KM 03-62
KERBAL SYSTEMS FIELD MANUAL

KERBAL OPERATING SYSTEM

HEADQUARTERS, KERBAL SPACE CENTER

Contents

Part I

Introduction

k-OS 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.

Installation

Like other mods, simply merge the contents of the zip file into your Kerbal Space Program folder.

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".
```

Part II

Tutorials

0.1 Quick Start

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.

0.1.1 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 Rightclick 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.

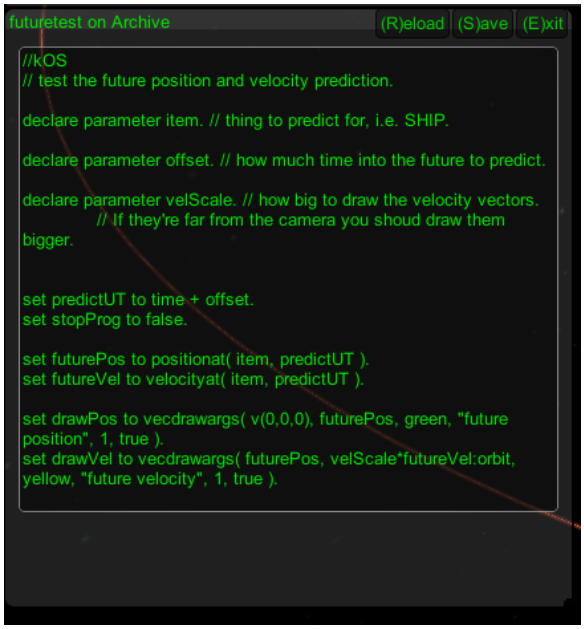
The terminal will respond by showing you this:

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.
```

(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:

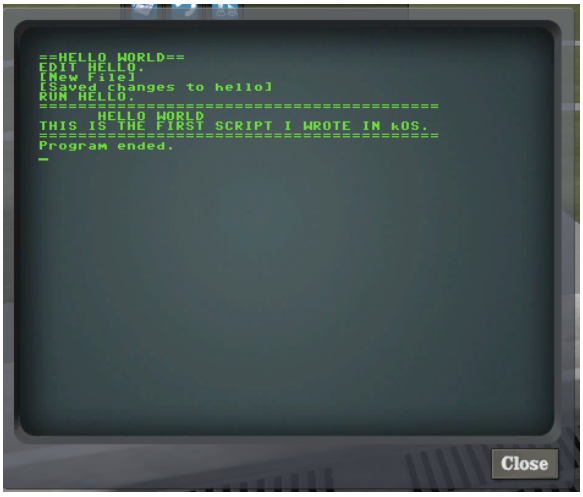
```
PRINT "=========================================".
PRINT "      HELLO WORLD".
PRINT "THIS IS THE FIRST SCRIPT I WROTE IN kOS.".
PRINT "=========================================".
```

Click ?Save? then ?Exit? in the editor popup 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.

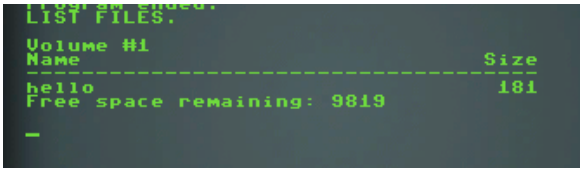


Step 7: Okay, but where is this program? To see where the ?HELLO? program has been saved, Issue the command LIST FILES like this:

```
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

0.1.2 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 Make any sort of rocket that can lift you to orbit that fills the following pattern:

It uses ONLY liquid fuel rockets. The example code here will assume this is the case. kOS can deal with solid fuel boosters as well, but to keep the example simple we?ll use liquid fuel only here.

Make the vessel?s staging list set up in the right order for a launch. (Make sure it has no need to manually rightclick parts to stage things weirdly, and no need to use action groups to activate stages weirdly). Make sure the vessel has plenty of torque power to stay steady without a lot of wobble. Make the vessel have at least these parts on it: battery power of at least 400 charge ability to recharge equal to at least 6 solar panel sections or 1 RTG unit the kOS SCS part somewhere in the stack, near the top bit where it won?t fall off due to staging. 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):

```
// My First Launcher.

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:

```
// My First Launcher.

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.
}

PRINT "Main throttle up.  2 seconds to stabalize it.".
LOCK THROTTLE TO 1.0.   // 1.0 is the max, 0.0 is idle.
WAIT 2. // give throttle time to adjust.
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. // pause here until ship is high up.

// 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 pogram 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:

```
// My First Launcher.

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.
}
PRINT "Main throttle up.  2 seconds to stabalize it.".
LOCK THROTTLE TO 1.0.   // 1.0 is the max, 0.0 is idle.


LOCK STEERING TO UP.  // This is the new line to add


WAIT 2. // give throttle time to adjust.
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. // pause here until ship is high up.

// 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 pogram 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. NOTE: Due to a bug sometimes reverting just to the launchpad does not work well and you need to revert all the way back to the VAB.:

```
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 STAGE:LIQUIDFUEL < 0.001 THEN {
    PRINT "No liquidfuel.  Attempting to stage.".
    STAGE.
    PRESERVE.
}
```

It says, ?Whenever the amount of liquid fuel in the current stage is so small it may as well be zero (< 0.001), 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.? The check for < 0.001 is because sometimes KSP won?t quite burn the last drop of fuel in a stage. 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 liquidfuel in it. 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:

```
// My First Launcher.

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.
}
PRINT "Main throttle up.  2 seconds to stabalize it.".
LOCK THROTTLE TO 1.0.   // 1.0 is the max, 0.0 is idle.
LOCK STEERING TO UP.
WAIT 2. // give throttle time to adjust.

// The section below replaces previous UNTIL loop:

WHEN STAGE:LIQUIDFUEL < 0.001 THEN {
    PRINT "No liquidfuel.  Attempting to stage.".
    STAGE.
    PRESERVE.
}
WAIT UNTIL SHIP:ALTITUDE > 70000. // pause here until ship is high up.

// 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 pogram 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. (again, assuming you made the entire vessel with only liquidfuel engines.)

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 excercise 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 to 10000m. Aim at 45 degrees down toward the east until 40000m. Thrust horizontally east after that. 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. Combining this with the WHEN command from before, we get this section:

```
WHEN SHIP:ALTITUDE > 10000 THEN {
    PRINT "Starting turn.  Aiming to 45 degree pitch.".
    LOCK STEERING TO HEADING(90,45). // east, 45 degrees pitch.
}
WHEN SHIP:ALTITUDE > 40000 THEN {
    PRINT "Starting flat part.  Aiming to horizon.".
    LOCK STEERING TO HEADING(90,0). // east, horizontal.
}
```

Note that these lack the command PRESERVE like the previous WHEN example had. This is because we want these to trigger just once and then never again. There?s no point in constantly telling kOS to reset the steering to the same thing over and over as the script runs.

Now, if you insert this new section to the script, we have a nice nifty example of a start of a launching script. Note that it works even if you insert it at the top of the script, because it sets up the triggers to occur LATER when the condition becomes true. They don?t execute right away:

```
// My First Launcher.

WHEN SHIP:ALTITUDE > 10000 THEN {
```

```
    PRINT "Starting turn.  Aiming to 45 degree pitch.".
    LOCK STEERING TO HEADING(90,45). // east, 45 degrees pitch.
}
WHEN SHIP:ALTITUDE > 40000 THEN {
    PRINT "Starting flat part.  Aiming to horizon.".
    LOCK STEERING TO HEADING(90,0). // east, horizontal.
}
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.
}
PRINT "Main throttle up.  2 seconds to stabalize it.".
LOCK THROTTLE TO 1.0.   // 1.0 is the max, 0.0 is idle.
LOCK STEERING TO UP.
WAIT 2. // give throttle time to adjust.

// The section below replaces previous UNTIL loop:

WHEN STAGE:LIQUIDFUEL < 0.001 THEN {
    PRINT "No liquidfuel.  Attempting to stage.".
    STAGE.
    PRESERVE.
}
WAIT UNTIL SHIP:ALTITUDE > 70000. // pause here until ship is high up.

// 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.
```

And here is it in action:

And toward the end:

If we assume you made a vessel that has enough fuel and power to get up to orbit, 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. You could complete the launching script by making sure once the vessel breaks the atmosphere it actually makes a circular orbit rather than just stopping after 70000m and coasting. 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 too much wasted energy fighting air friction. (The way KSP?s stock areodynamic model works, the optimal speed is terminal velocity, by the way). This is partly addressed in the PID Loop Tutorial. With more sophisticated staging checks, the script could be made to work with solid fuel engines as well. 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 termainl window as the script runs.

0.2 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!

0.2.1 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:

Sequential Loops with Condition Checking 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.

0.2.2 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 throtl TO 1.
LOCK THROTTLE TO throtl.
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 throtts to throtts + 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 throtts TO 1.
SET dthrott TO 0.
LOCK THROTTLE TO throtts.
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 throtts to throtts + 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 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).
}
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 throtT to throtT + 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. Certaintly, 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 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).

    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 throtT to throtT + 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.

0.3 PID Loops in k-OS

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.

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.

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 throtT to throtT + 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.

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.

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

bounds on the Integral term which addresses possible integral windup bounds on the throttle since it must stay in the range 0 to 1 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.
}
```

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.

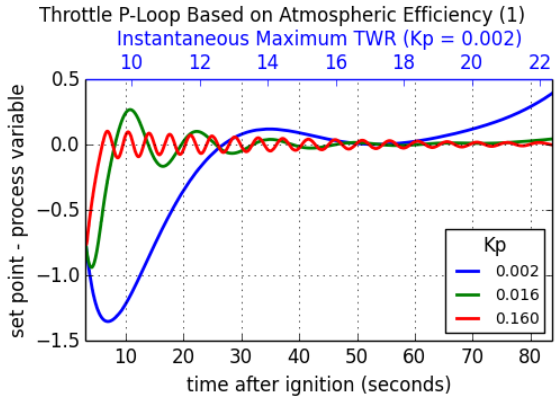
```
DECLARE PARAMETER Kp.

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 throtte TO 1.
LOCK THROTTLE TO throtte.
SAS ON.
STAGE.
WAIT 3.

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

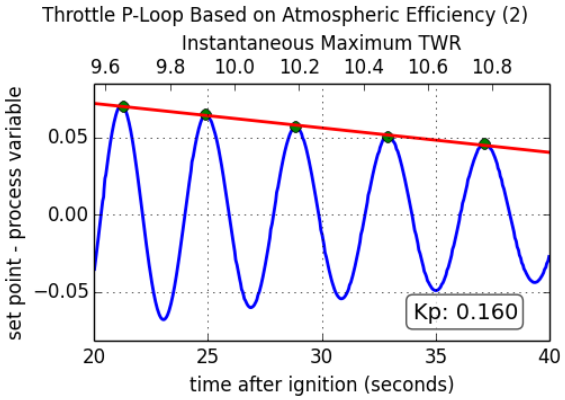
Give this script a short name, something like ?tune.txt? so that running is simple:

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

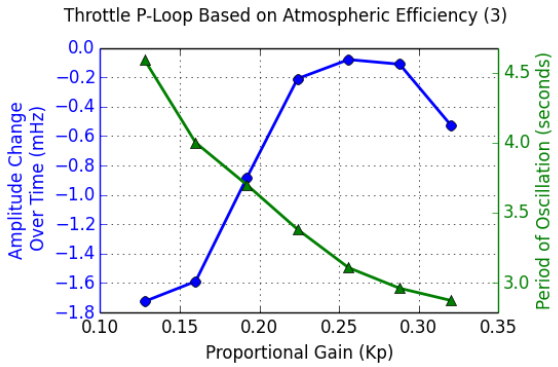
After every launch completes, you?ll have to go into the archive directory and rename the output log file. Something like ?$throttle_log.txt$?? >?$throttle$.01.log?$willhelpifyouincrementtheindexnumbereachtime.Toan$ $0.002, 0.016and0.160, includingthemaximumTWRwhenKp = 0.002asthetopx-axis.ThemaximumTWRdependenc$



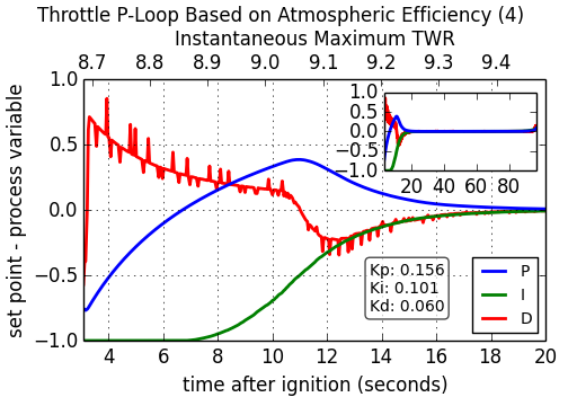
The value of 0.002 is obviously too low. The settling time is well over 20 seconds and the loop can?t keep up with the increase in terminal velocity at the higher altitudes reached after one minute. When Kp = 0.016, the behavior is far more well behaved, and though some oscillation exists, it?s damped 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:

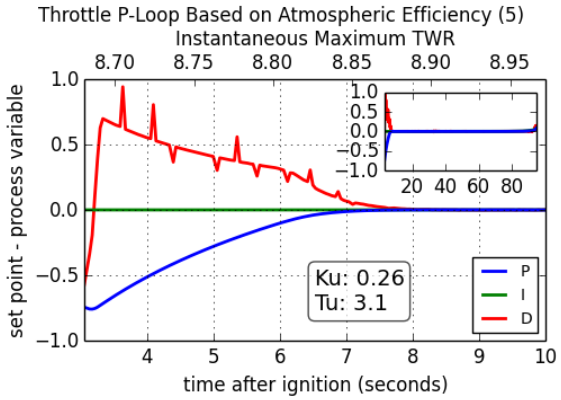


The period of oscillation was averaged over the interval and plotted on top of the amplitude change over time. Notice the turn over that occurs when Kp reaches approximately 0.26. This will mark the ?ultimate gain? and 3.1 seconds will be used as the associated period of oscillation. It is left as an 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.

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.

0.4 Execute Node Script

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 ad
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
    //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:delt
        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:delt
        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.

Code

Part III

Language

0.5 General Features

0.5.1 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.

0.5.2 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 e
```

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.

0.5.3 Short-circuiting 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.

0.5.4 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.

0.5.5 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.

0.5.6 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.

0.5.7 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.

0.6 Language Syntax

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.

0.6.1 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?.

0.6.2 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.

0.6.3 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().
```

0.6.4 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().
```

0.6.5 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:

Create a chunk of program instructions that you don?t intend to execute YET. 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.

0.6.6 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.

0.6.7 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.

0.7 Flow Control

0.8 Variables

0.9 User Functions

Part IV

Mathematics

0.10 Fundamental Constants

0.11 Mathematical Functions

0.12 Vectors

0.13 Directions

0.14 Geographic Coordinates

0.15 Reference Frames

Part V

Commands

0.16 Flight Control

0.16.1 Cooked Control

0.16.2 Raw Control

0.16.3 Pilot Input

0.16.4 Ship Systems

0.16.5 Time Warping

0.17 Prediction

0.18 Listing Data

0.19 Parts Information

0.20 File I/O

0.21 Termianl and GUI

0.22 Resource Transfer

Part VI

Structures

0.23 Orbits

0.23.1 Orbit

0.23.2 Orbitable (Vessels and Bodies)

0.23.3 Orbitable Velocity

0.24 Celestial Bodies

0.24.1 Atmosphere

0.24.2 Body

0.25 Vessels and Parts

0.25.1 Agregate Resource

0.25.2 Docking Port

0.25.3 Element

0.25.4 Engine

0.25.5 Gimbal

0.25.6 Maneuver Node

0.25.7 Part

0.25.8 Part Module

0.25.9 Resource

0.25.10 Sensor

0.25.11 Stage

0.25.12 Vessel

0.25.13 Vessel Sensors

0.26 Waypoints

0.27 Miscellaneous

0.27.1 Colors

0.27.2 Configuration of kOS

0.27.3 File Information

0.27.4 Part Highlighting

0.27.5 Iterator

0.27.6 List

0.27.7 Resource Transfer

0.27.8 Terminal

0.27.9 Time Span

Part VII

Addons

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. To help KOS scripts identify whether or not certain mod is installed and available following suffixed functions were introduced in version 0.17

```
ADDONS:AGX:AVAILABLE
```

Returns True if mod Action Group Extended is installed and available to KOS.

```
ADDONS:RT:AVAILABLE
```

Returns True if mod RemoteTech is installed and available to KOS. See more RemoteTech functions here.

```
ADDONS:KAC:AVAILABLE
```

Returns True if mod Kerbal Alarm Clock is installed and available to KOS.

```
ADDONS:IR:AVAILABLE
```

Returns True if mod Infernal Robotics is installed, available to KOS and applicable to current craft. See more here.

0.29 Addon Groups Extended

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.

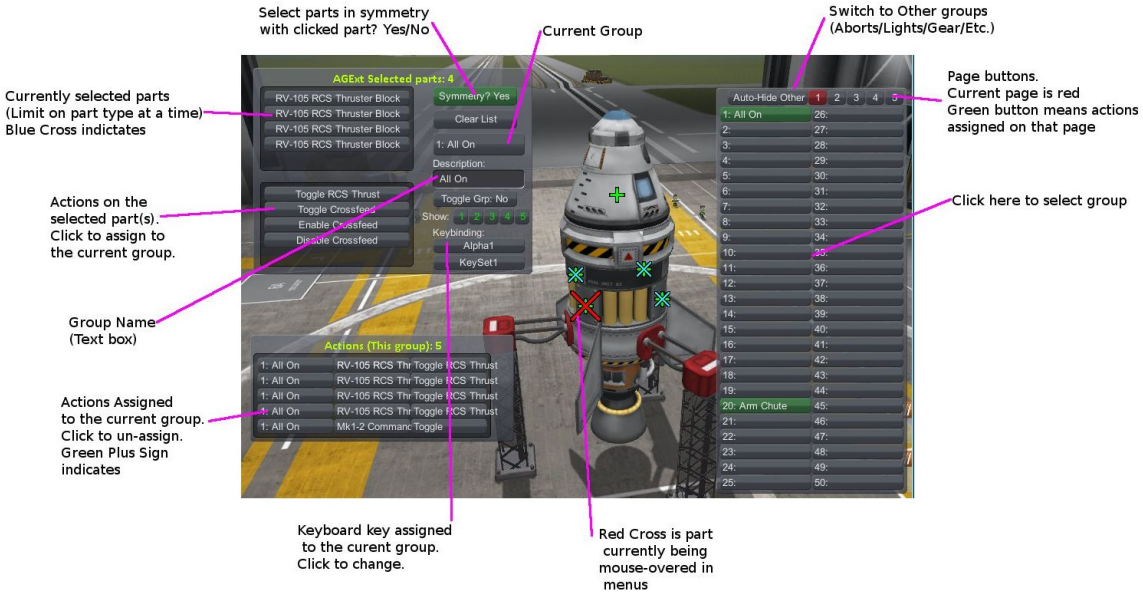
Anywhere you use AG1, you can use AG15 in the same way. (AG11 through AG250 explicitly behave the same as the 10 stock groups. Please file a bug report if they do not.)

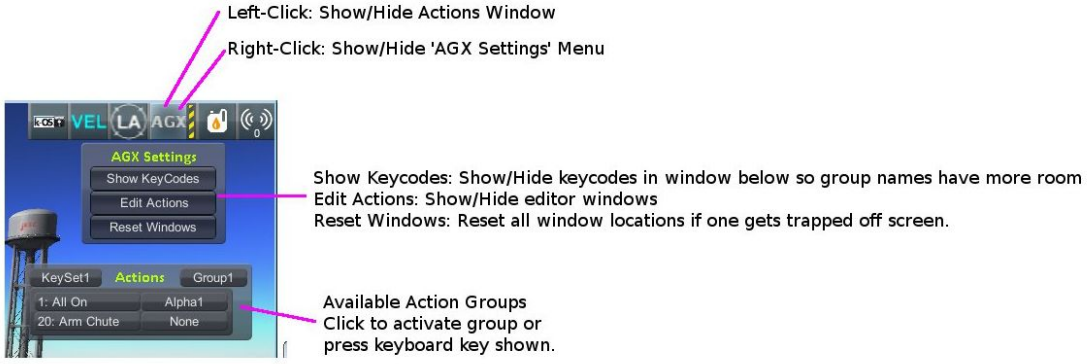
Script Trigger action: Installing AGX adds the ?Script Trigger? action to all kOS computer parts. This action is a null action that does not activate anything but serves as a placeholder to enhance action groups in kOS.

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.)

0.29.1 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.

0.29.2 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.

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.

0.29.3 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.

0.29.4 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 on
  print "Action group 15 clicked!".
  preserve.
}
```

0.29.5 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.

0.30 RemoteTech

RemoteTech is a modification for Squad?s ?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.

0.30.1 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).

0.31 Kerbal Alarm Clock

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.

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".
}
```

0.32 Infernal Robotics

Infernal Robotics introduces robotics parts to the game, letting you create moving or spinning contraptions that just aren?t possible under stock KSP.

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(readonly) 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. 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: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).
        }
    }
}
```

Part VIII

Changes