IN command. When you do this, the resulting list is a "flattening" of the tree of parts, created by use of a depth-first search starting from the root part. In the diagram shown here, the red numbers indicate one possible way the parts might be represented in LIST indeces if you used SHIP:PARTS on such a vessel. Note there is no guarantee it would look exactly like this, as it depends on exactly what order the parts were attached in the VAB.

Shortcuts to smaller lists of parts

If you know some of the properties of the parts you're interested in, you can ask kOS to give you a shorter list of parts that just includes those parts, using the following suffixes:

Return a List of just the parts who's name is "someNameHere":

```
SET ves TO SHIP. // or Target or Vessel("ship name").
SET PLIST TO ves:PARTSNAMED("someNameHere").
```



Demonstrating the tree layout of parts

Return a List of just the parts that have had some sort of activity attached to action group 1:

```
SET ves TO SHIP. // or Target or Vessel("ship name").
SET PLIST TO ves:PARTSINGROUP(AG1).
```

4.9.3 PartModules and the right-click menu:

Each Part, in turn has a list of what are called PartModules on it. A PartModule is a collection of variables and executable program hooks that gives the part some of its behaviors and properties. Without a PartModule, a part is really nothing more than a passive bit of structure that has nothing more than a shape, a look, and a strength to it. Some of the parts in the "structure" tab of the parts bin, like pure I-beams and girders, are like this - they have no PartModules on them. But all of the *interesting* parts you might want to do something with will have a PartModule on them. Through PartModules, **kOS will now allow you to manipulate or query anything that any KSP programmer, stock or mod, has added to the rightclick menu**, or action group actions, for a part.

PartModules, Stock vs Mods:

It should be noted that even if you play an entirely stock installation of KSP (well, stock other than for kOS, obviously, otherwise you wouldn't be reading this), you will still have PartModules on your Parts. Some people have muddied the terminology difference between "Mod" meaning "modification" and "Mod" meaning "module". It should be made absolutely clear that PartModules are a feature of stock KSP, and BOTH stock KSP parts and Modded KSP Parts use them. Even if all you want to do is affect the stock behavior of stock parts in a completely unmodded way, you'll still want to know about PartModules in order to do so.

PartModules and ModuleManager-like behavior:

Some Mods (meaning "modifications" here) operate by adding a new PartModule to every single part in the game. One example of such a mod is the Deadly Reentry mod. In order to track how fragile each part is and how well it withstands re-entry heat, the Deadly Re-entry mod adds a small module to each part in the game, even the stock parts that would normally have no mods at all on them.

Other Mods allow the user to add PartModule's to any part they feel like, through the use of the ModuleManager mod.

Because of these, it's impossible in this explanatory document to make blanket statements about which PartModules will exist on which Parts. Everything that is said here needs to be taken with a grain of salt, as depending on the mods you've installed on your game, you may find PartModules on your parts that are not normally on those parts for most other players.

4.9.4 What a PartModule means to a kOS script

There are 3 ways that a kOS script may interface with a PartModule.

TODO - TAKE SOME SCREENSHOTS TO PUT ALONGSIDE THIS TEXT, SHOWING EXAMPLES OF THESE THINGS IN THE USER INTERFACE. WE NEED A SCREENSHOT THAT SHOWS BOTH A KSPFIELD AND A KSPEVENT IN A PART'S RMB CONTEXT MENU, A SCREENSHOT THAT SHOWS FIELDS COMING FROM MULTIPLE PARTMODULES, AND A SCREENSHOT SHOWING THE KSPACTIONS IN THE VAB ACTION EDITOR.

KSPFields

A KSPField is a single variable that a PartModule attaches to a part. Some of the KSPFields are also displayed in the RMB context menu of a part. It has a current value, and if the field has had a "tweakable" GUI interface attached to it,

then it's also a settable field by the user manipulating the field in the context menu. In kOS, you can only access those KSPFields that are currently visible on the RMB context menu. We, the developers of kOS, instituted this rule out of respect for the developers of other mods and the stock KSP game. If they didn't allow the user to see or manipulate the variable directly in the GUI, then we shouldn't allow it to be manipulated or seen by a kOS script either.

KSPFields are read or manipulated by the following suffixes of PartModule

- :GETFIELD("name of field").
- :SETFIELD("name of field", new_value_for_field).

Note, that these are suffixes of the partmodule and NOT suffixes of the Part itself. This is because two different PartModule's on the same Part might have used the same field name as each other, and it's important to keep them separate.

KSPEvents

A KSPEvent, just like a KSPField, is a thing that a PartModule can put on the RMB context menu for a part. The difference is that a KSPEvent does not actually HAVE a value. It's not a variable. Rather it's just a button with a label next to it. When you press the button, it causes some sort of software inside the PartModule to run. An example of this is the "undock node" button you see on many of the docking ports.

Difference between a KSPEvent and a boolean KSPField: If you see a label next to a button in the RMB context menu, it might be a KSPEvent, OR it might be a boolean KSPField variable which is editable with a tweakable GUI. They look exactly the same in the user interface. To tell the difference, you need to look at what happens when you click the button. If clicking the button causes the button to depress inward and stay pressed in until you click it again, then this is a boolean value KSPField. If clicking the button pops the button in and then it pops out again right away, then this is a KSPEvent instead.

KSPEvents are manipulated by the following suffix of PartModule

- :DOEVENT("name of event").

This causes the event to execute once.

KSPActions:

A KSPAction is a bit different from a KSPField or KSPEvent. A KSPAction is like a KSPEvent in that it causes some software inside the PartModule to be run. But it doesn't work via the RMB context menu for the part. Instead KSPAction's are those things you see being made avaiable to you as options you can assign into an Action Group in the VAB or SPH. When you have the action group editor tab enabled in the VAB or SPH, and then click on a part, that part asks all of its PartModules if they have any KSPActions they'd like to provide access to, and gathers all those answers and lists them in the user interface for you to select from and assign to the action group.

kOS now allows you to access any of those actions without necessarily having had to assign them to any action groups if you didn't want to.

KSPActions are manipulated by the following suffix of PartModule

- :DOACTION("name of action", new_boolan_value).

The name of the action is the name you see in the action group editor interface, and the new boolean value is either True or False. Unlike KSPEvents, a KSPAction has two states, true and false. When you toggle the brakes, for example, they go from on to off, or from off to on. When you call :DOACTION, you are specifying if the KSPAction should behave as if you have just toggled the group on, or just toggled the group off. But instead of actually toggling an action group - you are just telling the single PartModule on a single Part to perform the same behavior it would have performed had that action been assigned to an action group. You don't *actually* have to assign the action to an action group for this to work.

4.9.5 Exploring what's there to find Field/Event/Action Names:

Okay, so you understand all that, but you're still thinking "but how do I KNOW what the names of part modules are, or what the names of the fields on them are? I didn't write all that C# source code for all the modules."

There are some additional suffixes that are designed to help you explore what's available so you can learn the answers to these questions. Also, some of the questions can be answered by other means:

What PartModules are there on a part?

To answer this question you can do one of two things:

A: **Use the part.cfg file** All parts in KSP come with a part.cfg file defining them, both for modded parts and stock parts. If you look at this file, it will contain sections looking something like this:

```
// Example snippet from a Part.cfg file:
MODULE
{
    name = ModuleCommand
```

That would tell you that this part has a PartModule on it called ModuleCommand. there can be multiple such modules per part. But it doesn't let you know about PartModules that get added afterward during runtime, by such things as the ModuleManager mod.

B: **Use the :MODULES suffix of Part:** If you have a handle on any part in kOS, you can print out the value of :MODULES and it will tell you the string names of all the modules on the part. For example:

```
FOR P IN SHIP:PARTS {
  LOG ("MODULES FOR PART NAMED " + P:NAME) TO MODLIST.
  LOG P:MODULES TO MODLIST.
}.
```

Do that, and the file MODLIST should now contain a verbose dump of all the module names of all the parts on your ship. You can get any of the modules now by using Part:GETMODULE("module name").

What are the names of the stuff that a PartModule can do?

These three suffixes tell you everything a part module can do:

```
SET MOD TO P:GETMODULE("some name here").
LOG ("These are all the things that I can currently USE GETFIELD AND SETFIELD ON IN " + MOD:NAME + "
LOG MOD:ALLFIELDS TO NAMELIST.
LOG ("These are all the things that I can currently USE DOEVENT ON IN " +  MOD:NAME + ":") TO NAMELIS
LOG MOD:ALLEVENTS TO NAMELIST.
LOG ("These are all the things that I can currently USE DOACTION ON IN " +  MOD:NAME + ":") TO NAMELI
LOG MOD:ALLACTIONS TO NAMELIST.
```

After that, the file NAMELIST would contain a dump of all the fields on this part module that you can use.

BE WARNED! Names are able to dynamically change!

Some PartModules are written to change the name of a field when something happens in the game. For example, you might find that after you've done this:

```
SomeModule:DOEVENT("Activate").
```

That this doesn't work anymore after that, and the "Activate" event now causes an error.

And the reason is that the PartModule chose to change the label on the event. It changed to the word "Deactivate" now. kOS can no longer trigger an event called "Activate" because that's no longer its name.

Be on the lookout for cases like this. Experiment with how the context menu is being manipulated and keep in mind that the list of strings you got the last time you exectued :ALLFIELDS a few minutes ago might not be the same list you'd get if you ran it now, because the PartModule has changed what is being shown on the menu.

4.10 Career Limits

The theme of KSP 0.90 is that some features of your space program don't work until after you've made some upgrades. kOS now supports enforcing these checks, as described below.

Warning:

4.10.1 The rules being enforced

These are rules inherited from the stock KSP game that kOS is simply adhering to:

- If your tracking center is not upgraded far enough, then you cannot see future orbit patches.
- If your mission control center AND tracking center are not upgraded enough, then you cannot add manuever nodes.

The following are rules invented by kOS that are thematically very similar to stock KSP, intended to be as close to the meaning of the stock game's rules as possible:

- In order to be allowed to execute the `PartModule` :DOACTION method, either your VAB or your SPH must be upgraded to the point where it will allow access to custom action groups. This is because otherwise the :DOACTION method would be a backdoor "cheat" past the restricted access to the action group feaures of various PartModules that the game is meant to have.
- In order to be allowed to add a `nametag` to parts inside the editor so they get saved to the craft file, you must have upgraded your current editor building (VAB or SPH, depending) to the point where it allows at least stock action groups. This is because name tags are a sort of semi-advanced feature.

You can see some of these settings by reading the values of the Career() global object, for example:

:: print Career():PATCHLIMIT. print Career():CANDOACTIONS.

4.10.2 Structure

structure `Career`

Table 4.2: Members

Suffix	Type	Get	Set
CANTRACKOBJECTS	boolean	yes	no
PATCHLIMIT	number	yes	no
CANMAKENODES	boolean	yes	no
CANDOACTIONS	boolean	yes	no

Career:**CANTRACKOBJECTS**

Type boolean

Access Get

If your tracking center allows the tracking of unnamed objects (asteroids, mainly) then this will return true.

Career:**PATCHLIMIT**

Type number

Access Get

If your tracking center allows some patched conics predictions, then this number will be greater than zero. The number represents how many patches beyond the current one that you are allowed to see. It influences attempts to call SHIP:PATCHES and SHIP:OBT:NEXTPATCH.

Career:**CANMAKENODES**

Type boolean

Access Get

If your tracking center and mission control buildings are both upgraded enough, then the game allows you to make manuever nodes (which the game calls "flight planning"). This will return true if you can make maneuver nodes.

Career:**CANDOACTIONS**

Type boolean

Access Get

If your VAB or SPH are upgraded enough to allow custom action groups, then you will also be allowed to execute the :DOACTION method of PartModules. Otherwise you can't. This will return a boolean letting you know if the condition has been met.

CHAPTER

FIVE

THE KERBOSCRIPT LANGUAGE

5.1 General Features of the KerboScript Language

5.1.1 Case Insensitivity

Everything in **KerboScript** is case-insensitive, including your own variable names and filenames. The only exception is when you perform a string comparison, (`"Hello"="HELLO"` will return false.)

Most of the examples here will show the syntax in all-uppercase to help make it stand out from the explanatory text.

5.1.2 Expressions

KerboScript uses an expression evaluation system that allows you to perform math operations on variables. Some variables are defined by you. Others are defined by the system. There are four basic types:

1. Numbers

You can use mathematical operations on numbers, like this:

```
SET X TO 4 + 2.5.
PRINT X.              // Outputs 6.5
```

The system follows the order of operations, but currently the implementation is imperfect. For example, multiplication will always be performed before division, regardless of the order they come in. This will be fixed in a future release.

2. Strings

Strings are pieces of text that are generally meant to be printed to the screen. For example:

```
PRINT "Hello World!".
```

To concatenate strings, you can use the + operator. This works with mixtures of numbers and strings as well:

```
PRINT "4 plus 3 is: " + (4+3).
```

3. Structures

Structures are variables that contain more than one piece of information. For example, a Vector has an X, a Y, and a Z component. Structures can be used with SET.. TO just like any other variable. To access the sub-elements of a structure, you use the colon operator (":"). Here are some examples:

```
PRINT "The Mun's periapsis altitude is: " + MUN:PERIAPSIS.
PRINT "The ship's surface velocity is: " + SHIP:VELOCITY:SURFACE.
```

Many structures also let you set a specific component of them, for example:

```
SET VEC TO V(10,10,10).  // A vector with x,y,z components
                         // all set to 10.
SET VEC:X to VEC:X * 4.  // multiply just the X part of VEC by 4.
PRINT VEC.               // Results in V(40,10,10).
```

4. Structure Methods

Structures also often contain methods. A method is a suffix of a structure that actually performs an activity when you mention it, and can sometimes take parameters. The following are examples of calling methods of a structure:

```
SET PLIST TO SHIP:PARTSDUBBED("my engines"). // calling a suffix
                                             // method with one
                                             // argument that
                                             // returns a list.
PLIST:REMOVE(0). // calling a suffix method with one argument that
                 // doesn't return anything.
PRINT PLIST:SUBLIST(0,4). // calling a suffix method with 2
                          // arguments that returns a list.
```

Note: New in version 0.15: Methods now perform the activity when the interpreter comes up to it. Prior to this version, execution was sometimes delayed until some later time depending on the trigger setup or flow-control.

For more information, see the *Structures Section*. A full list of structure types can be found on the *Structures* page. For a more detailed breakdown of the language, see the *Language Syntax Constructs* page.

5.1.3 Short-curcuiting booleans

Further reading: https://en.wikipedia.org/wiki/Short-circuit_evaluation

When performing any boolean operation involving the use of the AND or the OR operator, kerboscript will short-circuit the boolean check. What this means is that if it gets to a point in the expression where it already knows the result is a forgone conclusion, it doesn't bother calculating the rest of the expression and just quits there.

Example:

```
set x to true.
if x or y+2 > 10 {
    print "yes".
} else {
    print "no".
}.
```

In this case, the fact that x is true means that when evaluating the boolean expression `x or y+2 > 10` it never even bothers trying to add y and 2 to find out if it's greater than 10. It already knew as soon as it got to the `x or whatever` that given that x is true, the *whatever* doesn't matter one bit. Once one side of an OR is true, the other side can either be true or false and it won't change the fact that the whole expression will be true anyway.

A similar short circuiting happens with AND. Once the left side of the AND operator is false, then the entire AND expression is guaranteed to be false regardless of what's on the right side, so kerboscript doesn't bother calculating the righthand side once the lefthand side is false.

Read the link above for implications of why this matters in programming.

5.1.4 Late Typing

Kerboscript is a language in which there is only one type of variable and it just generically holds any sort of object of any kind. If you attempt to assign, for example, a string into a variable that is currently holding an integer, this does not generate an error. It simply causes the variable to change its type and no longer be an integer, becoming a string now.

In other words, the type of a variable changes dynamically at runtime depending on what you assign into it.

5.1.5 Lazy Globals (variable declarations optional)

Kerboscript is a language in which variables need not be declared ahead of time. If you simply set a variable to a value, that just "magically" makes the variable exist if it didn't already. When you do this, the variable will necessarily be *global* in scope. kerboscript refers to these variables created implicitly this way as "lazy globals". It's a system designed to make kerboscript easy to use for people new to programming.

But if you are an experienced programmer you might not like this behavior, and there are good arguments for why you might want to disable it. If you wish to do so, a syntax exists to do so called :ref:`NOLAZYGLOBAL`.

5.1.6 User Functions

Note: New in version 0.17: This feature did not exist in prior versions of kerboscript.

Kerboscript supports user functions which you can write yourself and call from your own scripts. *These are not structure methods (which as of this writing are a feature which only works for the built-in kOS types, and are not yet supported by the kerboscript language for user functions you write yourself).*

Example:

```
DECLARE FUNCTION DEGREES_TO_RADIANS {
  DECLARE PARAMETER DEG.

  RETURN CONSTANT():PI * DEG/180.
}.
```

```
SET ALPHA TO 45.
PRINT ALPHA + " degrees is " + DEGREES_TO_RADIANS(ALPHA) + " radians.".
```

For a more detailed description of how to declare your own user functions, see the *Language Syntax Constructs, User Functions* section.

5.1.7 Structures

Structures, *introduced above*, are variable *types* that contain more than one piece of information. All structures contain sub-values or *methods* that can be accessed with a colon (`:`) operator. Multiple structures can be chained together with more than one colon (`:`) operator:

```
SET myCraft TO SHIP.
SET myMass TO myCraft:MASS.
SET myVel TO myCraft:VELOCITY:ORBIT.
```

These terms are referred to as "suffixes". For example `Velocity` is a suffix of `Vessel`. It is possible to **set** some suffixes as well. The second line in the following example sets the `ETA` of a `NODE` 500 seconds into the future:

```
SET n TO Node( TIME:SECONDS + 60, 0, 10, 10).
SET n:ETA to 500.
```

The full list of available suffixes for each type *can be found here*.

5.2 KerboScript Syntax Specification

This describes what is and is not a syntax error in the **KerboScript** programming language. It does not describe what function calls exist or which commands and built-in variables are present. Those are contained in other documents.

Contents

5.2.1 General Rules

Whitespace consisting of consecutive spaces, tabs, and line breaks are all considered identical to each other. Because of this, indentation is up to you. You may indent however you like.

Note: Statements are ended with a **period** character (".").

The following are **reserved command keywords** and special operator symbols:

Arithmetic Operators:

```
+  -  *  /  ^  e  (  )
```

Logic Operators:

```
not  and  or  true  false  <>  >=  <=  =  >  <
```

Instructions and keywords:

```
set  to  if  else from until step do lock  unlock  print  at  on  toggle
wait  when  then  off  stage  clearscreen  add  remove  log
break  preserve  declare  parameter  switch  copy  rename
volume  file  delete  edit  run  compile  list  reboot  shutdown
for  unset  batch  deploy  in  all
```

Other symbols:

```
{  }  [  ]  ,  :  //
```

Comments consist of everything from a "//" symbol to the end of the line:

```
set x to 1. // this is a comment.
```

Identifiers: Identifiers consist of: a string of (letter, digit, or underscore). The first character must be a letter. The rest may be letters, digits or underscores. **Identifiers are case-insensitive**. The following are identical identifiers:

```
My_Variable  my_varible  MY_VARAIBLE
```

case-insensitivity The same case-insensitivity applies throughout the entire language, with all keywords except when comparing literal strings. The values inside the strings are still case-sensitive, for example, the following will print "unequal":

```
if "hello" = "HELLO" {
    print "equal".
} else {
    print "unequal".
}
```

Suffixes Some variable types are structures that contain sub-portions. The separator between the main variable and the item inside it is a colon character (`:`). When this symbol is used, the part on the right-hand side of the colon is called the "suffix":

```
list parts in mylist.
print mylist:length. // length is a suffix of mylist
```

Suffixes can be chained together, as in this example:

```
print ship:velocity:orbit:x.
```

In the above example you'd say "`velocity` is a suffix of `ship`", and "`orbit` is a suffix of `ship:velocity`", and "`x` is a suffix of `ship:velocity:orbit`".

5.2.2 Braces (statement blocks)

Anywhere you feel like, you may insert braces around a list of statements to get the language to treat them all as a single statement block.

For example: the IF statement expects one statement as its body, like so:

```
if x = 1
  print "it's 1".
```

But you can put multiple statements there as its body by surrounding them with braces, like so:

```
if x = 1 { print "it's 1".  print "yippieee.".  }
```

(Although this is usually preferred to be indented as follows):

```
if x = 1 {
  print "it's 1".
  print "yippieee.".
}
```

or:

```
if x = 1
{
  print "it's 1".
  print "yippieee.".
}
```

Kerboscript does not require proper indentation of the brace sections, but it is a good idea to make things clear.

You are allowed to just insert braces anywhere you feel like even when the language does not require it, as shown below:

```
declare x to 3.
print "x here is " + x.
{
  declare x to 5.
  print "x here is " + x.
  {
    declare x to 7.
    print "x here is " + x.
  }
}
```

The usual reason for doing this is to create a *local scope section* for yourself. In the above example, there are actually 3 *different* variables called 'x' - each with a different scope.

5.2.3 Functions (built-in)

There exist a number of built-in functions you can call using their names. When you do so, you can do it like so:

```
functionName( *arguments with commas between them* ).
```

For example, the `ROUND` function takes 2 arguments:

```
print ROUND(1230.12312, 2).
```

The `SIN` function takes 1 argument:

```
print SIN(45).
```

When a function requires zero arguments, it is legal to call it using the parentheses or not using them. You can pick either way:

```
// These both work:
CLEARSCREEN.
CLEARSCREEN().
```

5.2.4 Suffixes as Functions (Methods)

Some suffixes are actually functions you can call. When that is the case, these suffixes are called "method suffixes". Here are some examples:

```
set x to ship:partsnamed("rtg").
print x:length().
x:remove(0).
x:clear().
```

5.2.5 User Functions

Note: New in version 0.17: This feature did not exist in prior versions of kerboscript.

Help for the new user - What is a Function? In programming terminology, there is a commonly used feature of many programming languages that works as follows:

- 1. Create a chunk of program instructions that you don't intend to execute YET.
- 2. Later, when executing other parts of the program, do the following:
  - 2.A. Remember the current location in the program.
  - 2.B. Jump to the previously created chunk of code from (1) above.
  - 2.C. Run the instructions there.
  - 2.D. Return to where you remembered from (2.A) and continue from there.

This feature goes by many different names, with slightly different precise meanings: *Subroutines*, *Procedures*, *Functions*, etc. For the purposes of kerboscript, we will refer to all uses of this feature with the term *Function*, whether it *technically* fits the mathematical definition of a "function" or not.

In kerboscript, you can make your own user functions using the DECLARE FUNCTION command, which is structured as follows:

`declare function` *identifier* { *statements* } *optional dot (.)*

Functions are a long enough topic as to require a *separate documentation page, here.*

5.2.6 Built-In Special Variable Names

Some variable names have special meaning and will not work as identifiers. Understanding this list is crucial to using kOS effectively, as these special variables are the usual way to query flight state information. *The full list of reserved variable names is on its own page*.

5.2.7 What does not exist (yet?)

Concepts that many other languages have, that are missing from **KerboScript**, are listed below. Many of these are things that could be supported some day, but at the moment with the limited amount of developer time available they haven't become essential enough to spend the time on supporting them.

user-made structures or classes Several of the built-in variables of **kOS** are essentially "classes" with methods and fields, however there's currently no way for user code to create its own classes or structures. Supporting this would open up a *large* can of worms, as it would then make the **kOS** system more complex.

5.3 Flow Control

5.3.1 BREAK

Breaks out of a loop:

```
SET X TO 1.
UNTIL 0 {
    SET X TO X + 1.
    IF X > 10 { BREAK. } // Exits the loop when
                         // X is greater than 10
}
```

5.3.2 IF / ELSE

Checks if the expression supplied returns true. If it does, `IF` executes the following command block. Can also have an optional `ELSE` to execute when the `IF` condition is not true. `ELSE` can have another `IF` after it, to make a chain of `IF`/`ELSE` conditions:

```
SET X TO 1.
IF X = 1 { PRINT "X equals one.". }     // Prints "X equals one."
IF X > 10 { PRINT "X is greater than ten.". }  // Does nothing

// IF-ELSE structure:
IF X > 10 { PRINT "X is large".  } ELSE { PRINT "X is small".  }

// An if-else ladder:
IF X = 0 {
    PRINT "zero".
} ELSE IF X < 0 {
    PRINT "negative".
} ELSE {
```

```
    PRINT "positive".
}
```

Note: The period (`.`) is optional after the end of a set of curly braces like so:

```
// both of these lines are fine
IF TRUE { PRINT "Hello". }
IF TRUE { PRINT "Hello". }.
```

In the case where you are using the `ELSE` keyword, you must *not* end the previous `IF` body with a period, as that terminates the `IF` command and causes the `ELSE` keyword to be without a matching `IF`:

```
// works:
IF X > 10 { PRINT "Large". }  ELSE { PRINT "Small". }.

// syntax error - ELSE without IF.
IF X > 10 { PRINT "Large". }. ELSE { PRINT "Small". }.
```

5.3.3 LOCK

Locks an identifier to an expression. Each time the identifier is used in an expression, its value will be re-calculated on the fly:

```
SET X TO 1.
LOCK Y TO X + 2.
PRINT Y.          // Outputs 3
SET X TO 4.
PRINT Y.          // Outputs 6
```

LOCK follows the same scoping rules as the SET command. If the variable name used already exists in local scope, then the lock command creates a lock function that only lasts as long as the current scope and then becomes unreachable after that. If the variable name used does not exist in local scope, then LOCK will create it as a global variable, unless @NOLAZYGLOBAL is set to off, in which case it will be an error.

Note that a LOCK expression is extremely similar to a user function. Every time you read the value of the "variable", it executes the expression again.

Note: If a `LOCK` expression is used with a flight control such as `THROTTLE` or `STEERING`, then it will get continually evaluated in the background *each physics tick*.

5.3.4 UNLOCK

Releases a lock on a variable. See `LOCK`:

```
UNLOCK X.     // Releases a lock on variable X
UNLOCK ALL.   // Releases ALL locks
```

5.3.5 UNTIL loop

Performs a loop until a certain condition is met:

```
SET X to 1.
UNTIL X > 10 {       // Prints the numbers 1-10
    PRINT X.
    SET X to X + 1.
}
```

Note: If you are writing an UNTIL loop that looks much like the example above, consider the possibility of writing it as a *FROM* loop instead.

Note that if you are creating a loop in which you are watching a physical value that you expect to change each iteration, it's vital that you insert a small WAIT at the bottom of the loop like so:

```
SET PREV_TIME to TIME:SECONDS.
SET PREV_VEL to SHIP:VELOCITY.
SET ACCEL to V(9999,9999,9999).
PRINT "Waiting for accellerations to stop.".
UNTIL ACCEL:MAG < 0.5 {
    SET ACCEL TO (SHIP:VELOCITY - PREV_VEL) / (TIME:SECONDS - PREV_TIME).
    SET PREV_TIME to TIME:SECONDS.
    SET PREV_VEL to SHIP:VELOCITY.

    WAIT 0.001.  // This line is Vitally Important.
}
```

The full explanation why is *in the CPU hardware description page*.

5.3.6 FOR loop

Loops over a list collection, letting you access one element at a time. Syntax:

```
FOR variable1 IN variable2 { use variable1 here. }
```

Where:

- *variable1* is a variable to hold each element one at a time.
- *variable2* is a LIST variable to iterate over.

Example:

```
PRINT "Counting flamed out engines:".
SET numOUT to 0.
LIST ENGINES IN MyList.
FOR eng IN MyList {
    IF ENG:FLAMEOUT {
        set numOUT to numOUT + 1.
    }
}
PRINT "There are " + numOut + "Flamed out engines.".
```

Note: If you are an experienced programmer looking for something more like the for-loop from C, with its 3-part clauses of init, check, and increment in the header, see the *FROM* loop description. The kerboscript 'for' loop is more like a 'foreach' loop from other modern languages like C#.

5.3.7 FROM loop

Identical to the *UNTIL* loop, except that it also contains an explicit initializer and incrementer section in the header.

Syntax:

> FROM { one or more statements } UNTIL Boolean_expression STEP { one or more statements } DO one statement or a block of statements inside braces '{}'

Quick Example:

```
print "Countdown initiated:".
FROM {local x is 10.} UNTIL x = 0 STEP {set x to x-1.} DO {
  print "T -" + x.
}
```

Note: If you are an experienced programmer, you can think of the FROM loop as just being Kerboscript's version of the generic 3-part for-loop `for( int x=10; x > 0; -x ) {...}` that first appeared in C and is now so common to many programming languages, except that its Boolean check uses the reverse of that logic because it's based on UNTIL loops instead of WHILE loops.

What the parts mean

- FROM { one or more statements }
  - Perform these statements at the beginning before starting the first pass through the loop. They may contain local declarations of new variables. If they do, then the variables will be local to the body of the loop and won't be visible outside the loop. In this case the braces { and } are mandatory even when there is only one statement present. To create a a null FROM clause, give it an empty set of braces.
- UNTIL expression
  - Exactly like the *UNTIL* loop. The loop will run this expression at the start of each pass through the loop body, and if it's true, it will abort and stop running the loop. It checks before the initial first pass of the loop as well, so it's possible for the check to prevent the loop body from even executing once. Braces {.."}" are not used here because this is not technically a complete statement. It is just an expression that evaluates to a value.
- STEP { one or more statements }
  - Perform these statements at the bottom of each loop pass. The purpose is typically to increment or decrement the variable you declared in your FROM clause to get it ready for the next loop pass. In this case the braces { and } are mandatory even when there is only one statement present. To create a null FROM clause, give it an empty set of braces.
- DO one statement or a block of statements inside braxes {.."}":
  - This is where the loop body gets put. Much like with the UNTIL and FOR loops, these braces are not mandatory when there is only exactly one statement in the body, but are a very good idea to have anyway.

Why some braces are mandatory

Some braces are mandatory (for the FROM and STEP clauses) even when there is only one statement inside them, because the period that ends a single statement would look like it's terminating the entire FROM loop if it was open and bare. Wrapping it inside braces makes it more visually obvious that it's not the end of the FROM loop.

Why DO is mandatory

Other loop types don't require a keyword to begin the loop body. You can just start in with the opening left-brace {. The reason the additional DO keyword exists in the FROM loop is because otherwise you'd have two back-to-back brace sections (The end of the STEP clause would abut against the start of the loop body) without any punctuation between them, and that would look too much like it was starting a brand new thing from scratch.

Other formatting examples

```
// prints a count from 1 to 10:
FROM {local x is 1.} UNTIL x > 10 STEP {set x to x+1.} DO { print x.}

// Entire header in one line, body indented:
// ----------------------------
FROM {local x is 1.} UNTIL x > 10 STEP {set x to x+1.} DO {
  print x.
}

// Each header part on its own line, body indented:
// ----------------------------
FROM {local x is 1.}
UNTIL x > 10
STEP {set x to x+1.}
DO {
  print x.
}

// Fully exploded out: Each header part on its own line,
//  each clause indented separately:
// ----------------------------
FROM
{
  local x is 1.  // x will count upward from 1.
  local y is 10. // while y is counting downward from 10.
}
UNTIL
  x > 10 or y = 0
STEP
{
  set x to x+1.
  set y to y-1.
}
DO
{
  print "x is " + x + ", y is " + y.
}

// ETC.
```

Any such combination of indenting styles, or mix and match of them, is understood by the compiler. The compiler ignores the spacing and indenting. It is recommended that you pick just two of them and stick with them - one compact one to use for short headers, and one longer exploded one to use for more wordy headers when you have to split it up across lines.

The literal meaning of FROM

If you have a FROM loop, it ends up being exactly identical to an *UNTIL* loop written as follows:

If we assume that AAAA, BBBB, CCCC, and DDDD are placeholders referring to the actual script syntax, then in the generic case, the following is how all FROM loops work:

FROM LOOP:

```
FROM { AAAA } UNTIL BBBB STEP { CCCC } DO { DDDD }
```

Is exactly the same as doing this:

```
{ // start a brace to keep the scope of AAAA local to the loop.
    AAAA
    UNTIL BBBB {
        DDDD

        CCCC
    }
} // end a brace to throw away the local scope of AAAA
```

An example of why the FROM loop is useful

Given that the FROM loop is really just an alternate way to write a certain format of UNTIL loop, you might ask why bother having it. The reason is that in the long run it makes your script easier to edit and maintain. It makes things more self-contained and cut-and-pasteable:

Above, in the documentation for *UNTIL* loops, this example was given:

```
SET X to 1.
UNTIL X > 10 {       // Prints the numbers 1-10
    PRINT X.
    SET X to X + 1.
}
```

The same example, expressed as a FROM loop is this:

```
FROM {SET X to 1.} UNTIL X > 10 {SET X to X + 1.} DO {
    PRINT X.
}
```

Kerboscript FROM loop provides a way to place those sections in the loop header so they are declared up front and let people see the layout of how the loop iterates, leaving the body to just contain the statements to be done for that iteration.

If you are editing your script and need to cut a loop section and move it elsewhere, the FROM loop makes it more visually obvious how to cut that loop and move it. It makes the important parts of the loop be self contained in the header, so you don't leave the initializer behind when moving the loop.

5.3.8 WAIT

Halts execution for a specified amount of time, or until a specific set of criteria are met. Note that running a WAIT UNTIL statement can hang the machine forever if the criteria are never met. Examples:

```
WAIT 6.2.                    // Wait 6.2 seconds
WAIT UNTIL X > 40.           // Wait until X is greater than 40
WAIT UNTIL APOAPSIS > 150000. // You can see where this is going
```

Note that any WAIT statement, no matter what the actual expression is, will always result in a wait time that lasts at least *one physics tick*.

Note: The *WAIT* command only causes mainline code to be suspended. Trigger code such as WHEN, ON, LOCK STEERING, and LOCK THROTTLE, will continue executing while your program is sitting still on the WAIT command.

5.3.9 WHEN / THEN

Executes a command when a certain criteria are met. Unlike WAIT, WHEN does not halt execution. It starts a check in the background that will keep actively looking for the trigger condition while the rest of the code continues. When it triggers, the body after the THEN will execute exactly once, after which the trigger is removed unless the PRESERVE is used, in which case the trigger is not removed.

The body of a THEN or an ON statement interrupts the normal flow of a **kOS** program. When the event that triggers the body happens, the main **kOS** program is paused until the body of the THEN completes.

Warning: With the advent of *local variable scoping* in kOS version 0.17 and above, it's important to note that the variables used within the expression of a WHEN or an ON statement should be GLOBAL variables or the results are unpredictable. If local variables were used, the results could change depending on where you are within the execution at the time.

Warning: Do not make the body of a WHEN/THEN take a long time to execute. If you attempt to run code that lasts too long in the body of your WHEN/THEN statement, *it will cause an error*. Avoid looping during WHEN/THEN if you can. For details on how to deal with this, see the *tutorial on design patterns*.

Note: Changed in version 0.12: **IMPORTANT BREAKING CHANGE:** In previous versions of **kOS**, the body of a WHEN/THEN would execute simultaneously in the background with the rest of the main program. This behavior has changed as of version *0.12* of **kOS**, as described above, and scripts that used to rely on this behavior will not work with version *0.12* of **kOS**

Example:

```
WHEN BCount < 99 THEN PRINT BCount + " bottles of beer on the wall.

// Watch in the background for when the altitude is high enough.
// Once it is, then turn on the solar panels and action group 1
WHEN altitude > 70000 THEN {
    PRINT "ACTIVATING PANELS AND AG 1.".
    PANELS ON.
    AG1 ON.
}
```

A WHEN/THEN trigger is removed when the program that created it exits, even if it has not occurred yet. The PRESERVE can be used inside the THEN clause of a WHEN statement. If you are going to make extensive use of WHEN/THEN triggers, it's important to understand more details of how they *work in the kOS CPU*.

5.3.10 ON

The ON command is almost identical to the WHEN/THEN command. ON sets up a trigger in the background that will run the selected command exactly once when the boolean variable changes state from true to false or from false to true. This command is best used to listen for action group activations.

Just like with the WHEN/THEN command, the PRESERVE command can be used inside the code block to cause the trigger to remain active and not go away.

Warning: With the advent of *local variable scoping* in kOS version 0.17 and above, it's important to note that the variables used within the expression of a WHEN or an ON statement should be GLOBAL variables or the results are unpredictable. If local variables were used, the results could change depending on where you are within the execution at the time.

How does it differ from WHEN/THEN? The WHEN/THEN triggers are executed whenever the conditional expression *becomes true*. ON triggers are executed whenever the boolean variable *changes state* either from false to true or from true to false.

The body of an ON statement can be a list of commands inside curly braces, just like for WHEN/THEN. Also just like with WHEN/THEN, the body of the ON interrupts all of **KSP** while it runs, so it should be designed to be a short and finish quickly without getting stuck in a long loop:

```
ON AG3 {
   PRINT "Action Group 3 Activated!.
}
ON SAS PRINT "SAS system has been toggled.
ON AG1 {
    PRINT "Action Group 1 activated.".
    PRESERVE.
}
```

Warning: DO NOT make the body of an ON statement take a long time to execute. If you attempt to run code that lasts too long in the body of your ON statement, *it will cause an error*. For general help on how to deal with this, see the *tutorial on design patterns*.

Avoid looping during ON code blocks if you can. If you are going to make extensive use of ON triggers, it's important to understand more details of how they *work in the kOS CPU*.

5.3.11 PRESERVE

PRESERVE is a command keyword that is only valid inside of WHEN/THEN and ON code blocks.

When a WHEN/THEN or ON condition is triggered, the default behavior is to execute the code block body exactly once and only once, and then the trigger condition is removed and the trigger will never occur again.

To alter this, execute the PRESERVE command anywhere within the body of the code being executed and it tells the **kOS** computer to keep the trigger condition active. When it finishes executing the code block of the trigger, if PRESERVE has happened anywhere within that run of the block of code, it will not remove the trigger. Instead it will allow it to re-trigger, possibly as soon as the very next tick. If the PRESERVE keyword is executed again and again each time the trigger occurs, the trigger could remain active indefinitely.

The following example sets up a continuous background check to keep looking for if there's no fuel in the current stage, and if there is, then it activates the next stage, but no more often than once every half second. Once more than NUMSTAGES have happened, it allows the check to stop executing but it keeps the check alive until that happens:

```
SET NUMSTAGES TO 5.
SET COOLDOWN_START TO 0.

WHEN (TIME:SECONDS > COOLDOWN_START + 0.5) AND STAGE:LIQUIDFUEL = 0 {
    SET COOLDOWN_START TO TIME:SECONDS.
    STAGE.
    SET NUMSTAGES TO NUMSTAGES - 1.
```

```
    IF NUMSTAGES > 0 {
        PRESERVE.
    }
}

// Continue to the rest of the code
```

5.3.12 DECLARE FUNCTION

Covered in more depth *elsewhere in the documentation*, the DECLARE FUNCTION statement creates a user-defined function that you can then call elsewhere in the code.

5.3.13 Boolean Operators

All conditional statements, like IF, can make use of boolean operators. The order of operations is as follows:

- = < > <= >= <>
- AND
- OR
- NOT

Boolean is a type that can be stored in a variable and used that way as well. The constants True and False (case insensitive) may be used as values for boolean variables. If a number is used as if it was a Boolean variable, it will be interpreted in the standard way (zero means false, anything else means true):

```
IF X = 1 AND Y > 4 { PRINT "Both conditions are true". }
IF X = 1 OR Y > 4 { PRINT "At least one condition is true". }
IF NOT (X = 1 or Y > 4) { PRINT "Neither condition is true". }
IF X <> 1 { PRINT "X is not 1". }
SET MYCHECK TO NOT (X = 1 or Y > 4).
IF MYCHECK { PRINT "mycheck is true." }
LOCK CONTINUOUSCHECK TO X < 0.
WHEN CONTINUOUSCHECK THEN { PRINT "X has just become negative.". }
IF True { PRINT "This statement happens unconditionally." }
IF False { PRINT "This statement never happens." }
IF 1 { PRINT "This statement happens unconditionally." }
IF 0 { PRINT "This statement never happens." }
IF count { PRINT "count isn't zero.". }
```

5.4 Variables & Statements

5.4.1 DECLARE .. TO/IS

What it does:

Declares a variable, explicitly or implicitly defining what scope it has, and gives it an initial value.

Allowed Syntax:

All the following are legal "declare" statements:

The following alternate versions have identical meaning to each other:

- DECLARE *identifier* TO *expression dot*
- DECLARE *identifier* IS *expression dot*
- DECLARE LOCAL *identifier* TO *expression dot*
- DECLARE LOCAL *identifier* IS *expression dot*
- LOCAL *identifier* TO *expression dot*
- LOCAL *identifier* IS *expression dot*

The following alternate versions have identical meaning to each other:

- DECLARE GLOBAL *identifier* TO *expression dot*
- DECLARE GLOBAL *identifier* IS *expression dot*
- GLOBAL *identifier* TO *expression dot*

- GLOBAL *identifier* IS *expression dot*

Warning: New in version 0.17: ***BREAKING CHANGE** The meaning, and syntax, of this statement changed considerably in this update. Prior to this version, DECLARE always created global variables no matter where it appeared in the script. See 'initializer required' below.

Detailed Description of the syntax:

- The statement must begin with either the word DECLARE, LOCAL, or GLOBAL. If it begins with the word DECLARE it may optionally also contain the word LOCAL or GLOBAL afterward. *Note that if neither* GLOBAL *nor* LOCAL *is used, the behavior of* LOCAL *will be assumed implicitly. Therefore* DECLARE LOCAL *and just* DECLARE *and just* LOCAL *mean the same thing.*
- After that it must contain an identifier.
- After that it must contain either the word TO or the word IS, which mean the same thing here.
- After that it must contain some expression for the initial starting value of the variable.
- After that it must contain a dot ("period"), like all commands in kerboscript.

```
// These all do the exact same thing - make a local variable:
DECLARE X TO 1. // assumes local when unspecified.
LOCAL X IS 1.
DECLARE LOCAL X IS 1.

// These do the exact same thing - make a global variable:
GLOBAL X IS 1.
DECLARE GLOBAL X IS 1.
```

If neither the scope word GLOBAL nor the scope word LOCAL appear, a declare statement assumes LOCAL by default.

Any variable declared with DECLARE, DECLARE LOCAL, or LOCAL will only exist inside the code block section it was created in. After that code block is finished, the variable will no longer exist.

See Scoping:

If you don't know what the terms "global" or "local" mean, it's important to read the *section below about scoping.*

Note: It is implied that the outermost scope of a program file is the global scope. Therefore if you make a LOCAL variable at the outermost nesting level of your program it really ends up being GLOBAL. Note that GLOBAL variables are not only shared between functions of your script, but also can be seen by other programs you run from the current program, and visa versa.

Alternatively, a variable can be implicitly declared by any SET or LOCK statement, however doing so causes the variable to always have global scope. **The only way to make a variable be local instead of global is to declare it explicitly with one of these DECLARE statements**.

Note: Terminology: "declare statement": Note that the documentation will often refer to the phrase "declare statement" even when referring to a statement in which the optional keyword "declare" was left off. A statement such as LOCAL X IS 1. Will still be referred to as a "declare statement", even though the word "declare" never explicitly appeared in it.

Initializer required in DECLARE

New in version 0.17.

The syntax without the initializer, looking like so:

```
DECLARE x. // no initializer like "TO 1."
```

is **no longer legal syntax**.

Kerboscript now requires the use of the initializer clause (the "TO" keyword) after the identifier name so as to make it impossible for there to exist any uninitialized variables in a script.

5.4.2 DECLARE PARAMETER

If you put this statement in the main part of your script, it declares variables to be used as a parameter that can be passed in using the `RUN` command.

If you put this statement inside of a *Function body*, then it declares variables to be used as a parameter that can be passed in to that function when calling the function.

Just as with a *declare identifier statement*, in a `declare parameter` statement, the actual keyword `declare` need not be used. The word `parameter` may be used alone and that is legal syntax.

Program 1:

```
// This is the contents of program1:
DECLARE PARAMETER X.
PARAMETER Y. // omitting the word "DECLARE" - it still means the same thing.
PRINT "X times Y is " + X*Y.
```

Program 2:

```
// This is the contents of program2, which calls program1:
SET A TO 7.
RUN PROGRAM1( A, A+1 ).
```

The above example would give the output:

```
X times Y is 56.
```

It is also possible to put more than one parameter into a single `DECLARE PARAMETER` statement, separated by commas, as shown below:

```
DECLARE PARAMETER X, Y, CheckFlag.

// Or you could leave "DECLARE" off like so:
PARAMETER X, Y, CheckFlag.
```

Either of the above is exactly equivalent to:

```
PARAMETER X.
PARAMETER Y.
PARAMETER CheckFlag.
```

Note: Unlike normal variables, Parameter variables are always local to the program. When program A calls program B and passes parameters to it, program B can alter their values without affecting the values of the variables in program A.

Caveat This is only true if the values are primitive singleton values like numbers or booleans. If the values are Structures like Vectors or Lists, then they do end up behaving as if they were passed by reference, in the usual way that should be familiar to people who have used languages like Java or C# before.

Illegal to say `DECLARE GLOBAL PARAMETER` : Because parameters are always local to the location they were declared at, the keyword `GLOBAL` is illegal to use in a `DECLARE PARAMETER` statement.

The `DECLARE PARAMETER` statements can appear anywhere in a program as long as they are in the file at a point earlier than the point at which the parameter is being used. The order the arguments need to be passed in by the caller is the order the `DECLARE PARAMETER` statements appear in the program being called.

Note: Pass By Value

The following paragraph is important for people familiar with other programming languages. If you are new to programming and don't understand what it is saying, that's okay you can ignore it.

At the moment the only kind of parameter supported is a pass-by-value parameter, and pass-by reference parameters don't exist. Be aware, however, that due to the way kOS is implemented on top of a reference-using object-oriented language (CSharp), if you pass an argument which is a complex aggregate structure (i.e. a Vector, or a List - anything that kOS lets you use a colon suffix with), then the parameters will behave exactly like being passed by reference because all you're passing is the handle to the object rather than the object itself. This should be familiar behavior to anyone who has written software in Java or C# before.

5.4.3 SET

Sets the value of a variable. Implicitly creates a global variable if it doesnt already exist:

```
SET X TO 1.
SET X TO y*2 - 1.
```

This follows the *scoping rules explained below*. If the variable can be found in the current local scope, or any scope higher up, then it won't be created and instead the existing one will be used.

5.4.4 DEFINED

```
DEFINED identifier
```

Returns a boolean true or false according to whether or not an identifer is defined in such a way that you can use it from this part of the program. (i.e. is it declared and is it in scope and visible right now):

```
// This part prints 'doesn't exist":
if defined var1 {
  print "var1 exists".
} else {
  print "var1 doesn't exist."
}

local var1 is 0.

// But now it prints that it does exist:
if defined var1 {
  print "var1 exists".
} else {
  print "var1 doesn't exist."
}
```

The DEFINED operator pays attention to all the normal scoping rules described in the *scoping section below*. If an identifier does exist but is not usable from the current scope, it will return false.

Note that DEFINED does not work well on things that are not pure identifiers. for example:

```
print defined var1:suffix1.
```

is going to end up printing "False" because it's looking for pure identifiers, not complex suffix chains, and there's no identifier called "var1:suffix1".

Difference between SET and DECLARE LOCAL and DECLARE GLOBAL

The following three examples look very similar and you might ask why you'd pick one instead of the other:

```
SET X TO 1.
DECLARE LOCAL X TO 1.
DECLARE GLOBAL X TO 1.
```

They are slightly different, as follows:

`SET X TO 1.` Performs the following activity:

1. Attempt to find an already existing local X. If found, set it to 1.
2. Try again for each scoping level outside the current one.
3. If and only if it gets all the way out to global scope and it still hasn't found an X, then create a new X with value 1, and do so at global scope. This behavior is called making a "lazy global".

`DECLARE LOCAL X TO 1.` Performs the following activity:

1. Immediately make a new X right here at the local-most scope. Set it to 1.

`DECLARE GLOBAL X TO 1.` Performs the following activity:

1. Ignore whether or not there are any existing X's in a local scope.
2. Immediately go all the way to global scope and make a new X there. Set it to 1.

When to use GLOBAL

You should use a `DECLARE GLOBAL` statement only sparingly. It mostly exists so that a function can store values "in the caller" for the caller to get its hands on. It's generally a "sloppy" design pattern to use, and it's much better to keep everything local and only pass back things to the caller as return values.

5.4.5 LOCK

Declares that the identifier will refer to an expression that is always re-evaluated on the fly every time it is used (See also *Flow Control documentation*):

```
SET Y TO 1.
LOCK X TO Y + 1.
PRINT X.    // prints "2"
SET Y TO 2.
PRINT X.    // prints "3"
```

Note that because of how LOCK expressions are in fact implemented as mini functions, they cannot have local scope. A LOCK *always* has global scope.

By default a LOCK expression is GLOBAL when made. This is necessary for backward compatibility with older scripts that use LOCK STEERING from inside triggers, loops, etc, and expect it to affect the global steering value.

Calling a LOCK that was created in another file

If you try to call a lock that is declared in another program file you run, it does not work, and has never worked prior to kOS 0.17.0:

File1.ks:

```
run File2.
print "x's locked value is " + x.
```

File2.ks:

```
lock x to "this is x".
```

But now with the kerboscript of kOS 0.17.0, you can make it work by inserting empty parentheses after the lock name to help give the compiler the hint that you expected x to be a function call (which is what a lock really is):

Change this line:

```
print "x's locked value is " + x.
```

To this instead:

```
print "x's locked value is " + x().
```

and it should work.

Local lock

You can explicitly make a LOCK statement be LOCAL with the LOCAL keyword, like so:

LOCAL LOCK identifier TO expression.

But be aware that doing so with a cooked steering control such as THROTTLE or STEERING will not actually affect your ship. The automated cooked steering control is only reading the GLOBAL locks for these settings.

The purpose of making a LOCAL lock is if you only need to use the value temporarily for the duration of a function call, loop, or if-statement body, and then you don't care about it anymore after that.

Why do I care about a local lock?

You care because in order to make a LOCK work even after the variables it's using in its expression go out of scope (which is necessary for LOCK STEERING or LOCK THROTTLE to work if done from inside a user function call or trigger body), locks need to preserve a thing called a "closure". (http://en.wikipedia.org/wiki/Closure_(computer_programming)

When they do this, it means none of the local variables used in the function body they were declared in truly "go away" from memory. They live on, taking up space until the lock disappears. Making the lock be local tells the computer that it can make the lock disappear when it goes out of scope, and thus it doesn't need to hold that "closure" around forever.

The tl;dr version: It's more efficient for memory. If you know for sure that your lock isn't getting used after your current section of code is over, make it a local lock.

5.4.6 TOGGLE

Toggles a variable between TRUE or FALSE. If the variable in question starts out as a number, it will be converted to a boolean and then toggled. This is useful for setting action groups, which are activated whenever their values are inverted:

```
TOGGLE AG1. // Fires action group 1.
TOGGLE SAS. // Toggles SAS on or off.
```

This follows the same rules as *SET*, in that if the variable in question doesn't already exist, it will end up creating it as a global variable.

5.4.7 ON

Sets a variable to TRUE. This is useful for the RCS and SAS bindings:

```
RCS ON.  // Turns on the RCS
```

This follows the same rules as *SET*, in that if the variable in question doesn't already exist, it will end up creating it as a global variable.

5.4.8 OFF

Sets a variable to FALSE. This is useful for the RCS and SAS bindings:

```
RCS OFF.  // Turns off the RCS
```

This follows the same rules as *SET*, in that if the variable in question doesn't already exist, it will end up creating it as a global variable.

5.4.9 Scoping terms

Note: New in version 0.17: In prior versions of kerboscript, all identifiers other than DECLARE PARAMETER identifiers were always global variables no matter what, even if you used the DECLARE statement to make them.

What is Scope? The term *Scope* simply refers to asking the question "where in the code can this variable be used, and how long does it last before it goes away?" The *scope* of a variable is the section of the program's code that it "works" within. Any section of the program's code from which the variable cannot be seen is said to be "out of that variable's scope".

Global scope The simplest scope is called "global". Global scope simply means "this variable can be used from anywhere in the program". If you never use the DECLARE statement, then your variables in kerboscript will all be in *global scope*. For simple easy scripts used by beginners, this is often enough and you don't have to read the rest of this topic until you start advancing to more intermediate scripts.

Local Scope Kerboscript uses block scoping to keep track of local variable scope. This means you can have variables that are not only local to a function, but are in fact actually local to JUST the current curly-brace block of statements, even if that block of statements is, say, the body of an IF check, or the body of an UNTIL loop.

Why limit scope? You might be wondering why it's useful to limit the scope of a variable. Wouldn't it be easier just to make all variables global? The answer is twofold: (1) Once a program becomes large enough, trying to remember the name of every variable in the program, and having to keep coming up with new names for new variables, can be a large unmanageable chore, especially with programs written by more than one person collaborating together. (2) Even if you can keep track of all that in your head, there's a certain programming

technique known as recursion (http://en.wikipedia.org/wiki/Recursion#In_computer_science) in which you actually NEED to have local variable scope for the technique to even work at all.

If you need to have variables that only have local scope, either just to keep your code more manageable, or because you literally need local scope to allow for recursive function calls, then you use the `DECLARE LOCAL` statement (or just `LOCAL` for short) to create the variables.

5.4.10 Scoping syntax

Presumed defaults

The DECLARE keyword and the LOCK keyword have some default presumed scoping behaviors:

`DECLARE` Is assumed to always be LOCAL when not otherwise specified.

`FUNCTION` Is assumed to always be LOCAL when not otherwise specified.

`PARAMETER` Cannot be anything but LOCAL to the location it's mentioned. It is an error to attempt to declare a parameter with the GLOBAL keyword.

`LOCK` Is assumed to always be GLOBAL when not otherwise specified. this is necessary to preserve backward compatibility with how cooked controls such as LOCK STEERING and LOCK THROTTLE work.

Explicit scoping keywords

The `DECLARE`, `FUNCTION`, and `LOCK` commands can be given explicit `GLOBAL` or `LOCAL` keywords to define their intended scoping level:

```
//
// These are all synonymous with each other:
//
DECLARE X TO 1.
DECLARE LOCAL X TO 1.
LOCAL X TO 1. // 'declare' is implied and optional when scoping words are used
LOCAL X IS 1. // 'declare' is implied and optional when scoping words are used
//
// These are all synonymous with each other:
//
DECLARE GLOBAL X TO 1.
GLOBAL X TO 1. // 'declare' is implied and optional when scoping words are used
GLOBAL X IS 1. // 'declare' is implied and optional when scoping words are used
```

Even when the word 'DECLARE' is left off, the statement can still be referred to as a "declare statement". The word "declare" is implied by the use of LOCAL or GLOBAL and you are allowed to leave it off merely to reduce verbosity.

Explicit Scoping required for @lazyglobal off

Note that when operating under the *@LAZYGLOBAL OFF* directive the keywords LOCAL and GLOBAL are no longer optional for **declare identifier** statements, and are in fact required. You are not allowed to rely on these presumed defaults when you've turned off LAZYGLOBAL. (This only applies to trying to make a variable with **declare identifier to value**, and not to `declare parameter` or `declare function`.)

Locals stated at the global level are global

Note that if you put a statement at the outermost scope of the program, then there is effectively no difference between a DECLARE LOCAL (or just LOCAL for short) and a DECLARE GLOBAL (or just GLOBAL for short) statement. They are both going to make a variable at global scope because that's the scope the program was in when the statement was encountered.

Examples:

```
GLOBAL x IS 10. // X is now a global variable with value 10,
SET y TO 20. // Y is now a global variable (implicitly) with value 20.
LOCAL z IS 0.  // Z is now a global variable
               // because even though this says LOCAL, it was
               // stated at the outermost, global scope.

SET sum to -1. // sum is now an implicitly made global variable, containing -1.

// A function to return the mean average of all the items in the list
// passed into it, under the assumption all the items in the list are
// numbers of some sort:
FUNCTION calcAverage {
  PARAMETER inputList.

  LOCAL sum IS 0. // sum is now local to this function's body.
  FOR val IN inputList {
    SET sum TO sum + val.
  }.
  print "Inside calcAverage, sum is " + sum.
  RETURN sum / inputList:LENGTH.
}.

SET testList TO LIST(5,10,15);
print "average is " + calcAverage(testList).
print "but out here where it's global, sum is still " + sum.
```

This example will print:

```
Inside calcAverage, sum is 30
average is 10
but out here where it's global, sum is still -1
```

Thus proving that the variable called SUM inside the function is NOT the same variable as the one called SUM out in the global main code.

Nesting

The scoping rules are nested as well. If you attempt to use a variable that doesn't exist in the local scope, the next scope "outside" it will be used, and if it doesn't exist there, the next scope "outside" that will be used and so on, all the way up to the global scope. Only if the variable isn't found at the global scope either will it be implicitly created.

Scoping and Triggers:

Triggers such as:

- WHEN <expression> { <statements> }.

and

- ON <boolean variable> { <statements> }.

Do not work predictably when you use local variables in the <expression> part of them. They need to be designed to use global variables only, because they outlive the duration of any particular scoping braces. You can declare local variables within their <statements> in their bodies, just don't use local variables in the trigger conditions.

@LAZYGLOBAL directive

Often the fact that you can get an implicit global variable declared without intending to can lead to a lot of code maintenance headaches down the road. If you make a typo in a variable name, you end up creating a new variable instead of generating an error. Or you may just forget to mark the variable as local when you intended to.

If you wish to instruct kerboscript to alter its behavior and disable its normal implicit globals, and instead demand that all variables MUST be explicitly declared and may not use implied lazy scoping, the `@LAZYGLOBAL` compiler directive allows you to do that.

If you place the words:

```
@LAZYGLOBAL OFF.
```

At the start of your program, you will turn off the compiler's lazy global feature and it will require you to explicitly mention all variables you use in a declaration somewhere (with the exception of the built-in variables such as THROTTLE, STEERING, SHIP, and so on.)

@LAZYGLOBAL Can only exist at the top of your code.

The @LAZYGLOBAL compile directive is only allowed as the first non-comment thing in the program file. This is because it instructs the compiler to change its default behavior for the duration of the entire file's compile.

@LAZYGLOBAL Makes LOCAL and GLOBAL mandatory

Normally the keywords `local` and `global` can be left off as optional in declare **identifier** statements. But when you turn LAZYGLOBAL off, the compiler starts requiring them to be explicitly stated for **declare identifier** statements, to force yourself to be clear and explicit about the difference.

For example, this program, which is valid:

```
function foo {print "foo ". }
declare x is 1.

print foo() + x.
```

Starts giving errors when you add @LAZYGLOBAL OFF to the top:

```
@LAZYGLOBAL OFF.
function foo {print "foo ". }
declare x is 1.

print foo() + x.
```

Which you fix by explicitly stating the local keyword, as follows:

```
@LAZYGLOBAL OFF.
function foo {print "foo ". }  // This does not need the 'local' keyword added
declare local x is 1.          // But this does because it is a declare *identifier* statement.
                               // you could have also just said:
```

```
                                //     local x is 1.
                                // without the 'declare' keyword.

print foo() + x.
```

If you get in the habit of just writing your **declare identifier** statements like `local x is 1.` or `global x is 1.`, which is probably nicer to read anyway, the issue won't come up.

Longer Example of use

Example:

```
@LAZYGLOBAL off.
global num TO 1.
IF TRUE {
  LOCAL Y IS 2.
  SET num TO num + Y. // This is fine.  num exists already as a global and
                      // you're adding the local Y to it.
  SET nim TO 20. // This typo generates an error.  There is
                 // no such variable "nim" and @LAZYGLOBAL OFF
                 // says not to implicitly make it.
}.
```

Why `LAZYGLOBAL OFF`? The rationale behind `LAZYGLOBAL OFF.` is to primarily be used in cases where you're writing a library of function calls you intend to use elsewhere, and want to be careful not to accidentally make them dependent on globals outside the function itself.

The `@LAZYGLOBAL OFF.` directive is meant to mimic Perl's `use strict;` directive.

History: Kerboscript began its life as a language in which you never have to declare a variable if you don't want to. You can just create any variable implicitly by just using it in a SET statement.

There are a variety of programming languages that work like this, such as Perl, Javascript, and Lua. However, they all share one thing in common - once you want to allow the possibility of having local variables, you have to figure out how this should work with the implicit variable declaration feature.

And all those languages went with the same solution, which kerboscript now follows as well. Because implicit undeclared variables are intended to be a nice easy way for new users to ease into programming, they should always default to being global so that people who wish to keep programming that way don't need to understand or deal with scope.

5.5 KerboScript User Functions

This page covers functions created by you, the user of kerboscript, rather than the built-in functions provided by kOS.

5.5.1 Help for the new user - What is a Function?

In programming terminology, there is a commonly used feature of many programming languages that works as follows:

- 1. Create a chunk of program instructions that you don't intend to execute YET.
- 2. Later, when executing other parts of the program, do the following:
  - 1. Remember the current location in the program.
  - 2. Jump to the previously created chunk of code from (1) above.
  - 3. Run the instructions there.
  - 4. Return to where you remembered from (A) and continue from there.

This feature goes by many different names, with slightly different precise meanings: *Subroutines*, *Procedures*, *Functions*, etc. For the purposes of kerboscript, we will refer to all uses of this feature with the term *Function*, whether it *technically* fits the mathematical definition of a "function" or not.

Warning: Functions created in programs can only be called from programs, not from the interpreter terminal prompt.
If you attempt to create a function in a program, and then call it from the interactive prompt in the interpreter instead of calling it from inside a program, it will definitely not work properly, but the exact error you get will depend on several "random" factors. This may be fixed later by a later release, but for now, don't do it. For further explanation, see the section entitled *Functions and the interpreter terminal*

5.5.2 `DECLARE FUNCTION`

In kerboscript, you can make your own user functions using the DECLARE FUNCTION command, which has syntax as follows:

[`declare`] [`local`] `function` *identifier* { *statements* } *optional dot (.)*

The statement is called a "declare function" statement even when the optional word "declare" was left off.

The following are all identical in meaning:

```
declare function hi { print "hello". }
declare local function hi { print "hello". }
local function hi { print "hello". }
function hi { print "hello". }
```

Functions are presumed to have scope local to the location where they are declared when the explicit local scope keyword is missing.

At the moment, it is redundant to mention the `local` keyword, although it is allowed.

It is best to just leave all the optional keywords of and merely say `function` by itself.

example:

```
// Print the string you pass in, in one of the 4 corners
// of the terminal:
//   mode = 1 for upper-left, 2 for upper-right, 3
//          for lower-left, and 4 for lower-right:
//
function print_corner {
  parameter mode.
  parameter text.

  local row is 0.
  local col is 0.

  if mode = 2 or mode = 4 {
    set col to terminal:width - text:length.
  }.
  if mode = 3 or mode = 4 {
    set row to terminal:height - 1.
  }.

  print text at (col, row).
}.

// An example of calling it:

print_corner(4,"That's me in the corner").
```

A declare function command can appear anywhere in a kerboscript program, and once its been "parsed" by the compiler, the function can be called from anywhere in the program.

The best design pattern is probably to create your library of function calls as one or more separate .ks files that contain ONLY function definitions and nothing else in them. Then when you "run" the file containing the functions, what you're really doing is just loading the function definitions into memory so they can be called by other programs. At the top of your main script you can then "run" the other scripts containing the library of functions to get them compiled into memory.

5.5.3 DECLARE PARAMETER

If your function expects to have parameters passed into it, you can use the *DECLARE PARAMETER* command to do so. This is the same command as is used to declare parameters for running a whole script. By putting a DECLARE PARAMETER statement inside a function, you tell the kerboscript compiler that you want the parameter to be for that function, not for the whole script.

An example of using `declare parameter` can be seen in the example above, where it is used for the `mode` and `text` parameters.

(Again, even when the word 'declare' is missing, we still call them 'declare parameter' commands.)

5.5.4 Calling a function

To call a function you created, you call it the same way you call a built-in function, by putting a pair of parentheses to the right of it, as shown here:

```
function example_function {
  print "hello, this is my example.".
}

example_function().
```

If the function takes parameters, then you put them in the parentheses just like when running a program. You can see an example of this above in the previous example where it said:

```
print_corner(4,"That's me in the corner").
```

Functions and the terminal interpreter

You **cannot** call functions from the interpreter interactive command line if they were declared inside of script programs. If you do, you will get seemingly "random" errors. The reasons for this are complex, but the short version is because the memory the script files' pseudo-machine language instructions live in and the memory the interpreter's pseudo-machine langauge instructions live in are two different things.

The effect you may see if you attempt this is merely an "Unknown Identifer" error, or worse yet, it may end up jumping into random parts of your code that have nothing to do with the actual function call you're trying to make.

As a rule of thumb, in kOS 0.17.0, make sure you only use functions from inside script programs. Don't try to call them interactively from the interpreter prompt. You will get very strange and (seemingly) inexplicable errors.

In the future we may find a way to fix this problem, but for right now, just don't do it.

Calling a function without parentheses (please don't)

In some cases it is possible to call a function with the parentheses off, as shown below, but this is not recommended:

```
function example_function {
  print "hello, this is my example.".
}

example_function. // please don't do this, even if it works.
```

This is a holdover from the fact that functions and locks are really the same thing, and you need to be able to call a lock without the parentheses for old scripts written prior to kOS version 0.17.0 to continue working.

Omitting parentheses only works in the same file

One reason to avoid the above technique (of leaving the parentheses off) is that it really only works when you try to call a function that was declared in the same file. If you want to call a *library* function (a function you made for yourself in another file) then it does not work, for complex reason involving the compiler and late-time binding.

5.5.5 LOCAL .. TO

(aka: **local variables**)

Syntax:

- `DECLARE` *identifier* `TO` *expression dot*
- `LOCAL` *identifier* `IS` *expression dot*
- `DECLARE LOCAL` *identifier* `IS` *expression dot*

The above are all the same, although the version that just says `LOCAL identifier IS expr.` is preferred.

Examples:

```
declare x to 5.
local y is 2*x - 1.
declare local halfSpeed to SHIP:VELOCITY:ORBIT:MAG / 2.
```

If your function needs to make a local variable, it can do so using the *DECLARE* command. Whenever the DECLARE command is seen inside a function, the compiler assumes the variable is meant to be local to that function's block. This also works with recursion. If you recursively call a function again and again, there will be new copies stacked up of all the local variables made with DECLARE, but not of the variables implicitly made global without DECLARE.

An example of using `local` for a local variable can be seen in the example above, where it is used for the `row` and `col` variables.

A more in-depth explanation of kerboscript's scoping rules and how they work is found *on another page*

Initializers are now mandatory for the DECLARE statement

This is now **illegal** syntax:

```
declare x.  // no initial value for x given.
```

Warning: New in version 0.17: **Breaking Change:** The kerboscript from prior versions of kOS did allow you do make `declare` statements without any initializers in them (and in fact you couldn't provide an initializer for them in prior versions even if you wanted to.)

In order to avoid the issue of having uninitialized variables in kerboscript, any declare statement *requires* the use of the initializer clause.

> *This is especially important as kerboscript is a late typing language in which it is impossible for the compiler to choose some implied default initial value for the variable from some language spec. This is because until a value has been assigned into it, the compiler wouldn't even know what type of default to use - a string, an integer, a floating point number, etc.*

Difference between declare and set

You may think that:

```
local x is 5.
```

is identical to just not using a declare local statement at all, and just performing `set x to 5.` alone, but it is not. With `declare local` (or just `declare` or just `local`), a NEW variable called `x` will be made at the current local scope, temporarily hiding any existing `x` variables that may otherwise have been reachable in a more global scope. With `set`, if there already is an `x` variable you can use in a different scope higher than this scope, it will be used, and only if it doesn't exist will a new `x` be made (and that new `x` will be global, not local).

5.5.6 RETURN

`return` *expression(optional) dot(mandatory)*

examples:

```
return 3*x.

return.
```

If your function needs to exit early, and/or if it needs to pass a return value back to the user, you can use the RETURN statement to do so. RETURN accepts an optional argument - the value to pass back to the caller. Note that functions in kerboscript are very weakly typed with late binding. You cannot declare the expected return type for the function, and it's up to you to ensure that all possible returned values are useful and meaningful.

example:

```
// Note, in this example, the keyword 'declare' is
// spelled out explicitly.  You can choose to do so
// if you wish.  It's up to you what you aesthetically
// prefer.

// Calculate what component of a vessel's surface
// velocity is Northward:
declare function north_velocity {
  declare parameter which_vessel.

  return VDOT(which_vessel:velocity:surface, which_vessel:north:vector).
}.
```

5.5.7 Passing by value

Parameters to user functions in kerboscript are all pass-by-value, with an important caveat. "Pass by value" means that the function is working on a copy of the variable you passed in, rather than the original variable. This matters when the function tries to change the value of the parameter, as in this example:

```
function embiggen {
  parameter x.

  set x to x + 10.

  print "x has been embiggened to " + x.
}.

set global_val to 30.
print global_val.
embiggen(global_val).
print global_val.
```

The above example will print:

```
30
x has been embiggened to 40
30
```

Although the function added 10 to its OWN copy of the parameter, the caller's copy of the parameter remained unchanged.

Important exception to passing by value - structures

If the value being sent to the function as its parameter is a complex structure consisting of sub-parts (i.e. if it has suffixes) rather than being a simple single scalar value like a number, then the copy in the function is *really* a copy of the reference pointing to the object, so changes you make in the object really WILL change it, as shown here:

```
function half_vector {
  parameter vec. //vector passed in.

  print "full vector is " + vec.

  set vec:x to vec:x/2.
  set vec:y to vec:y/2.
  set vec:z to vec:z/2.

  print "half vector is " + vec.
}.

set global_vec to V(10,20,30).
half_vector(global_vec).
print "afterward, global_vec is now " + global_vec.
```

This will give the following result:

```
full vector is v(10,20,30)
half vector is v(5,10,15)
afterward, global_vec is now v(5,10,15)
```

Because a vector is a suffixed structure, it effectively acts as if it was passed in by reference instead of by value, and so when it was changed in the function, the caller's original copy is what was being changed.

This may be hard to get used to for new programmers, however experienced programmers who use some modern object-oriented languages will find this behavior very familiar. Only primitives are passed by value. Structures are passed by their reference rather than trying to make a deep copy of the object for the function to use.

This behavior is inherited from the fact that kerboscript is implemented on top of C#, which is one of several OOP languages that work like this.

5.5.8 Nesting functions inside functions

You are allowed to make a local function existing inside another function.

This means that the containing function is the only place the nested function can be called from.

Example:

```
function getMean {
  parameter aList.

  function getSum {
    parameter aList. // note, this is a local aList MASKING the other one.

    local sum is 0.
    for num in aList {
      set sum to sum + num.
    }.
    return sum.
  }.
```

```
  return getSum(aList) / aList:LENGTH.
}.

set L to LIST();
L:ADD(10).
L:ADD(9).
print "mean average is " + getMean(L).

// The following line will give an error because
// getSum is local inside of getMean, and isn't allowed
// to be called from here:
//
print "getSum is " + getSum(L).
```

5.5.9 Recursion

Recursive algorithms (http://en.wikipedia.org/wiki/Recursion#In_computer_science) are possible with kerboscript functions, provided you remember to always exclusively use local variables made with a declare statement in the body of the function, and never use global variables for something that you intended to be different per recursive call.

5.5.10 User Function Gotchas

Calling program's functions from the interpreter

As *explained above*, kOS 0.17.0 does not support the calling of a function from the interpreter console and if you attempt it you will get very strange and random errors that you might waste a lot of time trying to track down.

Inconsistent returns

Note that if you sometimes do and sometimes don't return a value, from the same function, as in the example here:

```
// A badly designed function, with inconsistency
// in whether or not it returns a value:
//
DECLARE FUNCTION foo {
   DECLARE PARAMETER x.
   IF X < 0 {
     RETURN. // no return value.
   } ELSE {
     RETURN "hello". // a string return value
   }.
}.
```

Then the kerboscript compiler is not clever enough to detect this and warn you about it. The internal stack will not get corrupted by this error, as some experienced programmers might expect upon hearing this (because secretly all kerboscript user functions return a value even if it's never used, so there's universally always something to pop off the stack even for the empty return statements.) However, you will still have to deal with the fact that the calling program might be getting nulls back some of the time if you make this programming error.

In general, make sure that if you *sometimes* return a value from a user function, that you *always* do so in every path through your function.

Accidentally using globals

It is possible to accidentally create global variables when you didn't meant to, just because you made a typo.

For example:

```
function mean {
  parameter the_list.
  local sum is 0.

  for item in the_list {
    set dum to sum + item. // typo - said 'dum' instead of 'sum'.
  }.

  return sum / the_list:length.
}.
```

The above example contains a typo that causes a global variable to be made where you didn't mean to. You wanted to say "sum" but said "dum" and instead of that being an error, kerboscript happily said "okay, well since you're setting a variable name that doesn't exist yet, I'll make it for you implicitly" (and it ends up being a global).

When you are writing libraries of code for yourself to call, this can really be annoying. And it's a very common problem with "sloppy" declaration languages that allow you to use variable names without declaring them first. Most such languages have provided a way to catch the problem, and allow you to instruct the compiler "please don't let me do that. Please force me to declare everything".

The way that is done in kerboscript is by using a `@LAZYGLOBAL` compiler directive, *as described here*.

Had the function above been compiled under a `@LAZYGLOBAL off.` compiler directive, the typo would be noticed:

```
@lazyglobal off.

local function mean {
  local parameter the_list.
  local sum is 0.

  for item in the_list {
    set dum to sum + item. // error - 'dum' is an unknown identifier.
  }.

  return sum / the_list:length.
}.
```

CHAPTER

SIX

MATHEMATICS AND BASIC GEOMETRY

6.1 Fundamental Constants

There is a bound variable called CONSTANT which contains some basic fundamental constants about the universe that you may find handy in your math operations.

New in version 0.18: Prior to kOS version 0.18, `constant` was a function call, and therefore to say `constant:pi`, you had to say `constant():pi`. The function call `constant()` still exists and still works, but the new way without the parentheses is preferred going forward, and the way with the parentheses may become deprecated later. For the moment, both ways of doing it work.

Identifier	Description
`G`	Newton's Gravitational Constant
`E`	Base of the natural log (Euler's number)
`PI`	π
`c`	Speed of light in a vacuum, in m/s.
`AtmToKPa`	Conversion constant: Atmospheres to kiloPascals.
`KPaToAtm`	Conversion constant: kiloPascals to Atmospheres.
`DegToRad`	Conversion constant: Degrees to Radians.
`RadToDeg`	Conversion constant: Radians to Degrees.

Constant:G

Newton's Gravitational Constant, 6.67384E-11:

```
PRINT "Gravitational parameter of Kerbin is:".
PRINT constant:G * Kerbin:Mass.
```

Constant:E

Natural Log base "e":

```
PRINT "e^2 is:".
PRINT constant:e ^ 2.
```

Constant:PI

Ratio of circumference of a circle to its diameter

Constant:C

Speed of light in a vacuum, in meters per second.

Note: In Kerbal Space Program, all physics motion is purely Newtonian. You can go faster than the speed of light provided you have enough delta-V, and no time dilation effects will occur. The universe will behave entirely linearly even at speeds near *c*.

This constant is provided mainly for the benefit of people who are playing with the mod "RemoteTech" installed, who may want to perform calculations about signal delays to hypothetical probes. (Note that if the probe already has a connection, you can *ask Remotetech directly* what the signal delay is.

Constant:AtmToKPa
A conversion constant.

If you have a pressure measurement expressed in atmospheres of pressure, you can multiply it by this to get the equivalent in kiloPascals (kiloNewtons per square meter).

Constant:KPaToATM
A conversion constant.

If you have a pressure measurement expressed in kiloPascals (kiloNewtons per square meter), you can multiply it by this to get the equivalent in atmospheres.

Constant:DegToRad
A conversion constant.

If you have an angle measured in degrees, you can multiply it by this to get the equivalent measure in radians. It is exactly the same thing as saying `constant:pi / 180`, except the result is pre-recorded as a constant number and thus no division is performed at runtime.

Constant:RadToDeg
A conversion constant.

If you have an angle measured in radians, you can multiply it by this to get the equivalent measure in degrees. It is exactly the same thing as saying `180 / constant:pi`, except the result is pre-recorded as a constant number and thus no division is performed at runtime.

6.2 Mathematical Functions

Function	Description
ABS(a)	absolute value
CEILING(a)	round up
FLOOR(a)	round down
LN(a)	natural log
LOG10(a)	log base 10
MOD(a,b)	modulus
MIN(a,b)	minimum
MAX(a,b)	maximum
RANDOM()	random number
ROUND(a)	round to whole number
ROUND(a,b)	round to nearest place
SQRT(a)	square root

ABS(a)
Returns absolute value of input:

```
PRINT ABS(-1). // prints 1
```

CEILING(a)
Rounds up to the nearest whole number:

```
PRINT CEILING(1.887). // prints 2
```

FLOOR(a)
Rounds down to the nearest whole number:

```
PRINT FLOOR(1.887). // prints 1
```

LN(a)
Gives the natural log of the provided number:

```
PRINT LN(2). // prints 0.6931471805599453
```

LOG10(a)
Gives the log base 10 of the provided number:

```
PRINT LOG10(2). // prints 0.30102999566398114
```

MOD(a,b)
Returns remainder from integer division. Keep in mind that it's not a traditional mathematical Euclidean division where the result is always positive. The result has the same absolute value as mathematical modulo operation but the sign is the same as the sign of dividend:

```
PRINT MOD(21,6). // prints 3
PRINT MOD(-21,6). // prints -3
```

MIN(a,b)
Returns The lower of the two values:

```
PRINT MIN(0,100). // prints 0
```

MAX(a,b)
Returns The higher of the two values:

```
PRINT MAX(0,100). // prints 100
```

RANDOM()
Returns a random floating point number in the range [0,1]:

```
PRINT RANDOM(). //prints a random number
```

ROUND(a)
Rounds to the nearest whole number:

```
PRINT ROUND(1.887). // prints 2
```

ROUND(a,b)
Rounds to the nearest place value:

```
PRINT ROUND(1.887,2). // prints 1.89
```

SQRT(a)
Returns square root:

```
PRINT SQRT(7.89). // prints 2.80891438103763
```

6.2.1 Trigonometric Functions

Function
SIN(a)
COS(a)
TAN(a)
ARCSIN(x)
ARCCOS(x)
ARCTAN(x)
ARCTAN2(x,y)

SIN (a)

Parameters

- **a** – (deg) angle

Returns sine of the angle

```
PRINT SIN(6). // prints 0.10452846326
```

COS (a)

Parameters

- **a** – (deg) angle

Returns cosine of the angle

```
PRINT COS(6). // prints 0.99452189536
```

TAN (a)

Parameters

- **a** – (deg) angle

Returns tangent of the angle

```
PRINT TAN(6). // prints 0.10510423526
```

ARCSIN (x)

Parameters

- **x** – (scalar)

Returns (deg) angle whose sine is x

```
PRINT ARCSIN(0.67). // prints 42.0670648
```

ARCCOS (x)

Parameters

- **x** – (scalar)

Returns (deg) angle whose cosine is x

```
PRINT ARCCOS(0.67). // prints 47.9329352
```

ARCTAN (x)

Parameters

- **x** – (scalar)

Returns (deg) angle whose tangent is x

```
PRINT ARCTAN(0.67). // prints 33.8220852
```

ARCTAN2 (y,x)

Parameters

- **y** – (scalar)
- **x** – (scalar)

Returns (deg) angle whose tangent is $\frac{y}{x}$

```
PRINT ARCTAN2(0.67, 0.89). // prints 36.9727625
```

The two parameters resolve ambiguities when taking the arctangent. See the wikipedia page about atan2 for more details.

6.3 Vectors

Contents

6.3.1 Creation

V (x,y,z)

Parameters

- **x** – (scalar) x coordinate
- **y** – (scalar) y coordinate
- **z** – (scalar) z coordinate

Returns *Vector*

This creates a new vector from 3 components in (x, y, z):

```
SET vec TO V(x,y,z).
```

Here, a new *Vector* called `vec` is created . The object *Vector* represents a three-dimensional euclidean vector To deeply understand most vectors in kOS, you have to understand a bit about the *underlying coordinate system of KSP*. If you are having trouble making sense of the direction the axes point in, go read that page.

Note: Remember that the XYZ grid in Kerbal Space Program uses a *left-handed* coordinate system.

6.3.2 Structure

structure Vector

Table 6.1: Members

Suffix	Type	Get	Set
X	scalar	yes	yes
Y	scalar	yes	yes
Y	scalar	yes	yes
MAG	scalar	yes	yes
NORMALIZED	*Vector*	yes	no
SQRMAGNITUDE	scalar	yes	no
DIRECTION	*Direction*	yes	yes
VEC	*Vector*	yes	no

Vector:**X**

Type scalar

Access Get/Set

The x component of the vector.

Vector:**Y**

Type scalar

Access Get/Set

The y component of the vector.

Vector:**Z**

Type scalar

Access Get/Set

The z component of the vector.

Vector:**MAG**

Type scalar

Access Get/Set

The magnitude of the vector, as a scalar number, by the Pythagorean Theorem.

Vector:**NORMALIZED**

Type *Vector*

Access Get only

This creates a unit vector pointing in the same direction as this vector. This is the same effect as multiplying the vector by the scalar `1 / vec:MAG`.

Vector:**SQRMAGNITUDE**

Type scalar

Access Get only

The magnitude of the vector, squared. Use instead of `vec:MAG^2` if you need to square of the magnitude as this skips the step in the Pythagorean formula where you take the square root in the first place. Taking the square root and then squaring that would introduce floating point error needlessly.

Vector:**DIRECTION**

Type *Direction*

Access Get/Set

GET: The vector rendered into a *Direction* (see note at the bottom of this page about information loss when doing this).

SET: Tells the vector to keep its magnitude as it is but point in a new direction, adjusting its (x, y, z) numbers accordingly.

Vector:**VEC**

Type *Vector*

Access Get only

This is a suffix that creates a *COPY* of this vector. Useful if you want to copy a vector and then change the copy. Normally if you `SET v2 TO v1`, then `v1` and `v2` are two names for the same vector and changing one would change the other.

6.3.3 Operations and Methods

Method / Operator	Return Type
** (asterisk)*	scalar or *Vector*
+ (plus)	*Vector*
- (minus)	*Vector*
- (unary)	*Vector*
VDOT, *VECTORDOTPRODUCT*, *	scalar
VCRS, *VECTORCROSSPRODUCT*	*Vector*
VANG, *VECTORANGLE*	scalar (deg)
VXCL, *VECTOREXCLUDE*	*Vector*

*****

Scalar multiplication or dot product of two `Vectors`. See also *VECTORDOTPRODUCT*:

```
SET a TO 2.
SET vec1 TO V(1,2,3).
SET vec2 TO V(2,3,4).
PRINT a * vec1.      // prints: V(2,4,6)
PRINT vec1 * vec2.   // prints: 20
```

Note that the *unary* minus operator is really a multiplication of the vector by a scalar of (-1):

```
PRINT -vec1.      // these two both print the
PRINT (-1)*vec1. // exact same thing.
```

+, -

Vector addition and subtraction by a scalar or another *Vector*:

```
SET a TO 2.
SET vec1 TO V(1,2,3).
SET vec2 TO V(2,3,4).
PRINT vec1 + vec2.  // prints: V(3,5,7)
PRINT vec2 - vec1.  // prints: V(1,1,1)
```

Note that the *unary* minus operator is the same thing as multiplying the vector by a scalar of (-1), and is not technically an addition or subtraction operator:

```
PRINT -vec1.      // these two both print the
PRINT (-1)*vec1. // exact same thing.
```

VDOT(v1,v2)
Same as *VECTORDOTPRODUCT(v1,v2)* and *v1 * v2*.

VECTORDOTPRODUCT(v1,v2)

Parameters

- **v1** – (*Vector*)
- **v2** – (*Vector*)

Returns The vector dot-product

Return type scalar

This is the dot product of two vectors returning a scalar number. This is the same as *v1 * v2*:

```
SET vec1 TO V(1,2,3).
SET vec2 TO V(2,3,4).

// These will all print the value: 20
PRINT vec1 * vec2.
PRINT VDOT(vec1, vec2).
PRINT VECTORDOTPRODUCT(vec1, vec2).
```

VCRS(v1,v2)
Same as *VECTORCROSSPRODUCT(v1,v2)*

VECTORCROSSPRODUCT(v1,v2)

Parameters

- **v1** – (*Vector*)
- **v2** – (*Vector*)

Returns The vector cross-product

Return type *Vector*

The vector cross product of two vectors in the order `(v1,v2)` returning a new *Vector*:

```
SET vec1 TO V(1,2,3).
SET vec2 TO V(2,3,4).

// These will both print: V(-1,2,-1)
PRINT VCRS(vec1, vec2).
PRINT VECTORCROSSPRODUCT(vec1, vec2).
```

When visualizing the direction that a vector cross product will point, remember that KSP is using a *left-handed* coordinate system, and this means a cross-product of two vectors will point in the opposite direction of what it would had KSP been using a right-handed coordinate system.

VANG(v1,v2)
Same as *VECTORANGLE(v1,v2)*.

VECTORANGLE(v1,v2)

Parameters

- **v1** – (*Vector*)
- **v2** – (*Vector*)

Returns Angle between two vectors

Return type scalar

This returns the angle between v1 and v2. It is the same result as:

$$\arccos\left(\frac{\vec{v_1}\cdot\vec{v_2}}{|\vec{v_1}\cdot\vec{v_2}|}\right)$$

or in **KerboScript**:

```
arccos( (VDOT(v1,v2) / VDOT(v1,v2):MAG ) )
```

VXCL (v1,v2)
Same as *VECTOREXCLUDE(v1,v2)*

VECTOREXCLUDE (v1,v2)
This is a vector, v2 with all of v1 excluded from it. In other words, the projection of v2 onto the plane that is normal to v1.

Some examples of using the *Vector* object:

```
// initializes a vector with x=100, y=5, z=0
SET varname TO V(100,5,0).

varname:X.    // Returns 100.
V(100,5,0):Y. // Returns 5.
V(100,5,0):Z. // Returns 0.

// Returns the magnitude of the vector
varname:MAG.

// Changes x coordinate value to 111.
SET varname:X TO 111.

// Lengthen or shorten vector to make its magnitude 10.
SET varname:MAG to 10.

// get vector pointing opposite to surface velocity.
SET retroSurf to (-1)*velocity:surface.

// use cross product to find normal to the orbit plane.
SET norm to VCRS(velocity:orbit, ship:body:position).
```

6.4 Directions

Contents

Direction objects represent a rotation starting from an initial point in **KSP**'s coordinate system where the initial state was looking down the $+z$ axis, with the camera "up" being the $+y$ axis. This exists primarily to enable automated steering.

In your thinking, you can largely think of Directions as being Rotations and Rotations as being Directions. The two concepts can be used interchangeably. Used on its own to steer by, a rotation from the default XYZ axes of the universe into a new rotation does in fact provide an absolute direction, thus the name Direction for these objects even though

in reality they are just Rotations. It's important to know that Directions are just rotations because you can use them to modify other directions or vectors.

Note: When dealing with Directions (which are Rotations) in kOS, it is important to remember that KSP uses a *left-handed* coordinate system. This affects the convention of which rotation direction is positive when calculating angles.

6.4.1 Creation

Method	Description
R(pitch,yaw,roll)	Euler rotation
Q(x,y,z,rot)	Quaternion
HEADING(dir,pitch)	Compass heading
LOOKDIRUP(lookAt,lookUp)	Looking along vector *lookAt*, rolled so that *lookUp* is upward.
ANGLEAXIS(degress,axisVector)	A rotation that would rotate the universe around an axis
ROTATEFROMTO(fromVec,toVec)	A rotation that would go from vectors fromVec to toVec
FACING	From SHIP or TARGET
UP	From SHIP
PROGRADE, etc.	From SHIP, TARGET or BODY

R(pitch,yaw,roll)

A *Direction* can be created out of a Euler Rotation, indicated with the *R()* function, as shown below where the `pitch`, `yaw` and `roll` values are in degrees:

```
SET myDir TO R( a, b, c ).
```

Q(x,y,z,rot)

A *Direction* can also be created out of a *Quaternion* tuple, indicated with the *Q()* function, as shown below where `x`, `y`, and `z` are a *Vector* to rotate around, and `rot` is how many degrees to rotate:

```
SET myDir TO Q( x, y, z, rot ).
```

HEADING(dir,pitch)

A *Direction* can be created out of a *HEADING()* function. The first parameter is the compass heading, and the second parameter is the pitch above the horizon:

```
SET myDir TO HEADING(degreesFromNorth, pitchAboveHorizon).
```

LOOKDIRUP(lookAt,lookUp)

A *Direction* can be created with the LOOKDIRUP function by using two vectors. This is like converting a vector to a direction directly, except that it also provides roll information, which a single vector lacks. *lookAt* is a vector describing the Direction's FORE orientation (its local Z axis), and *lookUp* is a vector describing the direction's TOP orientation (its local Y axis). Note that *lookAt* and *lookUp* need not actually be perpendicualr to each other - they just need to be non-parallel in some way. When they are not perpendicular, then a vector resulting from projecting *lookUp* into the plane that is normal to *lookAt* will be used as the effective *lookUp* instead:

```
// Aim up the SOI's north axis (V(0,1,0)), rolling the roof to point to the sun.
LOCK STEERING TO LOOKDIRUP( V(0,1,0), SUN:POSITION ).
//
// A direction that aims normal to orbit, with the roof pointed down toward the planet:
LOCK normVec to VCRS(SHIP:BODY:POSITION,SHIP:VELOCITY:ORBIT).  // Cross-product these for a norm
LOCK STEERING TO LOOKDIRUP( normVec, SHIP:BODY:POSITION).
```

ANGLEAXIS(degress,axisVector)

A *Direction* can be created with the ANGLEAXIS function. It represents a rotation of *degrees* around an

axis of *axisVector*. To know which way a positive or negative number of degrees rotates, remember this is a left-handed coordinate system:

```
// Pick a new rotation that is pitched 30 degrees from the current one, taking into account
// the ship's current orientation to decide which direction is the 'pitch' rotation:
//
SET pitchUp30 to ANGLEAXIS(-30,SHIP:STARFACING).
SET newDir to pitchUp30*SHIP:FACING.
LOCK STEERING TO newDir.
```

Note: The fact that KSP is using a *left-handed* coordinate system is important to keep in mind when visualizing the meaning of an ANGLEAXIS function call. It affects which direction is positive when calculating angles.

ROTATEFROMTO (fromVec,toVec)

A *Direction* can be created with the ROTATEFROMTO function. It is *one of the infinite number of* rotations that could rotate vector *fromVec* to become vector *toVec* (or at least pointing in the same direction as toVec, since fromVec and toVec need not be the same magnitude). Note the use of the phrase "**infinite number of**". Because there's no guarantee about the roll information, there are an infinite number of rotations that could qualify as getting you from one vector to another, because there's an infinite number of roll angles that could result and all still fit the requirement:

```
SET myDir to ROTATEFROMTO( v1, v2 ).
```

Suffix terms from other structures

A *Direction* can be made from many suffix terms of other structures, as shown below:

```
SET myDir TO SHIP:FACING.
SET myDir TO TARGET:FACING.
SET myDir TO SHIP:UP.
```

Whenever a *Direction* is printed, it always comes out showing its Euler Rotation, regardless of how it was created:

```
// Initializes a direction to prograde
// plus a relative pitch of 90
SET X TO SHIP:PROGRADE + R(90,0,0).

// Steer the vessel in the direction
// suggested by direction X.
LOCK STEERING TO X.

// Create a rotation facing northeast,
// 10 degrees above horizon
SET Y TO HEADING(45, 10).

// Steer the vessel in the direction
// suggested by direction Y.
LOCK STEERING TO Y.

// Set by a rotation in degrees
SET Direction TO R(0,90,0).
```

6.4.2 Structure

structure Direction

The suffixes of a *Direction* cannot be altered, so to get a new *Direction* you must construct a new one.

Suffix	Type	Description
PITCH	scalar (deg)	Rotation around x axis
YAW	scalar (deg)	Rotation around y axis
ROLL	scalar (deg)	Rotation around z axis
FOREVECTOR	*Vector*	This Direction's forward vector (z axis after rotation).
VECTOR	*Vector*	Alias synonym for *FOREVECTOR*
TOPVECTOR	*Vector*	This Direction's top vector (y axis after rotation).
UPVECTOR	*Vector*	Alias synonym for *TOPVECTOR*
STARVECTOR	*Vector*	This Direction's starboard vector (z axis after rotation).
RIGHTVEC-TOR	*Vector*	Alias synonym for *STARVECTOR*
INVERSE	*Direction*	The inverse of this direction.
`unary minus`	*Direction*	Using the negation operator "-" on a Direction does the same thing as using the :INVERSE suffix on it.

The *Direction* object exists primarily to enable automated steering. You can initialize a *Direction* using a *Vector* or a `Rotation`. *Direction* objects represent a rotation starting from an initial point in **KSP**'s coordinate system where the initial state was looking down the $+z$ axis, with the camera "up" being the $+y$ axis. So for example, a *Direction* pointing along the x axis might be represented as `R(0,90,0)`, meaning the initial z-axis direction was rotated *90 degrees* around the y axis.

If you are going to manipulate directions a lot, it's important to note that the order in which the rotations occur is:

1. First rotate around z axis.
2. Then rotate around x axis.
3. Then rotate around y axis.

What this means is that if you try to `ROLL` and `YAW` in the same tuple, like so: `R(0,45,45)`, you'll end up **rolling first and then yawing**, which might not be what you expected. There is little that can be done to change this as it's the native way things are represented in the underlying **Unity engine**.

Also, if you are going to manipulate directions a lot, it's important to note how **KSP**'s native coord system works.

Direction:**`PITCH`**

Type scalar (deg)

Access Get only

Rotation around the x axis.

Direction:**`YAW`**

Type scalar (deg)

Access Get only

Rotation around the y axis.

Direction:**`ROLL`**

Type scalar (deg)

Access Get only

Rotation around the z axis.

Direction:**FOREVECTOR**

Type *Vector*

Access Get only

Vector of length 1 that is in the same direction as the "look-at" of this Direction. Note that it is the same meaning as "what the Z axis of the universe would be rotated to if this rotation was applied to the basis axes of the universe". When you LOCK STEERING to a direction, that direction's FOREVECTOR is the vector the nose of the ship will orient to. SHIP:FACING:FOREVECTOR is the way the ship's nose is aimed right now.

Direction:**TOPVECTOR**

Type *Vector*

Access Get only

Vector of length 1 that is in the same direction as the "look-up" of this Direction. Note that it is the same meaning as "what the Y axis of the universe would be rotated to if this rotation was applied to the basis axes of the universe". When you LOCK STEERING to a direction, that direction's TOPVECTOR is the vector the roof of the ship will orient to. SHIP:FACING:TOPVECTOR is the way the ship's roof is aimed right now.

Direction:**STARVECTOR**

Type *Vector*

Access Get only

Vector of length 1 that is in the same direction as the "starboard side" of this Direction. Note that it is the same meaning as "what the X axis of the universe would be rotated to if this rotation was applied to the basis axes of the universe". When you LOCK STEERING to a direction, that direction's STARVECTOR is the vector the right wing of the ship will orient to. SHIP:FACING:STARVECTOR is the way the ship's right wing is aimed right now.

Direction:**INVERSE**

Type *Direction*

Access Get only

Struct Gives a *Direction* with the opposite rotation around its axes.

Note: The difference between a :struct:'Direction' and a "Vector"

`Vector` and a *Direction* can be represented with the exact same amount of information: a tuple of 3 floating point numbers. So you might wonder why it is that a `Vector` can hold information about the magnitude of the line segment, while a *Direction* cannot, given that both have the same amount of information. The answer is that a *Direction* does contain one thing a `Vector` does not. A *Direction* knows which way is "up", while a `Vector` does not. If you tell **kOS** to `LOCK STEERING` to a `Vector`, it will be able to point the nose of the vessel in the correct direction, but won't know which way you want the roof of the craft rotated to. This works fine for axial symmetrical rockets but can be a problem for airplanes.

6.4.3 Operations and Methods

You can use math operations on *Direction* objects as well. The next example uses a rotation of "UP" which is a system variable describing a vector directly away from the celestial body you are under the influence of:

Supported Direction Operators:

Direction Multiplied by Direction `Dir1 * Dir2` - This operator returns the result of rotating Dir2 by the rotation of Dir1. Note that the order of operations matters here. `Dir1*Dir2` is not the same as `Dir2*Dir1`. Example:

```
// A direction pointing along compass heading 330, by rotating NORTH by 30 degrees around UP axi
SET newDir TO ANGLEAXIS(30,SHIP:UP) * NORTH.
```

Direction Multiplied by Vector `Dir * Vec` - This operator returns the result of rotating the vector by Dir:

```
// What would the velocity of your ship be if it was angled 20 degrees to your left?
SET Vel to ANGLEAXIS(-20,SHIP:TOPVECTOR) * SHIP:VELOCITY:ORBIT.
// At this point Vel:MAG and SHIP:VELOCITY:MAG should be the same, but they don't point the same
```

Direction Added to Direction `Dir1 + Dir2` - This operator is less reliable because its exact behavior depends on the order of operations of the UnityEngine's X Y and Z axis rotations, and it can result in gimbal lock. It's supposed to perform a Euler rotation of one direction by another, but it's preferred to use Dir*Dir instead, as that doesn't experience gimbal lock, and does not require that you know the exact transformation order of Unity.

For vector operations, you may use the `:VECTOR` suffix in combination with the regular vector methods:

```
SET dir TO SHIP:UP.
SET newdir TO VCRS(SHIP:PROGRADE:VECTOR, dir:VECTOR)
```

6.4.4 Vectors and Directions

There are some consequences when converting from a *Direction* to a `Vector` and vice versa which should not be overlooked.

A `Vector` and a *Direction* can be represented with the exact same amount of information: a tuple of 3 floating point numbers. So you might wonder why it is that a `Vector` can hold information about the magnitude of the line segment, while a *Direction* cannot, given that both have the same amount of information. The answer is that a *Direction* does contain one thing a `Vector` does not. A *Direction* knows which way is "up", while a `Vector` does not. If you tell **kOS** to `LOCK STEERING` to a `Vector`, it will be able to point the nose of the vessel in the correct direction, but won't know which way you want the roof of the craft rotated to. This works fine for axial symmetrical rockets but can be a problem for airplanes.

Therefore if you do this:

```
SET MyVec to V(100,200,300).
SET MyDir to MyVec:DIRECTION.
```

Then `MyDir` will be a *Direction*, but it will be a *Direction* where you have no control over which way is "up" for it.

6.5 Geographic Coordinates

Contents

The GeoCoordinates object (also LATLNG) represents a latitude and longitude pair, which is a location on the surface of a *Body*.

6.5.1 Creation

LATLNG (lat,lng)

Parameters

- **lat** – (deg) Latitude
- **lng** – (deg) Longitude

Returns *GeoCoordinates*

This function creates a GeoCoordiantes object with the given latitude and longitude. Once created it can't be changed. The *GeoCoordinates:LAT* and *GeoCoordinates:LNG* suffixes are get-only and cannot be set. To switch to a new location, make a new call to *LATLNG()*.

It is also possible to obtain a GeoCoordiates from some suffixes of some other structures. For example:

```
SET spot to SHIP:GEOPOSITION.
```

6.5.2 Structure

structure GeoCoordinates

Suffix	Type	Args	Description
LAT	scalar (deg)	none	Latitude
LNG	scalar (deg)	none	Longitude
DISTANCE	scalar (m)	none	distance from *CPU Vessel*
TERRAINHEIGHT	scalar (m)	none	above or below sea level
HEADING	scalar (deg)	none	*absolute* heading from *CPU Vessel*
BEARING	scalar (deg)	none	*relative* direction from *CPU Vessel*
POSITION	*Vector* (3D Ship-Raw coords)	none	Position of the surface point.
ALTITUDEPOSITION	*Vector* (3D Ship-Raw coords)	scalar (altitude above sea level)	Position of a point above (or below) the surface point, by giving the altitude number.

GeoCoordinates:**LAT**
The latitude of this position on the surface.

GeoCoordinates:**LNG**
The longitude of this position on the surface.

GeoCoordinates:**DISTANCE**
Distance from the *CPU_Vessel* to this point on the surface.

GeoCoordinates:**TERRAINHEIGHT**
Distance of the terrain above "sea level" at this geographical position. Negative numbers are below "sea level."

GeoCoordinates:**HEADING**
The *absolute* compass direction from the *CPU_Vessel* to this point on the surface.

GeoCoordinates:**BEARING**
The *relative* compass direction from the *CPU_Vessel* to this point on the surface. For example, if

the vessel is heading at compass heading 45, and the geo-coordinates location is at heading 30, then *GeoCoordinates:BEARING* will return -15.

GeoCoordinates:**POSITION**
The ship-raw 3D position on the surface of the body, relative to the current ship's Center of mass.

GeoCoordinates:**ALTITUDEPOSITION**
The ship-raw 3D position above or below the surface of the body, relative to the current ship's Center of mass. You pass in an altitude number for the altitude above "sea" level of the desired location.

6.5.3 Examples Usage

```
SET spot TO LATLNG(10, 20).      // Initialize point at latitude 10,
                                 // longitude 20

PRINT spot:LAT.                  // Print 10
PRINT spot:LNG.                  // Print 20

PRINT spot:DISTANCE.             // Print distance from vessel to x
                                 // (same altitude is presumed)
PRINT spot:HEADING.              // Print the heading to the point
PRINT spot:BEARING.              // Print the heading to the point
                                 // relative to vessel heading

SET spot TO SHIP:GEOPOSITION.    // Make spot into a location on the
                                 // surface directly underneath the
                                 // current ship

SET spot TO LATLNG(spot:LAT,spot:LNG+5). // Make spot into a new
                                         // location 5  degrees east
                                         // of the old one

// Point nose of ship at a spot 100,000 meters altitude above a
// particular known latitude of 50 east, 20.2 north:
LOCK STEERING TO LATLNG(50,20.2):ALTITUDEPOSITION(100000).

// A nice complex example:
// -----------------
// Drawing an debug arrow in 3D space at the spot where the Geocoordinate 'spot' is:
// It starts at a position 100m above the ground altitude and is aimed down at
// the spot on the ground:
SET VD TO VECDRAWARGS(
              spot:ALTITUDEPOSITION(spot:TERRAINHEIGHT+100),
              spot:POSITION - spot:ALTITUDEPOSITION(TERRAINHEIGHT+100),
              red, "THIS IS THE SPOT", 1, true).

PRINT "THESE TWO NUMBERS SHOULD BE THE SAME:".
PRINT (SHIP:ALTITIUDE - SHIP:GEOPOSITION:TERRAINHEIGHT).
PRINT ALT:RADAR.
```

6.6 Reference Frames

This page describes the (x, y, z) reference frame used for most of **kOS**'s vectors. kOS inherits its reference frame mostly from the base Kerbal Space Program game itself. The coordinate system of Kerbal Space Program does

some strange things that don't make a lot of sense at first. For nomenclature, the following terms are used in this documentation:

Note: Be aware that Kerbal Space program (and in fact many of the games based on the Unity game engine) uses a **LEFT-handed** coordinate system. kOS inherits this behavior from KSP.

In all the reference frames mentioned below, the orientation of the axes is **left-handed**. This means that if you imagine opening your palm and pointing your fingers down the x-axis, then curling your fingers in the direction of the y-axis, then sticking your thumb up, that the direction your thumb would have to be pointing to accomplish this task is the direction of the z-axis **if and only if** you used your left hand to perform those steps. (If you do those steps with your right hand, you get a z-axis in the opposite direction and that is known as a **right handed** coordinate system).

This is an important thing to keep in mind, as most mathematics and physics textbooks tend to draw examples using a right handed coordinate system, and most students become familiar with that convention first. But for a variety of reasons, many computer graphics systems have a tradition of using left-handed systems instead, and Kerbal Space Program is one of them.

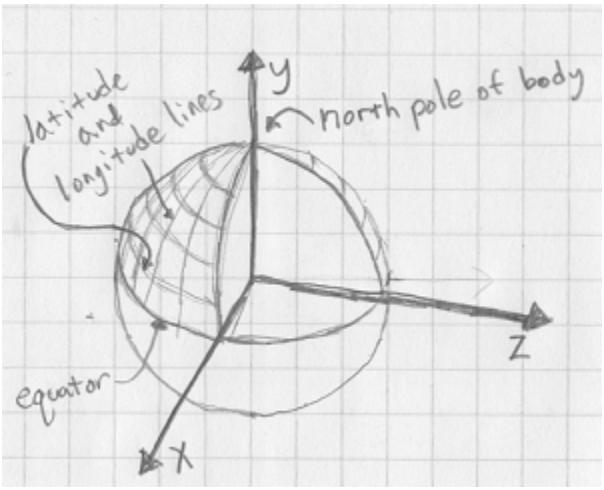
`SHIP-RAW`
: The name of the reference frame in which the origin point is *CPU Vessel (SHIP)*, and the rotation is identical to **KSP**'s native raw coordinate grid.

`SOI-RAW`
: The name of the reference frame in which the origin point is the center of the *SOI body*, and the rotation is identical to **KSP**'s native raw coordinate grid.

`RAW-RAW`
: The name of the reference frame in which both the origin point and the rotation of the axes is identical to **KSP**'s native raw coordinate grid. This is never exposed to the **KerbalScript** program, because the origin point is meaningless to work with.

Note: It is hoped that this may be expanded in the future, and conversion routines provided to let people pick a reference frame that makes sense depending on what the script is trying to do. At the moment the only reference frames used are SHIP-RAW and SOI-RAW, as they match closely to what **KSP** is using internally.

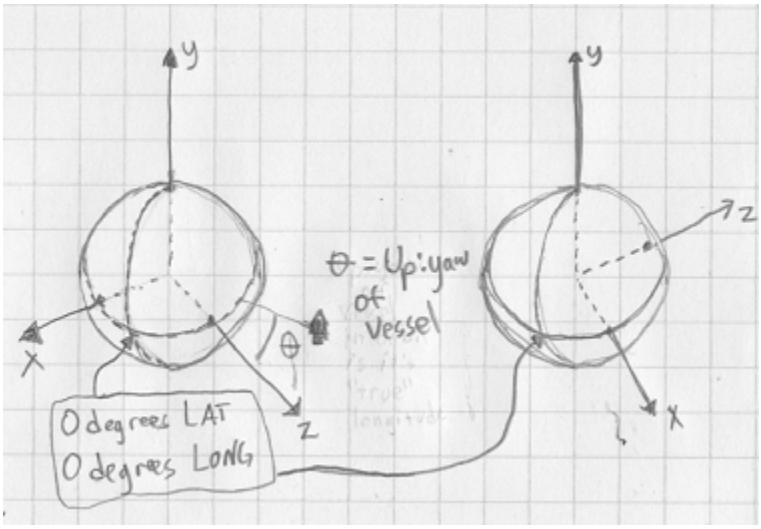
6.6.1 Raw Orientation



The Y axis of **KSP** is the only consistent thing. Imagine a ray starting in the center of the SOI body and pointing upward out through the north pole. That is the direction of the Y axis. (If you move to the SOI of a body with an

inclined spin, presumably it will also change the angle of the Y axis to point in the new direction of the body's spin axis).

The X and Z axes of the coordinate grid are then consequently aligned with the equator plane of the SOI body, 90 degrees to each other. **KSP** uses a left-handed coordinate system, so the Z axis will always be rotated 90 degrees to the east of the X axis.



However, the X and Z axes are hard to predict where they'll exactly be. They keep moving depending on where you are, to the point where it's impossible to get a fix on just which direction they'll point.

6.6.2 Origin Position

The origin position of the (x, y, z) coordinate grid in **KSP** is also a bit messy. It's usually *near* but not exactly *on* the current ship. **kOS** performs some conversions for you to make this a bit simpler and keep everything consistent.

Regardless of where the origin of the underlying **KSP** system is, in **kOS**, whenever a POSITION is reported, it will always be reported in a frame of reference where the origin is located at the *CPU Vessel*.

However, for the sake of VELOCITY, the origin point of all vectors is usually not SHIP, but rather it's the SOI body's center. This is because if the origin point was the SHIP, then the ship's velocity would always be zero in that frame of reference, and that would not be useful.

The makers of **kOS** are aware that this is not technically a proper frame of reference, because the origin point varies depending on if you're getting POSITION or getting VELOCITY. Fixing it at this point would break a lot of existing scripts, however.

So the rule of thumb is:

- For POSITION returned by **KSP**, the SHIP-RAW reference frame is used: centered on SHIP, with raw axes rotation.
- For VELOCITY returned by **KSP**, the SOI-RAW reference frame is used: centered on SOI Body, with raw axes rotation.

6.6.3 Converting

Converting between SHIP-RAW and SOI-RAW reference frames is a simple matter of adding or subtracting the SHIP:BODY:POSITION vector from the coordinate, to move the origin point. This is because both are using the same

axes rotation.

- Any SHIP-RAW vector Minus SHIP:BODY:POSITION Gives the vector in SOI-RAW coordinates.
- Any SOI-RAW vector Plus SHIP:BODY:POSITION Gives the vector in SHIP-RAW coordinates.

CHAPTER

SEVEN

COMMAND REFERENCE

7.1 Flight Control

7.1.1 Cooked Control

In this style of controlling the craft, you do not steer the craft directly, but instead select a goal direction and let kOS pick the way to steer toward that goal. This method of controlling the craft consists primarily of the following two commands:

LOCK THROTTLE TO value.
This sets the main throttle of the ship to *value*. Where *value* is a floating point number between 0.0 and 1.0. A value of 0.0 means the throttle is idle, and a value of 1.0 means the throttle is at maximum. A value of 0.5 means the throttle is at the halfway point, and so on.

LOCK STEERING TO value.
This sets the direction **kOS** should point the ship where *value* is a *Vector* or a *Direction* created from a *Rotation* or *Heading*:

Rotation

A Rotation expressed as `R(pitch,yaw,roll)`. Note that pitch, yaw and roll are not based on the horizon, but based on an internal coordinate system used by **KSP** that is hard to use. Thankfully, you can force the rotation into a sensible frame of reference by adding a rotation to a known direction first.

To select a direction that is 20 degrees off from straight up:

```
LOCK STEERING TO Up + R(20,0,0).
```

To select a direction that is due east, aimed at the horizon:

```
LOCK STEERING TO North + R(0,90,0).
```

`UP` and `NORTH` are the only two predefined rotations.

Heading

A heading expressed as `HEADING(compass, pitch)`. This will aim 30 degrees above the horizon, due south:

```
LOCK STEERING TO HEADING(180, 30).
```

Vector

Any vector can also be used to lock steering:

```
LOCK STEERING TO V(100,50,10).
```

Note that the internal coordinate system for `(X,Y,Z)` is quite complex to explain. To aim in the opposite of the surface velocity direction:

```
LOCK STEERING TO (-1) * SHIP:VELOCITY:SURFACE.
```

The following aims at a vector which is the cross product of velocity and direction down to the SOI planet - in other words, it aims at the "normal" direction to the orbit:

```
LOCK STEERING TO VCRS(SHIP:VELOCITY:ORBIT, BODY:POSITION).
```

Like all `LOCK` expressions, the steering and throttle continually update on their own when using this style of control. If you lock your steering to velocity, then as your velocity changes, your steering will change to match it. Unlike with other `LOCK` expressions, the steering and throttle are special in that the lock expression gets executed automatically all the time in the background, while other `LOCK` expressions only get executed when you try to read the value of the variable. The reason is that the **kOS** computer is constantly querying the lock expression multiple times per second as it adjusts the steering and throttle in the background.

Unlocking controls

If you `LOCK` the `THROTTLE` or `STEERING`, be aware that this prevents the user from manually controlling them. Until they unlock, the manual controls are prevented from working. You can free up the controls by issuing these two commands:

```
UNLOCK STEERING.
UNLOCK THROTTLE.
```

When the program ends, these automatically unlock as well, which means that to control a craft you must make sure the program doesn't end. The moment it ends it lets go of the controls.

Advantages/Disadvantages

The advantge of "Cooked" control is that it is simpler to write scripts for, but the disadvantage is that you have no control over the details of the motion. You can't dictate how fast or slow the craft rotates, or which axis it tries to rotate around first, and if your craft is wobbly, you can't dampen the wobbliness.

7.1.2 Raw Control

If you wish to have your kOS script manipulate a vessel's flight controls directly in a raw way, rather than relying on kOS to handle the flying for you, then this is the type of structure you will need to use to do it. This is offered as an alternative to using the combination of `LOCK STEERING` and `LOCK THROTTLE` commands. To obtain the CONTROL variable for a vessel, use its :CONTROL suffix:

```
SET controlStick to SHIP:CONTROL.
SET controlStick:PITCH to 0.2.
```

Unlike with so-called "Cooked" steering, "raw" steering uses the `SET` command, not the `LOCK` command. Using `LOCK` with these controls won't work. When controlling the ship in a raw way, you must decide how to move the controls in detail. Here is another example:

```
SET SHIP:CONTROL:YAW to 0.2.
```

This will start pushing the ship to rotate a bit faster to the right, like pushing the D key gently. All the following values are set between -1 and $+1$. Zero means the control is neutral. You can set to values smaller in magnitude than -1 and $+1$ for gentler control:

```
print "Gently pushing forward for 3 seconds.".
SET SHIP:CONTROL:FORE TO 0.2.
SET now to time:seconds.
WAIT until time:seconds > now + 3.
SET SHIP:CONTROL:FORE to 0.0.

print "Gently Pushing leftward for 3 seconds.".
SET SHIP:CONTROL:STARBOARD TO -0.2.
SET now to time:seconds.
WAIT until time:seconds > now + 3.
SET SHIP:CONTROL:STARBOARD to 0.0.

print "Starting an upward rotation.".
SET SHIP:CONTROL:PITCH TO 0.2.
SET now to time:seconds.
WAIT until time:seconds > now + 0.5.
SET SHIP:CONTROL:PITCH to 0.0.

print "Giving control back to the player now.".
SET SHIP:CONTROL:NEUTRALIZE to True.
```

One can use *SHIP:CONTROL:ROTATION* and *SHIP:CONTROL:TRANSLATION* to see the ship's current situation.

Raw Flight Controls Reference

These "Raw" controls allow you the direct control of flight parameters while the current program is running.

Note: The MAINTHROTTLE requires active engines and, of course, sufficient and appropriate fuel. The rotational controls YAW, PITCH and ROW require active reaction wheels with sufficient energy or *RCS* to be ON with properly placed thrusters and appropriate fuel. The translational controls FORE, STARBOARD and TOP require *RCS* to be ON with properly placed thrusters and appropriate fuel.

Suffix	Type, Range	Equivalent Key
MAINTHROTTLE	scalar [0,1]	LEFT-CTRL, LEFT-SHIFT
YAW	scalar [-1,1]	D, A
PITCH	scalar [-1,1]	W, S
ROLL	scalar [-1,1]	Q, E
ROTATION	*Vector*	(YAW,PITCH,ROLL)
YAWTRIM	scalar [-1,1]	ALT+D, ALT+A
PITCHTRIM	scalar [-1,1]	ALT+W, ALT+S
ROLLTRIM	scalar [-1,1]	ALT+Q, ALT+E
FORE	scalar [-1,1]	N, H
STARBOARD	scalar [-1,1]	L, J
TOP	scalar [-1,1]	I, K
TRANSLATION	*Vector*	(STARBOARD,TOP,FORE)
WHEELSTEER	scalar [-1,1]	A, D
WHEELTHROTTLE	scalar [-1,1]	W, S
WHEELSTEERTRIM	scalar [-1,1]	ALT+A, ALT+D
WHEELTHROTTLETRIM	scalar [-1,1]	ALT+W, ALT+S
NEUTRAL	boolean	Is **kOS** Controlling?
NEUTRALIZE	boolean	Releases Control

SHIP:CONTROL:MAINTHROTTLE
Set between 0 and 1 much like the cooked flying `LOCK THROTTLE` command.

SHIP:CONTROL:YAW
This is the rotation about the "up" vector as the pilot faces forward. Essentially left (-1) or right $(+1)$.

SHIP:CONTROL:PITCH
Rotation about the starboard vector up $(+1)$ or down (-1).

SHIP:CONTROL:ROLL
Rotation about the logintudinal axis of the ship left-wing-down (-1) or left-wing-up $(+1)$.

SHIP:CONTROL:ROTATION
This is a *Vector* object containing `(YAW, PITCH, ROLL)` in that order.

SHIP:CONTROL:YAWTRIM
Controls the `YAW` of the rotational trim.

SHIP:CONTROL:PITCHTRIM
Controls the `PITCH` of the rotational trim.

SHIP:CONTROL:ROLLTRIM
Controls the `ROLL` of the rotational trim.

SHIP:CONTROL:FORE
Controls the translation of the ship forward $(+1)$ or backward (-1).

SHIP:CONTROL:STARBOARD
Controls the translation of the ship to the right $(+1)$ or left (-1) from the pilot's perspective.

SHIP:CONTROL:TOP
Controls the translation of the ship up $(+1)$ or down (-1) from the pilot's perspective.

SHIP:CONTROL:TRANSLATION
Controls the translation as a *Vector* `(STARBOARD, TOP, FORE)`.

SHIP:CONTROL:WHEELSTEER
Turns the wheels left (-1) or right $(+1)$.

SHIP:CONTROL:WHEELTHROTTLE
Controls the wheels to move the ship forward $(+1)$ or backward (-1) while on the ground.

SHIP:CONTROL:WHEELSTEERTRIM
Controls the trim of the wheel steering.

SHIP:CONTROL:WHEELTHROTTLETRIM
Controls the trim of the wheel throttle.

SHIP:CONTROL:NEUTRAL
Returns true or false depending if **kOS** has any set controls. *This is not settable.*

SHIP:CONTROL:NEUTRALIZE
This causes manual control to let go. When set to true, **kOS** lets go of the controls and allows the player to manually control them again. *This is not gettable.*

Unlocking controls

Setting any one of `SHIP:CONTROL` values will prevent player from manipulating that specific control manually. Other controls will not be locked. To free any single control, set it back to zero. To give all controls back to the player you must execute:

```
SET SHIP:CONTROL:NEUTRALIZE to TRUE.
```

Advantages/Disadvantages

The control over *RCS* translation requires the use of Raw control. Also, with raw control you can choose how gentle to be with the controls and it can be possible to control wobbly craft better with raw control than with cooked control.

7.1.3 Pilot Input

This is not, strictly speaking, a method of controlling the craft. "Pilot" controls are a way to read the input from the pilot. Most of these controls share the same name as their flight control, prefixed with `PILOT` (eg `YAW` and `PILOTYAW`) the one exception to this is the `PILOTMAINTHROTTLE`. This suffix has a setter and allows you to change the behavior of the throttle that persists even after the current program ends:

```
SET SHIP:CONTROL:PILOTMAINTHROTTLE TO 0.
```

Will ensure that the throttle will be 0 when execution stops. These suffixes allow you to read the input given to the system by the user.

structure `Control`

Suffix	Type, Range	Equivalent Key
PILOTMAINTHROTTLE	scalar [0,1]	`LEFT-CTRL, LEFT-SHIFT`
PILOTYAW	scalar [-1,1]	`D, A`
PILOTPITCH	scalar [-1,1]	`W, S`
PILOTROLL	scalar [-1,1]	`Q, E`
PILOTROTATION	*Vector*	`(YAW,PITCH,ROLL)`
PILOTYAWTRIM	scalar [-1,1]	`ALT+D, ALT+A`
PILOTPITCHTRIM	scalar [-1,1]	`ALT+W, ALT+S`
PILOTROLLTRIM	scalar [-1,1]	`ALT+Q, ALT+E`
PILOTFORE	scalar [-1,1]	`N, H`
PILOTSTARBOARD	scalar [-1,1]	`L, J`
PILOTTOP	scalar [-1,1]	`I, K`
PILOTTRANSLATION	*Vector*	`(STARBOARD,TOP,FORE)`
PILOTWHEELSTEER	scalar [-1,1]	`A, D`
PILOTWHEELTHROTTLE	scalar [-1,1]	`W, S`
PILOTWHEELSTEERTRIM	scalar [-1,1]	`ALT+A, ALT+D`
PILOTWHEELTHROTTLETRIM	scalar [-1,1]	`ALT+W, ALT+S`
PILOTNEUTRAL	boolean	Is **kOS** Controlling?

`SHIP:CONTROL:PILOTMAINTHROTTLE`
Returns the pilot's input for the throttle. This is the only `PILOT` variable that is settable and is used to set the throttle upon termination of the current **kOS** program.

`SHIP:CONTROL:PILOTYAW`
Returns the pilot's rotation input about the "up" vector as the pilot faces forward. Essentially left (-1) or right $(+1)$.

`SHIP:CONTROL:PILOTPITCH`
Returns the pilot's rotation input about the starboard vector up $(+1)$ or down (-1).

`SHIP:CONTROL:PILOTROLL`
Returns the pilot's rotation input about the logintudinal axis of the ship left-wing-down (-1) or left-wing-up $(+1)$.

SHIP:CONTROL:PILOTROTATION
Returns the pilot's rotation input as a *Vector* object containing (YAW, PITCH, ROLL) in that order.

SHIP:CONTROL:PILOTYAWTRIM
Returns the pilot's input for the YAW of the rotational trim.

SHIP:CONTROL:PILOTPITCHTRIM
Returns the pilot's input for the PITCH of the rotational trim.

SHIP:CONTROL:PILOTROLLTRIM
Returns the pilot's input for the ROLL of the rotational trim.

SHIP:CONTROL:PILOTFORE
Returns the the pilot's input for the translation of the ship forward $(+1)$ or backward (-1).

SHIP:CONTROL:PILOTSTARBOARD
Returns the the pilot's input for the translation of the ship to the right $(+1)$ or left (-1) from the pilot's perspective.

SHIP:CONTROL:PILOTTOP
Returns the the pilot's input for the translation of the ship up $(+1)$ or down (-1) from the pilot's perspective.

SHIP:CONTROL:PILOTTRANSLATION
Returns the the pilot's input for translation as a *Vector* (STARBOARD, TOP, FORE).

SHIP:CONTROL:PILOTWHEELSTEER
Returns the the pilot's input for wheel steering left (-1) or right $(+1)$.

SHIP:CONTROL:PILOTWHEELTHROTTLE
Returns the the pilot's input for the wheels to move the ship forward $(+1)$ or backward (-1) while on the ground.

SHIP:CONTROL:PILOTWHEELSTEERTRIM
Returns the the pilot's input for the trim of the wheel steering.

SHIP:CONTROL:PILOTWHEELTHROTTLETRIM
Returns the the pilot's input for the trim of the wheel throttle.

SHIP:CONTROL:PILOTNEUTRAL
Returns true or false if the pilot is active or not.

Be aware that **kOS** can't control a control at the same time that a player controls it. If **kOS** is taking control of the yoke, then the player can't manually control it. Remember to run:

```
SET SHIP:CONTROL:NEUTRALIZE TO TRUE.
```

after the script is done using the controls, or the player will be locked out of control.

7.1.4 Ship Systems

e.g.:

```
set somepart to ship:partstagged("my favorite docking port")[0].
somepart:CONTROLFROM().
```

If you have a handle on a part, from LIST PARTS, you can select that part to set the orientation of the craft, just like using the "control from here" in the right-click menu in the game. For more information see *Part:CONTROLFROM*. All vessels must have at least one "control from" part on them somewhere, which is why there's no mechanism for un-setting the "control from" setting other than to pick another part and set it to that part instead.

RCS

Access Toggle ON/OFF

Turns the RCS **on** or **off**, like using `R` at the keyboard:

```
RCS ON.
```

SAS

Access Toggle ON/OFF

Turns the SAS **on** or **off**, like using `T` at the keybaord:

```
SAS ON.
```

SET SASMODE TO value.

Access Get/Set

Type string

Getting this variable will return the currently selected mode. Where `value` is one of the valid strings listed below, this will set the stock SAS mode for the cpu vessel:

```
SET SASMODE TO value.
```

It is the equivalent to clicking on the buttons next to the nav ball while manually piloting the craft, and will respect the current mode of the nav ball (orbital, surface, or target velocity). Valid strings for `value` are `"PROGRADE"`, `"RETROGRADE"`, `"NORMAL"`, `"ANTINORMAL"`, `"RADIALOUT"`, `"RADIALIN"`, `"TARGET"`, `"ANTITARGET"`, `MANEUVER`, `"STABILITYASSIST"`, and `"STABILITY"`. A null or empty string will default to stability assist mode, however any other invalid string will throw an exception. This feature will respect career mode limitations, and will throw an exception if the current vessel is not able to use the mode passed to the command. An exception is also thrown if `"TARGET"` or `"ANTITARGET"` are used, but no target is selected.

Warning: SASMODE does not work with RemoteTech
Due to the way that RemoteTech disables flight control input, the built in SAS modes do not function properly when there is no connection to the KSC or a Command Center. If you are writing scripts for use with RemoteTech, make sure to take this into account.

LIGHTS

Access Toggle ON/OFF

Turns the lights **on** or **off**, like using the `U` key at the keyboard:

```
LIGHTS ON.
```

BRAKES

Access Toggle ON/OFF

Turns the brakes **on** or **off**, like clicking the brakes button, though *not* like using the `B` key, because they stay on:

```
BRAKES ON.
```

TARGET

Access Get/Set

Type string

Where `name` is the name of a target vessel or planet, this will set the current target:

```
SET TARGET TO name.
```

Note that the above options also can refer to a different vessel besides the current ship, for example, `TARGET:THROTTLE` to read the target's throttle. But not all "set" or "lock" options will work with a different vessel other than the current one, because there's no authority to control a craft the current program is not attached to.

7.1.5 Time Warping

WARP

You may use the WARPTO(time) function to automatically warp to the specified time (given in seconds game universal time). If you need more precise control, you can use the other options below.:

```
WARPTO(TIME:SECONDS + 60 * 10). // warp to a time 10 minutes in the future
```

The *WARP* global variable can be set to change the game warp to a value between 0 and 7 (for rails warp) or 0 to 3 (for physics warp):

```
SET WARP TO 5. // Sets warp to 1000x
SET WARP TO 0. // Sets warp to 1x (real time)
```

You may also choose which warp mode you wish the WARP command to invoke- physics warp (capped at 4x) or rails warp:

```
SET WARPMODE TO "PHYSICS".
SET WARPMODE TO "RAILS".
```

WARPMODE can be set to a string to choose the warp. It must be one of the two strings mentioned above.

The difference is the same as that experienced in the game between physics warp and 'rails' warp (sometimes called 'time warp') although that term is confusingly ambiguous.

Table 7.1: RAILS WARP MODES

MODE	MEANING
0	1x
1	5x
2	10x
3	50x
4	100x
5	1000x
6	10000x
7	100000x

Table 7.2: PHYSICS WARP MODES

MODE	MEANING
0	1x
1	2x
2	3x
3	4x

Unless otherwise stated, all controls that a **kOS** CPU attempts will be done on the *CPU Vessel*. There are three styles of control:

Cooked Give a goal direction to seek, and let **kOS** find the way to maneuver toward it.

Raw Control the craft just like a manual pilot would do from a keyboard or joystick.

Pilot This is the stock way of controlling craft, the state of which can be read in **KerboScript**.

Warning: SAS OVERRIDES kOS
With the current implementation of flight control, you may now leave `SAS` turned on in `"STABILITYASSIST"` mode, and it will not override **kOS**'s attempts to steer the ship. However, it will fight and/or override **kOS**'s attempts to steer when using any other mode. In order for **kOS** to be able to turn the ship in other modes, you need to set `SAS OFF` or `SET SASMODE TO "STABILITYASSIST"`. You should take care in your scripts to manage the use of `SAS` and `SASMODE` appropriately. It is common for people writing **kOS** scripts to explicitly start them with a use of the `SAS OFF` and/or `SET SASMODE TO "STABILITYASSIST"` commands just in case you forgot to turn it off before running the script. You could also store the current state in a temporary variable, and re-set it at the conclusion of your script.

7.2 Predictions of Flight Path

Note: Manipulating the maneuver nodes

To alter the maneuver nodes on a vessel's flight plan, use the ADD and REMOVE commands as described on the *maneuver node manipulation page*.

Using the Add and Remove commands as described on that page, you may alter the flight plan of the CPU_vessel, however kOS does not automatically execute the nodes. You still have to write the code to decide how to successfully execute a planned maneuver node.

Warning: Be aware that a limitation of KSP makes it so that some vessels' manuever node systems cannot be accessed. KSP appears to limit the maneuver node system to only functioning on the current PLAYER vessel, under the presumption that its the only vessel that needs them, as ever other vessel cannot be manuevered. kOS can manuever a vessel that is not the player vessel, but it cannot overcome this limitation of the base game that unloads the maneuver node system for other vessels.
Be aware that the effect this has is that when you try to predict another vessel's position, it will sometimes give you answers that presume that other vessel will be purely drifting, and not following its maneuver nodes.

The following prediction functions do take into account the future maneuver nodes planned, and operate under the assumption that they will be executed as planned.

These return predicted information about the future position and velocity of an object.

POSITIONAT(orbitable,time)

Parameters

- **orbitable** – A *Vessel*, *Body* or other *Orbitable* object
- **time** – Time of prediction

Type orbitable *Orbitable*

Type time *TimeSpan*

Returns A position *Vector* expressed as the coordinates in the *ship-center-raw-rotation* frame

Returns a prediction of where the *Orbitable* will be at some *universal Timestamp*. If the *Orbitable* is a *Vessel*, and the *Vessel* has planned *maneuver nodes*, the prediction assumes they will be executed exactly as planned.

VELOCITYAT(orbitable,time)

Parameters

- **orbitable** – A *Vessel*, *Body* or other *Orbitable* object
- **time** – Time of prediction

Type orbitable *Orbitable*

Type time *TimeSpan*

Returns An *ObitalVelocity* structure.

Returns a prediction of what the *Orbitable's* velocity will be at some *universal Timestamp*. If the *Orbitable* is a *Vessel*, and the *Vessel* has planned `maneuver nodes`, the prediction assumes they will be executed exactly as planned.

ORBITAT(orbitable,time)

Parameters

- **orbitable** – A *Vessel*, *Body* or other *Orbitable* object
- **time** – Time of prediction

Type orbitable *Orbitable*

Type time *TimeSpan*

Returns An *Orbit* structure.

Returns the *Orbit patch* where the *Orbitable* object is predicted to be at some *universal Timestamp*. If the *Orbitable* is a *Vessel*, and the *Vessel* has planned *maneuver nodes*, the prediction assumes they will be executed exactly as planned.

Examples:

```
//kOS
// test the future position and velocity prediction.
// Draws a position and velocity vector at a future predicted time.

declare parameter item. // thing to predict for, i.e. SHIP.
declare parameter offset. // how much time into the future to predict.
declare parameter velScale. // how big to draw the velocity vectors.
              // If they're far from the camera you should draw them bigger.


set predictUT to time + offset.
set stopProg to false.

set futurePos to positionat( item, predictUT ).
set futureVel to velocityat( item, predictUT ).

set drawPos to vecdrawargs( v(0,0,0), futurePos, green, "future position", 1, true ).
set drawVel to vecdrawargs( futurePos, velScale*futureVel:orbit, yellow, "future velocity", 1, true )
```

Example Screenshot:

7.3 LIST Command

A *List* is a type of *Structure* that stores a list of variables in it. The `LIST` command either prints or crates a *List* object containing items queried from the game. For more information, see the *page about the List structure*.

7.3.1 FOR Loop

Lists need to be iterated over sometimes, to help with this we have the *FOR loop, explained on the flow control page*. The LIST Command comes in 4 forms:

1. **LIST.** When no parameters are given, the LIST command is exactly equivalent to the command:

```
LIST FILES.
```

2. **LIST ListKeyword.** This variant prints items to the termianl sceen. Depending on the *ListKeyword* used (see below), different values are printed.
3. **LIST ListKeyword IN YourVariable.** This variant takes the items that would otherwise have been printed to the terminal screen, and instead makes a *List* of them in YourVariable, that you can then iterate over with a *FOR loop* if you like.
4. **LIST ListKeyword FROM SomeVessel IN YourVariable.** This variant is just like variant (3), except that it gives a list of the items that exist on some other vessel that might not necessarily be the current *CPU_vessel*.

7.3.2 Available Listable Keywords

The *ListKeyword* in the above command variants can be any of the following:

Universal Lists

These generate *lists* that are not dependent on which *Vessel*:

Bodies *List* of *Celestial Bodies*

Targets *List* of possible target *Vessels*

Vessel Lists

These generate *lists* of items on the *Vessel*:

Resources *List* of *AggregateResources*

Parts *List* of *Parts*

Engines *List* of *Engines*

Sensors *List* of *Sensors*

Elements *List* of *Elements* that comprise the current vessel.

DockingPorts list of *DockingPorts* <*DockingPort*>

File System Lists

These generate *lists* about the files in the system:

Files *List* the *files* on the current Volume. (note below)

Volumes *List* all the *Files and Volumes* that exist.

Note: `LIST FILES.` is the default if you give the `LIST` command no parameters.

Examples:

```
LIST.  // Prints the list of files on current volume.
LIST FILES.  // Does the same exact thing, but more explicitly.
LIST VOLUMES. // which volumes can be seen by this CPU?
LIST FILES IN fileList. // fileList is now a LIST() containing file structures.
```

The file structures returned by `LIST FILES IN fileList.` are documented *on a separate page*.

Here are some more examples:

```
// Prints the list of all
// Celestial bodies in the system.
LIST BODIES.

// Puts the list of bodies into a variable.
LIST BODIES IN bodList.
// Iterate over everything in the list:
SET totMass to 0.
FOR bod in bodList {
    SET totMass to totMass + bod:MASS.
}.
PRINT "The mass of the whole solar system is " + totMass.

// Adds variable foo that contains a list of
// resources for my currently target vessel
LIST RESOURCES FROM TARGET IN foo.
FOR res IN foo {
    PRINT res:NAME. // Will print the name of every
                    // resource in the vessel
}.
```

7.4 Querying a vessel's parts

This is a quick list to get the idea across fast. The actual details of the meaning of these things is complex enough to warrant its own topic.

To get the parts of a vessel (such as your current vessel, called SHIP), you can do the following things:

These are equivalent. They get the full list of all the parts:

```
LIST PARTS IN MyPartList.
SET MyPartlist TO SHIP:PARTS.
```

This gets all the parts that have the name given, as either a nametag (Part:TAG), a title (Part:TITLE), or a name, (Part:NAME):

```
SET MyPartList to SHIP:PARTSDUBBED("something").
```

These are other ways to get parts that are more specific about what exact nomenclature system is being used:

```
SET MyPartList to SHIP:PARTSTAGGED("something"). // only gets parts with Part:TAG = "something".
SET MyPartList to SHIP:PARTSTITLED("something"). // only gets parts with Part:TITLE = "something".
SET MyPartList to SHIP:PARTSNAMED("something"). // only gets parts with Part:NAME = "something".
```

This gets all the PartModules on a ship that have the same module name:

```
SET MyModList to SHIP:MODULESNAMED("something").
```

This gets all the parts that have been defined to have some sort of activity occur from a particular action group:

```
SET MyPartList to SHIP:PARTSINGROUP( AG1 ). // all the parts in action group 1.
```

This gets all the modules that have been defined to have some sort of activity occur from a particular action group:

```
SET MyModList to SHIP:MODULESINGROUP( AG1 ). // all the parts in action group 1.
```

This gets the primary root part of a vessel (the command core that you placed FIRST when building the ship in the VAB or SPH):

```
SET firstPart to SHIP:ROOTPART.
```

This lets you query all the parts that are immediate children of the current part in the tree:

```
SET firstPart to SHIP:ROOTPART.
FOR P IN firstPart:CHILDREN {
  print "The root part as an immediately attached part called " + P:NAME.
}.
```

You could keep walking down the tree this way, or go upward with PARENT and HASPARENT:

TODO - **NEED TO MAKE A GOOD EXAMPLE OF WALKING THE PARTS TREE HERE WITH RECURSION ONCE THE SYNTAX IS NAILED DOWN FOR THAT.**

```
IF thisPart:HASPARENT {
  print "This part's parent part is "+ thisPart:PARENT:NAME.
}.
```

7.5 File I/O

For information about where files are kept and how to deal with volumes see the *Volumes* page in the general topics section of this documentation.

Note: All file names (program names) must be valid Identifiers. They can not contain spaces or special characters. For example, you can't have a file name called "this is my-file".

Warning: Changed in version 0.15: **Archive location and file extension change**
The Archive where KerboScript files are kept has been changed from `Plugins/PluginData/Archive` to `Ships/Script`, but still under the top-level **KSP** installation directory. The file name extensions have also changes from `.txt` to `.ks`.

7.5.1 Volume and Filename arguments

Any of the commands below which use filename arguments, **with the exception of the RUN command**, follow these rules:

- (expression filenames) A filename may be an expression which evaluates to a string.
- (bareword filenames) A filename may also be an undefined identifier which does not match a variable name, in which case the bare word name of the identifier will be used as the filename. If the identifier does match a variable name, then it will be evaluated as an expression and the variable's contents will be used as the filename.

- A bareword filename may contain file extensions with dots, provided it does not end in a dot.
- If the filename does not contain a file extension, kOS will pad it with a ".ks" extension and use that.

Putting the above rules together, you can refer to filenames in any of the following ways:

- copy myfilename to 1. // This is an example of a bareword filename.
- copy "myfilename" to 1. // This is an example of an EXPRESSION filename.
- copy myfilename.ks to 1. // This is an example of a bareword filename.
- copy myfilename.txt to 1. // This is an example of a bareword filename.
- copy "myfilename.ks" to 1. // This is an example of an EXPRESSION filename
- set str to "myfile" + "name" + ".ks". copy str to 1. // This is an example of an EXPRESSION filename

Limits:

The following rules apply as limitations to the bareword filenames:

- The **RUN command only works with bareword filenames**, not expression filenames. Every other command works with either type of filename.
- Filenames containing any characters other than A-Z, 0-9, underscore, and the period extension separator ('.'), can only be referred to using a string expression (with quotes), and cannot be used as a bareword expression (without quotes).
- If your filesystem is case-sensitive (Linux and sometimes Mac OSX, or even Windows if using some kinds of remote network drives), then bareword filenames will only work properly on filenames that are all lowercase. If you try to use a file with capital letters in the name on these systems, you will only be able to do so by quoting it.

Volumes too:

The rules for filenames also apply to volumes. You may do this for example:

- set volNum to 1. copy "myfile" to volNum.

7.5.2 `COMPILE program (TO compiledProgram).`

(experimental)

Arguments:

argument 1 Name of source file.

argument 2 Name of destination file. If the optional argument 2 is missing, it will assume it's the same as argument 1, but with a file extension changed to `*.ksm`.

Pre-compiles a script into an *Kerboscript ML Exceutable image* that can be used instead of executing the program script directly.

The RUN command (elsewhere on this page) can work with either *.ks script files or *.ksm compiled files.

The full details of this process are long and complex enough to be placed on a separate page.

Please see *the details of the Kerboscript ML Executable*.

7.5.3 COPY programFile FROM/TO voumeNumber.

Arguments

- argument 1: Name of target file.
- argument 2: Target volume.

Copies a file to or from another volume. Volumes can be referenced by their ID numbers or their names if theyve been given one. See LIST, SWITCH and RENAME.

Understanding how *volumes work* is important to understanding this command.

Example:

```
SWITCH TO 1.              // Makes volume 1 the active volume
COPY file1 FROM 0.        // Copies a file called file1.ks from volume 0 to volume 1
COPY file2 TO 0.          // Copies a file called file1.ks from volume 1 to volume 0
COPY file1.ks FROM 0.     // Copies a file called file1.ks from volume 0 to volume 1
COPY file2.ksm TO 0.      // Copies a file called file1.ksm from volume 1 to volume 0
COPY "file1.ksm" FROM 0. // Copies a file called file1.ksm from volume 0 to volume 1
COPY "file1" + "." + "ks" FROM 0. // Copies a file called file1.ks from volume 0 to volume 1
```

7.5.4 DELETE filename FROM volumeNumber.

Deletes a file. You can delete a file from the current volume, or from a named volume.

Arguments

- argument 1: Name of target file.
- argument 2: (optional) Target volume.

Example:

```
DELETE file1.          // Deletes file1.ks from the active volume.
DELETE "file1".        // Deletes file1.ks from the active volume.
DELETE file1.txt.      // Deletes file1.txt from the active volume.
DELETE "file1.txt".    // Deletes file1.txt from the active volume.
DELETE file1 FROM 1.   // Deletes file1.ks from volume 1
```

7.5.5 EDIT program.

Edits a program on the currently selected volume.

Arguments

- argument 1: Name of file for editing.

Note: The Edit feature was lost in version 0.11 but is back again after version 0.12.2 under a new guise. The new editor is unable to show a monospace font for a series of complex reasons involving how Unity works and how squad bundled the KSP game. The editor works, but will be in a proportional width font, which isn't ideal for editing code. The best way to edit code remains to use a text editor external to KSP, however for a fast peek at the code during play, this editor is useful.

Example:

```
EDIT filename.       // edits filename.ks
EDIT filename.ks.    // edits filename.ks
EDIT "filename.ks".  // edits filename.ks
EDIT "filename".     // edits filename.ks
EDIT "filename.txt". // edits filename.txt
```

7.5.6 LOG text TO filename.

Logs the selected text to a file on the local volume. Can print strings, or the result of an expression.

Arguments

- argument 1: Value you would like to log.
- argument 2: Name of file to log into.

Example:

```
LOG Hello to mylog.txt.    // logs to "mylog.txt".
LOG 4+1 to "mylog" .         // logs to "mylog.ks" because .ks is the default extension.
LOG 4 times 8 is:  + (4*8) to mylog.   // logs to mylog.ks because .ks is the default extension.
```

7.5.7 RENAME name1 TO name2.

Renames a file or volume.

Arguments

- argument 1: Volume/File Name you would like to change.
- argument 2: New name for $1.

Example:

```
RENAME VOLUME 1 TO AwesomeDisk
RENAME FILE MyFile TO AutoLaunch.
```

7.5.8 RUN <program>.

Runs the specified file as a program, optionally passing information to the program in the form of a comma-separated list of arguments in parentheses.

Arguments

- <program>: File to run.
- comma-separated-args: a list of values to pass into the program.

Example:

```
RUN AutoLaunch.ks.
RUN AutoLaunch.ksm.
RUN AutoLaunch.      // runs AutoLaunch.ksm if available, else runs AutoLaunch.ks.
RUN AutoLaunch( 75000, true, "hello" ).
RUN AutoLaunch.ks( 75000, true, "hello" ).
RUN AutoLaunch.ksm( 75000, true, "hello" ).
```

The program that is reading the arguments sees them in the variables it mentions in *DECLARE PARAMETER*.

Important exceptions to the usual filename rules for RUN

The RUN command does not allow the same sorts of generic open-ended filenames that the other file commands allow. This is very important.

RUN only works when the filename is a bareword filename. It cannot use expression filenames:

```
RUN "ProgName"   // THIS WILL FAIL.  Run needs a bareword filename.
SET ProgName to "MyProgram".
RUN ProgName     // THIS WILL FAIL also.  It will attempt to run a file
                 // called "ProgName.ksm" or "ProgName.ks", when it sees this,
                 // rather than "MyProgram".

The reasons for the exception to how filenames work for the RUN
command are too complex to go into in large detail here. Here's the
short version: While the kOS system does defer the majority of the
work
of actually compiling subprogram scripts until run-time, it still
has to
generate some header info about them at compile time, and the
filename
has to be set in stone at that time. Changing this would require a
large re-write of some of the architecture of the virtual machine.
```

7.5.9 SWITCH TO <volumeNumber>.

Switches to the specified volume. Volumes can be specified by number, or its name (if it has one). See LIST and RENAME. Understanding how *volumes work* is important to understanding this command.

Example:

```
SWITCH TO 0.                        // Switch to volume 0.
RENAME VOLUME 1 TO AwesomeDisk.     // Name volume 1 as AwesomeDisk.
SWITCH TO AwesomeDisk.              // Switch to volume 1.
PRINT VOLUME:NAME.                  // Prints "AwesomeDisk".
```

Special handling of files starting with "boot" (example `boot.ks`)

(experimental)

For users requiring even more automation, the feature of custom boot scripts was introduced. If you have at least 1 file in your Archive volume starting with "boot" (for example "boot.ks", "boot2.ks" or even "boot_custom_script.ks"), you will be presented with the option to choose one of those files as a boot script for your kOS CPU.

As soon as you vessel leaves VAB/SPH and is being initialised on the launchpad (e.g. its status is PRELAUNCH) the assigned script will be copied to CPU's local hard disk with the same name. If kOS is configured to start on the archive, the file will not be copied locally automatically. This script will be run as soon as CPU boots, e.g. as soon as

you bring your CPU in physics range or power on your CPU if it was turned off. You may get or set the name of the boot file using the *core:bootfilename* suffix.

Warning: Changed in version 0.18: **boot file name changed**
Previously boot files were copied to the local hard disk as "boot.ks". This behaviour was changed so that boot files could be handled consistently if kOS is configured to start on the Archive. Some scripts may have terminated with a generic "delete boot." line to clear the boot script. Going forward you should use the new core:bootfilename suffix when dealing the boot file.

Important things to consider:

- kOS CPU hard disk space is limited, avoid using complex boot scripts or increase disk space using MM config.
- Boot script runs immediately on initialisation, it should avoid interaction with parts/modules until physics fully load. It is best to wait for couple seconds or until certain trigger.

Possible uses for boot scripts:

- Automatically activate sleeper/background scripts which will run on CPU until triggered by certain condition.
- Create basic station-keeping scripts - you will only have to focus your probes once in a while and let the boot script do the orbit adjustment automatically.
- Create multi-CPU vessels with certain cores dedicated to specific tasks, triggered by user input or external events (Robotic-heavy Vessels)
- Anything else you can come up with

7.6 Terminal and game environment

CLEARSCREEN
Clears the screen and places the cursor at the top left:

```
CLEARSCREEN.
```

PRINT
Prints the selected text to the screen. Can print strings, or the result of an expression:

```
PRINT Hello.
PRINT 4+1.
PRINT 4 times 8 is:  + (4*8).
```

SET TERMINAL:WIDTH. GET TERMINAL:WIDTH
Gets or sets the terminal's width in characters. For more information see *terminal struct*.

SET TERMINAL:HEIGHT. GET TERMINAL:HEIGHT
Gets or sets the terminal's height in characters. For more information see *terminal struct*.

AT (col,line)

Parameters

- **col** – (integer) column starting with zero (left)
- **line** – (integer) line starting with zero (top)

Used in combination with *PRINT*. Prints the selected text to the screen at specified location. Can print strings, or the result of an expression:

```
PRINT Hello AT(0,10).
PRINT 4+1 AT(0,10).
PRINT 4 times 8 is:  + (4*8) AT(0,10).
```

MAPVIEW

Access Get/Set

Type boolean

A variable that controls or queries whether or not the game is in map view:

```
IF MAPVIEW {
    PRINT "You are looking at the map.".
} ELSE {
    PRINT "You are looking at the flight view.".
}.
```

You can switch between map and flight views by setting this variable:

```
SET MAPVIEW TO TRUE.  // to map view
SET MAPVIEW TO FALSE. // to flight view
```

REBOOT

Reboots the kOS module.

SHUTDOWN

Causes kOS module to shutdown.

7.6.1 GUI display tools

VECDRAW

See VECDRAWARGS, below

VECDRAWARGS

You can **draw visual vectors on the screen** in kOS to help debugging or to help show the player information. The full description can be found on the Vecdraw Page.

HUDTEXT

You can make text messages appear on the heads-up display, in the same way that the in-game stock messages appear, by calling the HUDTEXT function, as follows:

```
HUDTEXT( string Message,
         integer delaySeconds,
         integer style,
         integer size,
         RGBA colour,
         boolean doEcho).
```

Message The message to show to the user on screen

delaySeconds How long to make the message remain onscreen before it goes away. If another message is drawn while an old message is still displaying, both messages remain, the new message scrolls up the old message.

style Where to show the message on the screen: - 1 = upper left - 2 = upper center - 3 = lower right - 4 = lower center Note that all these locations have their own defined slightly different fonts and default sizes, enforced by the stock KSP game.

size A number describing the font point size: NOTE that the actual size varies depending on which of the above styles you're using. Some of the locations have a magnifying factor attached to their fonts.

colour The colour to show the text in, using one of the built-in colour names or the RGB constructor to make one up

doEcho If true, then the message is also echoed to the terminal as "HUD: message".

Examples:

```
HUDTEXT("Warning: Vertical Speed too High", 5, 2, 15, red, false).
HUDTEXT("docking mode begun", 8, 1, 12, rgb(1,1,0.5), false).
```

7.7 Transferring resources

Resource transfer is an important part of many missions in Kerbal Space Program, whether you are gassing up a lander or refuelling a station. Because of this, we need a way to describe a resource transfer in script and monitor that transfer to be sure it was successful.

To accomplish all of this we have two new functions

```
SET transferFoo TO TRANSFER(resourceName, fromParts, toParts, amount).
SET transferBar TO TRANSFERALL(resourceName, fromParts, toParts).
```

TRANSFER will move the specified amount of a resource, where TRANSFERALL will move the resource until the source is empty or the destination is full. Both functions return a new *ResourceTransfer*.

7.7.1 Resource

in all transfers, you must specify a resource by name (eg "oxidizer", "LIQUIDFUEL", etc). this term is not case sensitive.

7.7.2 Source and Destination

In all transfers, you must specify a source and a destination, both of these need to be one of these three types:

- *Part*
- *List* of *Part*
- *Element*

7.7.3 Examples

Building a list of *Element*, then transferring all of the oxidizer from one element to another:

```
LIST ELEMENTS IN elist.
SET foo TO TRANSFERALL("OXIDIZER", elist[0], elist[1]).
SET foo:ACTIVE to TRUE.
```

Finding two parts by index, then transferring all of the oxidizer from part 1 to part 2:

```
SET foo TO TRANSFERALL("OXIDIZER", SHIP:PARTS[0], SHIP:PARTS[1]).
SET foo:ACTIVE to TRUE.
```

Finding two lists of parts by tag, then transferring 100 units of liquidfuel:

```
SET sourceParts to SHIP:PARTSDUBBED("fooPart").
SET destinationParts to SHIP:PARTSDUBBED("barPart").
SET foo TO TRANSFER("liquidfuel", sourceParts, destinationParts, 100).
SET foo:ACTIVE to TRUE.
```

CHAPTER

EIGHT

STRUCTURE REFERENCE

A general discussion of structures *can be found here*.

- ***Mathematics***
  - *Fundamental Constants*
  - *Mathematical Functions*
  - *Trigonometric Functions*
  - `Vector`
  - `Direction`
  - `GeoCoordinates`
- `Waypoint`
- ***Orbits***
  - `Orbit`
  - `Orbitable`
  - `OrbitableVelocity`
- ***Celestial Bodies***
  - `Body`
  - `Atmosphere`
- ***Vessels***
  - `Vessel`
  - `Control`
  - `ManeuverNode`
  - `Engine`
  - `AggregateResource`
  - `DockingPort`
  - `Gimbal`
  - `Stage`
  - `Part`
  - `PartModule`

8.1 Orbits

8.1.1 Orbit

Contents

Variables of type *Orbit* hold descriptive information about the elliptical shape of a predicted orbit. Whenever there are multiple patches of orbit ellipses strung together, for example, when an encounter with a body is expected to alter the path, or when a maneuver node is planned, then each individual patch of the path is represented by one *Orbit* object.

Each *Orbitable* item such as a *Vessel* or celestial *Body* has an `:ORBIT` suffix that can be used to obtain its current *Orbit*.

Whenever you get the *Orbit* of a *Vessel*, be aware that its just the current *Orbit* patch that doesn't take into account any planetary encounters (slingshots) or maneuver nodes that may occur. For example, your vessel might never reach `SHIP:ORBIT:APOAPSIS` if you're going to intersect the Mun and be flung by it into a new orbit.

Warning: Some of the parameters listed here come directly from KSP's API and there is a bit of inconsistency with whether it uses radians or degrees for angles. As much as possible we have tried to present everything in kOS as degrees for consistency, but some of these may have slipped through. If you see any of these being reported in radians, please make a bug report.

Structure

structure `Orbit`

Table 8.1: Members

Suffix	Type (units)	Description
NAME	string	name of this orbit
APOAPSIS	scalar (m)	Maximum altitude
PERIAPSIS	scalar (m)	Minimum altitude
BODY	*Body*	Focal body of orbit
PERIOD	scalar (s)	orbital period
INCLINATION	scalar (deg)	orbital inclination
ECCENTRICITY	scalar	orbital eccentricity
SEMIMAJORAXIS	scalar (m)	semi-major axis
SEMIMINORAXIS	scalar (m)	semi-minor axis
LAN	scalar (deg)	Same as *LONGITUDEOFASCENDINGNODE*
LONGITUDEOFASCENDINGNODE	scalar (deg)	Longitude of the ascending node
ARGUMENTOFPERIAPSIS	scalar	argument of periapsis
TRUEANOMALY	scalar	true anomaly in degrees (not radians)
MEANANOMALYATEPOCH	scalar	mean anomaly in degrees (not radians)
TRANSITION	string	*Transition from this orbit*
POSITION	*Vector*	The current position
VELOCITY	*Vector*	The current velocity
NEXTPATCH	*Orbit*	Next *Orbit*
HASNEXTPATCH	boolean	Has a next *Orbit*

Orbit:**`NAME`**

Type string

Access Get only

a name for this orbit.

Orbit:**`APOAPSIS`**

Type scalar (m)

Access Get only

The max altitude expected to be reached.

Orbit:**`PERIAPSIS`**

Type scalar (m)

Access Get only

The min altitude expected to be reached.

Orbit:**`BODY`**

Type *Body*

Access Get only

The celestial body this orbit is orbiting.

Orbit:**`PERIOD`**

Type scalar (seconds)

Access Get only

orbital period

Orbit:**INCLINATION**

Type scalar (degree)

Access Get only

orbital inclination

Orbit:**ECCENTRICITY**

Type scalar

Access Get only

orbital eccentricity

Orbit:**SEMIMAJORAXIS**

Type scalar (m)

Access Get only

semi-major axis

Orbit:**SEMIMINORAXIS**

Type scalar (m)

Access Get only

semi-minor axis

Orbit:**LAN**

Same as *Orbit:LONGITUDEOFASCENDINGNODE*.

Orbit:**LONGITUDEOFASCENDINGNODE**

Type scalar (deg)

Access Get only

The Longitude of the ascening node is the "celestial longitude" where the orbit crosses the body's equator from its southern hemisphere to its northern hemisphere

Note that the "celestial longitude" in this case is NOT the planetary longitude of the orbit body. "Celestial longitudes" are expressed as the angle from the Solar Prime Vector <solarprimevector, not from the body's longitude. In order to find out where it is relative to the body's longitude, you will have to take into account `body:rotationangle`, and take into account that the body will rotate by the time you get there.

Orbit:**ARGUMENTOFPERIAPSIS**

Type scalar

Access Get only

argument of periapsis

Orbit:**TRUEANOMALY**

Type scalar

Access Get only

true anomaly in degrees. Even though orbital parameters are traditionally done in radians, in keeping with the kOS standard of making everything into degrees, they are given as degrees by kOS.

Orbit:**MEANANOMALYATEPOCH**

Type scalar

Access Get only

mean anomaly in degrees. Even though orbital parameters are traditionally done in radians, in keeping with the kOS standard of making everything into degrees, they are given as degrees by kOS.

Orbit:**TRANSITION**

Type string

Access Get only

Describes the way in which this orbit will end and become a different orbit, with a value taken *from this list*.

Orbit:**POSITION**

Type *Vector*

Access Get only

The current position of whatever the object is that is in this orbit.

Orbit:**VELOCITY**

Type *Vector*

Access Get only

The current velocity of whatever the object is that is in this orbit.

Orbit:**NEXTPATCH**

Type *Orbit*

Access Get only

When this orbit has a transition to another orbit coming up, this suffix returns the next Orbit patch after this one. For example, when escaping from a Mun orbit into a Kerbin orbit from which you will escape and hit a Solar orbit, then the current orbit's `:NEXTPATCH` will show the Kerbin orbit, and `:NEXTPATCH:NEXTPATCH` will show the solar orbit. The number of patches into the future that you can peek depends on your conic patches setting in your **Kerbal Space Program** Settings.cfg file.

Orbit:**HASNEXTPATCH**
boolean :access: Get only

If *:NEXTPATCH* will return a valid patch, this is true. If *:NEXTPATCH* will not return a valid patch because there are no transitions occurring in the future, then `HASNEXTPATCH` will be false.

Both `:NEXTPATCH` and `:HASNEXTPATCH` both only operate on the **current** momentum of the object, and do **not** take into account any potential changes planned with maneuver nodes. To see the possible new path you would have if a maneuver node gets executed exactly as planned, you need to first get the orbit that follows the manuever node, by looking at the maneuver node's :ORBIT suffix <node>, and then look at **it's** `:NEXTPATCH` ` and ` `":HASNEXTPATCH`.

Deprecated Suffix

Orbit:**PATCHES**

Type *List* of *Orbit* Objects

Access Get only

Note: Deprecated since version 0.15: To get the same functionality, you must use *Vessel:PATCHES* which is a suffix of the *Vessel* itself.

Transition Names

INITIAL Refers to the pure of a new orbit, which is a value you will never see from the *Orbit:TRANSITION* suffix (it refers to the start of the orbit patch, and *Orbit:TRANSITION* only refers to the end of the patch.

FINAL Means that no transition to a new orbit is expected. It this orbit is the orbit that will remain forever.

ENCOUNTER Means that this orbit will enter a new SOI of another orbital body that is smaller in scope and is "inside" the current one. (example: currently in Sun orbit, will enter Duna Orbit.)

ESCAPE Means that this orbit will enter a new SOI of another orbital body that is larger in scope and is "outside" the current one. (example: currently in Kerbin orbit, will enter Sun Orbit.)

MANEUVER Means that this orbit will end due to a manuever node that starts a new orbit?

8.1.2 Orbitable (Vessels and Bodies)

All objects that can move in orbit around other objects share some similar structure. To help keep things as consistent as possible, that similar structure is defined here. Everything you see here works for both *Vessels* and *Bodies*.

Note: SOI Body

Every where you see the term **SOI Body** in the descriptions below, it refers to the body at the center of the orbit of this object - the body in who's sphere of influence this object is located. It is important to make the distinction that if this object is itself a Body, the **SOI body** is the body being orbited, not the body doing the orbiting. I.e. When talking about the Mun, the **SOI body** means "Kerbin". When talking about Kerbin, the **SOI body** means "Sun".

structure `Orbitable`

These terms are all read-only.

Suffix	Type (units)
NAME	string
BODY	*Body*
HASBODY	boolean
HASORBIT	boolean
HASOBT	boolean
OBT	*Orbit*
ORBIT	*Orbit*
UP	*Direction*
NORTH	*Direction*
PROGRADE	*Direction*
SRFPROGRADE	*Direction*
RETROGRADE	*Direction*
SRFRETROGRADE	*Direction*
POSITION	*Vector*
VELOCITY	*OrbitableVelocity*
DISTANCE	scalar (m)
DIRECTION	*Direction*
LATITUDE	scalar (deg)
LONGITUDE	scalar (deg)
ALTITUDE	scalar (m)
GEOPOSITION	*GeoCoordinates*
PATCHES	*List* of *Orbits*
The Following are deprecated (use apoapsis and periapsis on *OBT*)	
APOAPSIS	scalar (m)
PERIAPSIS	scalar (m)

Orbitable:**NAME**

Type string

Access Get only

Name of this vessel or body.

Orbitable:**HASBODY**

Type boolean

Access Get only

True if this object has a body it orbits (false only when this object is the Sun, pretty much).

Orbitable:**HASORBIT**

Type boolean

Access Get only

Alias for HASBODY.

Orbitable:**HASOBT**

Type boolean

Access Get only

Alias for HASBODY.

Orbitable:**BODY**

Type *Body*

Access Get only

The *Body* that this object is orbiting. I.e. `Mun:BODY` returns `Kerbin`.

Orbitable:**OBT**

Type *Orbit*

Access Get only

The current single orbit "patch" that this object is on (not the future orbits it might be expected to achieve after maneuver nodes or encounter transitions, but what the current orbit would be if nothing changed and no encounters perturbed the orbit.

Orbitable:**ORBIT**

Type *Orbit*

Access Get only

This is an alias for OBT, as described above.

Orbitable:**UP**

Type *Direction*

Access Get only

pointing straight up away from the SOI body.

Orbitable:**NORTH**

Type *Direction*

Access Get only

pointing straight north on the SOI body, parallel to the surface of the SOI body.

Orbitable:**PROGRADE**

Type *Direction*

Access Get only

pointing in the direction of this object's **orbitable-frame** velocity

Orbitable:**SRFPROGRADE**

Type *Direction*

Access Get only

pointing in the direction of this object's **surface-frame** velocity. Note that if this Orbitable is itself a body, remember that this is relative to the surface of the SOI body, not this body.

Orbitable:**RETROGRADE**

Type *Direction*

Access Get only

pointing in the opposite of the direction of this object's **orbitable-frame** velocity

Orbitable:**SRFRETROGRADE**

Type *Direction*

Access Get only

pointing in the opposite of the direction of this object's **surface-frame** velocity. Note that this is relative to the surface of the SOI body.

Orbitable:**POSITION**

Type *Vector*

Access Get only

The position of this object in the *SHIP-RAW reference frame*

Orbitable:**VELOCITY**

Type *OrbitableVelocity*

Access Get only

The *orbitable velocity* of this object in the *SHIP-RAW reference frame*

Orbitable:**DISTANCE**

Type scalar (m)

Access Get only

The scalar distance between this object and the center of `SHIP`.

Orbitable:**DIRECTION**

Type *Direction*

Access Get only

pointing in the direction of this object from `SHIP`.

Orbitable:**LATITUDE**

Type scalar (deg)

Access Get only

The latitude in degrees of the spot on the surface of the SOI body directly under this object.

Orbitable:**LONGITUDE**

Type scalar (deg)

Access Get only

The longitude in degrees of the spot on the surface of the SOI body directly under this object. Longitude returned will always be normalized to be in the range [-180,180].

Orbitable:**ALTITUDE**

Type scalar (m)

Access Get only

The altitude in meters above the *sea level* surface of the SOI body (not the center of the SOI body. To get the true radius of the orbit for proper math calculations remember to add altitude to the SOI body's radius.)

Orbitable:**GEOPOSITION**

Type *GeoCoordinates*

Access Get only

A combined structure of the latitude and longitude numbers.

Orbitable:**PATCHES**

Type *List* of *Orbit* "patches"

Access Get only

The list of all the orbit patches that this object will transition to, not taking into account maneuver nodes. The zero-th patch of the list is the current orbit.

Orbitable:**APOAPSIS**

Type scalar (deg)

Access Get only

Deprecated since version 0.15: Use *OBT:APOAPSIS* instead.

Orbitable:**PERIAPSIS**

Type scalar (deg)

Access Get only

Deprecated since version 0.15: Use *OBT:PERIAPSIS* instead.

8.1.3 OrbitableVelocity

When any *Orbitable* object returns its *VELOCITY* suffix, it returns it as a structure containing a pair of both its orbit-frame velocity and its surface-frame velocity at the same instant of time. To obtain its velocity as a vector you must pick whether you want the oribtal or surface velocities by giving a further suffix:

structure OrbitableVelocity

Table 8.2: Members

Suffix	Type
ORBIT	*Vector*
SURFACE	*Vector*

OrbitableVelocity:**ORBIT**

Type *Vector*

Access Get only

Returns the orbital velocity.

OrbitableVelocity:**SURFACE**

Type *Vector*

Access Get only

Returns the surface-frame velocity. Note that this is the surface velocity relative to the surface of the SOI body, not the orbiting object itself. (i.e. Mun:VELOCITY:SURFACE returns the Mun's velocity relative to the surface of its SOI body, Kerbin).

Examples:

```
SET VORB TO SHIP:VELOCITY:ORBIT
SET VSRF TO SHIP:VELOCITY:SURFACE
SET MUNORB TO MUN:VELOCITY:ORBIT
SET MUNSRF TO MUN:VELOCITY:SURFACE
```

Note: At first glance it may seem that `Mun:VELOCITY:SURFACE` is wrong because it creates a vector in the opposite direction from `Mun:VELOCITY:ORBIT`, but this is actually correct. Kerbin's surface rotates once every 6 hours, and the Mun takes a lot longer than 6 hours to orbit Kerbin. Therefore, relative to Kerbin's surface, the Mun is going backward.

8.2 Celestial Bodies

8.2.1 Atmosphere

A Structure closely tied to *Body* A variable of type *Atmosphere* usually is obtained by the *:ATM* suffix of a *Body*. ALL The following values are read-only. You can't change the value of a body's atmosphere.

structure Atmosphere

Suffix	Type	Description
BODY	string	Name of the celestial body
EXISTS	bool	True if this body has an atmosphere
OXYGEN	bool	True if oxygen is present
SCALE	scalar	Used to find atmospheric density
SEALEVELPRESSURE	scalar (atm)	pressure at sea level
HEIGHT	scalar (m)	advertised atmospheric height

Atmosphere:**BODY**

Type string

Access Get only

The Body that this atmosphere is around - as a STRING NAME, not a Body object.

Atmosphere:**EXISTS**

Type bool

Access Get only

True if this atmosphere is "real" and not just a dummy placeholder.

Atmosphere:**OXYGEN**

Type bool

Access Get only

True if the air has oxygen and could therefore be used by a jet engine's intake.

Atmosphere:**SCALE**

Type scalar

Access Get only

A math constant plugged into a formula to find atmosphere density.

Atmosphere:**SEALEVELPRESSURE**

Type scalar (atm)

Access Get only

Number of Atm's at planet's sea level 1.0 Atm's = same as Kerbin.

Atmosphere:**HEIGHT**

Type scalar (m)

Access Get only

The altitude at which the atmosphere is "officially" advertised as ending. (actual ending value differs, see below).

Atmospheric Math

Note: [Section deleted]

This documentation used to contain a description of how the math for Kerbal Space Program's default stock atmospheric model works, but everything that was mentioned here became utterly false when KSP 1.0 was released with a brand new atmospheric model that invalided pretty much everything that was said here. Rather than teach people incorrect information, it was deemed that no documentation is better than misleading documentation, so this section below this point has been removed.

8.2.2 Body

This is any sort of planet or moon. To get a variable referring to a Body, you can do this:

```
// "name" is the name of the body,
// like "Mun" for example.
SET MY_VAR TO BODY("name").
```

Also, all bodies have hard-coded variable names as well. You can use the variable `Mun` to mean the same thing as `BODY("Mun")`.

Note: Changed in version 0.13: A Celestial Body is now also an *Orbitable*, and can use all the terms described for these objects too.

Predefined Celestial Bodies

All of the main celestial bodies in the game are reserved variable names. The following two lines do the exactly the same thing:

```
SET the_mun TO Mun.
SET the_mun TO Body("Mun").
```

- Sun
- Moho
- **Eve**
  - Gilly
- **Kerbin**
  - Mun
  - Minmus
- **Duna**

  - Ike
- **Jool**
  - Laythe
  - Vall
  - Tylo
  - Bop
  - Pol
- Eeloo

structure `Body`

Suffix	Type (units)
Every Suffix of *`Orbitable`*	
`NAME`	string
`DESCRIPTION`	string
`MASS`	scalar (kg)
`ALTITUDE`	scalar (m)
`ROTATIONPERIOD`	scalar (s)
`RADIUS`	scalar (m)
`MU`	scalar ($m^3 s^2$)
`ATM`	*`Atmosphere`*
`ANGULARVEL`	*`Direction`* in *SHIP-RAW*
`GEOPOSITIONOF`	*`GeoCoordinates`* in *SHIP-RAW*
`ALTITUDEOF`	scalar (m)
`SOIRADIUS`	scalar (m)
`ROTATIONANGLE`	scalar (deg)

Body:**`NAME`**
The name of the body. Example: "Mun".

Body:**`DESCRIPTION`**
Longer description of the body, often just a duplicate of the name.

Body:**`MASS`**
The mass of the body in kilograms.

Body:**`ALTITUDE`**
The altitude of this body above the sea level surface of its parent body. I.e. the altitude of Mun above Kerbin.

Body:**`ROTATIONPERIOD`**
The length of the body's day in seconds. I.e. how long it takes for it to make one rotation.

Body:**`RADIUS`**
The radius from the body's center to its sea level.

Body:**`MU`**
The Gravitational Parameter of the body.

Body:**`ATM`**
A variable that describes the atmosphere of this body.

Body:**`ANGULARVEL`**
Despite the name, this is technically not a velocity. It only tells you the axis of rotation, not the speed of rotation around that axis.

Body:**GEOPOSITIONOF**
: The geoposition underneath the given vector position. SHIP:BODY:GEOPOSITIONOF(SHIP:POSITION) should, in principle, give the same thing as SHIP:GEOPOSITION, while SHIP:BODY:GEOPOSITIONOF(SHIP:POSITION + 1000*SHIP:NORTH) would give you the lat/lng of the position 1 kilometer north of you. Be careful not to confuse this with :GEOPOSITION (no "OF" in the name), which is also a suffix of Body by virtue of the fact that Body is an Orbitable, but it doesn't mean the same thing.

Body:**ALTITUDEOF**
: The altitude of the given vector position, above this body's 'sea level'. SHIP:BODY:ALTITUDEOF(SHIP:POSITION) should, in principle, give the same thing as SHIP:ALTITUDE. Example: Eve:ALTITUDEOF(GILLY:POSITION) gives the altitude of gilly's current position above Eve, even if you're not actually anywhere near the SOI of Eve at the time. Be careful not to confuse this with :ALTITUDE (no "OF" in the name), which is also a suffix of Body by virtue of the fact that Body is an Orbitable, but it doesn't mean the same thing.

Body:**SOIRADIUS**
: The radius of the body's sphere of influence. Measured from the body's center.

Body:**ROTATIONANGLE**
: The rotation angle is the number of degrees between the *Solar Prime Vector* and the current positon of the body's prime meridian (body longitude of zero).

  The value is in constant motion, and once per body's day, its `:rotationangle` will wrap around through a full 360 degrees.

8.3 Vessels and Parts

8.3.1 AggregateResource

A ship can have many parts that contain resources (i.e. fuel, electric charge, etc). kOS has several tools for getting the summation of each resource.

This is the type returned as the elements of the list from `LIST RESOURCES.`

IN MyList <list command>

```
PRINT "THESE ARE ALL THE RESOURCES ON THE SHIP:".
LIST RESOURCES IN RESLIST.
FOR RES IN RESLIST {
    PRINT "Resource " + RES:NAME.
    PRINT "    value = " + RES:AMOUNT.
    PRINT "    which is "
          + ROUND(100*RES:AMOUNT/RES:CAPACITY)
          + "% full.".
}.
```

This is also the type returned by STAGE:RESOURCES

```
PRINT "THESE ARE ALL THE RESOURCES active in this stage:".
SET RESLIST TO STAGE:RESOURCES.
FOR RES IN RESLIST {
    PRINT "Resource " + RES:NAME.
    PRINT "    value = " + RES:AMOUNT.
    PRINT "    which is "
          + ROUND(100*RES:AMOUNT/RES:CAPACITY)
          + "% full.".
}.
```

structure `AggregateResource`

Suffix	Type	Description
NAME	string	Resource name
AMOUNT	scalar	Total amount remaining
CAPACITY	scalar	Total amount when full
PARTS	List	Parts containing this resource

AggregateResource:**`NAME`**

Access Get only

Type string

The name of the resource, i.e. "LIQUIDFUEL", "ELECTRICCHARGE", "MONOPROP".

AggregateResource:**`AMOUNT`**

Access Get only

Type scalar

The value of how much resource is left.

AggregateResource:**`CAPACITY`**

Access Get only

Type scalar

What AMOUNT would be if the resource was filled to the top.

AggregateResource:**`PARTS`**

Access Get only

Type List

Because this is a summation of the resources from many parts. kOS gives you the list of all parts that do or could contain the resource.

8.3.2 ALT

ALT is a special object that exists just to help you get the altitudes of interest for a vessel future. It is grandfathered in for the sake of backward compatibility, but this information is also available on the Vessel structure as well, which is the better new way to do it:

structure `ALT`

Suffix	Type	Description
APOAPSIS	Number, meters	altitude in meters of SHIP's apoapsis. Same as SHIP:APOAPSIS.
PERIAPSIS	Number, meters	altitude in meters of SHIP's periapsis. Same as SHIP:PERIAPSIS.
RADAR	Number, meters	Altitude of SHIP above the ground terrain, rather than above sea level.

8.3.3 CrewMember

Represents a single crew member of a vessel.

structure `CrewMember`

Suffix	Type	Description
NAME	string	crew member's name
GENDER	string	"Male" or "Female"
EXPERIENCE	scalar	experience level (number of stars)
TRAIT	string	"Pilot", "Engineer" or "Scientist"
TOURIST	Boolean	true if this crew member is a tourist
PART	*Part*	part in which the crew member is located

CrewMember:**`NAME`**

Type string

Access Get only

crew member's name

CrewMember:**`GENDER`**

Type string

Access Get only

"Male" or "Female"

CrewMember:**`EXPERIENCE`**

Type scalar

Access Get only

experience level (number of stars)

CrewMember:**`TRAIT`**

Type string

Access Get only

crew member's trait (specialization), for example "Pilot"

CrewMember:**`TOURIST`**

Type Boolean

Access Get only

true if this crew member is a tourist

CrewMember:**`PART`**

Type *Part*

Access Get only

Part this crew member is located in

8.3.4 DockingPort

Some of the Parts returned by *LIST PARTS* will be of type *DockingPort*.

Note: New in version 0.18: The spelling of suffixes *AQUIRERANGE*, *AQUIREFORCE*, and *AQUIRETORQURE* on the *DockingPort* structure has been corrected. Please use *ACQUIRERANGE*, *ACQUIREFORCE*, and *ACQUIRETORQURE* instead. Using the old incorrect spelling, a deprecation exception will be thrown, with instruction to use the new spelling.

structure `DockingPort`

Suffix	Type	Description
All suffixes of *Part*		A *DockingPort* is a kind of *Part*
ACQUIRERANGE	scalar	active range of the port
ACQUIREFORCE	scalar	force experienced when docking
ACQUIRETORQUE	scalar	torque experienced when docking
REENGAGEDDISTANCE	scalar	distance at which the port is reset
DOCKEDSHIPNAME	string	name of vessel the port is docked to
NODEPOSITION	vector	coords of where the docking node attachment point is in SHIP-RAW xyz
NODETYPE	string	two nodes are only dockable together if their NODETYPE strings match
PORTFACING	*Direction*	facing of the port
STATE	string	current state of the port
UNDOCK		callable to release the dock
TARGETABLE	boolean	check if this port can be targeted

Note: *DockingPort* is a type of *Part*, and therefore can use all the suffixes of *Part*. Shown below are only the suffixes that are unique to *DockingPort*.

DockingPort:**`ACQUIRERANGE`**

Type scalar

Access Get only

gets the range at which the port will "notice" another port and pull on it.

DockingPort:**`ACQUIREFORCE`**

Type scalar

Access Get only

gets the force with which the port pulls on another port.

DockingPort:**`ACQUIRETORQUE`**

Type scalar

Access Get only

gets the rotational force with which the port pulls on another port.

DockingPort:**`REENGAGEDDISTANCE`**

Type scalar

Access Get only

how far the port has to get away after undocking in order to re-enable docking.

DockingPort:**`DOCKEDSHIPNAME`**

Type string

Access Get only

name of vessel on the other side of the docking port.

DockingPort:**`NODEPOSITION`**

Type vector

Access Get only

The coordinates of the point on the docking port part where the port attachment spot is located. This is different from the part's position itself because that's the position of the center of the whole part. This is the position of the face of the docking port. Coordinates are in SHIP-RAW xyz coords.

DockingPort:**NODETYPE**

Type string

Access Get only

Each docking port has a node type string that specifies its compatibility with other docking ports. In order for two docking ports to be able to attach to each other, the values for their NODETYPEs must be the same.

The base KSP stock docking port parts all use one of the following three values:

- "size0" for all Junior-sized docking ports.
- "size1" for all Normal-sized docking ports.
- "size2" for all Senior-sized docking ports.

Mods that provide their own new kinds of docking port might use any other value they feel like here, but only if they are trying to declare that the new part isn't supposed to be able to connect to stock docking ports. Any docking port that is meant to connect to stock ports will have to adhere to the above scheme.

DockingPort:**PORTFACING**

Type *Direction*

Access Get only

Gets the facing of this docking port which may differ from the facing of the part itself if the docking port is aimed out the side of the part, as in the case of the inline shielded docking port.

DockingPort:**STATE**

Type string

Access Get only

One of the following string values:

Ready Docking port is not yet attached and will attach if it touches another.

Docked (docker) One port in the joined pair is called the docker, and has this state

Docked (dockee) One port in the joined pair is called the dockee, and has this state

Docked (same vessel) Sometimes KSP says this instead. It's unclear what it means.

Disabled Docking port will refuse to dock if it bumps another docking port.

PreAttached Temporary state during the "wobbling" while two ports are magnetically touching but not yet docked solidly. During this state the two vessels are still tracked as separate vessels and haven't become one yet.

DockingPort:**UNDOCK**()
Call this to cause the docking port to detach.

DockingPort:**TARGETABLE**

Type boolean

Access Get only

True if this part can be picked with `SET TARGET TO`.

8.3.5 Element

An element is a *docked component* of a *Vessel*. When you dock several vessels together to create one larger vessel, you can obtain the "chunks" of the larger vessel organized by which original vessel they came from. These "chunks" are called elements, and they are what the Element structure refers to.

A list of elements from the vessel can be created by using the command:

```
list elements in eList.
// now eList is a list of elements from the vessel.
```

Each item of that list is one of the elements. The rest of this page describes the elements and what they do.

Note: Element boundries are not formed between two docking ports that were launched coupled. a craft with such an arrangement will appear as one element until you uncoupled the nodes and redocked

---

structure `Element`

Suffix	Type	Description
NAME	string	The name of the docked craft
UID	string	Unique Identifier
PARTS	*List*	all *Parts*
DOCKINGPORTS	*List*	all *DockingPorts*
VESSEL	*Vessel*	the parent *Vessel*
RESOURCES	*List*	all *AggrgateResources*

Element:**`UID`**

Type string

Access Get only

A unique id

Element:**`NAME`**

Type string

Access Get/Set

The name of the Element element, is an artifact from the vessel the element belonged to before docking. Cannot be set to an empty string.

Element:**`PARTS`**

Type *List* of *Part* objects

Access Get only

A List of all the *parts* on the Element. `SET FOO TO SHIP:PARTS.` has exactly the same effect as `LIST PARTS IN FOO.`. For more information, see *ship parts and modules*.

Element:**`DOCKINGPORTS`**

Type *List* of *DockingPort* objects

Access Get only

A List of all the *docking ports* on the Element.

Element:**`VESSEL`**

Type *Vessel*

Access Get only

The parent vessel containing the element.

Element:**RESOURCES**

Type *List* of *AggregateResource* objects

Access Get only

A List of all the *AggregateResources* on the element.

8.3.6 Engine

Some of the Parts returned by *LIST PARTS* will be of type Engine. It is also possible to get just the Engine parts by executing `LIST ENGINES`, for example:

```
LIST ENGINES IN myVariable.
FOR eng IN myVariable {
    print "An engine exists with ISP = " + eng:ISP.
}.
```

structure Engine

Table 8.3: Members

Suffix	Type (units)	Description
All suffixes of *Part*		
ACTIVATE		Turn engine on
SHUTDOWN		Turn engine off
THRUSTLIMIT	scalar (%)	Tweaked thrust limit
MAXTHRUST	scalar (kN)	Untweaked thrust limit
MAXTHRUSTAT(pressure)	scalar (kN)	Max thrust at the specified pressure (in standard Kerbin atmospheres).
THRUST	scalar (kN)	Current thrust
AVAILABLETHRUST	scalar (kN)	Available thrust at full throttle accounting for thrust limiter
AVAILABLETHRUSTAT(pressure)	scalar (kN)	Available thrust at the specified pressure (in standard Kerbin atmospheres).
FUELFLOW	scalar (l/s maybe)	Rate of fuel burn
ISP	scalar	Specific impulse
ISPAT(pressure)	scalar	Specific impulse at the given pressure (in standard Kerbin atmospheres).
VACUUMISP	scalar	Vacuum specific impulse
VISP	scalar	Synonym for VACUUMISP
SEALEVELISP	scalar	Specific impulse at Kerbin sealevel
SLISP	scalar	Synonym for SEALEVELISP
FLAMEOUT	boolean	Check if no more fuel
IGNITION	boolean	Check if engine is active
ALLOWRESTART	boolean	Check if engine can be reactivated
ALLOWSHUTDOWN	boolean	Check if engine can be shutdown
THROTTLELOCK	boolean	Check if throttle can not be changed

Note: *Engine* is a type of *Part*, and therefore can use all the suffixes of *Part*. Shown below are only the suffixes that are unique to *Engine*.

Engine:**ACTIVATE**()
Call to make the engine turn on.

Engine:**SHUTDOWN**()
Call to make the engine turn off.

Engine:**THRUSTLIMIT**

Access Get/Set

Type scalar (%)

If this an engine with a thrust limiter (tweakable) enabled, what percentage is it limited to?

Engine:**MAXTHRUST**

Access Get only

Type scalar (kN)

How much thrust would this engine give at its current atmospheric pressure and velocity if the throttle was max at 1.0, and the thrust limiter was max at 100%. Note this might not be the engine's actual max thrust it could have under other air pressure conditions. Some engines have a very different value for MAXTHRUST in vacuum as opposed to at sea level pressure. Also, some jet engines have a very different value for MAXTHRUST depending on how fast they are currently being rammed through the air.

Engine:**MAXTHRUSTAT**(pressure)

Parameters

- **pressure** – atmospheric pressure (in standard Kerbin atmospheres)

Return type scalar (kN)

How much thrust would this engine give if both the throttle and thrust limtier was max at the current velocity, and at the given atmospheric pressure. Use a pressure of 0.0 for vacuum, and 1.0 for sea level (on Kerbin) (or more than 1 for thicker atmospheres like on Eve).

Engine:**THRUST**

Access Get only

Type scalar (kN)

How much thrust is this engine giving at this very moment.

Engine:**AVAILABLETHRUST**

Access Get only

Type scalar (kN)

Taking into account the thrust limiter tweakable setting, how much thrust would this engine give if the throttle was max at its current thrust limit setting and atmospheric pressure and velocity conditions.

Engine:**AVAILABLETHRUSTAT**(pressure)

Parameters

- **pressure** – atmospheric pressure (in standard Kerbin atmospheres)

Return type scalar (kN)

Taking into account the thrust limiter tweakable setting, how much thrust would this engine give if the throttle was max at its current thrust limit setting and velocity, but at a different atmospheric pressure you pass into it. The pressure is measured in ATM's, meaning 0.0 is a vacuum, 1.0 is seal level at Kerbin.

Engine:**FUELFLOW**

Access Get only

Type scalar (Liters/s? maybe)

Rate at which fuel is being burned. Not sure what the units are.

Engine:**ISP**

Access Get only

Type scalar

Specific impulse

Engine:**ISPAT**(pressure)

Parameters

- **pressure** – atmospheric pressure (in standard Kerbin atmospheres)

Return type scalar

Specific impulse at the given atmospheric pressure. Use a pressure of 0 for vacuum, and 1 for sea level (on Kerbin).

Engine:**VACUUMISP**

Access Get only

Type scalar

Vacuum specific impulse

Engine:**VISP**

Access Get only

Type scalar

Synonym for :VACUUMISP

Engine:**SEALEVELISP**

Access Get only

Type scalar

Specific impulse at Kerbin sealevel.

Engine:**SLISP**

Access Get only

Type scalar

Synonym for :SEALEVELISP

Engine:**FLAMEOUT**

Access Get only

Type boolean

Is this engine failed because it is starved of a resource (liquidfuel, oxidizer, oxygen)?

Engine:**IGNITION**

Access Get only

Type boolean

Has this engine been ignited? If both *Engine:IGNITION* and *Engine:FLAMEOUT* are true, that means the engine could start up again immediately if more resources were made available to it.

Engine:**ALLOWRESTART**

Access Get only

Type boolean

Is this an engine that can be started again? Usually True, but false for solid boosters.

Engine:**ALLOWSHUTDOWN**

Access Get only

Type boolean

Is this an engine that can be shut off once started? Usually True, but false for solid boosters.

Engine:**THROTTLELOCK**

Access Get only

Type boolean

Is this an engine that is stuck at a fixed throttle? (i.e. solid boosters)

8.3.7 ETA

ETA is a special object that exists just to help you get the times from now to certain events in a vessel's future. It always presumes you're operating on the current SHIP vessel:

structure ETA

Suffix	Type	Description
APOAPSIS	Number, seconds	Seconds until SHIP hits its apoapsis.
PERIAPSIS	Number, seconds	Seconds until SHIP hits its periapsis.
TRANSITION	Number, seconds	Seconds until SHIP hits the next orbit patch.

ETA:**APOAPSIS**

Type Number, seconds

Access Get only

Seconds until SHIP hits its apoapsis. If the ship is on an escape trajectory (hyperbolic orbit) such that you will never reach apoapsis, it will return zero.

ETA:**PERIAPSIS**

Type Number, seconds

Access Get only

Seconds until SHIP hits its periapsis. If the ship is on an intersect with the ground, such that you'll hit the ground first before you'd get to periapsis, it will still return the hypothetical number of seconds it would have taken to get to periapsis if you had the magical ability to pass through the ground as if it wasn't there.

ETA:**TRANSITION**

Type Number, seconds

Access Get only

Seconds until SHIP hits its the end of its current orbit patch and transitions into another one, ignoring the effect of any intervening manuever nodes it might hit before it gets there.

8.3.8 Gimbal

Many engines in KSP have thrust vectoring gimbals and in ksp they are their own module

structure `Gimbal`

Suffix	Type	Description
RANGE	scalar	The Gimbal's Possible Range of movement
RESPONSESPEED	scalar	The Gimbal's Possible Rate of travel
PITCHANGLE	scalar	Current Gimbal Pitch
YAWANGLE	scalar	Current Gimbal Yaw
ROLLANGLE	scalar	Current Gimbal Roll
LOCK	boolean	Is the gimbal free to travel?

Gimbal:**`RANGE`**

Type scalar

Access Get only

The maximum extent of travel possible for the gimbal along all 3 axis (Pitch, Yaw, Roll)

Gimbal:**`RESPONSESPEED`**

Type scalar

Access Get only

A Measure of the rate of travel for the gimbal

Gimbal:**`PITCHANGLE`**

Type scalar

Access Get only

The gimbals current pitch, has a range of -1 to 1. Will always be 0 when LOCK is true

Gimbal:**`YAWANGLE`**

Type scalar

Access Get only

The gimbals current yaw, has a range of -1 to 1. Will always be 0 when LOCK is true

Gimbal:**`ROLLANGLE`**

Type scalar

Access Get only

The gimbals current roll, has a range of -1 to 1. Will always be 0 when LOCK is true

Gimbal:**`LOCK`**

Type string

Access Get/Set

Can this Gimbal produce torque right now, when you set it to false it will snap the engine back to 0s for pitch,yaw and roll

8.3.9 Maneuver Node

Contents

A planned velocity change along an orbit. These are the nodes that you can set in the KSP user interface. Setting one through kOS will make it appear on the in-game map view, and creating one manually on the in-game map view will cause it to be visible to kOS.

Warning: Be aware that a limitation of KSP makes it so that some vessels' manuever node systems cannot be accessed. KSP appears to limit the maneuver node system to only functioning on the current PLAYER vessel, under the presumption that its the only vessel that needs them, as ever other vessel cannot be manuevered. kOS can manuever a vessel that is not the player vessel, but it cannot overcome this limitation of the base game that unloads the maneuver node system for other vessels.
Be aware that the effect this has is that when you try to use some of these commands on some vessels, they won't work because those vessels do not have their manuever node system in play. This is mostly only going to happen when you try to run a script on a vessel that is not the current player active vessel.

Creation

NODE(utime, radial, normal, prograde)

Parameters

- **utime** – (sec) Time of this maneuver
- **radial** – (m/s) Delta-V in radial-out direction
- **normal** – (m/s) Delta-V normal to orbital plane
- **prograde** – (m/s) Delta-V in prograde direction

Returns *ManeuverNode*

You can make a maneuver node in a variable using the *NODE* function:

```
SET myNode to NODE( TIME:SECONDS+200, 0, 50, 10 ).
```

Once you have a maneuver node in a variable, you use the *ADD* and *REMOVE* commands to attach it to your vessel's flight plan. A kOS CPU can only manipulate the flight plan of its *CPU vessel*.

Warning: When *constructing* a new node using the *NODE* function call, you use the universal time (you must add the ETA time to the current time to arrive at the value to pass in), but when using the suffix *ManeuverNode:ETA*, you do NOT use universal time, instead just giving the number of seconds from now.

Once you have created a node, it's just a hypothetical node that hasn't been attached to anything yet. To attach a node to the flight path, you must use the command *ADD* to attach it to the ship.

ADD

To put a maneuver node into the flight plan of the current *CPU vessel* (i.e. SHIP), just *ADD* it like so:

```
SET myNode to NODE( TIME:SECONDS+200, 0, 50, 10 ).
ADD myNode.
```

You should immediately see it appear on the map view when you do this. The *ADD* command can add nodes anywhere within the flight plan. To insert a node earlier in the flight than an existing node, simply give it a smaller *ETA* time and then *ADD* it.

Warning: As per the warning above at the top of the section, ADD won't work on vessels that are not the active vessel.

REMOVE

To remove a maneuver node from the flight path of the cur:rent *CPU vessel* (i.e. SHIP), just *REMOVE* it like so:

```
REMOVE myNode.
```

Warning: As per the warning above at the top of the section, REMOVE won't work on vessels that are not the active vessel.

NEXTNODE

NEXTNODE is a built-in variable that always refers to the next upcoming node that has been added to your flight plan:

```
SET MyNode to :global:`NEXTNODE`.
PRINT :global:`NEXTNODE`:PROGRADE.
REMOVE :global:`NEXTNODE`.
```

Currently, if you attempt to query *NEXTNODE* and there is no node on your flight plan, it produces a run-time error. (This needs to be fixed in a future release so it is possible to query whether or not you have a next node).

Warning: As per the warning above at the top of the section, NEXTNODE won't work on vessels that are not the active vessel.
If you need to query whether or not you have a *NEXTNODE*, the following has been suggested as a workaround in the meantime: Set a node really far into the future, beyond any reasonable amount of time. Add it to your flight plan. Then check *NEXTNODE* to see if it returns THAT node, or an earlier one. If it returns an earlier one, then that earlier one was there all along and is the real *NEXTNODE*. If it returns the fake far-future node you made instead, then there were no nodes before that point. In either case, remove the far-future node after you perform the test.
The special identifier *NEXTNODE* is a euphemism for "whichever node is coming up soonest on my flight path". Therefore you can remove a node even if you no longer have the maneuver node variable around, by doing this:

```
REMOVE :global:`NEXTNODE`.
```

Structure

structure ManeuverNode

Here are some examples of accessing the suffixes of a *ManeuverNode*:

```
// creates a node 60 seconds from now with
// prograde = 100 m/s
SET X TO NODE(TIME:SECONDS+60, 0, 0, 100).

ADD X.            // adds maneuver to flight plan

PRINT X:PROGRADE. // prints 100.
PRINT X:ETA.      // prints seconds till maneuver
```

```
PRINT X:DELTAV    // prints delta-v vector

REMOVE X.         // remove node from flight plan

// Create a blank node
SET X TO NODE(0, 0, 0, 0).

ADD X.                  // add Node to flight plan
SET X:PROGRADE to 500. // set prograde dV to 500 m/s
SET X:ETA to 30.       // Set to 30 sec from now

PRINT X:ORBIT:APOAPSIS.  // apoapsis after maneuver
PRINT X:ORBIT:PERIAPSIS. // periapsis after maneuver
```

Table 8.4: Members

Suffix	Type (units)	Access	Description
DELTAV	*Vector* (m/s)	Get only	The burn vector with magnitude equal to delta-V
BURNVECTOR	*Vector* (m/s)	Get only	Alias for *DELTAV*
ETA	scalar (s)	Get/Set	Time until this maneuver
PROGRADE	scalar (m/s)	Get/Set	Delta-V along prograde
RADIALOUT	scalar (m/s)	Get/Set	Delta-V along radial to orbited *Body*
NORMAL	scalar (m/s)	Get/Set	Delta-V along normal to the *Vessel*'s *Orbit*
ORBIT	*Orbit*	Get only	Expected *Orbit* after this maneuver

ManeuverNode:**DELTAV**

Access Get only

Type *Vector*

The vector giving the total burn of the node. The vector can be used to steer with, and its magnitude is the delta V of the burn.

ManeuverNode:**BURNVECTOR**

Alias for *ManeuverNode:DELTAV*.

ManeuverNode:**ETA**

Access Get/Set

Type scalar

The number of seconds until the expected burn time. If you SET this, it will actually move the maneuver node along the path in the map view, identically to grabbing the maneuver node and dragging it.

ManeuverNode:**PROGRADE**

Access Get/Set

Type scalar

The delta V in (meters/s) along just the prograde direction (the yellow and green 'knobs' of the maneuver node). A positive value is a prograde burn and a negative value is a retrograde burn.

ManeuverNode:**RADIALOUT**

Access Get/Set

Type scalar

The delta V in (meters/s) along just the radial direction (the cyan knobs' of the maneuver node). A positive value is a radial out burn and a negative value is a radial in burn.

ManeuverNode:**NORMAL**

Access Get/Set

Type scalar

The delta V in (meters/s) along just the normal direction (the purple knobs' of the maneuver node). A positive value is a normal burn and a negative value is an anti-normal burn.

ManeuverNode:**ORBIT**

Access Get only

Type *Orbit*

The new orbit patch that will begin starting with the burn of this node, under the assumption that the burn will occur exactly as planned.

8.3.10 Part

These are the generic properties every PART has. You can obtain a list of values of type Part using the *LIST PARTS command*.

structure Part

Table 8.5: Members

Suffix	Type	Description
NAME	string	Name of this part
TITLE	string	Title as it appears in KSP
MASS	scalar	Current mass of part and its resources
DRYMASS	scalar	Mass of part if all resources were empty
WETMASS	scalar	Mass of part if all resources were full
TAG	string	Name-tag if assigned by the player
CONTROLFROM	Void	Call to control-from to this part
STAGE	scalar	The stage this is associated with
UID	string	Unique identifying number of this part
ROTATION	*Direction*	The rotation of this part's x-axis
POSITION	*Vector*	The location of this part in the universe
FACING	*Direction*	the direction that this part is facing
RESOURCES	*List*	list of the *Resource* in this part
TARGETABLE	boolean	true if this part can be selected as a target
SHIP	*Vessel*	the vessel that contains this part
GETMODULE(name)	*PartModule*	Get one of the *PartModules* by name
MODULES	*List*	Names (string) of all *PartModules*
ALLMODULES	*List*	Same as *MODULES*
PARENT	*Part*	Adjacent *Part* on this *Vessel*.
HASPARENT	boolean	Check if this part has a parent *Part*
HASPHYSICS	boolean	Does this part have mass or drag
CHILDREN	*List*	List of attached *Parts*

Part:**NAME**

Access Get only

Type string

Name of part as it is used behind the scenes in the game's API code.

A part's *name* is the name it is given behind the scenes in KSP. It never appears in the normal GUI for the user to see, but it is used in places like Part.cfg files, the saved game persistence file, the ModuleManager mod, and so on.

Part:**TITLE**

Access Get only

Type string

The title of the part as it appears on-screen in the gui.

A part's *title* is the name it has inside the GUI interface on the screen that you see as the user.

Part:**TAG**

Access Get / Set

Type string

The name tag value that may exist on this part if you have given the part a name via the *name-tag system*.

A part's *tag* is whatever custom name you have given it using the *name-tag system described here*. This is probably the best naming convention to use because it lets you make up whatever name you like for the part and use it to pick the parts you want to deal with in your script.

WARNING: This suffix is only settable for parts attached to the *CPU Vessel*

This example assumes you have a target vessel picked, and that the target vessel is loaded into full-physics range and not "on rails". vessels that are "on rails" do not have their full list of parts entirely populated at the moment:

```
LIST PARTS FROM TARGET IN tParts.

PRINT "The target vessel has a".
PRINT "partcount of " + tParts:LENGTH.

SET totTargetable to 0.
FOR part in tParts {
    IF part:TARGETABLE {
        SET totTargetable TO totTargetable + 1.
    }
}

PRINT "...and " + totTargetable.
PRINT " of them are targetable parts.".
```

Part:**CONTROLFROM**

Access Callable function only

Type void

Call this function to cause the game to do the same thing as when you right-click a part on a vessel and select "control from here" on the menu. It rotates the control orientation so that fore/aft/left/right/up/down now match the orientation of this part. NOTE that this will not work for every type of part. It only works for those parts that KSP itself allows this for (control cores and docking ports). It accepts no arguments, and returns no value. All vessels must have at least one "control from" part on them somewhere, which is why there's no mechanism for un-setting the "control from" setting other than to pick another part and set it to that part instead.

WARNING: This suffix is only callable for parts attached to the *CPU Vessel*

Part:**STAGE**

Access Get only

Type scalar

the stage this part is part of.

Part:**UID**

Access Get only

Type string

All parts have a unique ID number. Part's uid never changes because it is the same value as stored in persistent.sfs. Although you can compare parts by comparing their uid it is recommended to compare parts directly if possible.

Part:**ROTATION**

Access Get only

Type *Direction*

The rotation of this part's X-axis, which points out of its side and is probably not what you want. You probably want the *Part:FACING* suffix instead.

Part:**POSITION**

Access Get only

Type *Vector*

The location of this part in the universe. It is expressed in the same frame of reference as all the other positions in kOS, and thus can be used to help do things like navigate toward the position of a docking port.

Part:**FACING**

Access Get only

Type *Direction*

the direction that this part is facing.

Part:**MASS**

Access Get only

Type scalar

The current mass or the part and its resources. If the part has no physics this will always be 0.

Part:**WETMASS**

Access Get only

Type scalar

The mass of the part if all of its resources were full. If the part has no physics this will always be 0.

Part:**DRYMASS**

Access Get only

Type scalar

The mass of the part if all of its resources were empty. If the part has no physics this will always be 0.

Part:**RESOURCES**

Access Get only

Type *List*

list of the *Resource* in this part.

Part:**TARGETABLE**

Access Get only

Type boolean

true if this part can be selected by KSP as a target.

Part:**SHIP**

Access Get only

Type *Vessel*

the vessel that contains this part.

Part:**GETMODULE**(name)

Parameters

- **name** – (string) Name of the part module

Returns *PartModule*

Get one of the *PartModules* attached to this part, given the name of the module. (See *Part:MODULES* for a list of all the names available).

Part:**MODULES**

Access Get only

Type *List* of strings

list of the names of *PartModules* enabled for this part.

Part:**ALLMODULES**
Same as *Part:MODULES*

Part:**PARENT**

Access Get only

Type *Part*

When walking the *tree of parts*, this is the part that this part is attached to on the way "up" toward the root part.

Part:**HASPHYSICS**

Access Get only

Type bool

This comes from a part's configuration and is an artifact of the KSP simulation.

For a list of stock parts that have this attribute and a fuller explanation see the KSP wiki page about massless parts.

Part:**HASPARENT**

Access Get only

Type boolean

When walking the *tree of parts*, this is true as long as there is a parent part to this part, and is false if this part has no parent (which can only happen on the root part).

Part:**CHILDREN**

Access Get only

Type *List* of *Parts*

When walking the *tree of parts*, this is all the parts that are attached as children of this part. It returns a list of zero length when this part is a "leaf" of the parts tree.

8.3.11 PartModule

Almost everything done with at right-click menus and action group action can be accessed via the *PartModule* objects that are attached to *Parts* of a *Vessel*.

The exact arrangement of *PartModule* to *Parts* to *Vessels*, and how to make use of a *PartModule* is a complex enough topic to warrant its own separate subject, described on the *Ship parts and Modules* page. If you have not read that page, it is recommended that you do so before using *PartModules*. The page you are reading now is meant as just a reference summary, not a tutorial.

Once you have a *PartModule*, you can use it to invoke the behaviors that are connected to its right-click menu and to its action groups.

structure PartModule

Table 8.6: Members

Suffix	Type	Description
NAME	string	Name of this part module
PART	*Part*	*Part* attached to
ALLFIELDS	*List* of strings	Accessible fields
ALLFIELDNAMES	*List* of strings	Accessible fields (name only)
ALLEVENTS	*List* of strings	Triggerable events
ALLEVENTNAMES	*List* of strings	Triggerable event names
ALLACTIONS	*List* of strings	Triggerable actions
ALLACTIONNAMES	*List* of strings	Triggerable event names
GETFIELD(name)		Get value of a field by name
SETFIELD(name,value)		Set value of a field by name
DOEVENT(name)		Trigger an event button
DOACTION(name,bool)		Activate action by name with True or False
HASFIELD(name)	boolean	Check if field exists
HASEVENT(name)	boolean	Check if event exists
HASACTION(name)	boolean	Check if action exists

PartModule:**NAME**

Access Get only

Test string

Get the name of the module. Note that it's the same as the name given in the MODULE section of the Part.cfg file for the part.

PartModule:**PART**

Access Get only

Test *Part*

Get the *Part* that this PartModule is attached to.

PartModule:**ALLFIELDS**

Access Get only

Test *List* of strings

Get a list of all the names of KSPFields on this PartModule that the kos script is CURRENTLY allowed to get or set with :GETFIELD or :SETFIELD. Note the Security access comments below. This list can become obsolete as the game continues running depending on what the PartModule chooses to do.

PartModule:**ALLFIELDNAMES**

Access Get only

Test *List* of strings

Similar to :ALLFIELDS except that it returns the string without the formatting to make it easier to use in a script. This list can become obsolete as the game continues running depending on what the PartModule chooses to do.

PartModule:**ALLEVENTS**

Access Get only

Test *List* of strings

Get a list of all the names of KSPEvents on this PartModule that the kos script is CURRENTLY allowed to trigger with :DOEVENT. Note the Security access comments below. This list can become obsolete as the game continues running depending on what the PartModule chooses to do.

PartModule:**ALLEVENTNAMES**

Access Get only

Test *List* of strings

Similar to :ALLEVENTS except that it returns the string without the formatting to make it easier to use in a script. This list can become obsolete as the game continues running depending on what the PartModule chooses to do.

PartModule:**ALLACTIONS**

Access Get only

Test *List* of strings

Get a list of all the names of KSPActions on this PartModule that the kos script is CURRENTLY allowed to trigger with :DOACTION. Note the Security access comments below.

PartModule:**ALLACTIONNAMES**

Access Get only

Test *List* of strings

Similar to :ALLACTIONS except that it returns the string without the formatting to make it easier to use in a script. This list can become obsolete as the game continues running depending on what the PartModule chooses to do.

PartModule:**GETFIELD**(name)

Parameters

- **name** – (string) Name of the field

Returns varies

Get the value of one of the fields that this PartModule has placed onto the rightclick menu for the part. Note the Security comments below.

PartModule:**SETFIELD**(name,value)

Parameters

- **name** – (string) Name of the field

Set the value of one of the fields that this PartModule has placed onto the rightclick menu for the part. Note the Security comments below.

WARNING: This suffix is only settable for parts attached to the *CPU Vessel*

PartModule:**DOEVENT**(name)

Parameters

- **name** – (string) Name of the event

Trigger an "event button" that is on the rightclick part menu at the moment. Note the Security comments below.

WARNING: This suffix is only callable for parts attached to the *CPU Vessel*

PartModule:**DOACTION**(name,bool)

Parameters

- **name** – (string) Name of the action
- **bool** – (boolean) Value to set: True or False

Activate one of this PartModule's action-group-able actions, bypassing the action group system entirely by just activating it for this one part directly. The boolean value decides whether you are toggling the action ON or toggling it OFF. Note the Security comments below.

WARNING: This suffix is only callable for parts attached to the *CPU Vessel*

PartModule:**HASFIELD**(name)

Parameters

- **name** – (string) Name of the field

Returns boolean

Return true if the given field name is currently available for use with :GETFIELD or :SETFIELD on this PartModule, false otherwise.

PartModule:**HASEVENT**(name)

Parameters

- **name** – (string) Name of the event

Returns boolean

Return true if the given event name is currently available for use with :DOEVENT on this PartModule, false otherwise.

PartModule:**HASACTION**(name)

Parameters

- **name** – (string) Name of the action

Returns boolean

Return true if the given action name is currently available for use with :DOACTION on this PartModule, false otherwise.

Notes

In all the above cases where there is a name being passed in to :GETFIELD, :SETFIELD, :DOEVENT, or :DOACTION, the name is meant to be the name that is seen by you, the user, in the GUI screen, and NOT necessarily the actual name of the variable that the programmer of that PartModule chose to call the value behind the scenes. This is so that you can view the GUI rightclick menu to see what to call things in your script.

Note: Security and Respecting other Mod Authors

There are often a lot more fields and events and actions that a partmodule can do than are usable via kOS. In designing kOS, the kOS developers have deliberately chosen NOT to expose any "hidden" fields of a partmodule that are not normally shown to the user, without the express permission of a mod's author to do so.

The access rules that kOS uses are as follows:

KSPFields

Is this a value that the user can normally see on the right-click context menu for a part? If so, then let kOS scripts GET the value. Is this a value that the user can normally manipulate via "tweakable" adjustments on the right-click context menu for a part, AND, is that tweakable a CURRENTLY enabled one? If so, then let KOS scripts SET the value, BUT they must set it to one of the values that the GUI would normally allow, according to the following rules.

- **If the KSPField is boolean:**
  - The value must be true, false, or 0 or 1.
- **If the KSPField is an integer:**
  - The value must be a whole number.
- **If the KSPField is a floating point sliding number:**
  - The GUI for this field will be defined as a slider with a min value, a max value, with a fixed increment interval where the detents are. When setting such a value, you will be constrained to the limits of this slider. For example: If a slider is defined to have a minimum value of 2.0, a maximum value of 5.0, and a minimum allowed delta increment of 0.1:
  - If you try to set it to 0, it will instead become 2, the minimum allowed value. If you try to set it to 9, it will instead become 5, the maximum allowed value. If you try to set it to 3.14159, it will instead become 3.1, because that's rounding to the nearest increment step the slider supports.
- **If the KSPField is a string:**
  - There is currently no way to set these because kOS uses the existence of a gui tweakable as "proof" that it's okay to modify the field, and in the stock game there are no gui tweakables for string fields. This may change in the future if mods that extend the tweakables system are taken into account.

KSPEvents

Is this an event that has a GUI button associated with it that is currently visible on the right-click menu? If the answer is yes, then it will also be triggerable by kOSScripts, otherwise it won't.

KSPActions

Is this an action that the KSP user would have been allowed to set as part of an action group during building in the VAB or SPH? If so, then allow a kOS script to use it, EVEN IF it has never actually been added to an action group for this vessel.

Note: If a KSPField, KSPEvent, or KSPAction has been disallowed, often in kOS it won't even appear to be a field of the PartModule at all.

This is necessary because for some modules, the number of fields you can use are far outnumberd by the number of fields that exist but are normally hidden from view. It would become unworkable if all of the unusable ones were exposed to kOS scripts to see as fields.

Note: Which KSPFields, KSPEvents, and KSPActions exist on a PartModule can change during runtime!

A PartModule is allowed to change the look and feel of its rightclick menu fields on the fly as the game runs. Therefore a field that didn't exist the last time you looked might now exist, and might not exist again next time. The list of what fields exist is context dependant. For example, a docking port may have an event button on it called "Undock Node", that only exists when that port is connected to another port. If it's not connected, the button may be gone. Similarly, a PartModule might toggle something by using a pair of two events that swap in and out depending on the current state. For example, many of the stock lights in the game have a "Turn on" button that after it's been clicked, gets replaced with a "Turn off" button until it's clicked again. A boolean toggle with a KSPFIeld would be simpler, but until "tweakables" existed in the main game, that wasn't an option so a lot of older Partmodules still do things the old way with two KSPEvents that swap in and out.

8.3.12 Resource

A single resource value a thing holds (i.e. fuel, electric charge, etc). This is the type returned by the *Part*:RESOURCES suffix

structure `Resource`

Suffix	Type	Description
NAME	string	Resource name
AMOUNT	scalar	Amount of this resource left
DENSITY	scalar	Density of this resource
CAPACITY	scalar	Maximum amount of this resource
TOGGLEABLE	boolean	Can this tank be removed from the fuel flow
ENABLED	boolean	Is this tank currently in the fuel flow

Resource:**`NAME`**

Access Get only

Type string

The name of the resource, i.e. "LIQUIDFUEL", "ELECTRICCHARGE", "MONOPROP".

Resource:**`AMOUNT`**

Access Get only

Type scalar

The value of how much resource is left.

Resource:**`DENSITY`**

Access Get only

Type scalar

The density value of this resource, expressed in Megagrams f mass per Unit of resource. (i.e. a value of 0.005 would mean that each unit of this resource is 5 kilograms. Megagrams [metric tonnes] is the usual unit that most mass in the game is represented in.)

Resource:**CAPACITY**

Access Get only

Type scalar

What AMOUNT would be if the resource was filled to the top.

Resource:**TOGGLEABLE**

Access Get only

Type boolean

Many, but not all, resources can be turned on and off, this removes them from the fuel flow.

Resource:**ENABLED**

Access Get/Set

Type boolean

If the resource is TOGGLEABLE, setting this to false will prevent the resource from being taken out normally.

8.3.13 ScienceExperimentModule

The type of structures returned by kOS when querying a module that contains a science experiment.

Some of the science-related tasks are normally not available to kOS scripts. It is for example possible to deploy a science experiment:

```
SET P TO SHIP:PARTSNAMED("GooExperiment")[1].
SET M TO P:GETMODULE("ModuleScienceExperiment").
M:DOEVENT("observe mystery goo").
```

Hovewer, this results in a dialog being shown to the user. Only from that dialog it is possible to reset the experiment or transmit the experiment results back to Kerbin. *ScienceExperimentModule* structure introduces a few suffixes that allow the player to perform all science-related tasks without any manual intervention:

```
SET P TO SHIP:PARTSNAMED("GooExperiment")[0].
SET M TO P:GETMODULE("ModuleScienceExperiment").
M:DEPLOY.
WAIT UNTIL M:HASDATA.
M:TRANSMIT.
```

structure ScienceExperimentModule

Suffix	Type	Description
All suffixes of *PartModule*		*ScienceExperimentModule* objects are a type of *PartModule*
DEPLOY()		Deploy and run the science experiment
RESET()		Reset this experiment if possible
TRANSMIT()		Transmit the scientific data back to Kerbin
DUMP()		Discard the data
INOPERABLE	boolean	Is this experiment inoperable
RERUNNABLE	boolean	Can this experiment be run multiple times
DEPLOYED	boolean	Is this experiment deployed
HASDATA	boolean	Does the experiment have scientific data

Note: A *ScienceExperimentModule* is a type of *PartModule*, and therefore can use all the suffixes of *PartModule*.

ScienceExperimentModule:**DEPLOY**()
Call this method to deploy and run this science experiment. This method will fail if the experiment already contains scientific data or is inoperable.

ScienceExperimentModule:**RESET**()
Call this method to reset this experiment. This method will fail if the experiment is inoperable.

ScienceExperimentModule:**TRANSMIT**()
Call this method to transmit the results of the experiment back to Kerbin. This will render the experiment inoperable if it is not rerunnable. This method will fail if there is no data to send.

ScienceExperimentModule:**DUMP**()
Call this method to discard the data obtained as a result of running this experiment. This will render the experiment inoperable if it is not rerunnable.

ScienceExperimentModule:**INOPERABLE**

Access Get only

Type boolean

True if this experiment is no longer operable.

ScienceExperimentModule:**RERUNNABLE**

Access Get only

Type boolean

True if this experiment can be run multiple times.

ScienceExperimentModule:**DEPLOYED**

Access Get only

Type boolean

True if this experiment is deployed.

ScienceExperimentModule:**HASDATA**

Access Get only

Type boolean

True if this experiment has scientific data stored.

8.3.14 Sensor

The type of structures returned by *LIST SENSORS IN SOMEVARIABLE*. This is not fully understood because the type of *PartModule* in the KSP API called ModuleEnviroSensor, which all Sensors are a kind of, is not well documented. Here is an example of using *Sensor*:

```
PRINT "Full Sensor Dump:".
LIST SENSORS IN SENSELIST.

// TURN EVERY SINGLE SENSOR ON
FOR S IN SENSELIST {
  PRINT "SENSOR: " + S:TYPE.
  PRINT "VALUE:  " + S:DISPLAY.
  IF S:ACTIVE {
    PRINT "     SENSOR IS ALREADY ON.".
  } ELSE {
    PRINT "     SENSOR WAS OFF.  TURNING IT ON.".
    S:TOGGLE().
  }
}
```

structure Sensor

Suffix	Type	Description
All suffixes of *Part*		*Sensor* objects are a type of *Part*
ACTIVE	boolean	Check if this sensor is active
TYPE		
DISPLAY	string	Value of the readout
POWERCONSUMPTION	scalar	Rate of required electric charge
TOGGLE()		Call to activate/deactivate

Note: A *Sensor* is a type of *Part*, and therefore can use all the suffixes of *Part*.

Sensor:**ACTIVE**

Access Get only

Type boolean

True of the sensor is enabled. Can SET to cause the sensor to activate or de-activate.

Sensor:**TYPE**

Access Get only

Sensor:**DISPLAY**

Access Get only

Type string

The value of the sensor's readout, usualy including the units.

Sensor:**POWERCONSUMPTION**

Access Get only

Type scalar

The rate at which this sensor drains ElectricCharge.

Sensor:**TOGGLE**()

Call this method to cause the sensor to switch between active and deactivated or visa versa.

8.3.15 Stage

Contents

- EXAMPLE
- *Stage*

You access the current stage for the vessel the kOS core is attached to with the STAGE: command.

Structure

structure `Stage`

Table 8.7: Members

Suffix	Type (units)	Access	Description
READY	bool	Get only	Is the craft ready to activate the next stage.
NUMBER	scalar	Get only	The current stage number for the craft
RESOURCES	*List*	Get only	the *List* of *Resource* in the current stage

Stage:**`READY`**

Access Get only

Type bool

Kerbal Space Program enforces a small delay between staging commands, this is to allow the last staging command to complete. This bool value will let you know if kOS can activate the next stage.

Stage:**`NUMBER`**

Access Get only

Type scalar

Every craft has a current stage, and that stage is represented by a number, this is it!

Stage:**`Resources`**

Access Get

Type *List*

This is a collection of the available *Resource* for the current stage.

8.3.16 Vessel

All vessels share a structure. To get a variable referring to any vessel you can do this:

```
// Get a vessel by it's name.
// The name is Case Sensitive.
SET MY_VESS TO VESSEL("Some Ship Name").

// Save the current vessel in a variable,
// in case the current vessel changes.
SET MY_VESS TO SHIP.

// Save the target vessel in a variable,
```

```
// in case the target vessel changes.
SET MY_VESS TO TARGET.
```

Note: New in version 0.13: A vessel is now a type of *Orbitable*. Much of what a Vessel can do can now by done by any orbitable object. The documentation for those abilities has been moved to the *orbitable page*.

structure `Vessel`

Suffix	Type	Description
Every suffix of *Orbitable*		
CONTROL	*Control*	Raw flight controls
BEARING	scalar (deg)	relative heading to this vessel
HEADING	scalar (deg)	Absolute heading to this vessel
MAXTHRUST	scalar	Sum of active maximum thrusts
MAXTHRUSTAT(pressure)	scalar	Sum of active maximum thrusts at the given atmospheric pressure
AVAILABLETHRUST	scalar	Sum of active limited maximum thrusts
AVAILABLETHRUSTAT(pressure)	scalar	Sum of active limited maximum thrusts at the given atmospheric pre
FACING	*Direction*	The way the vessel is pointed
MASS	scalar (metric tons)	Mass of the ship
WETMASS	scalar (metric tons)	Mass of the ship fully fuelled
DRYMASS	scalar (metric tons)	Mass of the ship with no resources
VERTICALSPEED	scalar (m/s)	How fast the ship is moving "up"
GROUNDSPEED	scalar (m/s)	How fast the ship is moving "horizontally"
AIRSPEED	scalar (m/s)	How fast the ship is moving relative to the air
TERMVELOCITY	scalar (m/s)	terminal velocity of the vessel
SHIPNAME	string	The name of the vessel
NAME	string	Synomym for SHIPNAME
STATUS	string	Current ship status
TYPE	string	Ship type
ANGULARMOMENTUM	*Vector*	In *SHIP_RAW*
ANGULARVEL	*Vector*	In *SHIP_RAW*
SENSORS	*VesselSensors*	Sensor data
LOADED	Boolean	loaded into KSP physics engine or "on rails"
LOADDISTANCE	*LoadDistance*	the *LoadDistance* object for this vessel
ISDEAD	Boolean	True if the vessel refers to a ship that has gone away.
PATCHES	*List*	*Orbit* patches
ROOTPART	*Part*	Root *Part* of this vessel
PARTS	*List*	all *Parts*
DOCKINGPORTS	*List*	all *DockingPorts*
ELEMENTS	*List*	all *Elements*
RESOURCES	*List*	all *AggrgateResources*
PARTSNAMED(name)	*List*	*Parts* by *NAME*
PARTSTITLED(title)	*List*	*Parts* by *TITLE*
PARTSTAGGED(tag)	*List*	*Parts* by *TAG*
PARTSDUBBED(name)	*List*	*Parts* by *NAME*, *TITLE* or *TAG*
MODULESNAMED(name)	*List*	*PartModules* by *NAME*
PARTSINGROUP(group)	*List*	*Parts* by action group
MODULESINGROUP(group)	*List*	*PartModules* by action group
ALLPARTSTAGGED()	*List*	*Parts* that have non-blank nametags
CREWCAPACITY	scalar	Crew capacity of this vessel

Continued on next

Table 8.8 – continued from previous page

Suffix	Type	Description
CREW()	*List*	all *CrewMembers*

Vessel:**CONTROL**

Type *Control*

Access Get only

The structure representing the raw flight controls for the vessel.

WARNING: This suffix is only gettable for *CPU Vessel*

Vessel:**BEARING**

Type scalar

Access Get only

relative compass heading (degrees) to this vessel from the *CPU Vessel*, taking into account the CPU Vessel's own heading.

Vessel:**HEADING**

Type scalar

Access Get only

absolute compass heading (degrees) to this vessel from the *CPU Vessel*

Vessel:**MAXTHRUST**

Type scalar

Access Get only

Sum of all the *engines' MAXTHRUSTs* of all the currently active engines In Kilonewtons.

Vessel:**MAXTHRUSTAT**(pressure)

Parameters

- **pressure** – atmospheric pressure (in standard Kerbin atmospheres)

Return type scalar (kN)

Sum of all the *engines' MAXTHRUSTATs* of all the currently active engines In Kilonewtons at the given atmospheric pressure. Use a pressure of 0 for vacuum, and 1 for sea level (on Kerbin).

Vessel:**AVAILABLETHRUST**

Type scalar

Access Get only

Sum of all the *engines' AVAILABLETHRUSTs* of all the currently active engines taking into acount their throttlelimits. Result is in Kilonewtons.

Vessel:**AVAILABLETHRUSTAT**(pressure)

Parameters

- **pressure** – atmospheric pressure (in standard Kerbin atmospheres)

Return type scalar (kN)

Sum of all the *engines' AVAILABLETHRUSTATs* of all the currently active engines taking into acount their throttlelimits at the given atmospheric pressure. Result is in Kilonewtons. Use a pressure of 0 for vacuum, and 1 for sea level (on Kerbin).

Vessel:**FACING**

Type *Direction*

Access Get only

The way the vessel is pointed.

Vessel:**MASS**

Type scalar (metric tons)

Access Get only

The mass of the ship

Vessel:**WETMASS**

Type scalar (metric tons)

Access Get only

The mass of the ship if all resources were full

Vessel:**DRYMASS**

Type scalar (metric tons)

Access Get only

The mass of the ship if all resources were empty

Vessel:**VERTICALSPEED**

Type scalar (m/s)

Access Get only

How fast the ship is moving. in the "up" direction relative to the SOI Body's sea level surface.

Vessel:**GROUNDSPEED**

Type scalar (m/s)

Access Get only

How fast the ship is moving in the two dimensional plane horizontal to the SOI body's sea level surface. The vertical component of the ship's velocity is ignored when calculating this.

Note: New in version 0.18.

The old name for this value was SURFACESPEED. The name was changed because it was confusing before. "surface speed" implied it's the scalar magnitude of "surface velocity", but it wasn't, because of how it ignores the vertical component.

Vessel:**AIRSPEED**

Type scalar (m/s)

Access Get only

How fast the ship is moving relative to the air. KSP models atmosphere as simply a solid block of air "glued" to the planet surface (the weather on Kerbin is boring and there's no wind). Therefore airspeed is generally the same thing as as the magnitude of the surface velocity.

Vessel:**TERMVELOCITY**

Type scalar (m/s)

Access Get only

terminal velocity of the vessel in freefall through atmosphere, based on the vessel's current altitude above sea level, and its drag properties. Warning, can cause values of Infinity if used in a vacuum, and kOS sometimes does not let you store Infinity in a variable.

Vessel:**SHIPNAME**

Type string

Access Get/Set

The name of the vessel as it appears in the tracking station. When you set this, it cannot be empty.

Vessel:**NAME**
Same as *Vessel:SHIPNAME*.

Vessel:**STATUS**

Type string

Access get only

The current status of the vessel possible results are: *LANDED*, *SPLASHED*, *PRELAUNCH*, *FLYING*, *SUB_ORBITAL*, *ORBITING*, *ESCAPING* and *DOCKED*.

Vessel:**TYPE**

Type string

Access Get/Set

The ship's type as described on the KSP wiki.

Vessel:**ANGULARMOMENTUM**

Type *Direction*

Access Get only

Given in *SHIP_RAW* reference frame. The vector represents the axis of the rotation, and its magnitude is the angular momentum of the rotation, which varies not only with the speed of the rotation, but also with the angular inertia of the vessel.

Note: Changed in version 0.15.4: This has been changed to a vector, as it should have been all along.

Vessel:**ANGULARVEL**

Type *Direction*

Access Get only

Given in *SHIP_RAW* reference frame. The vector represents the axis of the rotation, and its magnitude is the speed of that rotation (Presumably in degrees per second? This is not documented in the KSP API and may take some experimentation to discover if it's radians or degrees).

Note: Changed in version 0.15.4: This has been changed to a vector, as it should have been all along.

Vessel:**SENSORS**

Type *VesselSensors*

Access Get only

Structure holding summary information of sensor data for the vessel

Vessel:**LOADED**

Type Boolean

Access Get only

true if the vessel is fully loaded into the complete KSP physics engine (false if it's "on rails").

Vessel:**LOADDISTANCE**

Type *LoadDistance*

Access Get only

Returns the load distance object for this vessel. The suffixes of this object may be adjusted to change the loading behavior of this vessel. Note: these settings are not persistant across flight instances, and will reset the next time you lauch a craft from an editor or the tracking station.

Vessel:**ISDEAD**

Type Boolean

Access Get only

It is possible to have a variable that refers to a vessel that doesn't exist in the Kerbal Space Program universe anymore, but did back when you first got it. For example: you could do: SET VES TO VESSEL("OTHER"). WAIT 10. And in that intervening waiting time, the vessel might have crashed into the ground. Checking :ISDEAD lets you see if the vessel that was previously valid isn't valid anymore.

Vessel:**PATCHES**

Type *List*

Access Get only

The list of *orbit patches* that describe this vessel's current travel path based on momentum alone with no thrusting changes. If the current path has no transitions to other bodies, then this will be a list of only one orbit. If the current path intersects other bodies, then this will be a list describing the transitions into and out of the intersecting body's sphere of influence. SHIP:PATCHES[0] is always exactly the same as SHIP:OBT, SHIP:PATCHES[1] is the same as SHIP:OBT:NEXTPATCH, SHIP:PATCHES[2] is the same as SHIP:OBT:NEXTPATCH:NEXTPATCH, and so on. Note that you will only see as far into the future as your KSP settings allow. (See the setting CONIC_PATCH_LIMIT in your settings.cfg file)

Vessel:**ROOTPART**

Type *Part*

Access Get only

The ROOTPART is usually the first *Part* that was used to begin the ship design - the command core. Vessels in KSP are built in a tree-structure, and the first part that was placed is the root of that tree. It is possible to change the root part in VAB/SPH by using Root tool, so ROOTPART does not always point to command core or command pod. Vessel:ROOTPART may change in flight as a result of docking/undocking or decoupling of some part of a Vessel.

Vessel:**PARTS**

Type *List* of *Part* objects

Access Get only

A List of all the *parts* on the vessel. `SET FOO TO SHIP:PARTS.` has exactly the same effect as `LIST PARTS IN FOO.`. For more information, see *ship parts and modules*.

Vessel:**DOCKINGPORTS**

Type *List* of *DockingPort* objects

Access Get only

A List of all the *docking ports* on the Vessel.

Vessel:**ELEMENTS**

Type *List* of *Element* objects

Access Get only

A List of all the *elements* on the Vessel.

Vessel:**RESOURCES**

Type *List* of *AggregateResource* objects

Access Get only

A List of all the *AggregateResources* on the vessel. `SET FOO TO SHIP:RESOURCES.` has exactly the same effect as `LIST RESOURCES IN FOO.`.

Vessel:**PARTSNAMED**(name)

Parameters

- **name** – (string) Name of the parts

Returns *List* of *Part* objects

Part:NAME. The matching is done case-insensitively. For more information, see *ship parts and modules*.

Vessel:**PARTSTITLED**(title)

Parameters

- **title** – (string) Title of the parts

Returns *List* of *Part* objects

Part:TITLE. The matching is done case-insensitively. For more information, see *ship parts and modules*.

Vessel:**PARTSTAGGED**(tag)

Parameters

- **tag** – (string) Tag of the parts

Returns *List* of *Part* objects

Part:TAG value. The matching is done case-insensitively. For more information, see *ship parts and modules*.

Vessel:**PARTSDUBBED**(name)

Parameters

- **name** – (string) name, title or tag of the parts

Returns *List* of *Part* objects

name regardless of whether that name is the Part:Name, the Part:Tag, or the Part:Title. It is effectively the distinct union of :PARTSNAMED(val), :PARTSTITLED(val), :PARTSTAGGED(val). The matching is done case-insensitively. For more information, see *ship parts and modules*.

Vessel:**MODULESNAMED**(name)

Parameters

- **name** – (string) Name of the part modules

Returns *List* of *PartModule* objects

match the given name. The matching is done case-insensitively. For more information, see *ship parts and modules*.

Vessel:**PARTSINGROUP**(group)

Parameters

- **group** – (integer) the action group number

Returns *List* of *Part* objects

one action triggered by the given action group. For more information, see *ship parts and modules*.

Vessel:**MODULESINGROUP**(group)

Parameters

- **group** – (integer) the action group number

Returns *List* of *PartModule* objects

have at least one action triggered by the given action group. For more information, see *ship parts and modules*.

Vessel:**ALLPARTSTAGGED**()

Returns *List* of *Part* objects

nametag on them of any sort that is nonblank. For more information, see *ship parts and modules*.

Vessel:**CREWCAPACITY**

Type scalar

Access Get only

crew capacity of this vessel

Vessel:**CREW**()

Returns *List* of *CrewMember* objects

list of all *kerbonauts* aboard this vessel

8.3.17 VesselSensors

When you ask a Vessel to tell you its *Vessel:SENSORS* suffix, it returns an object of this type. It is a snapshot of sensor data taken at the moment the sensor reading was requested.

Note: These values are only enabled if you have the proper type of sensor on board the vessel. If you don't have a thermometer, for example, then the `:TEMP` suffix will always read zero.

If you store this in a variable and wait, the numbers are frozen in time and won't change as the vessel's condition changes.

structure VesselSensors

Table 8.9: Members

Suffix	Type	Description
ACC	*Vector*	Acceleration experienced by the *Vessel*
PRES	scalar	Atmospheric Pressure outside this *Vessel*
TEMP	scalar	Temperature outside this *Vessel*
GRAV	*Vector* (g's)	Gravitational acceleration
LIGHT	scalar	Sun exposure on the solar panels of this *Vessel*

VesselSensors:**ACC**

Access Get only

Type *Vector*

Accelleration the vessel is undergoing. A combination of both the gravitational pull and the engine thrust.

VesselSensors:**PRES**

Access Get only

Type scalar

The current pressure of this ship.

VesselSensors:**TEMP**

Access Get only

Type scalar

The current temperature.

VesselSensors:**GRAV**

Access Get only

Type *Vector*

Magnitude and direction of gravity acceleration where the vessel currently is. Magnitude is expressed in "G"'s (multiples of 9.802 m/s^2).

VesselSensors:**LIGHT**

Access Get only

Type scalar

The total amount of sun exposure that exists here - only readable if there are solar panels on the vessel.

8.4 Waypoints

Waypoints are the location markers you can see on the map view showing you where contracts are targeted for. With this structure, you can obtain coordinate data for the locations of these waypoints.

WAYPOINT(name)

Parameters

- **name** – (string) Name of the waypoint as it appears on the map or in the contract description

Returns *Waypoint*

This creates a new Waypoint from a name of a waypoint you read from the contract paramters. Note that this only works on contracts you've accpted. Waypoints for proposed contracts haven't accepted yet do not actually work in kOS.

SET spot TO WAYPOINT("herman's folly beta").

The name match is case-insensitive.

ALLWAYPOINTS()

Returns *List* of *Waypoint*

This creates a *List* of *Waypoint* structures for all accepted contracts. Waypoints for proposed contracts you haven't accepted yet do not appear in the list.

structure Waypoint

Table 8.10: Members

Suffix	Type
NAME	string
BODY	*BodyTarget*
GEOPOSITION	*GeoCoordinates*
POSITION	*Vector*
ALTITUDE	scalar
AGL	scalar
NEARSURFACE	boolean
GROUNDED	boolean
INDEX	scalar
CLUSTERED	boolean

Waypoint:**NAME**

Type string

Access Get only

Name of waypoint as it appears on the map and contract

Waypoint:**BODY**

Type *BodyTarget*

Access Get only

Celestial body the waypoint is attached to

Waypoint:**GEOPOSITION**

Type GeoCoordinates

Access Get only

The LATLNG of this waypoint

Waypoint:**POSITION**

Type Vector

Access Get only

The Vector position of this waypoint in 3D space, in ship-raw coords.

Waypoint:**ALTITUDE**

Type scalar

Access Get only

Altitude of waypoint **above "sea" level**. Warning, this a point somewhere in the midst of the contract altitude range, not the edge of the altitude range. It corresponds towhere the marker tip hovers on the map, which is not actually at the very edge of the contract condition's range. It represents a typical midling location inside the contract's altitude range.

Waypoint:**AGL**

Type scalar

Access Get only

Altitude of waypoint **above ground**. Warning, this a point somewhere in the midst of the contract altitude range, not the edge of the altitude range. It corresponds to where the marker tip hovers on the map, which is not actually at the very edge of the contract condition's range. It represents a typical midling location inside the contract's altitude range.

Waypoint:**NEARSURFACE**

Type boolean

Access Get only

True if waypoint is a point near or on the body rather than high in orbit.

Waypoint:**GROUNDED**

Type boolean

Access Get only

True if waypoint is actually glued to the ground.

Waypoint:**INDEX**

Type scalar

Access Get only

The integer index of this waypoint amongst its cluster of sibling waypoints. In other words, when you have a cluster of waypoints called "Somewhere Alpha", "Somewhere Beta", and "Somewhere Gamma", then the alpha site has index 0, the beta site has index 1 and the gamma site has index 2. When Waypoint:CLUSTERED is false, this value is zero but meaningless.

Waypoint:**CLUSTERED**

Type boolean

Access Get only

True if this waypoint is part of a set of clustered waypoints with greek letter names appended (Alpha, Beta, Gamma, etc). If true, there should be a one-to-one correspondence with the greek letter name and the :INDEX suffix. (0 = Alpha, 1 = Beta, 2 = Gamma, etc).

8.5 Configurations and Miscellany

8.5.1 Colors

Any place you need to specify a color in the game (at the moment this is just with *VECDRAW*, *HIGHLIGHT*. and HUDTEXT) You do so with a rgba color structure defined as follows:

Method 1: Use one of these pre-arranged named colors:

- **RED**
- **GREEN**
- **BLUE**
- **YELLOW**
- **CYAN**
- **MAGENTA**

  –PURPLE
  (Alias of *MAGENTA*)
- **WHITE**
- **BLACK**

RGB (r,g,b)
This global function creates a color from red green and blue values:

```
SET myColor TO RGB(r,g,b).
```

where:

r A floating point number from 0.0 to 1.0 for the red component.

g A floating point number from 0.0 to 1.0 for the green component.

b A floating point number from 0.0 to 1.0 for the blue component.

RGBA (r,g,b,a)
Same as *RGB()* but with an alpha (transparency) channel:

```
SET myColor TO RGBA(r,g,b,a).
```

`r, g, b` are the same as above.

a A floating point number from 0.0 to 1.0 for the alpha component. (1.0 means opaque, 0.0 means invisibly transparent).

structure RGBA

Table 8.11: Members

Suffix	Type	Description
:R or :RED	scalar	the red component of the color
:G or :GREEN	scalar	the green component of the color
:B or :BLUE	scalar	the blue component of the color
:A or :ALPHA	scalar	the alpha (how opaque: 1 = opaque, 0 = transparent) component of the color
:HTML or :HEX	string	the color rendered into a HTML tag string i.e. “#ff0000”. This format ignores the alpha channel and treats all colors as opaque.

Examples:

```
SET myarrow TO VECDRAW.
SET myarrow:VEC to V(10,10,10).
SET myarrow:COLOR to YELLOW.
SET mycolor TO YELLOW.
SET myarrow:COLOR to mycolor.
SET myarrow:COLOR to RGB(1.0,1.0,0.0).

// COLOUR spelling works too
SET myarrow:COLOUR to RGB(1.0,1.0,0.0).

// half transparent yellow.
SET myarrow:COLOR to RGBA(1.0,1.0,0.0,0.5).

PRINT GREEN:HTML. // prints #00ff00
```

HSV (h,s,v)

This global function creates a color from hue, saturation and value:

```
SET myColor TO HSV(h,s,v).

`More Information about HSV <http://en.wikipedia.org/wiki/HSL_and_HSV>`_,
```

where:

h A floating point number from 0.0 to 360.0 for the hue component.

s A floating point number from 0.0 to 1.0 for the saturation component.

v A floating point number from 0.0 to 1.0 for the value component.

HSVA (h,s,v,a)

Same as *HSV()* but with an alpha (transparency) channel:

```
SET myColor TO HSVA(h,s,v,a).
```

`h, s, v` are the same as above.

a A floating point number from 0.0 to 1.0 for the alpha component. (1.0 means opaque, 0.0 means invisibly transparent).

structure HSVA

The HSVA structure contains all of the suffixes from the RGBA structure in addition to these

Table 8.12: Members

Suffix	Type	Description
:H or :HUE	scalar	the hue component of the color. It is a value from 0.0 to 360.0
:S or :SATURATION	scalar	the saturation component of the color. It has a value from 0.0 to 1.0
:V or :VALUE	scalar	the value component of the color. It has a value from 0.0 to 1.0

Examples:

```
SET myarrow TO VECDRAW.
SET myarrow:VEC to V(10,10,10).
SET myarrow:COLOR to HSV(60,1,1). // Yellow
SET myarrow:COLOR:S to 0.5. // Light yellow
SET myarrow:COLOR:H to 0. // pink
```

8.5.2 Configuration of kOS

structure `Config`

Config is a special structure that allows your kerboscript programs to set or get the values stored in the kOS plugin's config file.

The options here can also be set by using the user interface panel shown here. This control panel is part of the *App Control Panel*

In either case, whether the setting is changed via the GUI panel, or via script code, these are settings that **affect the kOS mod in all saved games** as soon as the change is made. It's identical to editing the config file in the kOS installation directory, and in fact will actually change that file the next time the game saves its state.

Table 8.13: Members (all Gettable and Settable)

Suffix	Type	Default	Description
IPU	integer	150	Instructions per update
UCP	boolean	False	Use compressed persistence
STAT	boolean	False	Print statistics to screen
RT2	boolean	False	Enable RemoteTech2 integration
ARCH	boolean	False	Start on archive (instead of volume 1)
SAFE	boolean	False	Enable safe mode
AUDIOERR	boolean	False	Enable sound effect on kOS error
VERBOSE	boolean	False	Enable verbose exceptions
TELNET	boolean	False	activate the telnet server
TPORT	integer	5410	set the port the telnet server will run on
LOOPBACK	boolean	True	Force the telnet server to use loopback (127.0.0.1) address

Config:**`IPU`**

Access Get/Set

Type integer. range = [50,2000]

Configures the `InstructionsPerUpdate` setting.

This is the number of kRISC psuedo-machine-langauge instructions that each kOS CPU will attempt to execute from the main program per *physics update tick*.

This value is constrained to stay within the range [50..2000]. If you set it to a value outside that range, it will reset itself to remain in that range.

Config:**`UCP`**

Access Get/Set

Type boolean

Configures the `UseCompressedPersistence` setting.

If true, then the contents of the kOS local volume 'files' stored inside the campaign save's persistence file will be stored using a compression algorithm that has the advantage of making them take less space, but at the cost of making the data impossible to decipher with the naked human eye when looking at the persistence file.

Config:**`STAT`**

Access Get/Set

Type boolean

Configures the `ShowStatistics` setting.

If true, then executing a program will log numbers to the screen showing execution speed statistics.

Config:**RT2**

Access Get/Set

Type boolean

Configures the `EnableRT2Integration` setting.

If true, then the kOS mod will attempt to interact with the Remote Tech 2 mod, letting RT2 make decisions about whether or not a vessel is within communications range rather than having kOS use its own more primitive algorithm for it.

Due to a long stall in the development of the RT2 mod, this setting should still be considered experimental at this point.

Config:**ARCH**

Access Get/Set

Type boolean

Configures the `StartOnArchive` setting.

If true, then when a vessel is first loaded onto the launchpad or runway, the initial default volume will be set to volume 0, the archive, instead of volume 1, the local drive.

Config:**SAFE**

Access Get/Set

Type boolean

Configures the `EnableSafeMode` setting.

If true, then it enables the following error messages:

```
Tried to push NaN into the stack.
Tried to push Infinity into the stack.
```

They will be triggered any time any mathematical operation would result in something that is not a real number, such as dividing by zero, or trying to take the square root of a negative number, or the arccos of a number larger than 1. Performing such an operation will immediately terminate the program with one of the error messages shown above.

If false, then these operations are permitted, but the result may lead to code that does not function correctly if you are not careful about how you use it. Using a value that is not a real number may result in freezing Kerbal Space Program itself if that value is used in a variable that is passed into Kerbal Space Program's API routines. KSP's own API interface does not seem to have any protective checks in place and will faithfully try to use whatever values its given.

Config:**AUDIOERR**

Access Get/Set

Type boolean

Configures the `AudibleExceptions` setting.

If true, then it enables a mode in which errors coming from kOS will generte a sound effect of a short little warning bleep to remind you that an exception occurred. This can be useful when you are flying hands-off and need to realize your autopilot script just died so you can take over.

Config:**VERBOSE**

Access Get/Set

Type boolean

Configures the `VerboseExceptions` setting.

If true, then it enables a mode in which errors coming from kOS are very long and verbose, trying to explain every detail of the problem.

Config:**`TELNET`**

Access Get/Set

Type boolean

Configures the `EnableTelnet` setting.

When set to true, it activates a kOS telnet server in game that allows you to connect external terminal programs like Putty and Xterm to it. Turning the option off or on immediately toggles the server. (When you change it from false to true, it will start the server right then. When you change it from true to false, it will stop the server right then.) Therefore **to restart the server** after changing a setting like `TPORT`, DO this:

```
// Restart telnet server:
SET CONFIG:TELNET TO FALSE.
WAIT 0.5. // important to give kOS a moment to notice and kill the old server.
SET CONFIG:TELNET TO TRUE.
```

Of course, you can do the equivalent of that by using the GUI config panel and just clicking the button off then clicking it on.

Config:**`TPORT`**

Access Get/Set

Type boolean

Configures the `TelnetPort` setting.

Changes the TCP/IP port number that the kOS telnet server in game will listen to.

To make the change take effect you may have to stop, then restart the telnet server, as described above.

Config:**`LOOPBACK`**

Access Get/Set

Type boolean

Configures the `TelnetLoopback` setting.

If true, then it tells the kOS telnet server in game to refuse to use the computer's actual IP address, and instead use the loopback address (127.0.0.1). This is the default mode the kOS mod ships in, in order to make it impossible get external access to your computer.

To make the change take effect you may have to stop, then restart the telnet server, as described above.

8.5.3 File Information

File name and size information. You can obtain a list of values of type FileInfo using the *LIST FILES* command.

structure `FileInfo`

Table 8.14: Members

Suffix	Type	Description
NAME	string	Name of the file including extension
FILETYPE	string	Type of the file
SIZE	integer (bytes)	Size of the file

FileInfo:**NAME**

Access Get only

Type string

name of the file, including its file extension.

FileInfo:**FILETYPE**

Access Get only

Type string

Type of the file as a string. Can be one of the following:

ASCII A file containing ASCII text, like the result of a LOG command.

KERBOSCRIPT (unimplemented) A type of ASCII file containing Kerboscript ascii code. At the moment this type does not ever get returned. You will always get files of type ASCII instead.

KSM A type of file containing KerboMachineLanguage compiled code, that was created from the *COMPILE command*.

UNKNOWN Any other type of file.

FileInfo:**SIZE**

Access Get only

Type scalar

size of the file, in bytes.

8.5.4 Part Highlighting

Being able to *color* tint a *part* or a collection of parts can be a powerful visualization to show their placement and status. The part highlighting structure is defined as follows:

HIGHLIGHT(p,c)

This global function creates a part highlight:

```
SET foo TO HIGHLIGHT(p,c).
```

where:

p A single *part*, a list of parts or an *element*

c A *color*

structure HIGHLIGHT

Table 8.15: Members

Suffix	Type	Description
:COLOR	*color*	the color that will be used by the highlight
:ENABLED	bool	controls the visibility of the highlight

Example:

```
list elements in elist.

// Color the first element pink
SET foo TO HIGHLIGHT( elist[0], HSV(350,0.25,1) ).

// Turn the highlight off
SET foo:ENABLED TO FALSE


// Turn the highlight back on
SET foo:ENABLED TO TRUE
```

8.5.5 Iterator

An iterator can be obtained from *List:ITERATOR*. Once a *List* has given you an *Iterator* object, you can use it to access elements inside the *List*. An ITERATOR is a generic computer programming concept. In the general case it's a variable type that allows you to get the value at a position in some collection, as well as increment to the next item in the collection in order to operate on all objects in the collection one at a time. In kOS it operates on *Lists*.

A loop using an *Iterator* on a *List* might look like this:

```
// Starting with a list that was built like this
SET MyList To LIST( "Hello", "Aloha", "Bonjour").

// It could be looped over like this
SET MyCurrent TO MyList:ITERATOR.
MyCurrent:RESET().
PRINT "After reset, position = " + MyCurrent:INDEX.
UNTIL NOT MyCurrent:NEXT {
    PRINT "Item at position " + MyIter:INDEX + " is [" + MyIter:VALUE + "].".
}
```

Which would result in this output:

```
After reset, position = -1.
Item at position 0 is [Hello].
Item at position 1 is [Aloha].
Item at position 2 is [Bonjour].
```

structure `Iterator`

Table 8.16: Members

Suffix	Type	Description
RESET		Rewind to the just before the beginning
NEXT	boolean	Move iterator to the next item
ATEND	boolean	Check if iterator is at the end of the list
INDEX	integer	Current index starting from zero
VALUE	varies	The object currently being pointed to

Iterator:**RESET**()
Call this to rewind the iterator to just before the beginning of the list. After a call to *Iterator:RESET*, the iterator must be moved with *Iterator:NEXT* before it gets to the first value in the list.

Iterator:**NEXT**()

Returns boolean

Call this to move the iterator to the next item in the list. Returns true if there is such an item, or false if no such item exists because it's already at the end of the list.

Iterator:**ATEND**

Access Get only

Type boolean

Returns true if the iterator is at the end of the list and therefore cannot be "NEXTed", false otherwise.

Iterator:**INDEX**

Access Get only

Type integer

Returns the numerical index of how far you are into the list, starting the counting at 0 for the first item in the list. The last item in the list is numbered N-1, where N is the number of items in the list.

Note: If you have just used *Iterator:RESET* or have just created the ITERATOR, then the value of *Iterator:INDEX* is -1. It only becomes 0 after the first call to *Iterator:NEXT*.

Iterator:**VALUE**

Access Get only

Type varies

Returns the thing stored at the current position in the list.

8.5.6 KUniverse 4th wall methods

structure KUniverse
KUniverse is a special structure that allows your kerboscript programs to access some of the functions that break the "4th Wall". It serves as a place to access object directly connected to the KSP game itself, rather than the interaction with the KSP world (vessels, planets, orbits, etc.).

Table 8.17: Members and Methods

Suffix	Type	Get/Set	Description
CANREVERT	boolean	Get	Is any revert possible?
CANREVERTTOLAUNCH	boolean	Get	Is revert to launch possible?
CANREVERTTOEDITOR	boolean	Get	Is revert to editor possible?
REVERTTOLAUNCH	none	Method	Invoke revert to launch
REVERTTOEDITOR	none	Method	Invoke revert to editor
REVERTTO(name)	string	Method	Invoke revert to the named editor
ORIGINEDITOR	string	Get	Returns the name of this vessel's editor, "SPH" or "VAB".

KUniverse:**CANREVERT**

Access Get

Type boolean.

Returns true if either revert to launch or revert to editor is available. Note: either option may still be unavailable, use the specific methods below to check the exact option you are looking for.

KUniverse:**CANREVERTTOLAUNCH**

Access Get

Type boolean.

Returns true if either revert to launch is available.

KUniverse:**CANREVERTTOEDITOR**

Access Get

Type boolean.

Returns true if either revert to the editor is available. This tends to be false after reloading from a saved game where the vessel was already in existence in the saved file when you loaded the game.

KUniverse:**REVERTTOLAUNCH**

Access Method

Type None.

Initiate the KSP game's revert to launch function. All progress so far will be lost, and the vessel will be returned to the launch pad or runway at the time it was initially launched.

KUniverse:**REVERTTOEDITOR**

Access Method

Type None.

Initiate the KSP game's revert to editor function. The game will revert to the editor, as selected based on the vessel type.

KUniverse:**REVERTTO**(editor)

Parameters

- **editor** – The editor identifier

Returns none

Revert to the provided editor. Valid inputs are *"VAB"* and *"SPH"*.

KUniverse:**ORIGINEDITOR**

Access Get

Type string.

Returns the name of the orginating editor based on the vessel type. The value is one of:

•"SPH" for things built in the space plane hangar,

•"VAB" for things built in the vehicle assembly building.

•"" (empty string) for cases where the vehicle cannot remember its editor (when KUniverse:CANREVERTTOEDITOR is false.)

KUniverse:**DEFAULTLOADDISTANCE**

Access Get

Type *LoadDistance*.

Get or set the default loading distances for vessels loaded in the future. Note: this setting will not affect any vessel currently in the universe for the current flight session. It will take effect the next time you enter a flight scene from the editor or tracking station, even on vessels that have already existed beforehand. The act of loading a new scene causes all the vessels in that scene to inherit these new default values, forgetting the values they may have had before.

(To affect the value on a vessel already existing in the current scene you have to use the :LOADDISTANCE suffix of the Vessel structure.)

KUniverse:**ACTIVEVESSEL**

Access Get/Set

Type *Vessel*.

Returns the active vessel object and allows you to set the active vessel. Note: KSP will not allow you to change vessels by default when the current active vessel is in the atmosphere or under acceleration. Use **:method:'FORCEACTIVE'** under those circumstances.

KUniverse:**FORCEACTIVE**(vessel)

Parameters

- **vessel** – *Vessel* to switch to.

Returns none

Force KSP to change the active vessel to the one specified. Note: Switching the active vessel under conditions that KSP normally disallows may cause unexpected results on the initial vessel. It is possible that the vessel will be treated as if it is re-entering the atmosphere and deleted.

Examples:

Switch to an active vessel called "vessel 2":

```
SET KUNIVERSE:ACTIVEVESSEL TO VESSEL("vessel 2").
```

Revert to VAB, but only if allowed:

```
PRINT "ATTEMPTING TO REVERT TO THE Vehicle Assembly Building."
IF KUNIVERSE:CANREVERTTOEDITOR {
  IF KUNIVERSE:ORIGINEDITOR = "VAB" {
    PRINT "REVERTING TO VAB.".
    KUNIVERSE:REVERTTOEDITOR().
  } ELSE {
    PRINT "COULD REVERT, But only to space plane hanger, so I won't.".
  }
} ELSE {
  PRINT "Cannot revert to any editor.".
}
```

8.5.7 Lexicon

A *Lexicon* is an associative array, and is similar to the *LIST type*. If you are an experienced programmer who already knows what "associative array" means, you can probably skip this section and go to the next part of the page further down, otherwise read on:

In a normal array, or in kerboscript's **:ref:'LIST type <list>'_** you specify which item in the list you want by giving its integer position in the list.

But in a *Lexicon*, you store pairs of keys and values, where the keys can be any type of thing you like, not just integers, and then you specify which item you want by using that key's value.

Here's a small example:

```
set arr to lexicon().
arr:add( "ABC", 1234.1 ).
arr:add( "Carmine", 4.1 ).
print arr["ABC"]. // prints 1234.1
print arr["Carmine"]. // prints 4.1
```

Notice how it looks a lot like a list, but the values in the index brackets are strings instead of integers. This is the most common use of a lexicon, to use strings as the key index values (and in fact why it's called "lexicon"). However you can really use any value you feel like for the keys - strings, RGB colors, numbers, etc.

Lexicons are case-insensitive

One important difference between Lexicons in kerboscript and associative arrays in most other languages is that kerboscript Lexicons use case-insensitive keys by default (when the keys are strings). This behaviour can be changed with the `:CASESENSITIVE` flag described below.

Constructing a lexicon

If you wish to make your own lexicon from scratch you can do so with the LEXICON() built-in function.

// Make an empty lexicon with zero items in it: set mylexicon to lexicon().

The keys and the values of a lexicon can be any type you feel like, and do not need to be of a homogeneous type.

Structure

structure `Lexicon`

Lexicon:**`ADD`**(key, value)

Parameters

- **`key`** – (any type) a unique key
- **`value`** – (any type) a value that is to be associated to the key

Adds an additional pair to the lexicon.

Lexicon:**`CASESENSITIVE`**

Type Boolean

Access Get or Set

The case sensitivity behaviour of the lexicon when the keys are strings. By default, all kerboscript lexicons use case-insensitive keys, at least for those keys that are string types, meaning that mylexicon["AAA"] means the same exact thing as mylexicon["aaa"]. If you do not want this behaviour, and instead want the key "AAA" to be different from the key "aaa", you can set this value to true.

Be aware, however, that if you change this, it has the side effect of *clearing out* the entire contents of the lexicon. This is done so as to avoid any potential clashes when the rules about what constitutes a duplicate key changed after the lexicon was already populated. Therefore you should probably only set this on a brand new lexicon, right after you've created it, and never change it after that.

Lexicon:**`CASE`**

Type Boolean

Access Get or Set

Synonym for CASESENSITIVE (see above).

Lexicon:**REMOVE**(key)

Parameters

- **key** – the keyvalue of the pair to be removed

Remove the pair with the given key from the lexicon.

Lexicon:**CLEAR**()
Removes all of the pairs from the lexicon. Making it empty.

Lexicon:**LENGTH**

Type integer

Access Get only

Returns the number of pairs in the lexicon.

Lexicon:**COPY**()

Return type *Lexicon*

Access Get only

Returns a new lexicon that contains the same set of pairs as this lexicon. Note that this is a "shallow" copy, meaning that if there is a value in the list that refers to, for example, another Lexicon, or a Vessel, or a Part, the new copy will still be referring to the same object as the original copy in that value.

Lexicon:**HASKEY**(key)

Parameters

- **key** – (any type)

Returns boolean

Returns true if the lexicon contains the provided key

Lexicon:**HASVALUE**(key)

Parameters

- **key** – (any type)

Returns boolean

Returns true if the lexicon contains the provided value

Lexicon:**DUMP**

Type string

Access Get only

Returns a string containing a verbose dump of the lexicon's contents.

The difference between a DUMP and just the normal printing of a Lexicon is in whether or not it recursively shows you the contents of every complex object inside the Lexicon.

i.e:: // Just gives a shallow list: print mylexicon.

// Walks the entire tree of contents, descending down into // any Lists or Lexicons that are stored inside this Lexicon: print mylexicon:dump.

Lexicon:**KEYS**

Type List

Access Get only

Returns a List of the keys stored in this lexicon.

Lexicon:**VALUES**

Type List

Access Get only

Returns a List of the values stored in this lexicon.

Access to Individual Elements

lexicon[expression] operator: another syntax to access the element at position 'expression'. Works for get or set. Any arbitrary complex expression may be used with this syntax, not just a number or variable name.

FOR VAR IN LEXICON.KEYS { ... }. *A type of loop* in which var iterates over all the items of lexicon from item 0 to item LENGTH-1.

Implicit ADD when using index brackets with new key values

(a.k.a. The difference between GETTING and SETTING with nonexistant keys)

If you attempt to use a key that does not exist in the lexicon, to GET a value, as follows:

```
SET ARR TO LEXICON().
SET X TO ARR["somekey"].  // this will produce an error.
```

Then you will get a KOSKeyNotFoundException error, as you might expect, because the key `"somekey"` isn't there in the empty lexicon you just made.

However if you use a key that does not exist yet to SET a value rather than to GET a value, you don't get an error. Instead it actually implicitly ADDS the new value to the lexicon with that key. The example below will not give you an error:

```
SET ARR TO LEXICON().
SET ARR["somekey"] TO 100. // adds new value to the lexicon.
```

The above ends up doing the same thing as if you had done this:

```
SET ARR TO LEXICON().
ARR:ADD("somekey",100).
```

Note that while using `:ADD()` to make a new value in the lexicon will give you a duplicate key error if the value already does exist, using SET to create the value implicitly won't because it simply replaces the existing value in-place rather than trying to make a new one.

This gives a duplicate key error:

```
SET ARR TO LEXICON().
ARR:ADD("somekey",100).
ARR:ADD("somekey",200).  // error, because "somekey" already exists.
```

While this does not:

```
SET ARR TO LEXICON().
SET ARR["somekey"] to 100.
SET ARR["somekey"] to 200. // no error, because it replaces the value 100 with a 200.
```

In a nutshell, using [..] to set a value in a lexicon does this: If the key already exists, replace the value with the new value. If the key does not already exist, make it exist and give it this new value.

Examples

```
SET BAR TO LEXICON().        // Creates a new empty lexicon in BAR variable
BAR:ADD("FIRST",10).         // Adds a new element to the lexicon with the key of "FIRST"
BAR:ADD("SECOND",20).        // Adds a new element to the lexicon with the key of "SECOND"
BAR:ADD("LAST",30).          // Adds a new element to the lexicon with the key of "LAST"

PRINT BAR["FIRST"].             // Prints 10
PRINT BAR["SECOND"].             // Prints 20
PRINT BAR["LAST"].             // Prints 30

SET FOO TO LEXICON().            // Creates a new empty lexicon in FOO variable
FOO:ADD("ALTITUDE", ALTITUDE).  // Adds current altitude number to the lexicon
FOO:ADD("ETA", ETA:APOAPSIS).   // Adds current seconds to apoapsis to the lexicon at the index "ETA

// As a reminder, at this point your lexicon, if you did all the above
// steps in order, would look like this now:
//
//  FOO["ALTITUDE"] = 99999. // or whatever your altitude was when you added it.
//  FOO["ETA"] = 99. // or whatever your ETA:APOAPSIS was when you added it.

PRINT FOO:LENGTH.         // Prints 2
PRINT FOO:LENGTH().       // Also prints 2.  LENGTH is a method that, because it takes zero arguments,
SET x TO "ALTITUDE". PRINT FOO[x].  // Prints the same thing as FOO["ALTITUDE"].

FOO:REMOVE("ALTITUDE").              // Removes the element at "ALTITUDE" from the lexicon.
```

8.5.8 List

A *List* is a collection of any type in kOS. Many places throughout the system return variables of the *List* type, and you can create your own *List* variables as well. One of the places you are likely to find that kOS gives you a *List* is when you use the *LIST command* to list some query into one of your variables.

Constructing a list

Numerous built-in functions in kOS return a list. If you wish to make your own list from scratch you can do so with the LIST() built-in function. You pass a varying number of arguments into it to pre-populate the list with an initial list of items:

> // Make an empty list with zero items in it: set mylist to list().
>
> // Make a list with 3 numbers in it: set mylist to list(10,20,30).
>
> // Make a list with 3 strings in it: set mylist to list("10","20","30").
>
> // Make a two dimensional 2x3 list with heterogenious contents // mixing strings and numbers: set mylist to list(list("a","b","c"), list(1,2,3)).

The contents of a list can be any objects you feel like, and do not need to be of a homogeneous type.

Structure

structure `List`

Table 8.18: Members

Suffix	Type	Description
ADD(item)	None	append an item
INSERT(index,item)	None	insert item at index
REMOVE(index)	None	remove item by index
CLEAR	None	remove all elements
LENGTH	integer	number of elements in list
ITERATOR	*Iterator*	for iterating over the list
COPY	*List*	a new copy of this list
CONTAINS(item)	boolean	check if list contains an item
SUBLIST(index,length)	*List*	new list of given length starting with index
EMPTY	boolean	check if list if empty
DUMP	string	verbose dump of all contained elements

List:**`ADD`**(item)

Parameters

- **`item`** – (any type) item to be added

Appends the new value given to the end of the list.

List:**`INSERT`**(index,item)

Parameters

- **`index`** – (integer) position in list (starting from zero)
- **`item`** – (any type) item to be added

Inserts a new value at the position given, pushing all the other values in the list (if any) one spot to the right.

List:**`REMOVE`**(index)

Parameters

- **`index`** – (integer) position in list (starting from zero)

Remove the item from the list at the numeric index given, with counting starting at the first item being item zero

List:**`CLEAR`**

Use this for its side-effect. Whenever `myList:CLEAR` exists in an expression, `myList` will be zeroed out, regardless of what you do with the value of the expression:

```
SET dummy TO myList:CLEAR.
```

List:**`LENGTH`**

Type integer

Access Get only

Returns the number of elements in the list.

List:**`ITERATOR`**

Type *Iterator*

Access Get only

An alternate means of iterating over a list. See: *Iterator*.

List:**COPY**

Type *List*

Access Get only

Returns a new list that contains the same thing as the old list.

List:**CONTAINS**(item)

Parameters

- **index** – (integer) starting index (from zero)

Returns boolean

Returns true if the list contains an item equal to the one passed as an argument

List:**SUBLIST**(index,length)

Parameters

- **index** – (integer) starting index (from zero)
- **length** – (integer) resulting length of returned *List*

Returns *List*

Returns a new list that contains a subset of this list starting at the given index number, and running for the given length of items.

List:**EMPTY**

Type boolean

Access Get only

Returns true if the list has zero items in it.

List:**DUMP**

Type string

Access Get only

Returns a string containing a verbose dump of the list's contents.

Access to Individual Elements

All list indexes start counting at zero. (The list elements are numbered from 0 to N-1 rather than from 1 to N.)

list[expression] operator: another syntax to access the element at position 'expression'. Works for get or set. Any arbitrary complex expression may be used with this syntax, not just a number or variable name. This syntax is preferred over the older "#" syntax, which is kept only for backward compatibility.

FOR VAR IN LIST { ... }. *A type of loop* in which var iterates over all the items of list from item 0 to item LENGTH-1.

ITERATOR An alternate means of iterating over a list. See *Iterator*.

list#x ***(deprecated)*** operator: access the element at postion x. Works for get or set. X must be a hardcoded number or a variable name. This is here for backward compatibility. The syntax in the next bullet point is preferred over this.

Examples:

```
SET BAR TO LIST(5,3,6).  // Creates a new list with 3 integers in it.
SET FOO TO LIST().       // Creates a new empty list in FOO variable
FOO:ADD(5).              // Adds a new element to the end of the list
FOO:ADD( ALTITUDE ).     // Adds current altitude number to the end of the list
FOO:ADD(ETA:APOAPSIS).   // Adds current seconds to apoapsis to the end of the list

// As a reminder, at this point your list, if you did all the above
// steps in order, would look like this now:
//
//  FOO[0] = 5.
//  FOO[1] = 99999. // or whatever your altitude was when you added it.
//  FOO[2] = 99. // or whatever your ETA:APOAPSIS was when you added it.

PRINT FOO:LENGTH.        // Prints 3
PRINT FOO:LENGTH().      // Also prints 3.  LENGTH is a method that, because it takes zero arguments,
PRINT FOO#0.             // Prints 5, using deprecated old '#' syntax.
PRINT FOO[0].            // Prints 5, using newer preferred '[]' syntax.
PRINT FOO[1].            // Prints altitude number.
PRINT FOO[2].            // Prints eta:apoapsis number.
SET x TO 2. PRINT FOO#x. // Prints the same thing as FOO[2], using deprecated old '#' syntax.
SET x TO 2. PRINT FOO[x].// Prints the same thing as FOO[2].
SET y to 3. PRINT FOO[ y/3 + 1 ].
                         // Prints the same thing as FOO#2, using a mathematical expression as the in
SET FOO#0 to 4.          // Replace the 5 at position 0 with a 4.
FOO:INSERT(0,"skipper 1"). // Inserts the string "skipper 1" to the start of the list, pushing the re
FOO:INSERT(2,"skipper 2"). // Inserts the string "skipper 2" at position 2 of the list, pushing the r

// As a reminder, at this point your list, if you did all the above
// steps in order, would look like this now:
//
//  FOO[0] = "skipper 1".
//  FOO[1] = 5.
//  FOO[2] = "skipper 2".
//  FOO[3] = 99999. // or whatever your altitude was when you added it.
//  FOO[4] = 99. // or whatever your ETA:APOAPSIS was when you added it.

FOO:REMOVE( 1).              // Removes the element at index 1 from the list, moving everything else
FOO:REMOVE(FOO:LENGTH - 1).  // Removes whatever element happens to be at the end of the list, at pos

// As a reminder, at this point your list, if you did all the above
// steps in order, would look like this now:
//
//  FOO[0] = "skipper 1".
//  FOO[1] = "skipper 2".
//  FOO[2] = 99999. // or whatever your altitude was when you added it.

SET BAR TO FOO:COPY.     // Makes a copy of the FOO list
FOO:CLEAR.               // Removes all elements from the FOO list.
FOO:CLEAR().             // Also removes all elements from the FOO list.  The parentheses are optiona
FOR var in BAR {         // -.
  print var.             //   |- Print all the contents of FOO.
}.                       // -'
```

Multidimensional Arrays

A 2-D array is a *List* who's elements are themselves also `Lists`. A 3-D array is a *List* of *Lists* of *Lists*. Any number of dimensions is possible.

list[x][y] (or list#x#y) Access the element at position x,y of the 2-D array (list of lists). The use of the '#' syntax is deprecated and exists for backward compatibility only. The newer '[]' square-bracket syntax is preferred.

- The elements of the array need not be uniform (any mix of strings, numbers, structures is allowed).
- The dimensions of the array need not be uniform (row 1 might have 3 columns while row 2 has 5 columns):

```
SET FOO TO LIST(). // Empty list.
FOO:ADD( LIST() ). // Element 0 is now itself a list.
FOO[0]:ADD( "A" ). // Element 0,0 is now "A".
FOO[0]:ADD( "B" ). // Element 0,1 is now "B".
FOO:ADD(LIST()).   // Element 1 is now itself a list.
FOO[1]:ADD(10).    // Element 1,0 is now 10.
FOO[1]:ADD(20).    // Element 1,1 is now 20.
FOO:ADD(LIST()).   // Element 2 is now itself a list.

FOO[ FOO:LENGTH -1 ]:ADD(3.14159).
    // Element 2,0 is now 3.1519, using a more complex
    //     expression to dynamically obtain the current
    //     maximum index of '2'.

FOO[ FOO:LENGTH -1 ]:ADD(7).
    // Element 2,1 is now 7, using a more complex
    //     expression to dynamically obtain the current
    //     maximum index of '2'.

// FOO is now a 2x3 matrix looking like this:
//    A         B
//    10        20
//    3.14159   7

// or like this, depending on how you want
// to visualize it as a row-first or column-first table:
//    A    10     3.14159
//    B    20     7

PRINT FOO[0][0]. // Prints A.
PRINT FOO[0][1]. // Prints B.
PRINT FOO[1][0]. // Prints 10.
PRINT FOO[1][1]. // Prints 20.
PRINT FOO[2][0]. // Prints 3.14159.
PRINT FOO[2][1]. // Prints 7.

PRINT FOO#2#0.   // Prints 3.14159, using deprecated syntax.
```

Comparing two lists

Note that if you have two lists, LISTA and LISTB, and you tried to compare if they were the same, in this way:

```
if LISTA = LISTB {
  print "they are equal".
}
```

Then the check will only be true if LISTA and LISTB are both actually the same list - not just two lists with equal contents, but in fact just two variables pointing to the same list.

This is because a LIST is a complex structure object, and like most complex structure objects, the equality check is just testing whether or not they refer to the same object, not whether or not they have equivalent content.

To test if the contents are equivalent, you have to check them item by item, like so:

```
set still_same to true.
FROM {local i is 0.}
  UNTIL i > LISTA:LENGTH or not still_same
  STEP {set i to i + 1.}
DO
{
  set still_same to (LISTA[i] = LISTB[i]).
}
if still_same {
  print "they are equal".
}
```

8.5.9 Vessel Load Distance

structure `LoadDistance`

(The "on rails" settings)

LoadDistance describes the set of distances at which the game causes vessels to unload, and the distances that cause a vessel to become "packed". This requires some explanation.

Before entering into that explanation, first here's a list of example cases where you might want to use this feature to change these values:

- Trying to have one airplane follow another.
- Trying to have two rockets fly in formation into orbit together.
- Trying to have a rover race to a flag, with several rovers seeing who gets there first.

Basically, any time you might have more than just the current active vessel running a kOS script, this is a setting you probably will want to understand and tweak.

The explanation:

Most players of KSP eventually discover a concept called being "on rails". This is a term used by the player community to describe the fact that vessels that are far away from the active vessel aren't being micro-managed under the full physics engine. At that distance, changes in movement due to things like the atmosphere, are not applied. The vessel's physics are calculated based only on orbital motion, much like the effects of using timewarp (not physics warp).

But the actual behavior in the game is a bit more complex than that, and understanding it is necessary to use this structure.

The term "on rails" actually refers to two entirely different things that are controlled by separate settings, as described below:

*loaded* : A vessel is LOADED when all its parts are being rendered by the graphics engine and it's possible to actually see what it looks like. A vessel that is UNLOADED doesn't even have its parts in memory and is just a single dot in space with no dimensions. An unloaded vessel is literally impossible to see on your screen no matter how much you squint because it's not even being rendered on camera at all. Unloaded vessels only exist as a marker icon in space, with a possible label text.

*packed* : A vessel is PACKED when it is close enough to be *loaded* (see above), but still far enough away that its full capabilities aren't enabled. A vessel that is *loaded*, but still *packed* will be unable to have its parts interact, and the vessel will appear stuck in the same location, unmoving. You can *see* a vessel that is loaded but packed, but the vessel won't actually be able to *do* anything. In this state, the game is still treating the entire object as if it was one single part. It's added all the graphic models of all the parts together into one

conglomerate "object" (thus the term "packed") that exists purely so you can look at it, even though it doesn't actually work until you get closer and it becomes *unpacked*.

THE NEXT SENTENCE IS VERY IMPORTANT AND VITAL:

In order for a vessel to run kOS script code, the vessel must be BOTH LOADED and UNPACKED. Until both of those conditions occur, scripts cannot run on the vessel's kOS computer parts.

This structure allows you to read or change the stock KSP game's distance settings for how far a vessel has to get from the active vessel in order for it to trigger its UNLOAD or PACK states.

The distance settings are different for different vessel situations. It's important to first read the existing values before changing them, to see what the stock game thought were reasonable for them.

Some distances are very short. For example, the fact that the pack distance for a landed vessel is short is what allows landers to stay "parked" in place without tipping over when you leave them on a long distance EVA.

Each of these suffixes returns a *SituationLoadDistance*, which is a tuple of values for the loading and packing distances in that situation.

Wait between LOAD and PACK changes!

Due to a strange way the game behaves, it is unsafe to change both the load/unload distance and the unpack/pack distance at the same time in the same physics tick. If you are going to increase both, then increase the load/unload distances first, followed by a `WAIT 0.001.` to force a new physics tick and let the change take effect, then increase the unpack/pack distances after the wait is over.

Beware the space kraken when changing these:

There's a reason the stock game has these distance limitations. Setting them very large can degrade your performance, and can cause buggy inaccuracies in the position and velocity calculations that cause the game to think things have collided together when they haven't. This is the classic "space kraken" that KSP players talk about a lot. Computer floating point numbers get less precise the farther from zero they are. So allowing the game to try to perform microcalculations on tiny time scales using floating point numbers that have imprecision because they are large in magnitude (i.e. the positions of parts that are many kilometers away from you), can cause phantom collisions, which make the game explode things for "no reason".

These distance limits were put in place by SQUAD specifically for the purpose of trying to avoid the space kraken. If you set them too large again, you can risk invoking the Kraken again. They typically CAN be enlarged some, because the settings are very low to provide a overly large safety margin, but be careful with it. Don't go overboard and set the ranges to several thousand kilometers.

Also, don't set the PACK distance to be higher than the LOAD distance, as that is undefined behavior in the main game. Always keep the LOAD distance higher than or equal to the PACK distance.

Table 8.19: Members and Methods

Suffix	Type	Get/Set	Description
ESCAPING	*SituationLoadDistance*	Get	Load and pack Distances while escaping the current body
FLYING	*SituationLoadDistance*	Get	Load and pack Distances while flying in atmosphere
LANDED	*SituationLoadDistance*	Get	Load and pack Distances while landed on the surface
ORBIT	*SituationLoadDistance*	Get	Load and pack Distances while in orbit
PRELAUNCH	*SituationLoadDistance*	Get	Load and pack Distances while on the launch pad or runway
SPLASHED	*SituationLoadDistance*	Get	Load and pack Distances while splashed in water
SUBORBITAL	*SituationLoadDistance*	Get	Load and pack Distances while on a suborbital trajectory

8.5.10 Situation Load Distance

Each of the above

structure `SituationLoadDistance`

`SituationLoadDistance` is what is returned by each of the above suffixes mentioned in the LoadDistance suffix list above.

Order Matters. - Becuse of the protections in place to prevent you from setting some values bigger than others (see the descriptions below), sometimes the order in which you change the values matters and you have to be careful to change them in the correct order, or else the attempt to change them will be denied.

SituationLoadDistance:**`LOAD`**

Access Get/Set

Type scalar, in meters

Get or set the load distance. When another vessel is getting closer to you, because you are moving toward it or it is moving toward you, when that vessel becomes this distance *or closer* to the active vessel, it will transition from being *unloaded* to being *loaded*. See the description above for what it means for a vessel to be *loaded*.

This value must be greater than `UNLOAD`, and will automatically be adjusted accordingly.

SituationLoadDistance:**`UNLOAD`**

Access Get/Set

Type scalar, in meters

Get or set the unload distance. When another vessel is becoming more distant as you move away from it, or it moves away from you, when that vessel becomes this distance *or greater* from the active vessel, it will transition from being *loaded* to being *unloaded*. See the description above for what it means for a vessel to be *loaded*.

This value must be less than `LOAD`, and will automatically be adjusted accordingly.

SituationLoadDistance:**`UNPACK`**

Access Get/Set

Type scalar, in meters

Get or set the unpack distance. When another vessel is getting closer to you, because you are moving toward it or it is moving toward you, when that vessel becomes this distance *or closer* to the active vessel, it will transition from being *packed* to being *unpacked*. See the description above for what it means for a vessel to be *packed*.

This value must be less than `PACK`, and will automatically be adjusted accordingly.

SituationLoadDistance:**`PACK`**

Access Get/Set

Type scalar, in meters

Get or set the pack distance. When another vessel is getting farther away from you, because you are moving away from it or it is moving away from you, when that vessel becomes this distance *or greater* from the active vessel, it will transition from being *unpacked* to being *packed*. See the description above for what it means for a vessel to be *packed*.

This value must be greater than `UNPACK`, and will automatically be adjusted accordingly.

===Examples===

Print out all the current settings:

```
SET distances TO KUNIVERSE:DEFAULTLOADDISTANCE.

PRINT "escaping distances:".
print "    load: " + distances:ESCAPING:LOAD + "m".
print "  unload: " + distances:ESCAPING:UNLOAD + "m".
print "  unpack: " + distances:ESCAPING:UNPACK + "m".
print "    pack: " + distances:ESCAPING:PACK + "m".
PRINT "flying distances:".
print "    load: " + distances:FLYING:LOAD + "m".
print "  unload: " + distances:FLYING:UNLOAD + "m".
print "  unpack: " + distances:FLYING:UNPACK + "m".
print "    pack: " + distances:FLYING:PACK + "m".
PRINT "landed distances:".
print "    load: " + distances:LANDED:LOAD + "m".
print "  unload: " + distances:LANDED:UNLOAD + "m".
print "  unpack: " + distances:LANDED:UNPACK + "m".
print "    pack: " + distances:LANDED:PACK + "m".
PRINT "orbit distances:".
print "    load: " + distances:ORBIT:LOAD + "m".
print "  unload: " + distances:ORBIT:UNLOAD + "m".
print "  unpack: " + distances:ORBIT:UNPACK + "m".
print "    pack: " + distances:ORBIT:PACK + "m".
PRINT "prelaunch distances:".
print "    load: " + distances:PRELAUNCH:LOAD + "m".
print "  unload: " + distances:PRELAUNCH:UNLOAD + "m".
print "  unpack: " + distances:PRELAUNCH:UNPACK + "m".
print "    pack: " + distances:PRELAUNCH:PACK + "m".
PRINT "splashed distances:".
print "    load: " + distances:SPLASHED:LOAD + "m".
print "  unload: " + distances:SPLASHED:UNLOAD + "m".
print "  unpack: " + distances:SPLASHED:UNPACK + "m".
print "    pack: " + distances:SPLASHED:PACK + "m".
PRINT "suborbital distances:".
print "    load: " + distances:SUBORBITAL:LOAD + "m".
print "  unload: " + distances:SUBORBITAL:UNLOAD + "m".
print "  unpack: " + distances:SUBORBITAL:UNPACK + "m".
print "    pack: " + distances:SUBORBITAL:PACK + "m".
```

Change the settings while flying or landed or splashed or on launchpad or runway, For the purpose of allowing more vessels to fly around the Kerbal Space Center at a greater distances from each other:

```
// 30 km for in-flight
// Note the order is important.  set UNLOAD BEFORE LOAD,
// and PACK before UNPACK.  Otherwise the protections in
// place to prevent invalid values will deny your attempt
// to change some of the values:
SET KUNIVERSE:DEFAULTLOADDISTANCE:FLYING:UNLOAD TO 29500.
SET KUNIVERSE:DEFAULTLOADDISTANCE:FLYING:LOAD TO 30000.
WAIT 0.001. // See paragraph above: "wait between load and pack changes"
SET KUNIVERSE:DEFAULTLOADDISTANCE:FLYING:PACK TO 29999.
SET KUNIVERSE:DEFAULTLOADDISTANCE:FLYING:UNPACK TO 29000.
WAIT 0.001. // See paragraph above: "wait between load and pack changes"

// 30 km for parked on the ground:
// Note the order is important.  set UNLOAD BEFORE LOAD,
// and PACK before UNPACK.  Otherwise the protections in
// place to prevent invalid values will deny your attempt
// to change some of the values:
```

```
SET KUNIVERSE:DEFAULTLOADDISTANCE:LANDED:UNLOAD TO 29500.
SET KUNIVERSE:DEFAULTLOADDISTANCE:LANDED:LOAD TO 30000.
WAIT 0.001. // See paragraph above: "wait between load and pack changes"
SET KUNIVERSE:DEFAULTLOADDISTANCE:LANDED:PACK TO 39999.
SET KUNIVERSE:DEFAULTLOADDISTANCE:LANDED:UNPACK TO 29000.
WAIT 0.001. // See paragraph above: "wait between load and pack changes"

// 30 km for parked in the sea:
// Note the order is important.  set UNLOAD BEFORE LOAD,
// and PACK before UNPACK.  Otherwise the protections in
// place to prevent invalid values will deny your attempt
// to change some of the values:
SET KUNIVERSE:DEFAULTLOADDISTANCE:SPLASHED:UNLOAD TO 29500.
SET KUNIVERSE:DEFAULTLOADDISTANCE:SPLASHED:LOAD TO 30000.
WAIT 0.001. // See paragraph above: "wait between load and pack changes"
SET KUNIVERSE:DEFAULTLOADDISTANCE:SPLASHED:PACK TO 29999.
SET KUNIVERSE:DEFAULTLOADDISTANCE:SPLASHED:UNPACK TO 29000.
WAIT 0.001. // See paragraph above: "wait between load and pack changes"

// 30 km for being on the launchpad or runway
// Note the order is important.  set UNLOAD BEFORE LOAD,
// and PACK before UNPACK.  Otherwise the protections in
// place to prevent invalid values will deny your attempt
// to change some of the values:
SET KUNIVERSE:DEFAULTLOADDISTANCE:PRELAUNCH:UNLOAD TO 29500.
SET KUNIVERSE:DEFAULTLOADDISTANCE:PRELAUNCH:LOAD TO 30000.
WAIT 0.001. // See paragraph above: "wait between load and pack changes"
SET KUNIVERSE:DEFAULTLOADDISTANCE:PRELAUNCH:PACK TO 29999.
SET KUNIVERSE:DEFAULTLOADDISTANCE:PRELAUNCH:UNPACK TO 29000.
WAIT 0.001. // See paragraph above: "wait between load and pack changes"
```

8.5.11 PIDLoop

The *PIDLoop* has multiple constructors available. Valid syntax can be seen here:

```
// Create a loop with default parameters
// kp = 1, ki = 0, kd = 0
// maxoutput = maximum number value
// minoutput = minimum number value
SET PID TO PIDLOOP().

// Other constructors include:
SET PID TO PIDLOOP(KP).
SET PID TO PIDLOOP(KP, KI, KD).
// you must specify both minimum and maximum output directly.
SET PID TO PIDLOOP(KP, KI, KD, MINOUTPUT, MAXOUTPUT).

// remember to update both minimum and maximum output if the value changes symmetrically
SET LIMIT TO 0.5.
SET PID:MAXOUTPUT TO LIMIT.
SET PID:MINOUTPUT TO -LIMIT.

// call the update suffix to get the output
SET OUT TO PID:UPDATE(TIME:SECONDS, IN).

// you can also get the output value later from the PIDLoop object
SET OUT TO PID:OUTPUT.
```

Please see the bottom of this page for information on the derivation of the loop's output.

Note: New in version 0.18: While the *PIDLOOP* structure was added in version 0.18, you may feel free to continue to use any previously implemented PID logic. This loop is intended to be a basic and flexible PID, but you may still find benefit in using customized logic.

---

structure `PIDLoop`

Suffix	Type	Description
LASTSAMPLETIME	scalar	decimal value of the last sample time
KP	scalar	The proportional gain factor
KI	scalar	The integral gain factor
KD	scalar	The derivative gain factor
INPUT	scalar	The most recent input value
SETPOINT	scalar	The current setpoint
ERROR	scalar	The most recent error value
OUTPUT	scalar	The most recent output value
MAXOUTPUT	scalar	The maximum output value
MINOUTPUT	scalar	The maximum output value
ERRORSUM	scalar	The time weighted sum of error
PTERM	scalar	The proportional component of output
ITERM	scalar	The integral component of output
DTERM	scalar	The derivative component of output
CHANGERATE	scalar (/s)	The most recent input rate of change
RESET	none	Reset the integral component
UPDATE(time, input)	scalar	Returns output based on time and input

PIDLoop:**`LASTSAMPLETIME`**

Type scalar

Access Get only

The value representing the time of the last sample. This value is equal to the time parameter of the `UPDATE` method.

PIDLoop:**`KP`**

Type scalar

Access Get/Set

The proportional gain factor.

PIDLoop:**`KI`**

Type scalar

Access Get/Set

The integral gain factor.

PIDLoop:**`KD`**

Type scalar

Access Get/Set

The derivative gain factor

PIDLoop:**`INPUT`**

Type scalar

Access Get only

The value representing the input of the last sample. This value is equal to the input parameter of the `UPDATE` method.

PIDLoop:**`SETPOINT`**

Type scalar

Access Get/Set

The current setpoint. This is the value to which input is compared when `UPDATE` is called. It may not be synced with the last sample.

PIDLoop:**`ERROR`**

Type scalar

Access Get only

The calculated error from the last sample (setpoint - input).

PIDLoop:**`OUTPUT`**

Type scalar

Access Get only

The calculated output from the last sample.

PIDLoop:**`MAXOUTPUT`**

Type scalar

Access Get/Set

The current maximum output value. This value also helps with regulating integral wind up mitigation.

PIDLoop:**`MINOUTPUT`**

Type scalar

Access Get/Set

The current minimum output value. This value also helps with regulating integral wind up mitigation.

PIDLoop:**`ERRORSUM`**

Type scalar

Access Get only

The value representing the time weighted sum of all errrors. It will be equal to `ITERM` / `KI`. This value is adjusted by the integral windup mitigation logic.

PIDLoop:**`PTERM`**

Type scalar

Access Get only

The value representing the proportional component of `OUTPUT`.

PIDLoop:**`ITERM`**

Type scalar

Access Get only

The value representing the integral component of OUTPUT. This value is adjusted by the integral windup mitigation logic.

PIDLoop:**DTERM**

Type scalar

Access Get only

The value representing the derivative component of OUTPUT.

PIDLoop:**CHANGERATE**

Type scalar

Access Get only

The rate of change of the INPUT during the last sample. It will be equal to (input - last input) / (change in time).

PIDLoop:**RESET**()

Returns none

Call this method to clear the ERRORSUM and ITERM components of the PID calculation.

PIDLoop:**UPDATE**(time, input)

Parameters

- **time** – (scalar) the decimal time in seconds
- **input** – (scalar) the input variable to compare to the setpoint

Returns scalar representing the calculated output

Upon calling this method, the PIDLoop will calculate the output based on this this basic framework (see below for detailed derivation): output = error * kp + errorsum * ki + (change in input) / (change in time) * kd. This method is usually called with the current time in seconds (*TIME:SECONDS*), however it may be called using whatever rate measurement you prefer. Windup mitigation is included, based on MAXOUTPUT and MINOUTPUT. Both integral components and derivative components are guarded against a change in time greater than 1s, and will not be calculated on the first iteration.

PIDLoop Update Derivation

The internal update method of the *PIDLoop* structure is the equivalent of the following in kerboscript

```
// assume that the terms LastSampleTime, Kp, Ki, Kd, Setpoint, MinOutput, and MaxOutput are previous
declare function Update {
    declare parameter sampleTime, input.
    set Error to Setpoint - input.
    set PTerm to error * Kp.
    set ITerm to 0.
    set DTerm to 0.
    if (LastSampleTime < sampleTime) {
        set dt to sampleTime - LastSampleTime.
        if dt < 1 {
            // only calculate integral and derivative if the time
            // difference is less than one second, and their gain is not 0.
            if Ki <> 0 {
                ITerm = (ErrorSum + Error) * dt * Ki.
            }
            set ChangeRate to (input - LastInput) / dt.
            if Kd <> 0 {
```

```
                DTerm = ChangeRate * Kd.
            }
        }
    }
    set Output to pTerm + iTerm + dTerm.
    // if the output goes beyond the max/min limits, reset it and adjust ITerm.
    if Output > MaxOutput {
        set Output to MaxOutput.
        // adjust the value of ITerm as well to prevent the value
        // from winding up out of control.
        if (Ki != 0) and (LastSampleTime < sampleTime) {
            set ITerm to Output - Pterm - DTerm.
        }
    }
    else if Output < MinOutput {
        set Output to MinOutput.
        // adjust the value of ITerm as well to prevent the value
        // from winding up out of control.
        if (Ki != 0) and (LastSampleTime < sampleTime) {
            set ITerm to Output - Pterm - DTerm.
        }
    }
    LastSampleTime = sampleTime.
    if Ki <> 0 set ErrorSum to ITerm / Ki.
    else ErrorSum = 0.
    return Output.
}
```

8.5.12 Resource Transfer

Usage is covered elsewhere <../commands/resource_transfer.html>‘__

Structure

structure `ResourceTransfer`

Table 8.20: Members

Suffix	Type (units)	Access	Description
STATUS	string	Get only	The string status of the transfer (eg “Inactive”, “Transferring”, “Failed”, “Finished”)
MESSAGE	string	Get only	A message about the current status
GOAL	scalar	Get only	This is how much of the resource will be transferred.
TRANSFERRED	scalar	Get only	This is how much of the resource has been transferred.
RESOURCE	string	Get only	The name of the resource (eg oxidizer, liquidfuel)
ACTIVE	bool	Get / Set	Setting this value will either start, pause or restart a transfer. Default is false.

RESOURCETRANSFER:**`STATUS`**

Access Get only

Type string

This enumerated type shows the status of the transfer. the possible values are:

- **Inactive (default)**
  - Transfer is stopped
- **Finished**
  - Transfer has reached its goal
- **Failed**
  - There was an error in the transfer, see `MESSAGE` for details
- **Transferring**
  - The transfer is in progress.

RESOURCETRANSFER:**`MESSAGE`**

Access Get only

Type string

This shows the detail related to `STATUS`

RESOURCETRANSFER:**`GOAL`**

Access Get only

Type scalar

If you specified an amount to transfer in your transfer request, it will be shown here. If you did not, this will return the sentinel value -1.

RESOURCETRANSFER:**`TRANSFERRED`**

Access Get only

Type scalar

Returns the amount of the specified resource that has been transferred by this resource transfer.

RESOURCETRANSFER:**`RESOURCE`**

Access Get only

Type string

The name of the resource that will be transferred. (eg, oxidizer, liquidfuel)

RESOURCETRANSFER:**`ACTIVE`**

Access Get / Set

Type bool

When getting, this suffix is simply a shortcut to tell you if `STATUS` is Transferring. Setting true will change the status of the transfer to Transferring, setting false will change status to inactive.

8.5.13 SteeringManager

The SteeringManager is a bound variable, not a suffix to a specific vessel. This prevents access to the SteeringManager of other vessels. You can access the steering manager as shown below:

```
// Display the ship facing, target facing, and world coordinates vectors.
SET STEERINGMANAGER:SHOWFACINGVECTORS TO TRUE.

// Change the torque calculation to multiply the available torque by 1.5.
SET STEERINGMANAGER:ROLLTORQUEFACTOR TO 1.5.
```

Note: New in version 0.18: The *SteeringManager* was added to improve the accuracy of kOS's cooked steering. While this code is a significant improvement over the old system, it is not perfect. Specifically it does not properly calculate the effects of control surfaces, nor does it account for atmospheric drag. It also does not adjust for asymmetric RCS or Engine thrust. It does allow for some modifications to the built in logic through the torque adjustments and factors. However, if there is a condition for which the new steering manager is unable to provide accurate control, you should continue to fall back to raw controls.

structure `SteeringManager`

Suffix	Type	Description
PITCHPID	*PIDLoop*	The PIDLoop for the pitch direction.
YAWPID	*PIDLoop*	The PIDLoop for the yaw direction.
ROLLPID	*PIDLoop*	The PIDLoop for the roll direction.
ENABLED	bool	Returns true if the *SteeringManager* is currently controlling the vessel
TARGET	*Direction*	The direction that the vessel is currently steering towards
RESETPIDS()	none	Called to call *RESET* on all steering PID loops.
SHOWFACINGVECTORS	bool	Enable/disable display of ship facing, target, and world coordinates vectors.
SHOWANGULARVECTORS	bool	Enable/disable display of angular rotation vectors
SHOWTHRUSTVECTORS	bool	Enable/disable display of engine thrust vectors
SHOWRCSVECTORS	bool	Enable/disable display of rcs thrust vectors
SHOWSTEERINGSTATS	bool	Enable/disable printing of the steering information on the terminal
WRITECSVFILES	bool	Enable/disable logging steering to csv files.
PITCHTS	scalar (s)	Settling time for the pitch torque calculation
YAWTS	scalar (s)	Settling time for the yaw torque calculation
ROLLTS	scalar (s)	Settling time for the roll torque calculation
MAXSTOPPINGTIME	scalar (s)	The maximum amount of stopping time to limit angular turn rate.
ANGLEERROR	scalar (deg)	The angle between vessel:facing and target directions
PITCHERROR	scalar (deg)	The angular error in the pitch direction
YAWERROR	scalar (deg)	The angular error in the yaw direction
ROLLERROR	scalar (deg)	The angular error in the roll direction
PITCHTORQUEADJUST	scalar (kN)	Additive adjustment to pitch torque (calculated)
YAWTORQUEADJUST	scalar (kN)	Additive adjustment to yaw torque (calculated)
ROLLTORQUEADJUST	scalar (kN)	Additive adjustment to roll torque (calculated)
PITCHTORQUEFACTOR	scalar	Multiplicative adjustment to pitch torque (calculated)
YAWTORQUEFACTOR	scalar	Multiplicative adjustment to yaw torque (calculated)
ROLLTORQUEFACTOR	scalar	Multiplicative adjustment to roll torque (calculated)

SteeringManager:**`PITCHPID`**

Type *PIDLoop*

Access Get only

Returns the PIDLoop object for the pitch direction. This allows direct manipulation of the gain parameters, and other components of the *PIDLoop* structure. The loop's *MAXOUTPUT* value will be overwritten on every sample, with it being set to limit the maximum turning rate to that which can be stopped in a MAXSTOPPINGTIME seconds (calculated based on available torque, and the ship's moment of inertia).

SteeringManager:**YAWPID**

Type *PIDLoop*

Access Get only

Returns the PIDLoop object for the yaw direction. This allows direct manipulation of the gain parameters, and other components of the *PIDLoop* structure. The loop's *MAXOUTPUT* value will be overwritten on every sample, with it being set to limit the maximum turning rate to that which can be stopped in a MAXSTOPPINGTIME seconds (calculated based on available torque, and the ship's moment of inertia).

SteeringManager:**ROLLPID**

Type *PIDLoop*

Access Get only

Returns the PIDLoop object for the roll direction. This allows direct manipulation of the gain parameters, and other components of the *PIDLoop* structure. The loop's *MAXOUTPUT* value will be overwritten on every sample, with it being set to limit the maximum turning rate to that which can be stopped in a MAXSTOPPINGTIME seconds (calculated based on available torque, and the ship's moment of inertia).

SteeringManager:**ENABLED**

Type bool

Access Get only

Returns true if the SteeringManager is currently controlling the vessel steering.

SteeringManager:**TARGET**

Type *Direction*

Access Get only

Returns direction that the is currently being targeted. If steering is locked to a vector, this will return the calculated direction. If steering is locked to "kill", this will return the vessel's last facing direction.

SteeringManager:**RESETPIDS**()

Returns none

Returns direction that the is currently being targeted. If steering is locked to a vector, this will return the calculated direction. If steering is locked to "kill", this will return the vessel's last facing direction.

SteeringManager:**SHOWFACINGVECTORS**

Type bool

Access Get/Set

Setting this suffix to true will cause the steering manager to display graphical vectors (see *VecDraw*) representing the forward, top, and starboard of the facing direction, as well as the world x, y, and z axis orientation (centered on the vessel). Setting to false will hide the vectors, as will disabling locked steering.

SteeringManager:**SHOWANGULARVECTORS**

Type bool

Access Get/Set

Setting this suffix to true will cause the steering manager to display graphical vectors (see *VecDraw*) representing the current and target angular velocities in the pitch, yaw, and roll directions. Setting to false will hide the vectors, as will disabling locked steering.

SteeringManager:**SHOWTHRUSTVECTORS**

Type bool

Access Get/Set

Setting this suffix to true will cause the steering manager to display graphical vectors (see *VecDraw*) representing the thrust and torque for each active engine. Setting to false will hide the vectors, as will disabling locked steering.

SteeringManager:**SHOWRCSVECTORS**

Type bool

Access Get/Set

Setting this suffix to true will cause the steering manager to display graphical vectors (see *VecDraw*) representing the thrust and torque for each active RCS block. Setting to false will hide the vectors, as will disabling locked steering.

SteeringManager:**SHOWSTEERINGSTATS**

Type bool

Access Get/Set

Setting this suffix to true will cause the steering manager to clear the terminal screen and print steering data each update.

SteeringManager:**WRITECSVFILES**

Type bool

Access Get/Set

Setting this suffix to true will cause the steering manager log the data from all 6 PIDLoops calculating target angular velocity and target torque. The files are stored in the *[KSP Root]GameDatakOSPluginsPluginDatakOS* folder, with one file per loop and a new file created for each new manager instance (i.e. every launch, every revert, and every vessel load). These files can grow quite large, and add up quickly, so it is recommended to only set this value to true for testing or debugging and not normal operation.

SteeringManager:**PITCHTS**

Type scalar

Access Get/Set

Represents the settling time for the PID calculating pitch torque based on target angular velocity. The proportional and integral gain is calculated based on the settling time and the moment of inertia in the pitch direction. Ki = (moment of inertia) * (4 / (settling time)) ^ 2. Kp = 2 * sqrt((moment of inertia) * Ki).

SteeringManager:**YAWTS**

Type scalar

Access Get/Set

Represents the settling time for the PID calculating yaw torque based on target angular velocity. The proportional and integral gain is calculated based on the settling time and the moment of inertia in the yaw direction. Ki = (moment of inertia) * (4 / (settling time)) ^ 2. Kp = 2 * sqrt((moment of inertia) * Ki).

SteeringManager:**ROLLTS**

Type scalar

Access Get/Set

Represents the settling time for the PID calculating roll torque based on target angular velocity. The proportional and integral gain is calculated based on the settling time and the moment of inertia in the roll direction. Ki = (moment of inertia) * (4 / (settling time)) ^ 2. Kp = 2 * sqrt((moment of inertia) * Ki).

SteeringManager:**MAXSTOPPINGTIME**

Type scalar (s)

Access Get/Set

This value is used to limit the turning rate when calculating target angular velocity. The ship will not turn faster than what it can stop in this amount of time. The maximum angular velocity about each axis is calculated as: (max angular velocity) = MAXSTOPPINGTIME * (available torque) / (moment of inertia).

SteeringManager:**ANGLEERROR**

Type scalar (deg)

Access Get only

The angle between the ship's facing direction forward vector and the target direction's forward. This is the combined pitch and yaw error.

SteeringManager:**PITCHERROR**

Type scalar (deg)

Access Get only

The pitch angle between the ship's facing direction and the target direction.

SteeringManager:**YAWERROR**

Type scalar (deg)

Access Get only

The yaw angle between the ship's facing direction and the target direction.

SteeringManager:**ROLLERROR**

Type scalar (deg)

Access Get only

The roll angle between the ship's facing direction and the target direction.

SteeringManager:**PITCHTORQUEADJUST**

Type scalar (kNm)

Access Get/Set

You can set this value to provide an additive bias to the calculated available pitch torque. (available torque) = ((calculated torque) + PITCHTORQUEADJUST * PITCHTORQUEFACTOR.

SteeringManager:**YAWTORQUEADJUST**

Type scalar (kNm)

Access Get/Set

You can set this value to provide an additive bias to the calculated available yaw torque. (available torque) = ((calculated torque) + YAWTORQUEADJUST) * YAWTORQUEFACTOR.

SteeringManager:**ROLLTORQUEADJUST**

Type scalar (kNm)

Access Get/Set

You can set this value to provide an additive bias to the calculated available roll torque. (available torque) = ((calculated torque) + ROLLTORQUEADJUST * ROLLTORQUEFACTOR.

SteeringManager:**PITCHTORQUEFACTOR**

Type scalar (kNm)

Access Get/Set

You can set this value to provide an multiplicative factor bias to the calculated available pitch torque. (available torque) = ((calculated torque) + PITCHTORQUEADJUST * PITCHTORQUEFACTOR.

SteeringManager:**YAWTORQUEFACTOR**

Type scalar (kNm)

Access Get/Set

You can set this value to provide an multiplicative factor bias to the calculated available yaw torque. (available torque) = ((calculated torque) + YAWTORQUEADJUST) * YAWTORQUEFACTOR.

SteeringManager:**ROLLTORQUEFACTOR**

Type scalar (kNm)

Access Get/Set

You can set this value to provide an multiplicative factor bias to the calculated available roll torque. (available torque) = ((calculated torque) + ROLLTORQUEADJUST * ROLLTORQUEFACTOR.

8.5.14 String

A *String* is an immutable sequence of characters in kOS.

Creating strings

Unlike other structures, strings are created with a special syntax:

```
// Create a new string
SET s TO "Hello, Strings!".
```

Strings are immutable. This means, once a string has been created, it can not be directly modified. However, new strings can be created out of existing strings. For example:

```
// Create a new string with "Hello" replaced with "Goodbye"
SET s TO "Hello, Strings!".
SET t TO s:REPLACE("Hello", "Goodbye").
```

NOTE: All string operations are currently case insensive, there are future plans to add mechanisms that will let you choose which style you perfer

Structure

structure `String`

Table 8.21: Members

Suffix	Type	Description
CONTAINS(string)	boolean	True if the given string is contained within this string
ENDSWITH(string)	boolean	True if this string ends with the given string
FIND(string)	integer	Returns the index of the first occurrence of the given string in this string (starting from 0)
FINDAT(string, startAt)	integer	Returns the index of the first occurrence of the given string in this string (starting from startAt)
FINDLAST(string)	integer	Returns the index of the last occurrence of the given string in this string (starting from 0)
FINDLASTAT(string, startAt)	integer	Returns the index of the last occurrence of the given string in this string (starting from startAt)
INDEXOF(string)	integer	Alias for FIND(string)
INSERT(index, string)	*String*	Returns a new string with the given string inserted at the given index into this string
LASTINDEXOF(string)	integer	Alias for FINDLAST(string)
LENGTH	integer	Number of characters in the string
PADLEFT(width)	*String*	Returns a new right-aligned version of this string padded to the given width by spaces
PADRIGHT(width)	*String*	Returns a new left-aligned version of this string padded to the given width by spaces
REMOVE(index,count)	*String*	Returns a new string out of this string with the given count of characters removed starting at the given index
REPLACE(oldString, newString)	*String*	Returns a new string out of this string with any occurrences of oldString replaced with newString
SPLIT(separator)	*String*	Breaks this string up into a list of smaller strings on each occurrence of the given separator
STARTSWITH(string)	boolean	True if this string starts with the given string
SUBSTRING(start, count)	*String*	Returns a new string with the given count of characters from this string starting from the given start position
TOLOWER	*String*	Returns a new string with all characters in this string replaced with their lower case versions
TOUPPER	*String*	Returns a new string with all characters in this string replaced with their upper case versions
TRIM	*String*	returns a new string with no leading or trailing whitespace
TRIMEND	*String*	returns a new string with no trailing whitespace
TRIMSTART	*String*	returns a new string with no leading whitespace

String:**`CONTAINS`**(string)

Parameters

- **`string`** – *String* to look for

Return type boolean

True if the given string is contained within this string.

String:**ENDSWITH** (string)

Parameters

- **string** – *String* to look for

Return type boolean

True if this string ends with the given string.

String:**FIND** (string)

Parameters

- **string** – *String* to look for

Return type *String*

Returns the index of the first occurrence of the given string in this string (starting from 0).

String:**FINDAT** (string, startAt)

Parameters

- **string** – *String* to look for
- **startAt** – integer index to start searching at

Return type *String*

Returns the index of the first occurrence of the given string in this string (starting from startAt).

String:**FINDLAST** (string)

Parameters

- **string** – *String* to look for

Return type *String*

Returns the index of the last occurrence of the given string in this string (starting from 0)

String:**FINDLASTAT** (string, startAt)

Parameters

- **string** – *String* to look for
- **startAt** – integer index to start searching at

Return type *String*

Returns the index of the last occurrence of the given string in this string (starting from startAt)

String:**INDEXOF** (string)
Alias for FIND(string)

String:**INSERT** (index, string)

Parameters

- **index** – integer index to add the string at
- **string** – *String* to insert

Return type *String*

Returns a new string with the given string inserted at the given index into this string

String:**LASTINDEXOF** (string)
Alias for FINDLAST(string)

String:**LENGTH**

Type integer

Access Get only

Number of characters in the string

String:**PADLEFT**(width)

Parameters

- **width** – integer number of characters the resulting string will contain

Return type *String*

Returns a new right-aligned version of this string padded to the given width by spaces.

String:**PADRIGHT**(width)

Parameters

- **width** – integer number of characters the resulting string will contain

Return type *String*

Returns a new left-aligned version of this string padded to the given width by spaces.

String:**REMOVE**(index,count)

Parameters

- **index** – integer position of the string from which characters will be removed from the resulting string
- **count** – integer number of characters that will be removing from the resulting string

Return type *String*

Returns a new string out of this string with the given count of characters removed starting at the given index.

String:**REPLACE**(oldString,newString)

Parameters

- **oldString** – *String* to search for
- **newString** – *String* that all occurances of oldString will be replaced with

Return type *String*

Returns a new string out of this string with any occurrences of oldString replaced with newString.

String:**SPLIT**(separator)

Parameters

- **separator** – *String* delimiter on which this string will be split

Returns *List*

Breaks this string up into a list of smaller strings on each occurrence of the given separator. This will return a list of strings, none of which will contain the separator character(s).

String:**STARTSWITH**(string)

Parameters

- **string** – *String* to look for

Return type boolean

True if this string starts with the given string .

String:**SUBSTRING**(start,count)

Parameters

- **start** – (integer) starting index (from zero)
- **count** – (integer) resulting length of returned *String*

Returns *String*

Returns a new string with the given count of characters from this string starting from the given start position.

String:**TOLOWER**

Type *String*

Access Get only

Returns a new string with all characters in this string replaced with their lower case versions

String:**TOUPPER**

Type *String*

Access Get only

Returns a new string with all characters in this string replaced with their upper case versions

String:**TRIM**

Type *String*

Access Get only

returns a new string with no leading or trailing whitespace

String:**TRIMEND**

Type *String*

Access Get only

returns a new string with no trailing whitespace

String:**TRIMSTART**

Type *String*

Access Get only

returns a new string with no leading whitespace

Access to Individual Characters

All string indexes start counting at zero. (The characters are numbered from 0 to N-1 rather than from 1 to N.)

string[expression] operator: access the character at position 'expression'. Any arbitrary complex expression may be used with this syntax, not just a number or variable name.

FOR VAR IN STRING { ... }. *A type of loop* in which var iterates over all the characters of the string from 0 to LENGTH-1.

Examples:

```
                                                                    // CORRECT OUTPUTS
SET s TO "Hello, Strings!".                                         // ----------
PRINT "Original String:                " + s.                       // Hello, Strings!
PRINT "string[7]:                      " + s[7].                    // S
PRINT "LENGTH:                         " + s:LENGTH.                // 15
PRINT "SUBSTRING(7, 6):                " + s:SUBSTRING(7, 6).       // String
PRINT "CONTAINS(''ring''):             " + s:CONTAINS("ring").      // True
PRINT "CONTAINS(''bling''):            " + s:CONTAINS("bling").     // False
PRINT "ENDSWITH(''ings!''):            " + s:ENDSWITH("ings!").     // True
PRINT "ENDSWITH(''outs!''):            " + s:ENDSWITH("outs").      // False
PRINT "FIND(''l''):                    " + s:FIND("l").             // 2
PRINT "FINDLAST(''l''):                " + s:FINDLAST("l").         // 3
PRINT "FINDAT(''l'', 0):               " + s:FINDAT("l", 0).        // 2
PRINT "FINDAT(''l'', 3):               " + s:FINDAT("l", 3).        // 3
PRINT "FINDLASTAT(''l'', 9):           " + s:FINDLASTAT("l", 9).    // 3
PRINT "FINDLASTAT(''l'', 2):           " + s:FINDLASTAT("l", 2).    // 2
PRINT "INSERT(7, ''Big ''):            " + s:INSERT(7, "Big ").     // Hello, Big Strings!

PRINT " ".
PRINT "                                |---- 18 ----|".
PRINT "PADLEFT(18):                    " + s:PADLEFT(18).           //    Hello, Strings!
PRINT "PADRIGHT(18):                   " + s:PADRIGHT(18).          // Hello, Strings!
PRINT " ".

PRINT "REMOVE(1, 3):                   " + s:REMOVE(1, 3).                 // Ho, Strings!
PRINT "REPLACE(''Hell'', ''Heaven''): " + s:REPLACE("Hell", "Heaven").     // Heaveno, Strings!
PRINT "STARTSWITH(''Hell''):           " + s:STARTSWITH("Hell").           // True
PRINT "STARTSWITH(''Heaven''):         " + s:STARTSWITH("Heaven").         // False
PRINT "TOUPPER:                        " + s:TOUPPER().                    // HELLO, STRINGS!
PRINT "TOLOWER:                        " + s:TOLOWER().                    // hello, strings!

PRINT " ".
PRINT "''  Hello!  '':TRIM():          " + "  Hello!  ":TRIM().           // Hello!
PRINT "''  Hello!  '':TRIMSTART():     " + "  Hello!  ":TRIMSTART().      // Hello!
PRINT "''  Hello!  '':TRIMEND():       " + "  Hello!  ":TRIMEND().        //   Hello!

PRINT " ".
PRINT "Chained: " + "Hello!":SUBSTRING(0, 4):TOUPPER():REPLACE("ELL", "ELEPHANT").  // HELEPHANT
```

8.5.15 Terminal

The TERMINAL identifier refers to a special structure that lets you access some of the information about the screen you are running on.

Structure

structure `Terminal`

Table 8.22: Members

Suffix	Type	Get/Set	Description
WIDTH	integer	get and set	Terminal width in characters
HEIGHT	integer	get and set	Terminal height in characters
REVERSE	Boolean	get and set	Determines if the screen is displayed with foreground and background colors swapped.
VISUALBEEP	Boolean	get and set	Turns beeps into silent visual screen flashes instead.

Terminal:**WIDTH**

Access Get/Set

Type integer.

If you read the width it will return a number of character cells wide the terminal is. If you set this value, it will cause the terminal to resize. If there's multiple terminals connected to the same CPU part via telnet clients, then kOS will attempt to keep them all the same size, and one terminal being resized will resize them all. (caveat: Some terminal types cannot be resized from the server side, and therefore this doesn't always work in both directions).

This setting is different per kOS CPU part. Different terminal windows can have different settings for this value.

Terminal:**HEIGHT**

Access Get/Set

Type integer.

If you read the width it will return a number of character cells tall the terminal is. If you set this value, it will cause the terminal to resize. If there's multiple terminals connected to the same CPU part via telnet clients, then kOS will attempt to keep them all the same size, and one terminal being resized will resize them all. (caveat: Some terminal types cannot be resized from the server side, and therefore this doesn't always work in both directions).

This setting is different per kOS CPU part. Different terminal windows can have different settings for this value.

Terminal:**REVERSE**

Access Get/Set

Type Boolean.

If true, then the terminal window is currently set to show the whole screen in reversed color - swapping the background and foreground colors. Both the telnet terminals and the in-game GUI terminal respond to this setting equally.

Note, this setting can also be toggled with a radio-button on the in-game GUI terminal window.

This setting is different per kOS CPU part. Different terminal windows can have different settings for this value.

Terminal:**VISUALBEEP**

Access Get/Set

Type Boolean.

If true, then the terminal window is currently set to show any BEEP characters by silently flashing the screen for a moment (inverting the background/foreground for a fraction of a second), instead of making a sound.

Note, this setting can also be toggled with a radio-button on the in-game GUI terminal window.

This will only typically affect the in-game GUI terminal window, and **not a telnet client's** terminal window.

To affect the window you are using in a telnet session, you will have to use whatever your terminal or terminal emulator's local settings panel has for it. Most do have some sort of visual beep setting, but it is usually not settable via a control character sequence sent across the connection. The terminals are designed to assume it's a local user preference that isn't overridable by the software you are running.

This setting is different per kOS CPU part. Different terminal windows can have different settings for this value.

8.5.16 Time Span

In several places the game uses a *TimeSpan* format. This is a strucure that gives the time in various formats. It also allows you to peform arithmetic on the time.

TimeSpan represents *SIMULATED* time

When you are examining a *TimeSpan* you are looking at the "in character" **simulated** time, not the "out of character" real world time. This is a very important distinction to remember, as the following points illustrate:

- A *TimeSpan* does not count the time that was passing while the game was paused.
- If you turn off your computer and don't play the game for several days, the *TimeSpan* does not count this time.
- If your game lags and stutters such that the simulation is taking 2 seconds of real time to calculate 1 second of game time, then the number of seconds that have passed according to a *TimeSpan* will be fewer than the number of seconds that have passed in the real world.

This allows you to use a *TimeSpan* such as is returned by the TIME special variable to make correct physics calculations.

Special variable TIME

TIME

Access Get only

Type *TimeSpan*

The special variable **TIME** is used to get the current time.

Any time you perform arithmetic on **TIME** you get a result back that is also a *TimeSpan*. In other words, TIME is a *TimeSpan*, but TIME + 100 is also a *TimeSpan*.

Note that Kerbals do not have the concept of "months":

```
TIME                // Gets the current universal time
TIME:CLOCK          // Universal time in H:M:S format(1:50:26)
TIME:CALENDAR       // Year 1, day 134
TIME:YEAR           // 1
TIME:DAY            // 134
TIME:HOUR           // 1
TIME:MINUTE         // 50
TIME:SECOND         // 26
TIME:SECONDS        // Total Seconds since campaign began
```

Using TIME to detect when the physics have been updated 'one tick'

kOS programs run however fast your computer's animation rate will allow, which can flow and change from one moment to the next depending on load. However, the physics of the universe get updated at a fixed rate according to your game settings (the default, as of KSP 0.25, is 25 physics updates per second)

You can use the TIME special variable to detect whether or not a real physics 'tic' has occurred yet, which can be important for scripts that need to take measurements from the simulated universe. If no physics tic has occurred, then TIME will still be exactly the same value.

Warning: Please be aware that the kind of calendar *TimeSpan*'s use will depend on your KSP settings. The main KSP game supports both Kerbin time and Earth time and changing that setting will affect how *TimeSpan* works in kOS.
The difference is whether 1 day = 6 hours or 1 day = 24 hours.

Warning: Beware the pitfall of confuising the *TimeSpan:SECOND* (singular) suffix with the *TimeSpan:SECONDS* (plural) suffix.

TimeSpan:SECOND
: This is the number of **remainder** seconds leftover after all whole-number minutes, hours, days, and years have been subtracted out, and it's never outside the range [0..60). It's essentially the 'seconds hand' on a clock.

TimeSpan:SECONDS
: This is the number of seconds total if you want to represent time as just a simple flat number without all the components. It's the total count of the number of seconds since the beginning of time (Epoch). Because it's a floating point number, it can store times less than 1 second. Note this is a measure of how much simulated Kerbal time has passed since the game began. People experienced at programming will be familiar with this concept. It's the Kerbal's version of "unix time".
The epoch (time zero) in the KSP game is the time at which you first started the new campaign. All campaign games begin with the planets in precisely the same position and the clock set to zero years, zero days, zero hours, and so on.

Warning: Beware that the times returned from *FileInfo* for the time a file was modified or created are NOT in this *TimeSpan* structure but instead are just raw strings. That is because they represent the time the file was affected in the real world and NOT times taken from the KSP simulation clock. That is a necessity because your files in the Archive exist globally across all multiple saved games. Different saved games won't have synchronized calendars with each other.

structure `TimeSpan`

Suffix	Type	Description
CLOCK	string	"HH:MM:SS"
CALENDAR	string	"Year YYYY, day DDD"
SECOND	integer (0-59)	Second-hand number
MINUTE	integer (0-59)	Minute-hand number
HOUR	integer (0-5)	Hour-hand number
DAY	integer (1-426)	Day-hand number
YEAR	integer	Year-hand number
SECONDS	Number (float)	Total Seconds since Epoch

TimeSpan:**`CLOCK`**

Access Get only

Type string

Time in (HH:MM:SS) format.

TimeSpan:**CALENDAR**

Access Get only

Type string

Day in "Year YYYY, day DDD" format. (Kerbals don't have 'months'.)

TimeSpan:**SECOND**

Access Get only

Type integer (0-59)

Second-hand number.

TimeSpan:**MINUTE**

Access Get only

Type integer (0-59)

Minute-hand number

TimeSpan:**HOUR**

Access Get only

Type integer (0-5) or (0-23)

Hour-hand number. Kerbin has six hours in its day.

TimeSpan:**DAY**

Access Get only

Type integer (1-426) or (1-356)

Day-hand number. Kerbin has 426 days in its year.

TimeSpan:**YEAR**

Access Get only

Type integer

Year-hand number

TimeSpan:**SECONDS**

Access Get only

Type Number (float)

Total Seconds since Epoch. Epoch is defined as the moment your current saved game's universe began (the point where you started your campaign). Can be very precise.

8.5.17 Drawing Vectors on the Screen

VECDRAW() or VECDRAWARGS()

Build the suffix fields one at a time using the *VecDraw* empty construction function. This creates an empty *VecDraw* with nothing populated yet. You have to follow it up with calls to the suffixes as shown here:

```
SET anArrow TO VECDRAW().
SET anArrow:VEC TO V(a,b,c).
SET anArrow:SHOW TO true.
// At this point you have done the minimal necessary to make the arrow appear
```

```
    // and it shows up on the scren immediately.

    // Further options can also be set:
    SET anArrow:START TO V(0,0,0).
    SET anArrow:COLOR TO RGB(1,0,0).
    SET anArrow:LABEL TO "See the arrow?".
    SET anArrow:SCALE TO 5.0.
```

VECDRAW(start, vec, color, label, scale, show) or VECDRAWARGS(start, vec, color, label, scale, show)

This variant builds the *VecDraw* all at once. It lets you specify all of the attributes in a list of arguments at once:

```
    SET anArrow TO VECDRAW(
          V(0,0,0),
          V(a,b,c),
          RGB(1,0,0),
          "See the arrow?",
          5.0,
          TRUE
        ).

    // Saying VECDRAWARGS gives the same exact result:
    SET anArrow TO VECDRAWARGS(
          V(0,0,0),
          V(a,b,c),
          RGB(1,0,0),
          "See the arrow?",
          5.0,
          TRUE
        ).
```

The above two examples make the same thing. The arrow should be visible on both the map view and the in-flight view, but on the map view it will have to be a long arrow to be visible. *VecDraw*'s do not auto-update for changes in the vector like a LOCK would, but if you repeatedly SET the :VEC suffix in a loop, it will adjust the arrow picture to match as you do so:

```
set xAxis to VECDRAWARGS( V(0,0,0), V(1,0,0), RGB(1.0,0.5,0.5), "X axis", 5, TRUE ).
set yAxis to VECDRAWARGS( V(0,0,0), V(0,1,0), RGB(0.5,1.0,0.5), "Y axis", 5, TRUE ).
set zAxis to VECDRAWARGS( V(0,0,0), V(0,0,1), RGB(0.5,0.5,1.0), "Z axis", 5, TRUE ).
```

To make a *VecDraw* disappear, you can either set its *VecDraw:SHOW* to false or just UNSET the variable, or re-assign it. An example using *VecDraw* can be seen in the documentation for *POSITIONAT()*.

Note: Historical note: Once upon a time VECDRAW() took zero arguments, and VECDRAWARGS() took the 6 arguments. This was before we added the ability to have varying numbers of arguments for built-in functions. Now that varying arguments exist, both names have become synonyms for the same thing, for backward compatibility. You can use them interchangably.

CLEARVECDRAWS()

Sets all visible vecdraws to invisible, everywhere in this kOS CPU. This is useful if you have lost track of the handles to them and can't turn them off one by one, or if you don't have the variable scopes present anymore to access the variables that hold them. The system does attempt to clear any vecdraws that go "out of scope", however the "closures" that keep local variables alive for LOCK statements and for other reasons can keep them from every truely going away in some circumstances. To make the arrow drawings all go away, just call CLEARVECDRAWS() and it will have the same effect as if you had done `SET varname:show to FALSE` for all vecdraw varnames in the entire system.

structure VecDraw

This is a structure that allows you to make a drawing of a vector on the screen in map view or in flight view.

Table 8.23: Members

Suffix	Type	Description
START	*Vector*	Start position of the vector
VEC	*Vector*	The vector to draw
COLOR	*Color*	Color of the vector
COLOUR		Same as *COLOR*
LABEL	string	Text to show next to vector
SCALE	integer	Scale *START* and *VEC*
SHOW	boolean	Draw vector to screen

VecDraw:**START**

Access Get/Set

Type *Vector*

Optional, defaults to V(0,0,0) - position of the tail of the vector to draw in SHIP-RAW coords. V(0,0,0) means the ship Center of Mass.

VecDraw:**VEC**

Access Get/Set

Type *Vector*

Mandatory - The vector to draw, SHIP-RAW Coords.

VecDraw:**COLOR**

Access Get/Set

Type *Color*

Optional, defaults to white. This is the color to draw the vector. There is a hard-coded fade effect where the tail is a bit more transparent than the head.

VecDraw:**COLOUR**

Access Get/Set

Type *Color*

Alias for *VecDraw:COLOR*

VecDraw:**LABEL**

Access Get/Set

Type string

Optional, defaults to “”. Text to show on-screen at the midpoint of the vector.

VecDraw:**SCALE**

Access Get/Set

Type integer

Optional, defauls to 1. Scalar to multiply by both the START and the VEC

VecDraw:**SHOW**

Access Get/Set

Type boolean

Set to true to make the arrow appear, false to hide it. Defaults to false until you're ready to set it to true.

8.6 Core

Core represents your ability to identify and interact directly with the running kOS processor. You can use it to access the parent vessel, or to perform operations on the processor's part. You obtain a CORE structure by using the bound variable `core`.

structure CORE

Table 8.24: Members

Suffix	Type
PART	*Part*
VESSEL	*Vessel*
ELEMENT	*Element*
VERSION	*Version*
BOOTFILENAME	*String*
CURRENTVOLUME	*Volume*

CORE:**PART**

Type Part

Access Get only

The Part object for the current processor.

CORE:**VESSEL**

Type *VesselTarget*

Access Get only

The vessel containing the current processor.

CORE:**ELEMENT**

Type *Element*

Access Get only

The element object containing the current processor.

CORE:**VERSION**

Type *VersionInfo*

Access Get only

The kOS version currently running.

CORE:**BOOTFILENAME**

Type *String*

Access Get or Set

The filename for the boot file on the current processor. This may be set to an empty string *""* or to *"None"* to disable the use of a boot file.

CORE:**CURRENTVOLUME**

Type *Volume*

Access Get only

The currently selected volume for the current processor. This may be useful to prevent deleting files on the Archive, or for interacting with multiple local hard disks.

CHAPTER

NINE

ADDON REFERENCE

This section is for ways in which kOS has special case exceptions to its normal generic behaviours, in order to accommodate other KSP mods. If you don't use any of KSP mods mentioned, you don't need to read this section.

9.1 Action Groups Extended

- Download: https://github.com/SirDiazo/AGExt/releases
- Forum thread, including full instructions: http://forum.kerbalspaceprogram.com/threads/74195

Increase the action groups available to kOS from 10 to 250. Also adds the ability to edit actions in flight as well as the ability to name action groups so you can describe what a group does.

Includes a Script Trigger action that can be used to control a running program and visual feedback if an action group is currently activated.

Usage: Adds action groups AG11 through AG250 to kOS that are interacted with the same way as the AG1 through AG10 bindings in base kOS are.

Anywhere you use `AG1`, you can use `AG15` in the same way. (AG11 through AG250 explicitly behave the same as the 10 stock groups. Please file a bug report if they do not.)

Script Trigger action: Installing AGX adds the "Script Trigger" action to all kOS computer parts. This action is a null action that does not activate anything but serves as a placeholder to enhance action groups in kOS.

When an action group has the Script Trigger action assigned, on that action group you can now:

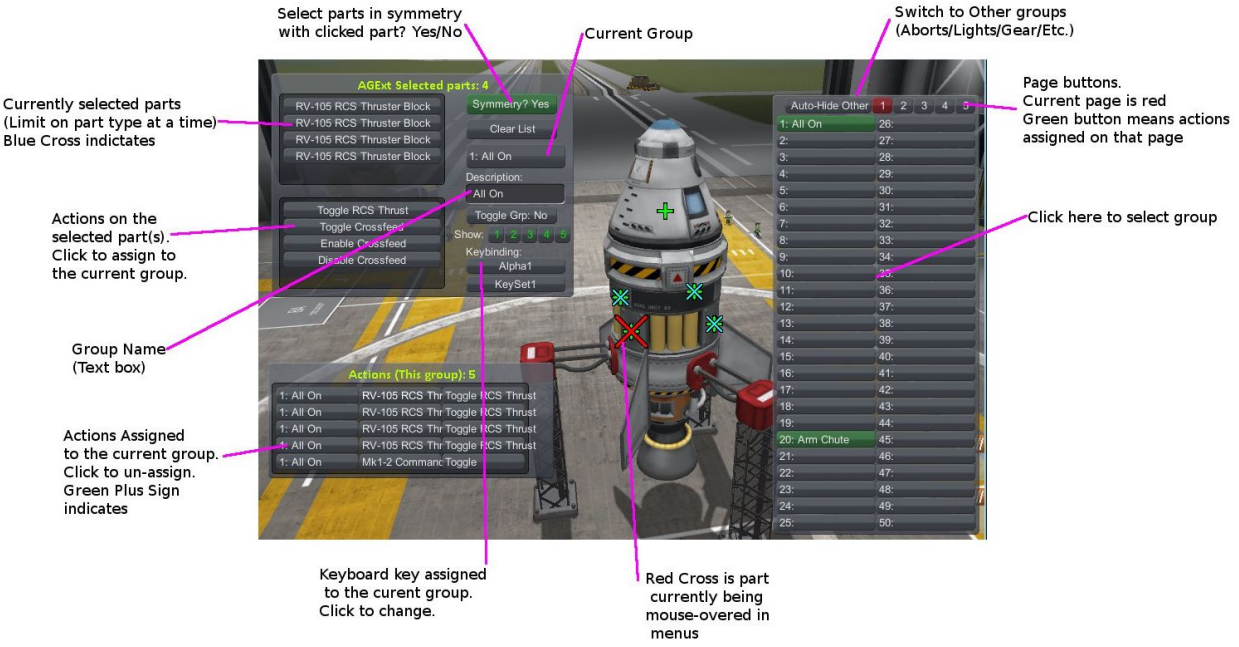
- Name the action group so you remember what that action group does in your code when you trigger it.
- Activate the action group with a mouse click on-screen, no more tying up your entire keyboard with various script trigger keys.
- Enable group state feedback so you can have your script change the groups state as feedback as to what the script is doing. Green being On and Red being Off. (Toggle option in AGX.)

Basic Quick Start:

Note that this mod only adds action groups 11 through 250, it does not change how action groups 1 through 10 behave in any way and groups 11 through 250 should behave the same way.

Known limitations (Action groups 11 through 250 only):

- On a nearby vessel that is not your current focus, an action group with no actions assigned will always return a state of False and can not be set to a state of true via the "AG15 on." command. Assign the Script Trigger action as a work-around for this.


Select parts in symmetry with clicked part? Yes/No
Current Group
Switch to Other groups (Aborts/Lights/Gear/Etc.)
Currently selected parts (Limit on part type at a time) Blue Cross indictates
Page buttons. Current page is red Green button means actions assigned on that page
Actions on the selected part(s). Click to assign to the current group.
Click here to select group
Group Name (Text box)
Actions Assigned to the current group. Click to un-assign. Green Plus Sign indicates
Keyboard key assigned to the curent group. Click to change.
Red Cross is part currently being mouse-overed in menus
AGExt Selected parts: 4
RV-105 RCS Thruster Block
Symmetry? Yes
Clear List
1: All On
Description:
All On
Toggle RCS Thrust
Toggle Crossfeed
Enable Crossfeed
Disable Crossfeed
Toggle Grp: No
Show: 1 2 3 4 5
Keybinding:
Alpha1
KeySet1
Actions (This group): 5
Auto-Hide Other 1 2 3 4 5
20: Arm Chute


Left-Click: Show/Hide Actions Window
Right-Click: Show/Hide 'AGX Settings' Menu
VEL
LA
AGX
AGX Settings
Show KeyCodes
Edit Actions
Reset Windows
KeySet1
Actions
Group1
1: All On
Alpha1
20: Arm Chute
None
Show Keycodes: Show/Hide keycodes in window below so group names have more room
Edit Actions: Show/Hide editor windows
Reset Windows: Reset all window locations if one gets trapped off screen.
Available Action Groups
Click to activate group or
press keyboard key shown.

- At this point, AG11 through AG250 do not officially support RemoteTech through kOS. (Support will happen once all three mods involved have updated to KSP version 1.0 and made any internal changes necessary.) All three mods can be installed at the same time without issue, just be aware there may be unexpected behaviour when using action groups 11 through 250 from a kOS script in terms of RemoteTech signal delay and connection state.

Action state monitoring

Note that the state of action groups is tracked on a per-action basis, rather then on a per-group basis. This results in the group state being handled differently.

- The Script Trigger action found on the kOS computer module is not subject to the below considerations and is the recommended action to use when interacting with a running kOS script.
- The state of actions are monitored on the part and updated automatically. A closed solar panel will return a state of false for all it's actions. (Extend Panels, Retract Panels, Toggle Panels) When you extend the solar panel with either the Extend Panels or Toggle Panels action, all three actions will change to a state of True. Retract the panels and the state of all three actions will become False. Note that this state will update in any action group that contains that action, not just the action group that was activated.
- This can result in an action group have actions in a mixed state where some actions are on and some are off. In this case querying the group state will result in a state of False. For the purposes of the group state being True or False, if *all* actions in the action group are true, the group state will return true. If *any* actions in the group are false,the group state with return False.
- When an action triggers an animation, the state of the action will be uncertain until the animation finishes playing. Some parts will report True during the animation and some will report False. It depends on how the part creator set things up and not something AGX can control.
- For clarity, visual feedback can be provided of the current state of an action group. When editing action groups, find the "Toggle Grp." button just below the text entry field for the group name in the main AGX window and enable it. (It is enabled/disabled for the current action group when you click the button.) Once you do this, the text displaying that group will change from gray to colored. Green: Group is activated (state True). Red: Group is deactivated (state False). Yellow: Group is in a mixed state, will return a state False when queried.
- It is okay to activate an already activated group and deactivate a non-activated group. Actions in the group will still try to execute as normal. Exact behaviour of a specific action will depend on how the action's creator set things up.

Example code:

Print to the terminal any time you activate action group 15. Use this to change variables within a running kOS script and the "Script Trigger" action found on the kOS computer part:

```
AG15 on. // Activate action group 15.
print AG15. // Print action group 15's state to the terminal. (True/False)

on AG15 { //Prints "Action group 15 clicked!" to the console when AG15 is toggled, either via "AG15 o
  print "Action group 15 clicked!".
  preserve.
}
```

Animation Delay:

- Using the above code on a stock solar panel's Toggle Panels action, the player activates AG15, AG15's state goes from false to true and the actions are triggered. `AG15 False -> True` and prints to the terminal.
- On it's next update pass (100ms to 250ms later), AGX checks AG15's state and sees the solar panel is still deploying which means that AG15's state is false and so sets it that way. `AG15 True -> False` and prints to the terminal.

- A few seconds later, the solar panel finishes it's deployment animation. On it's next update pass AGX checks AG15's state and sees the solar panel is now deployed which means that AG15's state is now true and so sets it that way. `AG15 False -> True` and prints to the terminal a third time.

As a workaround, you need to add a cooldown:

```
declare cooldownTimeAG15 to 0.
on AG15 {
  if cooldownTimeAG15 + 10 < time:seconds {
    print "Solar Panel Toggled!".
    set cooldownTimeAG15 to time.
  }
  preserve.
}
```

Note the 10 in the second line, that is your cooldown time in seconds. Set this to a number of seconds that is longer then your animation time and the above code will limit whatever is inside the IF statement so it can only activate after 10 seconds have passed since the previous activation and will not try to activate a second time while the solar panel animation is still playing.

9.2 RemoteTech

RemoteTech is a modification for Squads "Kerbal Space Program" (KSP) which overhauls the unmanned space program. It does this by requiring unmanned vessels have a connection to Kerbal Space Center (KSC) to be able to be controlled. This adds a new layer of difficulty that compensates for the lack of live crew members.

- Download: http://kerbalstuff.com/mod/134/RemoteTech
- Sources: https://github.com/RemoteTechnologiesGroup/RemoteTech
- Documentation: http://remotetechnologiesgroup.github.io/RemoteTech/

9.2.1 Interaction with kOS

When you have RemoteTech installed you can only interact with the core's terminal when you have a connection to KSC on any unmanned craft. Scripts launched when you still had a connection will continue to execute even if your unmanned craft loses connection to KSC. But you should note, that when there is no connection to KSC the archive volume is inaccessible. This will require you to plan ahead and copy necessary scripts for your mission to probe hard disk, if your kerbals and/or other scripts need to use them while not connected.

If you launch a manned craft while using RemoteTech, you are still able to input commands from the terminal even if you do not have a connection to the KSC. The archive will still be inaccessible without a connection to the KSC. Under the current implementation, there is no delay when accessing the archive with a local terminal. This implementation may change in the future to account for delays in reading and writing data over the connection.

It is possible to activate/deactivate RT antennas, as well as set their targets using kOS:

```
SET p TO SHIP:PARTSNAMED("mediumDishAntenna")[0].
SET m to p:GETMODULE("ModuleRTAntenna").
m:DOEVENT("activate").
m:SETFIELD("target", "mission-control").
// or
m:SETFIELD("target", mun).
m:SETFIELD("target", "minmus").
```

Acceptable values for *"target"* are: *"no-target"*, *"active-vessel"*, *"mission-control"*, a *Body*, a *Vessel*, or a string containing the name of a body or vessel.

Starting version 0.17 of kOS you can access structure RTAddon via *ADDONS:RT*.

structure RTAddon

Suffix	Type	Description
AVAILABLE	bool(readonly)	True if RT is installed and RT integration enabled.
DELAY(vessel)	double	Get shortest possible delay to given *Vessel*
KSCDELAY(vessel)	double	Get delay from KSC to given *Vessel*
HASCONNECTION(vessel)	bool	True if given *Vessel* has any connection
HASKSCCONNECTION(vessel)	bool	True if given *Vessel* has connection to KSC
HASLOCALCONTROL(vessel)	bool	True if given *Vessel* has local control

RTADDON:**AVAILABLE**

Type bool

Access Get only

True if RT is installed and RT integration enabled.

RTAddon:**DELAY**(vessel)

Parameters

- **vessel** – *Vessel*

Returns (double) seconds

Returns shortest possible delay for *vessel* (Will be less than KSC delay if you have a local command post).

RTAddon:**KSCDELAY**(vessel)

Parameters

- **vessel** – *Vessel*

Returns (double) seconds

Returns delay in seconds from KSC to *vessel*.

RTAddon:**HASCONNECTION**(vessel)

Parameters

- **vessel** – *Vessel*

Returns bool

Returns True if *vessel* has any connection (including to local command posts).

RTAddon:**HASKSCCONNECTION**(vessel)

Parameters

- **vessel** – *Vessel*

Returns bool

Returns True if *vessel* has connection to KSC.

RTAddon:**HASLOCALCONTROL**(vessel)

Parameters

- **vessel** – *Vessel*

Returns bool

Returns True if *vessel* has local control (and thus not requiring a RemoteTech connection).

9.3 Kerbal Alarm Clock

- Download: https://github.com/TriggerAu/KerbalAlarmClock/releases
- Alternative download https://kerbalstuff.com/mod/231/Kerbal%20Alarm%20Clock
- Forum thread, including full instructions: http://forum.kerbalspaceprogram.com/threads/24786

The Kerbal Alarm Clock is a plugin that allows you to create reminder alarms at future periods to help you manage your flights and not warp past important times.

Creator of the KAC provides API for integration with other mods. In KOS we provide limited access to KAC alarms via following structure and functions.

Access structure KACAddon via *ADDONS:KAC*.

structure KACAddon

Suffix	Type	Description
AVAILABLE	bool(readonly)	True if KAC is installed and KAC integration enabled.
ALARMS()	List	List all alarms

KACAddon:**AVAILABLE**

Type bool

Access Get only

True if KAC is installed and KAC integration enabled. Example of use:

```
if ADDONS:KAC:AVAILABLE
{
    //some KAC dependent code
}
```

KACAddon:**ALARMS**()

Returns List of *KACAlarm* objects

List **all** the alarms set up in Kerbal Alarm Clock. Example of use:

```
for i in ADDONS:KAC:ALARMS
{
        print i:NAME + " - " + i:REMAINING + " - " + i:TYPE+ " - " + i:ACTION.
}
```

structure KACAlarm

Suffix	Type	Description
ID	string (readonly)	Unique identifier
NAME	string	Name of the alarm
ACTION	string	What should the Alarm Clock do when the alarm fires
TYPE	string (readonly)	What type of Alarm is this - affects icon displayed and some calc options
NOTES	string	Long description of the alarm (optional)
REMAINING	scalar (s)	Time remaining until alarm is triggered
REPEAT	bool	Should the alarm be repeated once it fires
REPEATPERIOD	scalar (s)	How long after the alarm fires should the next alarm be set up
ORIGINBODY	string	Name of the body the vessel is departing from
TARGETBODY	string	Name of the body the vessel is arriving at

KACAlarm:**ID**

Type string

Access Get only

Unique identifier of the alarm.

KACAlarm:**NAME**

Type string

Access Get/Set

Name of the alarm. Displayed in main KAC window.

KACAlarm:**ACTION**

Type string

Access Get/Set

Should be one of the following

- *MessageOnly* - Message Only-No Affect on warp
- *KillWarpOnly* - Kill Warp Only-No Message
- *KillWarp* - Kill Warp and Message
- *PauseGame* - Pause Game and Message

If set incorrectly will log a warning in Debug log and revert to previous or default value.

KACAlarm:**TYPE**

Type string

Access Get only

Can only be set at Alarm creation. Could be one of the following as per API

- Raw (default)
- Maneuver
- ManeuverAuto
- Apoapsis
- Periapsis
- AscendingNode
- DescendingNode
- LaunchRendevous
- Closest
- SOIChange
- SOIChangeAuto
- Transfer
- TransferModelled
- Distance
- Crew

•EarthTime

Warning: Unless you are 100% certain you know what you're doing, create only "Raw" AlarmTypes to avoid unnecessary complications.

KACAlarm:**NOTES**

Type string

Access Get/Set

Long description of the alarm. Can be seen when alarm pops or by double-clicking alarm in UI.

Warning: This field may be reserved in the future version of KAC-KOS integration for automated script execution upon triggering of the alarm.

KACAlarm:**REMAINING**

Type double

Access Get only

Time remaining until alarm is triggered.

KACAlarm:**REPEAT**

Type bool

Access Get/Set

Should the alarm be repeated once it fires.

KACAlarm:**REPEATPERIOD**

Type double

Access Get/Set

How long after the alarm fires should the next alarm be set up.

KACAlarm:**ORIGINBODY**

Type string

Access Get/Set

Name of the body the vessel is departing from.

KACAlarm:**TARGETBODY**

Type string

Access Get/Set

Name of the body the vessel is arriving to.

9.3.1 Available Functions

Function	Description
ADDALARM(AlarmType, UT, Name, Notes)	Create new alarm of AlarmType at UT
LISTALARMS(alarmType)	List alarms with type *alarmType*.
DELETEALARM(alarmID)	Delete alarm with ID = alarmID

ADDALARM(AlarmType, UT, Name, Notes)

Creates alarm of type *KACAlarm:ALARMTYPE* at *UT* with *Name* and *Notes* attributes set. Attaches alarm to current *CPU Vessel*. Returns *KACAlarm* object if creation was successful and empty string otherwise:

```
set na to addAlarm("Raw",time:seconds+300, "Test", "Notes").
print na:NAME. //prints 'Test'
set na:NOTES to "New Description".
print na:NOTES. //prints 'New Description'
```

LISTALARMS (alarmType)

If *alarmType* equals "All", returns *List* of *all* *KACAlarm* objects attached to current vessel or have no vessel attached. Otherwise returns *List* of all *KACAlarm* objects with *KACAlarm:TYPE* equeal to *alarmType* and attached to current vessel or have no vessel attached.:

```
set al to listAlarms("All").
for i in al
{
    print i:ID + " - " + i:name.
}
```

DELETEALARM (alarmID)

Deletes alarm with ID equal to alarmID. Returns True if successful, false otherwise:

```
set na to addAlarm("Raw",time:seconds+300, "Test", "Notes").
if (DELETEALARM(na:ID))
{
    print "Alarm Deleted".
}
```

9.4 Infernal Robotics

- Download: http://kerbal.curseforge.com/ksp-mods/220267
- Alternative download: https://kerbalstuff.com/mod/8/Magic_Smoke_Industries_Infernal_Robotics
- Forum thread, including full instructions: http://forum.kerbalspaceprogram.com/threads/116064

Infernal Robotics introduces robotics parts to the game, letting you create moving or spinning contraptions that just aren't possible under stock KSP. .. figure:: http://i.imgur.com/O94LBvF.png

Starting version 0.20 of the Infernal Robotics, mod creators introduced API to for easier access to robotic features.

Access structure IRAddon via *ADDONS:IR*.

structure IRAddon

Suffix	Type	Description
AVAILABLE	bool(read-only)	Returns True if mod Infernal Robotics is installed, available to KOS and applicable to current craft.
GROUPS	List (readonly)	Lists all Servo Groups for current focused vessel
ALLSERVOS	List (readonly)	Lists all Servos for current focused vessel

IRAddon:**AVAILABLE**

Type bool

Access Get only

Returns True if mod Infernal Robotics is installed, available to KOS and applicable to current craft. Example of use:

```
if ADDONS:IR:AVAILABLE
{
    //some IR dependent code
}
```

IRAddon:**GROUPS**

Type List of *IRControlGroup* objects

Access Get only

Lists all Servo Groups for current focused vessel. Example of use:

```
for g in ADDONS:IR:GROUPS
{
    Print g:NAME + " contains " + g:SERVOS:LENGTH + " servos".
}
```

IRAddon:**ALLSERVOS**

Type List of *IRServo* objects

Access Get only

Lists all Servos for current focused vessel. Example of use:

```
for s in ADDONS:IR:ALLSERVOS
{
    print "Name: " + s:NAME + ", position: " + s:POSITION.
}
```

structure IRControlGroup

Suffix	Type	Description
NAME	string	Name of the Control Group
SPEED	float	Speed multiplier set in the IR UI
EXPANDED	bool	True if Group is expanded in IR UI
FORWARDKEY	string	Key assigned to forward movement
REVERSEKEY	string	Key assigned to reverse movement
SERVOS	List (readonly)	List of servos in the group
MOVERIGHT()	void	Commands servos in the group to move in positive direction
MOVELEFT()	void	Commands servos in the group to move in negative direction
MOVECENTER()	void	Commands servos in the group to move to default position
MOVENEXTPRESET()	void	Commands servos in the group to move to next preset
MOVEPREVPRESET()	void	Commands servos in the group to move to previous preset
STOP()	void	Commands servos in the group to stop

IRControlGroup:**NAME**

Type string

Access Get/Set

Name of the Control Group (cannot be empty).

IRControlGroup:**SPEED**

Type float

Access Get/Set

Speed multiplier as set in the IR user interface. Avoid setting it to 0.

IRControlGroup:**EXPANDED**

Type bool

Access Get/Set

True if Group is expanded in IR UI

IRControlGroup:**FORWARDKEY**

Type string

Access Get/Set

Key assigned to forward movement. Can be empty.

IRControlGroup:**REVERSEKEY**

Type string

Access Get/Set

Key assigned to reverse movement. Can be empty.

IRControlGroup:**SERVOS**

Type List of *IRServo* objects

Access Get only

Lists Servos in the Group. Example of use:

```
for g in ADDONS:IR:GROUPS
{
    Print g:NAME + " contains " + g:SERVOS:LENGTH + " servos:".
    for s in g:servos
    {
        print "    " + s:NAME + ", position: " + s:POSITION.
    }
}
```

IRControlGroup:**MOVERIGHT**()

Returns void

Commands servos in the group to move in positive direction.

IRControlGroup:**MOVELEFT**()

Returns void

Commands servos in the group to move in negative direction.

IRControlGroup:**MOVECENTER**()

Returns void

Commands servos in the group to move to default position.

IRControlGroup:**MOVENEXTPRESET**()

Returns void

Commands servos in the group to move to next preset

IRControlGroup:**MOVEPREVPRESET**()

Returns void

Commands servos in the group to move to previous preset

IRControlGroup:**STOP**()

Returns void

Commands servos in the group to stop

structure `IRServo`

Suffix	Type	Description
NAME	string	Name of the Servo
UID	int	Unique ID of the servo part (part.flightID).
HIGHLIGHT	bool (set-only)	Set Hightlight status of the part.
POSITION	float (readonly)	Current position of the servo.
MINCFGPOSITION	float (readonly)	Minimum position for servo as defined by part creator in part.cfg
MAXCFGPOSITION	float (readonly)	Maximum position for servo as defined by part creator in part.cfg
MINPOSITION	float	Minimum position for servo, from tweakable.
MAXPOSITION	float	Maximum position for servo, from tweakable.
CONFIGSPEED	float (readonly)	Servo movement speed as defined by part creator in part.cfg
SPEED	float	Servo speed multiplier, from tweakable.
CURRENTSPEED	float (readonly)	Current Servo speed.
ACCELERATION	float	Servo acceleration multiplier, from tweakable.
ISMOVING	bool (readonly)	True if Servo is moving
ISFREEMOVING	bool (readonly)	True if Servo is uncontrollable (ex. docking washer)
LOCKED	bool	Servo's locked status, set true to lock servo.
INVERTED	bool	Servo's inverted status, set true to invert servo's axis.
PART	*Part*	A reference to a Part containing servo module.
MOVERIGHT()	void	Commands servo to move in positive direction
MOVELEFT()	void	Commands servo to move in negative direction
MOVECENTER()	void	Commands servo to move to default position
MOVENEXTPRESET()	void	Commands servo to move to next preset
MOVEPREVPRESET()	void	Commands servo to move to previous preset
STOP()	void	Commands servo to stop
MOVETO(position, speedMult)	void	Commands servo to move to *position* with *speedMult* multiplier

IRServo:**`NAME`**

Type string

Access Get/Set

Name of the Control Group (cannot be empty).

IRServo:**`UID`**

Type int

Access Get

Unique ID of the servo part (part.flightID).

IRServo:**`HIGHLIGHT`**

Type bool

Access Set

Set Hightlight status of the part.

IRServo:**POSITION**

Type float

Access Get

Current position of the servo.

IRServo:**MINCFGPOSITION**

Type float

Access Get

Minimum position for servo as defined by part creator in part.cfg

IRServo:**MAXCFGPOSITION**

Type float

Access Get

Maximum position for servo as defined by part creator in part.cfg

IRServo:**MINPOSITION**

Type float

Access Get/Set

Minimum position for servo, from tweakable.

IRServo:**MAXPOSITION**

Type float

Access Get/Set

Maximum position for servo, from tweakable.

IRServo:**CONFIGSPEED**

Type float

Access Get

Servo movement speed as defined by part creator in part.cfg

IRServo:**SPEED**

Type float

Access Get/Set

Servo speed multiplier, from tweakable.

IRServo:**CURRENTSPEED**

Type float

Access Get

Current Servo speed.

IRServo:**ACCELERATION**

Type float

Access Get/Set

Servo acceleration multiplier, from tweakable.

IRServo:**ISMOVING**

Type bool

Access Get

True if Servo is moving

IRServo:**ISFREEMOVING**

Type bool

Access Get

True if Servo is uncontrollable (ex. docking washer)

IRServo:**LOCKED**

Type bool

Access Get/Set

Servo's locked status, set true to lock servo.

IRServo:**INVERTED**

Type bool

Access Get/Set

Servo's inverted status, set true to invert servo's axis.

IRServo:**PART**

Type *Part*

Access Get

Returns reference to the *Part* containing servo module. Please note that Part:UID does not equal IRServo:UID.

IRServo:**MOVERIGHT**()

Returns void

Commands servo to move in positive direction

IRServo:**MOVELEFT**()

Returns void

Commands servo to move in negative direction

IRServo:**MOVECENTER**()

Returns void

Commands servo to move to default position

IRServo:**MOVENEXTPRESET**()

Returns void

Commands servo to move to next preset

IRServo:**MOVEPREVPRESET**()

Returns void

Commands servo to move to previous preset

IRServo:**STOP**()

Returns void

Commands servo to stop

IRServo:**MOVETO**(position, speedMult)

Parameters

- **position** – (float) Position to move to
- **speedMult** – (float) Speed multiplier

Returns void

Commands servo to move to *position* with *speedMult* multiplier.

Example code:

```
print "IR Iavailable: " + ADDONS:IR:AVAILABLE.

Print "Groups:".

for g in ADDONS:IR:GROUPS
{
    Print g:NAME + " contains " + g:SERVOS:LENGTH + " servos:".
    for s in g:servos
    {
        print "    " + s:NAME + ", position: " + s:POSITION.
        if (g:NAME = "Hinges" and s:POSITION = 0)
        {
            s:MOVETO(30, 2).
        }
        else if (g:NAME = "Hinges" and s:POSITION > 0)
        {
            s:MOVETO(0, 1).
        }
    }
}
```

To help KOS scripts identify whether or not certain mod is installed and available following suffixed functions were introduced in version 0.17

9.5 ADDONS:AGX:AVAILABLE

Returns True if mod Action Group Extended is installed and available to KOS.

9.6 ADDONS:RT:AVAILABLE

Returns True if mod RemoteTech is installed and available to KOS. See more RemoteTech functions here.

9.7 ADDONS:KAC:AVAILABLE

Returns True if mod Kerbal Alarm Clock is installed and available to KOS.

9.8 ADDONS:IR:AVAILABLE

Returns True if mod Infernal Robotics is installed, available to KOS and applicable to current craft. See more here.

CHAPTER

TEN

CONTRIBUTE

10.1 How to Contribute to this Project

Do you know or are willing to learn C# and the **KSP** public API? Great, we could use your help! The source code for **kOS** is kept on github and is currently maintained by Chris Woerz and Steven Mading. The first steps you will want to take is to clone the latest version of **kOS** from github and try to build it using your favorite C# compiler suite.

10.2 How to Edit this Documentation

This documentation was written using reStructuredText and compiled into HTML using Sphinx and the Read The Docs Theme. It is maintained by Steven Mading, Chris Woerz and Johann Goetz

CHAPTER

ELEVEN

CHANGES FROM VERSION TO VERSION

This is a slightly more verbose version of the new features mentioned in the CHANGELOG, specifically for new features and for users familiar with older versions of the documentation who want only a quick update to the docs without reading the entire set of documentation again from scratch.

11.1 Changes in 0.17.3

11.1.1 New Looping control flow, the FROM loop

There is now a new kind of loop, *the FROM loop*, which is a bit like the typical 3-part for-loop seen in a lot of other languages with a separate init, check, and increment section.

11.1.2 Short-Circuit Booleans

Previously, kerboscript's AND and OR operators were not short-circuiting. *Now they are*.

11.1.3 New Infernal Robotics interface

There are a few new helper addon utilities for the Infernal Robotics mod, on the *IR addon page*.

11.1.4 New RemoteTech interface

There are a few new helper addon utilities for the RemoteTech mod, on the *RemoteTech addon page*.

11.1.5 Deprecated INCOMMRANGE

Reading from the INCOMMRANGE bound variable will now throw a deprecation exception with instructions to use the new `RTAddon` structure for the RT mod.

11.1.6 Updated thrust calculations for 1.0.x

KSP 1.0 caused the thrust calculations to become a LOT more complex than they used to be and kOS hadn't caught up yet. For a lot of scripts, trying to figure out a good throttle setting is no longer a matter of just taking a fraction of the engine's MAXTHRUST.

We fixed the existing suffixes of MAXTHRUST and AVAILABLETHRUST for `engine` and `vessel` to account for the new changes in thrust based on ISP at different altitudes. MAXTHRUST is now the max the engine can put out at the CURRENT atmospheric pressure and current velocity. It might not be the maximum it could put out under other conditions. The AVAILABLETHRUST suffix is now implemented for engines (it was previously only available on vessels). There are also new suffixes MAXTHRUSTAT (engines and vessels), AVAILABLETHRUSTAT (engines and vessels), and ISPAT (engines only) to read the applicable value at a given atmospheric pressure.

11.1.7 New CORE struct

The *core* bound variable gives you a structure you can use to access properties of the current in-game CPU the script is running on, including the vessel part it's inside of, and the vessel it's inside of, as well as the currently selected volume. Moving forward this will be the struct where we enable features that interact with the processor itself, like local configuration or current operational status.

11.1.8 Updated boot file name handling

Boot files are now copied to the local hard disk using their original file name. This allows for uniform file name access either on the archive or local drive and fixes boot files not working when kOS is configured to start on the Archive. You can also get or set the boot file using the BOOTFILENAME suffix of the *CORE* bound variable.

11.1.9 Docking port, element, and vessel references

You can now get a list of docking ports on any element or vessel using the DOCKINGPORTS suffix. Vessels also expose a list of their elements (the ELEMENTS suffix) and an element will refernce it's parent vessel (the VESSEL suffix).

11.1.10 New sounds and terminal features

For purely cosmetic purpopses, there are new sound features and a few terminal tweaks.

- A terminal keyclick option for the in-game GUI terminal.
- The ability to BEEP when printing ascii code 7 (BEL), although the only way currently to achieve this is with the KSlib's spec_char.ksm file, as kOS has no BEL char, but this will be addressed later.
- A sound effect on exceptions, which can be turned off on the CONFIG panel.

11.1.11 Clear vecdraws all at once

For convenience, you can clear all vecdraws off the screen at once now with the *clearvecdraws()* function.

11.2 Changes in 0.17.0

11.2.1 Variables can now be local

Previously, the kOS runtime had a serious limitation in which it could only support one flat namespace of global-only variables. Considerable archetecture re-work has been done to now support *block-scoping* in the underlying runtime, which can be controlled through the use of *local declarations* in your kerboscript files.

11.2.2 Kerboscript has User Functions

The primary reason for the local scope variables rework was in support of the new *user functions feature* which has been a long-wished-for feature for kOS to support.

11.2.3 Community Examples Library

There is now a *new fledgling repository of examples and library scripts* that we hope to be something the user community contributes to. Some of the examples shown in the kOS 0.17.0 release video are located there. The addition of the ability to make user functions now makes the creation of such a library a viable option.

11.2.4 Physics Ticks not Update Ticks

The updates have been *moved to the physics update* portion of Unity, instead of the animation frame rate updates. This may affect your preferred CONFIG:IPU setting. The new move creates a much more uniform performance across all users, without penalizing the users of faster computers anymore. (Previously, if your computer was faster, you'd be charged more electricity as the updates came more often).

11.2.5 Ability to use SAS modes from KSP 0.90

Added a new *third way to control the ship*, by leaving SAS on, and just telling KSP which mode (prograde, retrograde, normal, etc) to put the SAS into.

11.2.6 Blizzy ToolBar Support

If you have the Blizzy Toolbar mod installed, you should be able to put the kOS control panel window under its control.

11.2.7 Ability to define colors using HSV

When a color is called for, such as with VECDRAW or HIGHLIGHT, you can now use the *HSV color system (hue, saturation, value)* instead of RGB, if you prefer.

11.2.8 Ability to highlight a part in color

Any time your script needs to communicate something to the user about which part or parts it's dealing with, it can use KSP's *part highlighting feature* to show a part.

11.2.9 Better user interface for selecting boot scripts

The selection of *boot scripts for your vessel* has been improved.

11.2.10 Disks can be made bigger with tweakable slider

All parts that have disk space now have a slider you can use in the VAB or SPH editors to tweak the disk space to choose whether you want it to have 1x, 2x, or 4x as much as its default size. Increasing the size increases its price and its weight cost.

11.2.11 You Can Transfer Resources

You can now use kOS scripts to *transfer resources between parts* for things like fuel, in the same way that a manual user can do by using the right-click menus.

11.2.12 Kerbal Alarm Clock support

If you have the Kerbal Alarm Clock Mod isntalled, you can now *query and manipulate its alarms* from within your kOS scripts.

11.2.13 Query the docked elements of a vessel

You can get the *docked components of a joined-together vessel* as separate collections of parts now.

11.2.14 Support for Action Groups Extended

While there was some support for the Action Groups Extended mod before, it has *been greatly improved*.

11.2.15 LIST constructor can now initialize lists

You can now do this:

```
set mylist to list(2,6,1,6,21).
```

to initialize a *list of values* from the start, so you no longer have to have a long list of list:ADD commands to populate it.

11.2.16 ISDEAD suffix for Vessel

Vessels now have an :ISDEAD suffix you can use to detect if the vessel has gone away since the last time you got the handle to it. (for example, you LIST TARGETS IN FOO, then the ship foo[3] blows up, then foo[3]:ISDEAD should become true to clue you in to this fact.)

CHAPTER

TWELVE

ABOUT KOS AND KERBOSCRIPT

CHAPTER

THIRTEEN

INTRODUCTION TO KOS AND KERBOSCRIPT

KerboScript is the language used to program the CPU device attached to your vessel and **kOS** is the operating system that interprets the code you write. The program can be as simple as printing the current altitude of the vessel and as complicated as a six-axis autopilot controller taking your vessel from the launchpad to Duna and back! With **kOS**, the sky is *not* the limit.

This mod *is* compatible with RemoteTech, you just have to make sure you copy the program onto the local CPU before it goes out of range of KSC.

13.1 Installation

Like other mods, simply merge the contents of the zip file into your Kerbal Space Program folder.

13.2 KerboScript

KerboScript is a programming language that is derived from the language of planet Kerbin, which *sounds* like gibberish to non-native speakers but for some reason is *written* exactly like English. As a result, **KerboScript** is very English-like in its syntax. For example, it uses periods as statement terminators.

The language is designed to be easily accessible to novice programmers, therefore it is case-insensitive, and types are cast automatically whenever possible.

A typical command in **KerboScript** might look like this:

```
PRINT "Hello World".
```

CHAPTER

FOURTEEN

INDICES AND TABLES

- genindex
- modindex
- search