System&Panel SDK V1.5

1.Introduction

System & Panels SDK (hence to be refereed as SPSDK) has been developed over a long period of time (a bit longer than I can remember), as an attempt to provide a (semi) automatic way to build the internal systems of a spaceship as well as the panels for Orbiter add-ons. It is based on a somewhat-limited parser that reads configuration data from two ASCII-text files (a .prd and a .cfg), and also on a simple API interface that add-on creators can use to interact with the SPSDK from C++ code.

One of the goals of this SDK has been to achieve a level whereas C++ code is no longer needed, and as such, complex spaceships to be implemented in Orbiter without the need for a C++ compiler. In it's current state SPSDK is very near that goal, and although some C++ code is still required I can realistically hope, with feedback from users and some extra-time spent on this issue, that future version will contain enough Orbiter features to eliminate the C++ code entirely from building new ships.

NOTE : As of version 1.5, is it possible to build multi-stage ships entirely from scripts using the provided spsdk.dll file. Complex panels and ships requiring mesh animations, will still need a C++ code for the time being.

SPSDK as a hole, is actually composed of 5 large "engines" that control the behavior of the spaceship as follows:

1.Thermal engine

Based on some properties defined by the user in the .cfg file, this engine will perform all the thermal computations needed on the spaceship.
All the "physical" systems on the spaceship, as you will see, have a position vector parameter built in, defined in the .cfg file, whereas the user can specify the actual location (in meters) of that item, in ship coordinates. Using this location, the direction towards the Sun and towards Earth, the thermal engine will compute a thermal state for each of these systems as follows:

a) The item will get warmer (or colder) based on the temperature of items nearby and distance to them (CONDUCTION):

$$\Phi_Q = \frac{k * \Delta T}{d}$$

Stating, that the heat flux per unit area is determined proportional with the thermal conductivity of the items (k), the difference in temperature, and reverse proportional with the distance between the items.

b) The item will get colder by radiative cooling:

$$\Phi_Q = -\sigma(T - 3.0)^4$$

Stating that the item will loose thermal energy per unit area at a rate proportional with its temperature (in Kelvin) to the fourth power. The –3 in the formula is to account for the space-background heating.

c) The item will get warmer by radiative heating from the Sun:

$$\Phi_Q = 1372 * \cos(\angle ToSun)$$

A 1372W radiation is assumed in near-Earth orbit from the Sun, and this radiation is proportional with the cosines of the angle of incidence of the Sun's rays.

Obviously, if the item is not pointing towards the Sun, there will be no heating from this. Nor will there be a heating if the Earth is blocking the Sun.

d) The item will get warmer by radiative heating from Earth
Identical with point c), only this time the Earth is considered as the source. From a thermodynamically pov, the Earth can be considered as a radiating blackbody at –18ºC, so about 190W are added per unit area.

e) Lastly, Earth's albedo is taken into account.

$$\Phi_Q = 300 * \cos(\angle ToEarth) * \%EarthVisible$$

If the ship is orbiting over the sunny part of Earth, about 300W of energy is added to the unit area.

Because of all these thermal computations, a Earth-orbiting spacecraft might in fact never reach a true thermodynamically equilibrium, which a fancy way of saying that temperatures will constantly change, as the ship rotates around it's axis, but more so as the ship rotates around the Earth, into and out of sunlight. With temperature differences in the order of 100's of degrees, additional equipment and extra care must be taken when designing a spaceship, in order to maintain all equipment, and crew, within decided thermal limits.

2.Hydraulic

The hydraulic engine is responsible with managing fluids and gases used throughout the ship. Based on thermal data provided by the thermal engine, the hydraulic engine will maintain it's fluids and gases in accordance with thermodynamically and chemical laws:

All fluids and gasses that are in contact (sharing a volume) will constantly drive towards a thermodynamically equilibrium, whereas pressure, temperature and chemical potential equalize throughout the volume. Liquids will boil if pressure drops below the vapor pressure at that temperature or gases will condense if the pressure rises above that. Temperatures will drop if liquids boil, or will rise if gasses condense, etc… Though all in all a lot of shortcuts and simplifications are used, mainly to keep computation within limits, the resulting behavior of the gases and fluids are reasonably in accordance with Real World Behavior [tm.]

3. Electric

Probably the most common-type of all the engines, the electrical engine maintains all of the electrical equipment on the spaceship. Volts, Amperes, power are all maintained by this engine. Electrical equipment can be interconnected by this engine, whereas power consumers will draw power from producers, etc. While there is a way to chain-link multiple electrical equipment to a single electrical circuit (or bus), the engine will always link them in parallel, there is no way to link multiple electrical items in a serial way on the bus using this SDK. This has been decided for a number of reasons, including the improbability of serial-linking multiple consumers, to the complexities added by having parallel and serial circuits throughout the ship.

4. Vessel Management

This engine is responsible with all the vessel -related data. Things like loading meshes, defining thrusters and ejecting discarded stages are handled here.

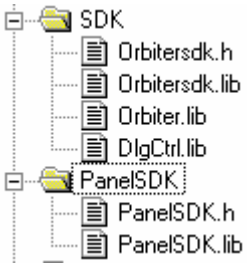
5. PanelSDK

The last engine will be responsible with controlling all the panel events, resources and management needed to properly display a panel in Orbiter. This engine, is also the one responsible with the API interaction from the user. A list of instrument types, the bitmaps used (i.e. graphical resources), the behavior of these instruments, etc., will be specified to this engine, via a .prd (Panel Resources Descriptor) file. From these defined instruments (or gauges) actual panels will be build in the ship's .cfg file. Both of these files should be located in the Orbiter's "Config" subdirectory.

Chapter1 – interacting with SPSDK

Firstly, for a ship to make use of this SDK, the module of the ship will need to include, in addition with the OrbiterSDK.lib and OrbiterSDK.h two more files: PanelSDK.lib and PanelSDK.h

Also, because of the OpenGL libraries used by the ADI (which is standard included in the SPSDK) you will have to additionally link the opengl32.lib glu32.lib files (Project->Settings->Link then type these two additional libs)



PanelSDK.h will contain all of the API calls needed for the module to interact with the SPSDK.

All of the SPSDK <->ship module interaction will be done via the Panel engine. This engine, in the form of a class named PanelSDK. It is this class that your vessel will have to contain an instance of. Please check the sample project included with the SPSDK to get an idea how to operate the SPSDK API.

The following is a list of calls to PanelSDK currently supported:

void Panel.RegisterVessel(this);

Called upon ship creation (in the vessel constructor), this sends a pointer towards your vessel to the SPSDK.

void Panel.RegisterCustomPointer("NAME",(void*)&Variable);

Also called inside the ship's constructor, this call is used to define a number of "custom" variables that the user wants to use throughout the config files. Any reference throughout the config files to "SHIP:NAME" will point towards "Variable".

void Panel.InitFromFile("configfile");

This call sends the name of the .cfg file (the extension is automatically added) that contains the configuration we want for our vessel.

void *Panel.GetPointerByString("SHIP:NAME");

This call will return a void* (you need to downcast is) pointer to the item you specified. In this sample, a void pointer to the "Variable" will be return, as previously defined

void Panel.Timestep(time);

Called on each timestep. This updates various items throughout the SPSDK.

void Panel.Load(scn);
void Panel.Save(scn);

Used to load and save both panel and internal data.

void Panel.SetStage(Stage);

SPSDK has the capability to define different configurations for different stages of the spaceship, to account for the jettisoning of some equipment during staging. The actual

differences in the staging are defined in the <SEPARATION> section of the .cfg file. A call to SetStage() will then ask the SPSDK to “reconfigure” our internals based on our new stage number. Up to 99 stages can be defined. While it is not necessary to call stages in a directly increasing order (i.e. SetStage(0) then call SetStage(3) is OK), it is not possible to “revert’ to a lower stage number (i.e. SetStage(3) then call SetStage(2) is NOT OK). This call is to be made AFTER all the thrusters & fuel resources have been deleted, but BEFORE the thrusters and resources for the new stage have been defined.

void Panel.LoadPanel(id);
void Panel.PanelEvent(id,event,surf);
void Panel.MouseEvent(id,event,mx,my);
void Panel.KeybEvent(key,keystate);
void Panel.MFDEvent(mfd);

All of the panel,keyboard and mouse events should be “captured” by the SPSDK. Again, please check the included sample project for a detailed look on how these functions should be called. Note that users can freely “add” their own panel data on top of the panel items defined in the configs.

Another special item to note hare are QueryStrings used in the .cfg file:

Throughout the configuration file SPSDK uses heavily on QueryStrings as text-like pointers of different values. These QueryStrings act very much like normal variables used in normal C++ language, or pointers.

The basic format for a QueryString is composed of the engine we whish to query, the name of the item (or system) from the respective engine, the sub-section (or component), the sub-sub-section, etc. recursively.
E.g.
HYDRAULIC:H2OTANK:TEMP
[enginename]:[itemname]:[component]

Specifies the temperature of an object named “H2OTANK” which is a hydraulic system.

Or,
HYDRAULIC:H2OTANK:IN:ON
[enginename]:[itemname]:[component]:[sub-component]

etc.

Valid engine names are HYDRAULIC for all hydraulic elements and ELECTRIC for electrical systems. THERMAL, though a SPSDK engine, does not have any items to actually select by QueryString.

To select by QueryString an instrument located on a panel, instead of the engine name (i.e. HYDRAULIC or ELECTRIC) you specify the actual name of the panel, as defined per .cfg file.

To select customary variables defined via RegisterCustomPointer(), you must specify “SHIP:NAME” whereas NAME is to be replaced with the string you provided to the RegisterCustomPointer().

Please note that, unlike modern day compilers, the SPSDK parser will not account for the various data types used by QueryString, nor will it re-cast them. It is very much possible to specify a pointer to a double value (such as temperature of a tank) as the index of a switch, which requires an int value. The SPSDK will not complain in this case, but the results are quite undefined. Throughout this manual, I will specify which queries contain what types of values, and

what types you need to provide to configurations, but it is the user's job to keep these pointers within required types.

Chapter2 – the PRD file

The .prd (Panel Resource Descriptor) file should be located in the "Config" subfolder of Orbiter. The name of the file (w/out the extension) is to be defined on the *PanelType=filename* line in the .cfg file of the vessel.
The .prd file describes the various instruments that the panel of your ship will use. Once you define your instrument types in the .prd file, you can then used these instruments in the .cfg file to build up and configure your panel.

All the instruments are defined in a list in the form of

<INSTRUMENTNAME>
CLASS THE CLASS OF THE INSTRUMENT
…
… (VARIOUS PARAMETERS)
…
</INSTRUMENTNAME>

<OTHERINSTRUMENT>
CLASS THE CLASS OF THE INSTRUMENT
… (VARIOUS PARAMETERS)
</ OTHERINSTRUMENT>
..etc..

Each type of instrument is to begin with a <NAME> line and end with a </NAME> whereas the NAME is the string to be used in the .cfg file when referring to this particular instrument. All instruments must belong to a specific CLASS, which defines their behavior.

There are a total of 11 possible classes, or types, of instruments

1. CLASS SWITCH

Switches are just that, a basic type of instrument providing a two-way or a three-way selection from the user.

<u>Parameters in the PRD file:</u>

BITMAP *path_to_bitmap*

Name and path to the bitmap to be used as switch. Note that the layout of this bitmap must be a 3 x 2 image of the switch in the various positions:
The upper-row of 3 bitmaps must contain the bitmaps for a lights-off of the switch, whereas the lower-row of 3 bitmaps contain a lights-on display of the switch. The corresponding left-most column represents the switch in it's +1 position, the middle bitmap in the 0 position and the right bitmap in the –1 position. The RGB 255,255,255 color is assumed transparent.

SIZE *x y*
The size in pixels of the switch. Note that this is the size of 1 switch, not of the entire bitmap. The actual bitmap must be 3*x pixels wide and 2*y pixels high.

Then in the .CFG file, when you want to add a SWITCH-CLASS instrument,

Defining a SWITCH in .cfg file

<SWITCHINSTRUMENT> x y nr_pos spring SRC(int) [name]

x y – these are the xy coordinates of the switch on the panel
nr_pos – can be 2 or 3. A 2-pos switch only uses the up and down positions (+1 and –1), whilst a 3-pos switch will use all the up, middle, down positions (+1,0,-1)
spring - for 3-pos switches, if spring is 1 the button will revert to the middle position once the mouse button has been released. Otherwise, the button will remain in the set position.
SRC a QueryString towards an INT pointer that specifies what will this switch control. The range of possible uses go from a customary defined pointer (via the RegisterCusomPointer) in which case a SHIP:SOMENAME will be used, to different switch handles from the electrical or hydraulical engines, such as closing/opening a valve or turning some electrical equipment on/off.
[name] – optionally a name, to be used in QueryString when reffering to the switch.

Note that a left-click on a SWITCH will ADD to the position of the switch (i.e. –1 to 0, or down to middle) while a right-click on a SWITCH will REMOVE from the position (i.e. 0 to –1 or middle to down)

Components you can use on a SWITCH-CLASS instrument in a QueryString:

SRC(int*) - a pointer to the actual location this switch is changing
SRCP(int**) - a pointer to the pointer of the actual location this switch is changing

2. CLASS CSWITCH

CSWITCH is nearly identical in use with switch, with the exception of .cfg definition where

<CSWICHINSTRUMENT> x y 2 0 1 SRC(int) [name]

is to be used, with a “2 0 1” inserted before the SRC string, CSWITCHES are special switches in that they can control panel lighting trough the use of PanelName:LIGHT and PanelName:TEXTLIGHT QueryStrings.

3. CLASS TEXT

Text class is not really an instrument class as is a “panel element”. TEXT class objects are just that: text you can display on the panel.
You can add capitons to the text (i.e. /------ TEXT------\ )
You can also control whether this text is lighted or not. By assigning the PanelName:TEXTLIGHT pointer as a SRC to a CSWITCH, that respective cswitch will then control the lighting (ie.color) of all the TEXT-CLASS instruments on that respective panel.

Parameters in the PRD file:

COLOR *r g b*
RGB values of the color of text when panel lighting is on. Range is 0-255 for each value

NOCOL *r g b*
RGB values of the color of text when panel lighting is off. Range is 0-255 for each value

FONT name size
Actual name and size of the font used to display the text. Note that bitmap fonts are not used, and rather simple GDI fonts. Make sure you use default-windows fonts that are guaranteed to exist in this form on all user computers. You might get surprises otherwise.

Defining a TEXT in .cfg file

<TEXTINSTRUMENT> x y length dir <actual text on screen>

x y - these are the xy coordinates of the instrument on the panel
length - the actual length of the text in pixels. Note that this value does not stretch the text. Rather it provides the length of the caption line, if one is used.
d*ir* - the direction of the end of caption line:
a 1 will provide a /-----TEXT -----\ caption
a 0 will provide no caption line around the text.
a -1 will provide a \-----TEXT-----/ caption

Of course, you can choose not to use TEXT classes at all, and put all of your panel-aid text embedded into the background bitmap. If your panel is a complex one, please do so. If your panel text however is not very abundant (like it is on Apollo CSM), you might want to choose using the TEXT class and defining your texts as panel instruments. It provides an easier way of positioning via the ASCII text files, plus the nice feature of turning the lights on and off on the text.

4.CLASS ROTATIONAL

Rotational class is any instrument requiring a rotating needle to display a value (a digital clock). The needle itself is drawn by SPSDK, with the user only providing the background bitmap.

Parameters in the PRD file:

BITMAP path_to_bitmap
Name and path to the bitmap to be used. Note that the layout of this bitmap must be a 1 x 2 image of the background of the clock. The upper image will contain the background of the lights off clock, while the lower image will contain the lights-on background.

SIZE *x y*
Size in pixels of the background of the instrument. Note that the actual bitmap must be x pixels in width and 2*y pixels high.

CENTER x y
x y coordinates of the center of the clock, that is the actual axis of rotation of the needle.

MAX val deg
Maximum position of the clock. That is the needle will be rotated “deg” degrees when the clock will indicate the “val” value. Note that 0 degrees means the needle is pointing downwards and positive rotation values are clockwise.(i.e. 90 is left, 180 is up ,270 is right, etc.)

MIN val deg
Minimum position of the clock. That is the needle will be rotated “deg” degrees when the clock will indicate the “val” value.

NEEDLE w h
The size of the needle in pixels.
w is the actual width of the needle at base (center) while h is the actual height of the needle measured from center to top. Obviously h should be small enough for the needle to fit inside the background bitmap at all times (i.e. at any rotation angle).

Defining a ROTATIONAL instrument in .cfg file

<ROTATIONALINSTRUMENT> x y SRC(double) [name]

x y - these are the xy coordinates of the instrument on the panel
SRC - a QueryString towards an DOUBLE pointer that specifies what value this instrument is to monitor. The range of possible uses go from a customary defined pointer (via the RegisterCusomPointer) in which case a SHIP:SOMENAME will be used, to different double values needing monitoring, such as critical temperatures, pressures ,etc..
[name] – optionally a name, to be used in QueryString when referring to the instrument.

Components you can use on a ROTATIONAL-CLASS instrument in a QueryString:

SRC(double*) - a pointer to the actual location this instrument is monitoring
SRCP(double**)- a pointer to the pointer of the actual location this instrument is monitoring

5. CLASS LINEAR

Linear class is any instrument requiring a needle moving in a linear motion to display a value. The needle itself is drawn by SPSDK, with the user only providing the background bitmap.

Parameters in the PRD file:

BITMAP path_to_bitmap
Name and path to the bitmap to be used as switch. Note that the layout of this bitmap must be a 2 x 1 image of the background of the instrument. The leftmost image will contain the background of the lights off, while the rightmost image will contain the lights-on background.

SIZE *x y*
Size in pixels of the background of the instrument. Note that the actual bitmap must be 2*x pixels in width and y pixels high.

CENTER x y
x y coordinates of the center of the clock, that is the actual axis of rotation of the needle.

MAX xmax ymax
Maximum position of the instrument. The actual x,y coordinates of the needle when the instrument is displaying it's maximum value (maximum value is specified in the CFG file)

MIN xmin ymin
Minimum position of the instrument. The actual x,y coordinates of the needle when the instrument is displaying it's minimum value (maximum value is specified in the CFG file)

You can use a ymin=ymax value which will make the needle move in a horizontal way, or advertedly you can use a xmin=xmax which will make the needle move in a vertical way.

NEEDLE w h
The size of the needle in pixels.
w is the actual width of the needle while h is the actual height of the needle measured from center to top. Obviously, the w and h values must be chosen so that the needle will remain inside the background image at all times

ROTATE ang

Unlike the ROTATIONAL class, the LINEAR class displays a needle that “translates” at a fixed rotation angle. This angle can be specified here. A 0 deg specifies a downward pointing needle, 90 deg means left, 180 – up , 270- right.

Defining a LINEAR instrument in .cfg file

<LINEARINSTRUMENT> x y min max scale bias SRC(double) [name]

x y - these are the xy coordinates of the instrument on the panel
Maximum position of the instrument. The actual x,y coordinates of the needle when the instrument is displaying it’s maximum value (maximum value is specified in the CFG file)
min max – the actual range of values displayed by the instrument
scale bias - scaling and bias of the value indicated. When computing needle positioning, the scale and bias is used before comparing it with the min max values.
E.g.

<LINEARINSTRUMENT> 100 100 0.0 100.0 1.0 –273.15 HYDRAULIC:O2TANK:TEMP

In this example, the SRC is defined as the temperature of the O2TANK, which is a double value, the temperature of the tank in Kelvin. By using a scale of 1.0 and a bias of –273.15, the actual value indicated will be TEMP*1.0 – 273.15, which is in fact the Celsius value of that temperature. Because of the min and max values, this means this instrument will display the temperature of the O2TANK in degrees Celsius between 0 deg C and 100 deg C

SRC - a QueryString towards a DOUBLE pointer that specifies what value this instrument is to monitor. The range of possible uses go from a customary defined pointer (via the RegisterCusomPointer) in which case a SHIP:SOMENAME will be used, to different double values needing monitoring, such as critical temperatures, pressures, etc..
[name] – optionally a name, to be used in QueryString when referring to the instrument.

Components you can use on a LINEAR-CLASS instrument in a QueryString:

SRC(double*) - a pointer to the actual location this instrument is monitoring
SRCP(double**)- a pointer to the pointer of the actual location this instrument is monitoring

6. TB CLASS

TB(TalkBack) class provides a simple way to monitor events. TB class will display a different bitmap if a pre-set condition is true. This makes it very efficient to use as talkback indicators, alarms, etc..

Parameters in the PRD file:

BITMAP path_to_bitmap
Name and path to the bitmap to be used. Note that the layout of this bitmap must be a 2 x 1 image of the background of the instrument if the TB is not blinking, or a 3x1 image of the instrument if the TB is to blink. The leftmost image will contain the background of the off status, while the second image will contain the on status, finally if present a third right-most image can be provided, and the TB will alternate between the right and middle bitmaps to create a blinking effect.

SIZE *x y*
Size in pixels of the background of the instrument. Note that the actual bitmap must be 2*x (or 3*x) pixels in width and y pixels high.

BLINK freq
Frequency of blinking. Zero if no blinking is desired, otherwise number of blink status changes per second.

Defining a TB instrument in .cfg file

<TBINSTRUMENT> x y SRC(double) condition [name]

x y - these are the xy coordinates of the instrument on the panel
SRC(double) - a pointer to a double value this instrument monitors
condition - the actual condition that will trigger this TB to go on. Acceptable values are:
ZERO - TB will light when SRC = 0
NON_ZERO - TB will light when SRC <> 0
GR_ZERO - TB will light when SRC > 0
SM_ZERO - TB will light when SRC < 0
[name] – optionally a name, to be used in QueryString when referring to the instrument.

Components you can use on a TB-CLASS instrument in a QueryString:

SRC(double*) - a pointer to the actual location this instrument is monitoring
SRCP(double**)- a pointer to the pointer of the actual location this instrument is monitoring

7. DIGIT

Digit class, though probably can be thought of other interesting uses, was primarily designed to be used as a digital clock instrument. It will display a digital number on the panel, with the actual digits provided in bitmap form.

Parameters in the PRD file:

BITMAP *path_to_bitmap*
Name and path to the bitmap to be used. This bitmap must contain a list of all digits [0..9] plus a tenth open space for non-digits. Plus and minus signs are not supported

SIZE *x y*
Size in pixels of one digit. Note that the actual bitmap must be 10*x pixels in width and y pixels high.

Defining a DIGIT instrument in .cfg file

<DIGITHINSTRUMENT> x y number SRC(double) [name]

x y - these are the xy coordinates of the instrument on the panel

number - number of digits this instrument will have (i.e. 3 means a 999 display)
SRC(double) - a pointer to a double value this instrument monitors
[name] – optionally a name, to be used in QueryString when referring to the instrument.

Components you can use on a DIGIT-CLASS instrument in a QueryString:

SRC(double*) - a pointer to the actual location this instrument is monitoring
SRCP(double**)- a pointer to the pointer of the actual location this instrument is monitoring

8.MFD

Obviously MFD class instruments provide a simple way to include standard Orbiter MFDs onto your panels.

Parameters in the PRD file:

SIZE *x y*
Total size of the MFD, side buttons and margins included

BORDER *w h*
The width and height of the "border" around the MFD, in pixels. This border is used for drawing buttons, etc..

FONT *name size*
The name and size of the font used to display button text. See explanations at TEXT class.

BUTTONS *left right*
Number of buttons on the left and right sides of the MFD. A minimum of 12 buttons is currently needed to display all buttons currently in use by default MFDs.

BUTTONSIZE *yoffs ydist*
Offset in pixels of the first button, and distance between the buttons on the y-axis.

BUTTONCOL *r g b*
RGB color of the button text. Range is 0 - 255 for each value.

Defining a MFD instrument in .cfg file

<MFDINSTRUMENT> x y index

x y - these are the xy coordinates of the instrument on the panel
index - all the MFDs must be indexed, to differentiate in-between. You can define as much as 5 MFDs per panel. MFDs defined with index 0 can use the LeftShift and those defined with index 1 can use the RightShift.

9.SELECT

SELECT class, though very similar with DIGIT, has a somewhat different function. You can use the SELECT class instruments to provide a way for the user to select from a larger list of values (think of rotary dials)

Parameters in the PRD file:

BITMAP *path_to_bitmap*
Name and path to the bitmap to be used. This bitmap must contain a list of all the various positions of the rotary dial. The number of position is only limited by the size of your bitmap, but you should make sure you have enough "position" on the bitmap for all the values you whish to select.
SIZE *x y*
Size of the instrument in pixels. Note that the bitmap should be u*x pixels in width and y pixels in height, where u is the number of positions.

MAX *max*
The maximum value selectable (int)

MIN *min*
The minimum value selectable (int). Note that the min value does not always need to be 0. You can define a SELECT instrument with max=10 and min=-10. In this case your bitmap needs to contain 20 images and the instrument will have 20 possible positions.

Defining a SELECT instrument in .cfg file

<SELECTINSTRUMENT> x y pos name

x y - these are the xy coordinates of the instrument on the panel
pos – the initial position of the select (must be in the min-max range of the class)
name –To be used in QueryString when referring to the instrument. Name is still optional, though there isn't much use for an SELEC instrument if it has no name to refer by.

Components you can use on a SELECT-CLASS instrument in a QueryString:

SEL(int*) - a pointer to the value this instrument has been set at.

9b. SELECTEL

A special kind of class, SELECTEL does not need to be specified in the PRD file. SELECTEL is specified directly in the CFG file.
SELECTEL allows changing the SRC value of some instruments based on another index-value. For instance ,you can link the SRC of a LINEAR instrument in function of the value selected by a SELECT instrument. This will cause the LINEAR instrument to monitor the respective value the pilot has selected by rotating a SELECT to the desired position.

Defining a SELECTEL in .cfg file

<SELECTEL> *TRGP(double**) SRC(int*) min max current*
SRC(double)*
SRC(double)*
…
</SELECTEL>

TRGP(double**) – pointer to the location where the instrument is holding the pointer to the value it is monitoring. E.g. LINEARINSTRUMENT:SRCP
SRC(int*) - pointer to a location providing the index of the item to select. Can be the SEL component of a SELECT class, or the SRC component of a SWITCH class
SRC(double*) - a list of double pointer to use starting from the min to max values.

E.g. In the .cfg file:

<SWITCHINSTRUMENT> 100 100 3 0 SHIP:SOMEVALUE SELSW
<SELECTINSTRUMENT> 100 200 3 SELSEL

<LINEARINSTRUMENT> 200 100 0.0 100.0 1.0 –273.0 HYDRAULIC:O2TANK:TEMP TEMPG
<LINEARINSTRUMENT> 200 100 0.0 100.0 1000.0 0.0 HYDRAULIC:O2TANK:PRESS PRESSG

<SELECTEL> PanelName:TEMPG:SRCP PanelName:SELSW:SRC –1 1 –1
HYDRAULIC:O2TANK:TEMP
HYDRAULIC:H2TANK:TEMP
HYDRAULIC:N2TANK:TEMP
</SELECTEL>

<SELECTEL> PanelName:PRESSG:SRCP PanelName:SELSEL:SEL 0 3 0
HYDRAULIC:O2TANK:PRESS
HYDRAULIC:H2TANK:PRESS
HYDRAULIC:N2TANK:PRESS
</SELECTEL>

The first two lines define a 3-pos switch and a 3 pos select.
The second two lines define 2 linear gauges, one indicating tank temp, and other pressure
The first SELECTEL states that:
When the switch is in the down pos, the temp gauge will indicate temp of O2TANK
When the switch is in the middle pos, the temp gauge will indicate temp of H2TANK
When the switch is in the up pos, the temp gauge will indicate temp of N2TANK
Default value is –1 (O2TANK)
The second SELECTEL states that:
When the SELECT is in the 0 position, the press gauge will indicate press of O2TANK
When the SELECT is in the 1 position, the press gauge will indicate press of H2TANK
When the SELECT is in the 2 position, the press gauge will indicate press of N2TANK
Default value is 0 (O2TANK)

10.ADI

ADI class provides for a way to display a spherical shaped ball on the panel. The ball can be lighted into any color desired, can be textured and rotated to any desirable angle.

Parameters in the PRD file:

BITMAP *path_to_bitmap*

Name and path to the bitmap to be used. Unlike previous instruments this bitmap functions more like a bitmaks around which the ADI is to be drawn. The RGB 0 0 0 color is considered transparent, and the ADI is drawn in the middle of this bitmap (where is black).

BALTEX *path_to_bitmap*

Same as BITMAP, but this bitmap is used to texture the ball.. the bitmap should to be in a 256x256 size, but any pow2 values will do.

SIZE *x y*

Actual size of the instrument (not of the ball) .Should be the same size as BITMAP

ZOOM *–y*

Size of the ball. Larger values mean smaller balls.. adjust to your needs.

LIGHT *x y z*

3d vector for lighting. When panel lights are turned on, there will be a light cast onto the ball . You can use this vector to adjust lighting direction on your ball for consistency with the rest of the bitmaps

AMBIENT r g b

RGB color of ambient lighting of the ball.

LIGHTCOL r g b

RGB color of light cast over the ball when panel lighting is on.

Defining a ADI class in .cfg file

<ADIINSTRUMENT> x y yaw pitch roll [name]

x y - these are the xy coordinates of the instrument on the panel

yaw pitch roll – QueryString pointers to 3 doubles that provide rotational control over the ball. Remember that you can use SELECTEL to select different rotational sources for the ADI ball.

[name] – optionally a name, to be used in QueryString when referring to the instrument.

Components you can use on a ADI-CLASS instrument in a QueryString:

YAW(double*) - a pointer to the actual location providing yaw orientation
YAWP(double**)- a pointer to the pointer of the actual location providing yaw orientation
PITCH(double*) - a pointer to the actual location providing pitch orientation
PITCHP(double**)- a pointer to the pointer of the actual location providing pitch orientation
ROLL(double*) - a pointer to the actual location providing roll orientation
ROLLP(double**)- a pointer to the pointer of the actual location providing roll orientation

11. PLOT

Probably the most complex class, PLOT allows you to define instruments that plot various paths on a 2D graphic scroll tape. The most obvious use of this is the Acc vs. Vel EMS re-entry instrument in the Apollo CSM, but I'm sure you can find others.

Parameters in the PRD file:

BITMAP *path_to_bitmap*
Actual name and path to the bitmap holding the scrolling tape over whom the plot is to draw.

MASK1 *path_to_bitmap*
Path to a bitmap holding a bitmaks around the scroll tape. This functions much in the way of the bitmask around the ADI class. RGB 0 0 0 is considered transparent, and the scroll tape (and plot) is drawn where there is black on this bitmap.

MASK2 *path_to_bitmap*
Name and path to a bitmap holding the "bug" bitmap, or the actual image to be plotted over the graph.

SIZE *x y*
Size of the instrument in pixels, should be the same as the size of MASK1

BUGSIZE *x y*
Size of the bug ,(i.e. the size of MASK2)

TRACE 0/1
If 1, the bug leaves a "trace"

MAX *x y*
The x,y pixel values of the bug at max-x and max-y position

MIN *x y*
The x,y pixel values of the bug at min-x and min-y position

Defining a PLOT class in .cfg file

<PLOTCLASS> x y Y-SRC(double) min-y max-y X-SRC(double*) min-x max-x x-res [name]*

x y - these are the xy coordinates of the instrument on the panel
Y-SRC(double)* – pointer to a double value controlling the y axis positioning of the bug
min-y max-y – the range of values to be displayed on the y-axis
X-SRC(double)* – pointer to a double value controlling the x axis positioning of the bug
min-x max-x – the range of values to be displayed on the x-axis
x-res – the resolution of the x-axis. (i.e. the number of units per pixel deflection). This value is used to "scale" the x-axis deflection and thus control the total length of the scroll tape used. The value represents the number of x-axis units monitored per pixel.

Components you can use on a PLOT-CLASS instrument in a QueryString:

ARM(int*) – pointer to a on/off controller. 1 means instrument is plotting
XAXIS(double*) - a pointer to the actual location providing x-axis values

XAXISP(double**)- a pointer to the pointer of the actual location providing x-axis values
YAXIS(double*) - a pointer to the actual location providing y-axis values
YAXISP(double**)- a pointer to the pointer of the actual location providing y-axis values

Chapter 3 – the CFG file

Introduction

The entire .cfg file of the ship is now split into 8 main sections.

The first section, deals mostly with Orbiter related definitions:

ClassName = name
Module = name
PanelType = name

ClassName and Module points to a .dll file
Note: as of version 1.5, the name of the cfg file that contains the definition data for your ship can be defined via a scenario file , by using the

<SOURCE> file_name

parameter and defining your vessel as a "spsdk" type of vessel. In this case, the *ClassName* and *Module* parameters are not used from this cfg file.

PanelType points to a. prd file whereas all the panel's instruments look and behavior is defined

The next 7 sections deal with the actual definition of the ship elements

<VESSEL>
Defines all the visual aspects of the ship. Thuster definitions and diferent stage configurations are also defined in here

<KEYBOARD>
Maps a series of keys to specific actions, such as triggering animations (not implemented yet) or triggering stagings.

<THERMAL>
Deals primarily with the definition of the thermal properties of the hull. Outer skin material characteristics and such can be defined here.

<HYDRAULICAL>
Defines the internal hydraulical parts of the vessel. Basic rule of thumb for the definition of a hydraulic item is any system on the ship that does not require electrical power to operate.

<ELECTRICAL>
Defines the internal electrical parts of the vessel. Complementing the hydraulical part, the electrical items are any system ships that use (or produce) electrical power.

<PANEL>
Defines the layout of the panels that provide controls and monitor capabilities for the items (systems) defined in the previous sections.

<SEPARATION>
The final part of the CFG defines the various items and equipment that are lost during staging (for multi-staging ships) and the items that are "jettisoned" durring staging.

1.<VESSEL>

The VESSEL section is split into 3 main categories:

<ELEMENTS>

This is where you define the diferent “bits” that make up your multi-stage ship:

<COMPONENT_NAME> mesh_name total_mass min_stage max_stage
<x y z >
</>
COMPONENT_NAME – the name of this particular component (eg. Booster, external tank,etc..)
mesh_name – the name of the mesh for this component (w/o the .msh extension)
total_mass – total mass of this component (including fuel and such) (in kg)
min_stage – at what stage should this component “appear” in the stack (usually 1)
max_stage – at what stage should this component “dissapear” from the stack.

Then, on the next lines, in consequtive order, on each line, a mesh offset vector for each of the configurations where the mesh is present.(please refer to Generic_ship.cfg for an example of this)

<PROPELLANTS>

Much like you defined the mesh elements that compose your multi-stage ship, this is where you define the various propellant tanks your vessel will use. The syntax is :
NAME total_mass min_stage max_stage

NAME – the name of this propellant
total_mass – total mass of the fully-loaded tank (in kg)
min_stage – see <ELEMENTS>
max_stage – see <ELEMENTS>

<CONFIG> X

The third and final section deal with the actual configuration differences inbetween the configs (stages). The SPSDK already knows what meshes and what tanks to put in each stage, but here we must define the various atmospheric parameters and the list of thrusters:

Most of the params defined here are quite straightforward and are passed on directly from the OrbiterSDK. Consult the OrbiterSDK for a more detailed explanation of what each of them does.

<THRUSTERS>
<x y z> <x y z> thrust name_propellant isp1 isp2 group <length width>

<x y z> < x y z> - two vectors signifying the position and direction of each thruster
thrust – the actual thrust of this thrusters
name_propelant – as defined in the <PROPELLANTS> section
isp1 isp2 – the 1 atm and the vaccum isp of this thruster
group – the group to which this thruster belongs :
THGROUP_ATT_RIGHT
THGROUP_ATT_LEFT
…etc..
The group definitions are the same as the ones defined in the OrbiterSDK. Chech the OrbiterSDK header for a complete enumeration of the possible thruster groups.

<length width> - the actual size of the visual flame for this thruster

<DOCK> <x y z> < x y z> <x y z>
Also in the <CONFIG> section,you can define the various docking ports :

First vector – the postion
Second vector – direction for the port
Third vector – the “up” direction of your port

you can define as much as 10 diferent configurations (or stages) for your vessel. The actual stage the ship will be at can be controled from the <CONFIG> parameter stored in the scenario file

2.<KEYBOARD>

All of the keys defined in this section share the same name and layout as the ones defined in the OrbiterSDK , in the form of : OAPI_KEY_ *

Please refer to the OrbiterSDK header file for a complete list of key mappings.

Animations (not implemented)

Separations

OAPI_KEY_* SEPARATION stage_number

Will link the key provided in this line with the separation action that will go to the “stage_number” stage.
Please note that staging is only allowed in consecutive order (ie from stage to stage+1)

Eg.

OAPI_KEY_K SEPARATION 2
OAPI_KEY_L SEPARATION 3

In this example, pressing the “k” key will separate the vessel to configuration 2 (ie. second stage), whilst pressing the “l” key will separate the vessel to configuration 3 (or third stage). If the vessel is currently in configuration 1, pressking the “l” key will have no effect.

Note: you can link keyboard events to cswitches, thus perfoming separations by the flip of a switch:

<CSWITCH> 100 100 2 0 **2** OAPI_KEY_K

Defining a CSWITCH belonging to class **2** , then providing not a QueryString to a value, but the name of the key you wish to trigger. Activating this switch, will then have the same effect as pressing the respective key.

3.<THERMAL>

<RESOLUTION> - number of “patches” per side of the hull. Normally ,the ship is considered a paralelipedic structure with cross-sections as defined in Orbiter API via the SetCrossSections() function. This number controls the number of “patches”, or thermal points to be computed on

each side of the hull. Larger numbers will give a more accurate result of radiative heating, but is more computationally expensive.
<CONDUCTIVE> YES/NO -perform conductive heating calculations. Better simulation of thermal conditions, but prohibitively expensive. Recommend to keep this off by default.

<RADIATIVE> - YES/NO – perform radiative heating calculations.

<HEATCAPACITY> - $Jg^{-1}K^{-1}$ - property of the hull material
<CONDUCTIVITY> - $Js^{-1}m^{-1}$ - property of the hull material
<REFLECTIVITY> - dimensionless- reflectivity coefficient of the hull material. This affects the amount of heat received from radiative heating. Larger numbers mean more energy is reflected, thus less is absorbed by the ship.
<RADIATIVITY> - dimensionless- radiativity coefficient of the hull material. This affects the amount of heat dissipated from radiative cooling.
<HULLMASS> - mass of hull in kg. Density distribution is assumed homogenous.

4.<HYDRAULICAL>

- starts the hydraulical engine part of the definitions. Must be ended with
</HYDRAULICAL> before moving onto other areas

The hydraulical engine deals with all of the spaceship systems that do not use, or produce, electrical power. Hydraulical systems are basically split into two major sections: tanks and pipes. Obviously, tanks-like systems are used as space-holders for the actual fluids and gasses used by the spaceship, whilst the pipe-like systems are used to transport these fluids from tank to tank.

Note that QueryStrings used in the HYDRAULICAL section can only refer to other hydraulical objects. Thus, the “HYDRAULICAL:” part, i.e. the name of the engine, is always assumed to be HYDRAULICAL and is not needed to be used in the <HYDRAULICAL> part of the CFG.
Hydraulical items used by the engine:

TANK

<TANK> **name <x y z> volume in_valve out_valve [isolation]**
CHM **type mass vapor_mass enthalpy**
…
</TANK>

Tanks are basic holders for the hydraulic systems. All liquids and gasses are moved from tank to tank trough pipes.
name – name by which this tank is known. Used so that you can refer to it later.
<x y z> - vector position of the tank. Used by the thermal engine to compute radiation from the sun (i.e. is it facing the Sun or not, etc..)
volume - the volume of the tank in Liters. Volume is used by hydraulic engine (obviously!) but also by the thermal engine: all tanks are assumed perfect spheres. Based on the volume ,the thermal engine computes a “surface area”. The bigger the volume of the tank, the more energy it radiates away, but also the more it heats from the Sun
in_valve - is the in valve open or closed by default
out_valve - is the out valve open or closed by default
(there are 2 valves / tank. a IN valve and a OUT valve)
[isolation] – optional can specify a thermic isolation of the tank. Range is [0..1] whereas 0 specifies the tank is perfectly isolated (no heat radiative heat transfer), and 1 means not isolated and tank will transfer heat with the values specified to the thermal engine

On the second definition lines, are the chemicals that are present in the tank. Any number of CHM lines can be entered, but for now only 5 substances are known to the Hydraulic engine.
type - type of chemical (0 – O2, 1 – H2, 2 – H20 , 3 – N2, 4 – CO2)
mass – amount of that substance in grams
vapor_mass – how much of this substance is in vapor state ,in grams (vapor_mass <mass)
enthalpy - total enthalpy of this substance (ie. thermal energy)

Components that you can use on a TANK:

IN	-valve*(see VALVE components below)
OUT	-valve*
TEMP	- double*
MASS	- double*
TEMP	- double*
VOLUME	- double*

Components that you can use on a VALVE:

OPEN	-int* (used by switches to open/close)
PZ	-double* (non zero if valve is moving, use for TBs)
ISOPEN	-double* (0 if closed, 1 if open, also used for TBs

PIPE

<PIPE> **name**
NAME:COMPONENT NAME:COMPONENT [ONEWAY]
NAME:COMPONENT NAME:COMPONENT PREG max_p [min_p ONEWAY]
NAME:COMPONENT NAME:COMPONENT BURST max_p min_p
NAME:COMPONENT NAME:COMPONENT PVALVE max_p
…
<PIPE>

The pipes are defined in a list. The <PIPE> directive starts that list, and the </PIPE> directive ends the current group of pipes. You can define as many pipes as you want in a PIPE list.

name – the common name of the pipes. You can define as many groups of PIPEs that you want, or you can group them into one big PIPE list. Note that it is useful to group pipes by their purpose(you'll see later when we talk about <SEPARATOR>). This allows entire groups of pipes to be deleted by their purpose, rather than individually

Each line in the list defines a new pipe (connector) in-between two VALVES. For each line you specify TWO valves that are to be connected and the type of pipe in-between.

NAME:COMPONENT (valve pointer) – the first valve QueryString
NAME:COMPONENT (valve pointer) - the second valve QueryString

[ONEWAY] – optional parameter of the pipe. It means that liquids can only flow from the first to the second valve(if the second valve is at a lower pressure). If not specified, liquids (or gasses) will flow freely wherever there is a lower pressure

PREG max_p –specifies there is a pressure regulator on this pipe . The pressure from the first valve will never exceed this value when flowing to the second valve.

Ex.

H2O_TANK:IN WASTE_TANK:OUT PREG 101300.0

Specifies a pipe, from the IN valve of the H2O_TANK to the OUT valve of the WASTE_TANK. If the pressure in the H20_TANK is 200.000 pascals, and the pressure in the WASTE_TANK is 50.000 pascals, there will be a flow from H2O_TANK to the WASTE_TANK. However, once the pressure in the WASTE_TANK reaches (or exceeds) 101300 pascals, there will be no more flow, since the water coming from H20_TANK is "regulated" by a pressure regulator to 101300 pascals of pressure.

BURST max_p min_p - the BURST parameter indicates there is a burst disc on the pipe. This disc will open once the pressure difference between the first valve and the second valve exceeds max_p and will resettle back to close again, once the pressure difference gets lower than min_p

Ex.

H2O_TANK:IN WASTE_TANK:OUT BURST 100000.0 50000.0

1) Pressure in H2O_TANK is (say) 170.000 pascals and the pressure in WASTE_TANK is 100.000 pascals. Difference is 70.000 , so there is no flow. The burst disc is closed.
2) Hours later , the H2O_TANK pressure rises to 260.000 pascals. The pressure difference is 110.000 pascals , the burst disc opens, so now water flows from H2O_TANK to WASTE_TANK
3) After a few minutes the pressure in H2O_TANK has decreased to 200.000 pascals, but the pressure in the WASTE_TANK has increased to 160.000 pascals. The difference is now only 40.000 pascals, so the burst disc will close.

PVALVE – this parameter, indicates that the pipe must function as a pressure bladder. There will be no actual flow of materials from the first valve to the second valve, but there will be a sharing of volume. If the second valve is at a lower pressure than the second volume, there will be a flow from the first valve to the second, but there will be no actual mixing of the substances. Once the two pressures will equalize, the flow will stop. Note that the "bladder" can only develop inwards on the second valve:

Ex.

N2_PRESS:IN O2_TANK:IN PVALVE 20000.0

If the pressure in N2_PRESS tank is larger than the pressure in the O2_TANK, some nitrogen will flow from the N2_PRESS tank to the O2_TANK, but the oxygen will not mix with the nitrogen. there is a bladder separating them.

Say, initially, the N2_PRESS has 40 liters and O2_TANK has 20 liters. Nitrogen will keep flowing inside the O2_TANK until the two pressures balance out. Say, now the nitrogen occupies 40 liters inside the N2_PRESS and some 5 liters inside the O2_TANK = 45 liters. Oxygen occupies 20-5 =15 liters.

Notice, that if the pressure of the oxygen is raised (for one reason or another), oxygen can push the nitrogen BACK into the N2_PRESS tank, but it will never flow into N2_PRESS. Oxygen can attain a maximum of 20 liters (i.e. .the size of the O2_TANK).

Because the HYDRAULIC can only access objects from the HYDRAULIC system, the SYSTEM part is always assumed to be HYDRAULIC when defining hydraulic object, hence this is not needed for pipes.

A PIPE has no components that you can reefer to

VENT

<VENT> **name in_open out_open**
<x y z> <x y z> size
....
</VENT>

Vents act just like tanks, only that everything that flows into them is vented into outer space (via a thruster).

name - same as before
in_open/ out_open - see Tank

<x y z> <x y z> - defines the position and direction of the jet flowing into space. you can define as much as 4 "holes" per vent
size - the size of the "hole" in cm

Components that you can use on a VENT:

*same as TANK

RADIATOR

<RADIATOR> **name <x y z> temp**
area radiation mass
</RADIATOR>

Radiators are just that. Hydraulic objects that dump excessive heat into outerspace by thermal radiation.. these are later used by the electrical COOLING LOOP objects, which will pump thermal agent into them for cooling, then looping the colder thermal agent to the warm equipment.
temp - the initial temperature of the radiator's surface
area – actual surface area of the radiator
radiation - "radiativity" properly of the material. i.e. how much does it radiate..
mass – bulk mass of the heat sink

note that specific heat is assumed 0.15(aluminum), since it speeds up computations A LOT having ALL of the "external" skins of the spacecraft at the same specific heat.

Components that you can use on a RADIATOR:

TEMP - double*

CREW

<CREW> **name nr ITEM(tank)**
name - you can actually name the crew and then use this to delete upon staging
nr – actual number of people to calculate
ITEM(tank) – pointer to a hydraulic item (must be a tank, let's be serious) where the crew is located

What this item will do , is simulate the breathing activity of the crew. Oxygen is consumed (at a rate of ~250ml per minute per person) and carbon dioxide is expelled (at around 200ml per minute) . Also a small percentage of water vapor is produced.

Components that you can use on a CREW:

none

5. <ELECTRIC>

-electric section must be ended with </ELECTRIC>. Normally (with a few exceptions) electrical system is now assumed by default (i.e. HYDRAULIC:ITEM must now be explicitly used to select an item from the hydraulic section)

Electrical engine contains and handles all items on the spaceship that uses or produces electrical power in any sort of way. There is a way to chain-link multiple electrical equipment to a single electrical circuit (or bus), but the engine will always link them in parallel, as there is no way to link multiple electrical items in a serial way on the bus using this SDK.

Hydraulical items used by the engine:

BATTERY

<BATTERY> **name max_power source**

max_power - maximum power stored inside the batt. (in watts second)
source – pointer to another electrical equipment, used to re-load the battery. Use NOPOWER if the battery is not to be recharged
Components that you can use on a BATTERY:

LOAD	- int* (a 2 pos. switch, to turn battery re-loading on/off)
VOLTS	- double*
AMPS	- double*
RESET	- int*
SRC	- electrical object* (source)

Note: RESET is a special items for electrical items and functions much like a on / off switch . The proper way to use this is with a 3pos /springed switch, whereas a momentary “up” position will turn the equipment on, and a momentary “down” position will turn the equipment “off”

FCELL

<FCELL> **name <x y z> max_amp O2 H2 WASTE**

max_amp - the maximum amperage this fcell can produce (all Fcells produce @ 28.8 Volts)
<**x y z**> - location of the fuel cell in vessel coordinates
O2 - pointer to a valve that brings oxygen reactant
H2 - pointer to a valve that brings hydrogen reactant
WASTE – pointer to a valve where the FC will put the resulting water (from the reaction)
Note the valves are automatically assumed to belong to the hydraulic section, so HYDRAULIC: is not required to specify any of the O2 H2 and WASTE valves, only TANK:VALVE.

Components that you can use on a BATTERY:

VOLTS	- double*
AMPS	- double*
START	- int*
RUNNING	- double* (for TB)
TEMP	- double*
DPH	- double*

Note: START is the actual component to use on a fuel cell for ON/OFF switching (not RESET) DPH is a number reflecting the changes in the Ph inside the fuel cell stack. While the fuel cells WILL purge themselves automatically (if on) there are a number of items to be careful about . While purging, the fuel cell can only produce about 60% of max_amp, so care must be taken not to overload the fuel cell while purging. Also a large quantity of reactants might be dumped on to the WASTE tank, which usually leads to a large rise in pressure and further on to a H2O venting into space(which might produce a rotation or a translation). DPH provides a way to monitor the Ph inside the fuel stack and know when to expect a fuel cell purge.

DCBUS / ACBUS

<DCBUS> **name source**
<ACBUS> **name source**

source – name of an electrical system that this bus will draw power from.

Besides the obvious difference that DC busses provide direct current while AC busses provide alternative one, note that DC busses operate at 28.8V while the AC busses will operate at ~36V. Be careful when linking electrical systems, so that equipment requiring alternative current are not connected to direct current or viceversa.

Components that you can use on a DCBUS / ACBUS:

VOLTS - double*
AMPS - double*
RESET - int*
SRC - electrical object* (source)

SOCKET

<SOCKET> **name ip source sel1 sel2 sel3**
ip - initial setting (1,2 or 3)
source - the electrical item this socket controls
sel1, sel2, sel2 - up to three possible electrical sources to select from

Sockets are used to route power from and to electrical equipment. As described later on, switches on the panel can be linked to these sockets, so that the pilot can choose the power source for an item by simply flipping a switch:
E.g.
<SOCKET> SK1 1 DC1 FC1 BAT FC2
…
<SWITCH> 100 100 3 0 ELECTRIC:SK1:CONNECT

The first line (located in the Electric section) defines a socket, named SK1 that controls the (source) DC1 bus. The three “options” of power for DC1 are ,in order, fuel cell 1 (FC1) , a battery (BAT) or fuel cell 2 (FC2). The second line (located in the Panel section) will provide a switch that, when in the down position will cause the DC1 bus to “connect” and draw power from FC1. When the same switch will be in the middle position, DC1 will draw power from BAT and so on, when the switch will be in the “up” position DC1 will draw power from FC2.

Components that you can use on a SOCKET:

CONNECT -int

COOLING

<COOLING> **name on source heat_t min_t max_t**
name length
….
</COOLING>

on – whether this cooling loop is on(1) or off(0) by default
source (DC) - name of an electrical equipment from which this item draws power(can be changed via sockets)
heat_t – the heat transfer capabilities of the cooling agent. All in all this number contains a large number of parameters for the coolant loop. The type of coolant used, diameter of the pipe, etc… all of whom are linear and thus combined inside one single number. What it all comes down to is: large number provides better thermal transfer while lower number provides less.
min_t – the minimum temperature. When set to "auto" , and the coolant temperature reaches below min_t, the coolant pump will be turned off.
max_t - the maximum temperature. When set to "auto", and the coolant temperature exceeds max_t, the coolant pump will be turned on.
name length - each subsequent line (maximum of 6) define the various items that the coolant loop traverses for cooling. Basic rule of thumb is that only items that have a <x y z> parameters should be used, as they are the only ones that have thermal computations attached. In this case both ELECTRIC: and HYDRAULIC: must be specified, since items can be comprised of both. length specifies the length of the coolant pipe trough that item (in cm.) and is basically a weight parameter as to how much should this coolant loop cool (or warm) this particular item.

Note that the actual flow of the coolant loop is considered in the same order as defined in the list. Putting a very warm item AHEAD of a very cold item, means that the second colder item might actually be warmed , since the coolant in the loop might have been previously warmed . It is generally a good idea to include a RADIATOR as the last item in the list, to provide a way for the coolant to dump the excessive heat into outerspace, and generally to keep the coolant at low temperatures. You can adjust the length of the piping trough the radiators to keep the coolant at lower or higher temperatures.

Basically, one could thing at the <COOLING> object as a big temperature "fudging " item, whereas all of the items in the list tend (very slowly) to reach a common temperature.

Components that you can use on a COOLING:

PUMP	-int*
HMAX	-int*
HMIN	-int*
ISON	-double* (for TB use)
MAXT	-double*
MINT	-double*

Note: the PUMP component will set the different modes of operation via a panel switch:
Ie. Switch in the "down" (-1)position will manually force the pump ON. Switch in the middle (0) position will manually force the pump OFF. Switch in the up(+1) position will cause a "auto" mode.
HMAX is a int to be used with a 3pos/springed switch. Holding this switch up will increase the max_t temperature used by the auto mode at the rate of 1 deg / second, while holding the switch in the down position will decrease the max_t by 1 deg / second.
MAXT is the actual max_t value used by the auto mode.
Same for HMIN and MINT.

ATMREGEN

<ATMREGEN> **name on source delta_press in_valve out_valve**

on – whether the fans are on(1) or off(0) by default
source (DC)- pointer to electrical power source
delta_press - the fan capacity in pascals
in_valve (VALVE)- the input of the regenerator
out_valve (VALVE)– the output

All of the ,rather complex, atmospheric regeneration systems usually used on a spaceship are comprised into this one big item called ATMREGEN, mainly for simplicity and to save some extra computations. What this item will do, is take air from one source (in_valve) via a DC powered fan, remove all of the water vapor and carbon dioxide from it, the release it via the out_valve.

Components that you can use on a ATMREGEN:

PUMP	-int*
ISON	-double* (for TB use)

Note: Here, PUMP only has 2 possible values to be used via a 2pos switch: down position meaning OFF and up position meaning ON

HEATER

<HEATER> **on source(AC) temp watts target**
name,on,source – same as before
temp - when set to "auto" mode , this is the temperature this boiler will attempt to maintain
watts - specifies the power consumption of the heater. Larger number uses more power, but heats up more
target - pointer to item that is to be heater. Again, note that only objects with <x y z> are actually valid targets, as they are the only ones with thermal computations attached.

Components that you can use on a HEATER:

PUMP	-int*
HMAX	-int*
MAXT	-double*
ISON	-double* (for TB use)

Note:see text at COOLING for a description on PUMP, HMAX and MAXT

6.<PANEL> PanelName

- starts the panel definition. Must be ended with </PANEL>

PanelName – the name of the panel is to be used to refer to instruments on the respective panel on QueryStrings

Besides all the instruments defined in the PRD file, there are a number of other items you can use in the <PANEL> sections

SIZE x y
The total size in pixels of the panel background bitmap

BITMAP path_to_bitmap
Much the same as in the PRD section, this is the path to the background bitmap

TRANSPARENT r g b

The RGB color that the panel uses for transparency

ATTACH PANEL_ATTACH_TOP(LEFT/RIGHT/BOTTOM)
PANEL_MOVEOUT_TOP(LEFT/RIGHT/BOTTOM)
Defines the way the panel is to scroll onto the screen.

NEIGHBOURS left right top bottom
Indexes of the panel neighbors in the respective directions. Note that this does not use names, but rather indexes. Indexes are computed starting with 0 as the first defined panels and increasing by 1 for each subsequent panel.

Components you can use on a PANEL

LIGHT (int*) - to be used by a CSWITCH class instrument. The switch will then control the light on/of status on the panel
TEXTLIGHT(int*) - to be used by a CSWITCH class instrument. The switch will then control the light on/of status of the text on the panel
MFDMODE(int*) - to be used by a CSWITCH class instrument. The switch will block/allow the use of MFDs on the panel. Off status means no MFDs can be opened.
HUDMODE(int*) - to be used by a 3-pos CSWITCH class instrument. The switch will then set one of three modes of the HUD: OFF / Surface / Orbital

7.<SEPARATION>

This is not actually an engine, but here we can specify what objects we want to discard when we are staging. (for multistaging ships)

For example,
<CONFIG> 7
<DELETE> HYDRAULIC:H2O_TANK
</CONFIG>

means that when reaching configuration 7 (SM jettisoning) , we will loose the H2O_TANK ... Please remember to delete the pipes using this tank aswell, since this may crash the sim if you don't.

Note that the names are compared with wildcards at the end:
Ex
<DELETE> HYDRAULIC:H2O_TANK
will delete ALL the tanks whose name BEGINS with H2O_TANK. It will delete H2O_TANK and it will delete H2O_TANK2 as well. For pipes, all the pipes belonging to that name group will be deleted.

You can specify both ELECTRIC and HYDRAULIC items for <SEPARATION>, but note that, for obvious reasons, SHIP and PANEL items cannot be deleted upon separation.

A bit on the legal disclaimer

Last but least, this software has been developed over a large amount of time, mainly as a nighttime hobby. As such , though heavily tested, a number of bugs or omissions are bound to occur. I'd appreciate any feedback ranging from bug-reports, to suggestions or plain old praises. You can contact the developer at

poenaru.radu@rcc.rondo-ganahl.com