FrontPanel

User Manual
Version 2.0

An Electronic Equipment Front Panel Simulation Library

John Kichury

jjk439 at gmail dot com

Table of Contents

Introduction

Vintage computer simulators such as simh[1] have provided a great service by preserving computer history. They allow those who have worked with old computers to reminisce, and those who have arrived later to experience a little of what it was like in those early days. One thing that is missing however is an accurate visual representation of what the front panel really looks like in action. Many early systems had lights that monitored the address and data buses as well as various processor status signals. Some simulators exist which make a good attempt at providing working lights and switches, but don't capture the real appearance of what the actual hardware looked like while functioning.

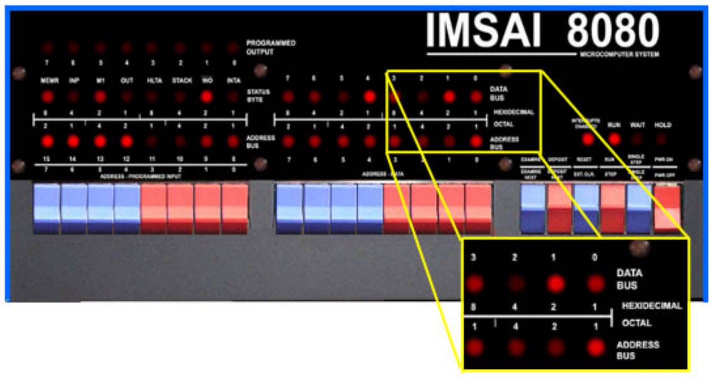
This project attempts to provide a general interface which allows front panels, lights and switches to be included in simulation projects. When the simulator is run on a modern computer with a 1.6GHz or faster CPU and a reasonably fast video card, the visual appearance can be surprisingly realistic.

New with FrontPanel 2.0

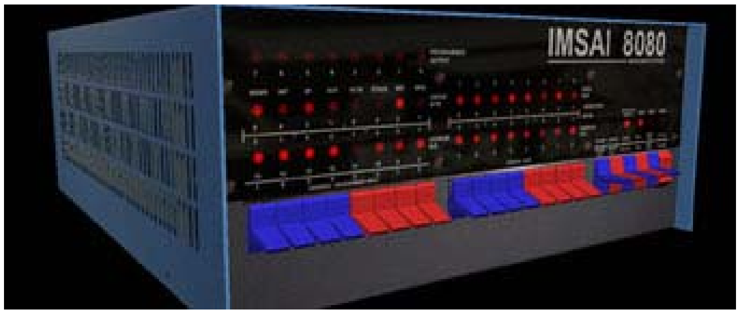
FrontPanel 2.0 contains new features that allow 3D representations of the hardware being simulated for even better realism. It is now possible to view and interact with a 3D model of a vintage computer system. For an example see the Imsai3D demo that is included with the FrontPanel 2.0 software. Example screenshots are shown later in this document.

Example Screenshots

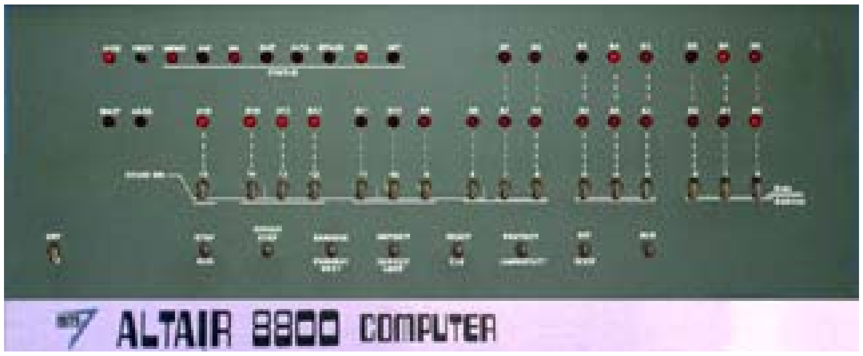
Below are screenshots taken of various systems running in simh integrated with *FrontPanel*. All of the examples with source code can be found in the *FrontPanel* software directory which you can experiment with. In all of the cases below, the simulator was actually running code when the image was saved. Look closely and you will notice the varying light intensities as they appeared at that precise moment in time.



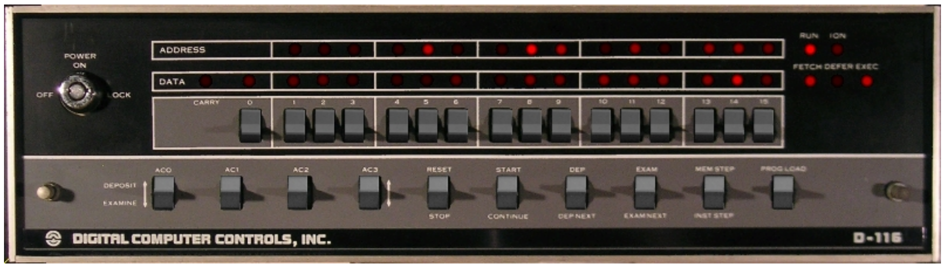
IMSAI 8080
Screenshot running in *FrontPanel*



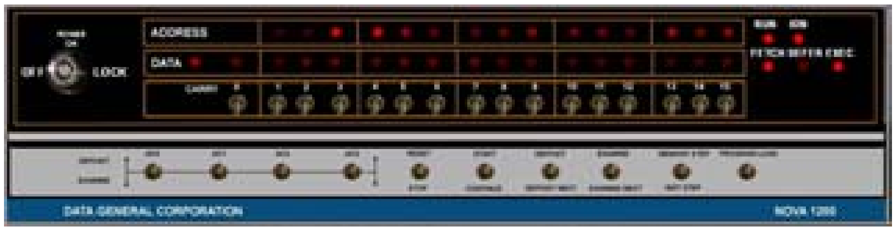
IMSAI 8080 3D Model
Screenshot running in *FrontPanel*



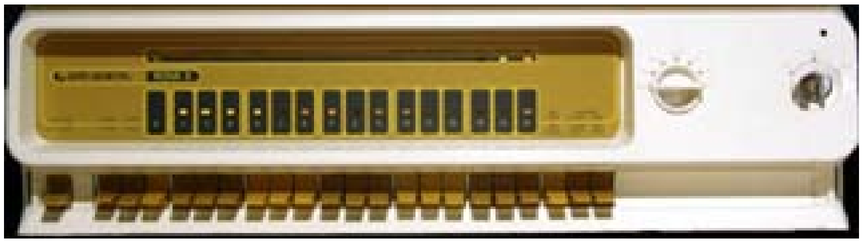
Altair 8800
Screenshot running in *FrontPanel*



Digital Computer Controls DCC-D116
Screenshot running in *FrontPanel*



Data General Nova 1200
Screenshot running in *FrontPanel*


Data General Nova 3
Screenshot running in *FrontPanel*

How FrontPanel works

The lamps or LEDs located on the front panel of actual systems were operated by a driver circuit attached directly to the signal being monitored (e.g. bits of an address or data bus). As such, rapid voltage transitions at nanosecond or microsecond durations cause bursts of light for very short periods of time giving an illumination characteristic that is challenging to simulate. Modern computer graphics systems aren't fast enough to draw the lights at intervals such that short transitions are represented. Further, you don’t want the computations required for drawing a representation of the lights to impede the performance of the simulation too much.

FrontPanel is a C++ class library with C bindings that can be linked with a simulator. It can be used with simulators written in either C or C++. A text configuration file is created that describes the physical appearance of the panel, placement of lights and switches etc. Function calls for initializing and binding light and switch functions on the front panel are placed in the simulation source code. Upon calling fp_init and passing the path to the configuration file as an argument, a graphics window is opened up which displays the front panel. All graphics operations (drawing, user switch selection etc.) is carried out in a separate thread while the actual simulation code runs in the main process. This allows the graphics functions and simulation functions to be decoupled and run at different rates. The simulator can execute the instructions of the computer it is simulating and periodically send data sampling requests to *FrontPanel*. The graphics thread models light intensities based on computed signal durations or duty cycles.

The front panel appearance can be simple using basic graphics elements such as polygons and lines, or increased realism may be attained by using digital images of the actual system being simulated.

System Requirements

Currently *FrontPanel* has been developed on the Linux operating system and has been tested on OpenSuSe 10.2.

Required Hardware and Software

X86 system running Linux
gcc and g++ compilers and development libraries
X11 and OpenGL development libraries installed
Pthread library (libpthread)
Jpeg Library (libjpeg and libjpeg-devel)

The best performance and most realism will be attained if the system has a CPU running at 1.6GHz or faster, and have a video card that has a GPU such as from nVidia or ATI.

Steps for adding a light and switch panel to a simulation.

1) Create a text configuration file that describes the physical appearance of the panel, which includes basic graphics objects and location, size and color of lights and switches.

2) Modify the source code of your simulator to include a simulation clock. Add subroutine calls for binding lights and switches to data values, and place data sampling calls within the simulation loop for providing the necessary timing information to *FrontPanel*.

The Frontpanel Configuration File

The *FrontPanel* configuration file is a text file containing directives for describing the characteristics of the front panel. Below is a list of recognized statements that may appear within the configuration file.

Comments may appear in the configuration file and must begin with a pound (#) sign.

color <r> <g> <b>

Defines a color for the current object red, green and blue components of the color are specified as 3 real numbers in the range of 0 through 1.0 Color must appear within an object.

# comment

Any line beginning with a pound (#) sign is considered a comment.

envmap

Specifies that the texture mapping on the current object be done using environment mapping which produces a reflective property. Instead of the texture being frozen and rotating with the object, the texture is used as a reflection.This is useful for representing shiny objects such as chrome, or in the case of an IMSAI 8080, the reflective plexiglass portion of the front panel.

For environment mapping to take affect, the object must have a texture defined (see the texture directive) and have surface normals applied to the geometry.

instance <object_name>

Allows the current object to "instance" or use geometric data from the object named in the argument. The *instance* command must appear within an object definition.

light <name> <options>

Defines a light or LED that is to appear as an indicator on the equipment being simulated. See the section below entitled "Describing Lights" for more information on the options for this statement.

lightcolor <r> <g> <b>

Specifies the default color for subsequent light definitions. Colors may also be defined on a per light basis (see the Describing Lights section of this manual).

lightsize <x> <y>

Specifies the default size for subsequent light definitions. Sizes may also be defined on a per light basis (see the Describing Lights section of this manual).

line

Begins the definition of a line that will be drawn as a graphics element, and must appear within an 'object'. Lines may be 2D or 3D by the definition of subsequent 'v' (vertex) statements.

material

If specified before object definitions, this directive defines a material property which may be used by graphics objects. The arguments are as follows:

n Ar Ag Ab Aa Dr Dg Db Da Sr Sg Sb Sa shine Er Eg Eb Ea

where:

n	is an integer material number by which this material will be referenced by objects. .
Ar Ag Ab Aa	are the floating point ambient light red, green, blue and alpha components in the range of 0.0 – 1.0.
Dr Dg Db Da	are the diffuse color components
Sr Sg Sb Sa	are the specular color components

shine	is the shininess
Er Eg Eb Ea	are the emission components.

If material is specified within an object definition, there must only be one argument which specifies the integer identifier of a previously defined material.

message <text string>

Prints a text message. The text following the command ‘message’ to be printed to stdout at the time the configuration file is being read in. This is useful for displaying copyright notices, instructions etc.

n <x> <y> <z> Normal

Defines a surface normal vector for the preceding vertex statement. The normal is used for lighting calculations when the objects are drawn. The x, y, and z values are floating point and must define a unit vector (vector of length 1.0) that is normal (perpendicular) to the surface that the polygon or tristrip is representing.

object [<name>]

Begins the definition of an object with optional name. Objects contain a collection of graphics elements such as lines, polygons, triangle strips, and optional parameters.

perspective

The default viewing mode for the graphics scene is orthographic. Perspective defines the default viewing mode to be in perspective, which will allow interactive navigation of the scene. Interactive navigation is only permitted while in perspective mode. Toggling between orthographic and perspective mode can also be accomplished by pressing the ‘v’ key on the keyboard.

polygon

Begins the definition of a polygon and must appear within an object. Vertices, vertex normals, and texture coordinate statements follow the polygon directive.

referenced

Specifies that the current object is to be referenced another entity such as a switch, light, or other object. An object that contains the “referenced” directive is only displayed when the object referencing or instancing it is drawn.

rotate <h> <p> <r>

If specified before an object, this directive defines the heading and pitch rotation values of the entire scene being visualized when viewing mode is set to perspective. If specified within an object, this directive defines the heading, pitch and roll values (in degrees) that the object will be rotated by when displayed.

switch <name> <options>

Defines a switch. See the section entitled “Describing Switches” for more information.

texture <path_to_jpeg_file>

Defines a jpeg image file to be applied to the polygons and triangle strips. This directive must appear within an object.

texture_scale <x> <y>

Scales the texture applied the object

texture_translate <x> <y>

Translates or shifts the texture that is applied to the object.

t <s> <t> **Texture coordinates**

Defines texture coordinates for the current polygon vertex

tristrip

Defines the beginning of a triangle strip. The very next text lines must be vertex, optional normal, and optional texture coordinates (v, n and t statements respectively), that describes the geometry of a strip of connected triangles.

v <x> <y> [z] **Vertex**

Defines a vertex for a polygon trimesh or line. Coordinates may be 2-D or optionally 3-D

Graphics Object

An ‘object’ is a graphics entity that contains one or more lines and/or polygons (triangles, rectangles etc.) It can optionally have a name and may be referenced by other entities such as switches to define their appearance. Objects are drawn in the order they are defined.

The following object could be used to create the background for a dark gray 19-inch by 5.25-inch panel.

```
object front_panel          - begins an object called “front_panel”
color 0.3 0.3 0.3           - defines a color
polygon                     - begins the definition of a polygon
v 0. 0.                     - vertex at x=0 y=0
v 19 0                      - vertex at x=19.0 y=5.25
v 19 5.25                   - vertex at x=19.0 y=5.25
v 0 5.25                    - vertex at x=0 y=5.25
```

Realism

Realistic appearances of computer systems may be achieved by using images or digital photographs taken of parts of an actual system. Images of vintage computer systems are available on the internet, and the rights to use them are in many cases granted for non-commercial purposes. One must be sure not to violate any copyright or usage rules before using acquired images of course.

Graphics objects comprised of polygons may have images mapped to them by using the *texture* directive. A polygon representing the front panel may have an image of an actual front panel mapped to it. An image processing utility such as Gimp or Photoshop may be used to process the image so it suits your needs. Aspects such as image size, aspect ratio, brightness and contrast may be manipulated.

Steps for creating a realistic system

The following describes the steps one can take for creating a realistic front panel.

1) Obtain an Image. Obtain a digital image in jpeg format of the front for the system you wish to simulate. The higher the resolution the better the results will be. If you can’t find a picture that is taken straight on and the system is angled, you can use an image processing program such as Gimp or Photoshop to crop and warp the image so it appears to be as orthogonal as possible. If the image is not already in jpeg format, convert it to jpeg with one of the image programs. This image will be used as the background for the panel and will be discussed later.

2) Obtain Dimensions. Obtain the width and height dimensions of the physical front panel making sure the aspect ratio closely matches the digital image you are to use. We will use these dimensions to describe a polygon that will serve as the background for the panel we are simulating.
.

3) Create Panel Configuration File. Using a text editor, create a configuration file and enter the object definition for the panel as the first object in the file. Shown below is the definition for a panel 19 inches wide and 5.25 inches high (using the polygon directive). Define the color to be white using the color directive (color 1 1 1). The image we acquired in step 1 will be "textured" or mapped onto the polygon and blended with the object's color. The texture directive has the image name that you wish to map onto the polygon. Save the configuration file and you can view it with the fpviewer program located in the *FrontPanel* directory if you obtained and compiled the software.

```
object panel
color 1 1 1
texture some_image.jpg
polygon
v 0. 0.
v 19 0
v 19 5.25
v 0 5.25
```

You may specify as many polygons as you wish. Usually one will suffice, however if you require "holes", you may place several polygons strategically such that they create the surface that will be the panel. The texture will be automatically aligned to fit across all of the defined polygons. If the alignment of the image is not correct when you view the panel, corrections can be made by using the other texture directives. The texture_translate directive allows the image to be shifted in the x and y direction. It slides the image left/right up/down. The example below will slide the image 0.5 units to the right and 0.25 units down.

```
texture_translate 0.5 -0.25
```

The image can also be magnified or shrunk by using the texture_scale directive. The following example will magnify the image by 1.2 units horizontally, and leave the vertical size alone at 1.0.

```
texture_scale 1.2 1.0
```

4) Define Light Placements. Once you are satisfied with the appearance of the front panel using the fpviewer utility, it is time to define the light placement. This entails knowing the coordinates of the lights. Coordinates are easily obtained by using the fpviewer. When the graphics window of a simulation or the fpviewer is in focus (usually by clicking on the window), typing the letter 'c' will toggle a cursor and display its x and y coordinates.. The arrow keys can be used for positioning the cursor. Holding the shift key will cause coarse steps for quick positioning. Position the cursor where a light is to be and note the coordinates. Then enter them into the "pos" (position) option in the light definition within the configuration file. See the section entitled "Describing Lights" for more details. Below is an example of a light named LED_DATA_00 having coordinates 4.97, 3.70.

```
light LED_DATA_00 pos=4.97,3.70
```

5) Define Switch Placements. If you desire functional switches, use the cursor as described for the light placement in step 4 above for identifying the coordinates of the switches. Refer to the section entitled "Defining Switches" for more details.

Describing Lights

Lights are items that may have their state changed by actions in your simulator. Their name, position, size and color are defined in the configuration file. The programming interface is used to bind a light to a specific bit within a variable in your program. Each time the fp_sampleData() function is called, the brightness of the light will be affected by the state of the bit it is bound to. More on this is described in the programming interface section of this manual. The light statement must appear all on one line in the configuration file.

Lights are defined with the 'light' directive which has the following syntax:

```
light <name> [options]

A light must have a name by which it will later be referenced from the
programming interface.

options include:

      color=<r>,<g>,<b>        - overrides the default light color.
                                  r,g,b components are comma-separated.

      group=<group_number>     - optional integer which may be used for
                                 finer control over data sampling.

      object=<object_name>     - specifies the usage of a graphics
                                 object instead of an internal light
                                 representation.

      pos=<x>,<y>              - x and y coordinates describing the
                                 position

      size=<x>,<y>             - x and y size
```

Examples:

```
light LED_DATA_00 pos=4.97,3.70
light LED_DATA_01 pos=5.87,3.70
light LED_FETCH pos=16.35,3.7 group=0
```

Describing Switches

Switches are items that may have their state changed by clicking on them with the mouse. They come in two forms; a simple toggle or on-off type typically used for address and data switches, or a center positioned momentary commonly used for deposit/deposit-next type switches. The switch location and type are defined in the configuration file. The programming interface allows the binding of a switch to a specific bit within a variable in your program. Further, a callback may be added to any switch to define a function or subroutine to be called when the switch is clicked on. More on this is described in the programming interface section of this manual.

Switches are described with the ‘switch’ directive which has the following syntax:

Switch <name> type=<type> operate=<operation> objects=<object>,<object>,<object> pos=x,y,z size=x,y,z

Parameters for a switch are as follows:

type=<type>

Switch type specifies the graphics representation of the switch. For now only ‘object’ is permitted. Other values may be used in future versions of the *FrontPanel*.

type=object

Type object means this switch is drawn by using a graphics object defined within the configuration file. See the ‘object’ directive above. You would have an object to represent each state of the switch (up, down, and center if it is a momentary switch). The “objects” parameter (described below) is used for controlling which object is drawn for the various switch states.

operate=<operation>

The operation defines the behavior of the switch when it is clicked on with a mouse. Available operations are ‘toggle’, and ‘mom_off_mom’ as described below.

operate=toggle

Toggle defines a common on/off style switch which has two positions – up and down.

operate=mom_off_mom

Mom_off_mom defines a switch where it is normally centered and the user may click the switch up or down. The switch will return to the center position upon releasing the mouse button.

objects=object_down,object_up,object_center

The ‘objects’ parameter defines which graphics object to be drawn depending on the current state of the switch. If the switch operation is ‘operate=toggle’, it will have two possible states. When the switch is down, the graphics object whose name appears in the first argument (‘object_down’ in the above example) will be drawn to represent the switch. If the switch is up, the graphics object whose name appears in the second argument (‘object_up’ in the above example) is used. If the switch operation is ‘operate=mom_off_mom’, the object whose name appears in the third argument will be used. The switch statement must appear all on one line in the configuration file.

Example:

```
switch S1 type=object operate=toggle objects=sw_down,sw_up
pos=15.2,2.45,0. size=.40,0.68,1
```

3D Geometric Objects

Beginning with version 2.0, FrontPanel supports 3D geometric objects defined within the configuration file. A 3D replica of an entire system including 3D functioning switches may be created. The FrontPanel software distribution contains an example of a 3D IMSAI system simulation.

The vertices associated with polygons, lines and triangle strips may have a Z component allowing those elements to be positioned anywhere in the 3-dimensional space. When describing a 3D polygon, make certain that the coordinates are ordered according to the right-hand rule. That is, vertices are ordered in a counter-clockwise direction when the polygon is viewed from the front. If material properties are used, polygons must include vertex normals for the lighting to work. Vertex normals are unit (length = 1.0) vectors that are perpendicular to the point on the surface of the object the vertex is representing. Many 3D modeling software packages will export the models in a human-readable format. The Wavefront obj format is one of them. It is then possible to extract the necessary data and reformat it into the form that the frontpanel configuration file requires.

3D Navigation

The default viewing mode is Orthographic projection. Perspective mode can be entered via the ‘perspective’ directive in the configuration file, or by typing the ‘v’ key on the keyboard while the graphics window is in focus. The ‘v’ key will toggle between orthographic and perspective modes. While in perspective mode, 3D navigation is enabled. The following is a list of keyboard and mouse functions that may be used.

Keyboard Functions

V	- toggle between orthographic and perspective viewing modes.
Z, z	- Zoom in/out respectively

Mouse Functions

Left Mouse-button	- If the cursor is over a switch when the mouse button is pressed, the switch will change state. If the cursor is not over a selectable item such as a switch when the button is pressed, the system enters rotation mode. Holding the button down while moving the mouse will cause the scene to rotate.
Shift + Leftmouse	- Pan the view left/right up/down.

The Programming Interface

The programming interface for *FrontPanel* is the mechanism for defining which bits within variables in your program activate lights, when data is sampled, and how it responds to switch selections. The configuration file described above defines the positions of the lights and switches and their names. The programming interface is a collection of functions that allow you to link or bind the lights and switches (using their names) to data bits in your program.

The basic framework for a simulator program source file might be as follows:

Add #include “frontpanel.h” to the top of source files requiring access to the *FrontPanel* interface. The “frontpanel.h” file is located in the include directory within the *FrontPanel* directory if you downloaded and built the software.

Below are the basic steps for integrating *FrontPanel* into your simulation program. The section entitled “Frontpanel API Functions” contains details on all of the functions available to you for interfacing your simulator.

1. Initialize the *FrontPanel* which reads your configuration file.

```
if(!fp_init("my_panel.conf")) error();
```

2. Bind a simulation clock to the *FrontPanel*.

```
fp_bindSimclock(&simclock);
```

3. Bind a runflag to the *FrontPanel*.

```
fp_bindRunFlag(&running);
```

4. Bind lights to specific bits within variables in your program.

```
if(!fp_bindLight16("LED_ADDR_{15-00}",&address, 1))
printf("error binding data lights\n");
```

5. Bind switches to specific bits within variables in your program.

```
fp_bindSwitch32("SW_{00-15}",&switches32,&switches32,1);
```

6. Add callbacks to switches to define functions to be called upon a button click.

```
fp_addSwitchCallback("SW_{00-15}",switch_callback,1);
```

7. Simulation Loop

```
while(!done)
{
   Increment your simulation clock.
          simclock += somevalue;

  Set values to your variables such as address=PC, data=mem[PC]
  Call fp_sampleData();
  Perform the instruction simulation
          …
          …
} /* end simulation loop */
```

Frontpanel API Functions

Control Functions

fp_framerate - framerate control

```
fp_framerate(float fps)
```

Argument:

```
      fps   floating point value representing the rendering rate in
            frames per second.
```

Description:

Controls the rate at which the graphics window is refreshed during simulation. An fps value of 30.0, will result in a frame rate not to exceed 30 frames per second. Note that if the simulator is running on a slow system the frame rate may drop below the specified rate.

fp_ignoreBindErrors - Ignore Binding Errors

```
void    fp_ignoreBindErrors(int val);
```

Argument:

```
      val   1 = ignore, 0 = report errors
```

Description:

By default, if a bindLight or bindSwitch function fails due to the specified name not being found, an error message is printed to stdout. The fp_ignoreBindErrors function can be used to suppress the error messages.
This is useful when the same simulation binary is used with multiple configuration files. An example is the DG Nova systems. The Nova 1200 has both address and data lights whereas the Nova 3 only has one set of lights.

fp_init - initialize frontpanel

```
int fp_init(char *config_filename );
```

Argument: cfg_fname - path to config file.
Return Value: 1 = success, 0 = failure.

Description:

Specifies the configuration file and initializes the *FrontPanel* system. If syntax errors are found while reading the configuration file they will be reported on stderr. The configuration file is read and parsed which defines graphics objects, lights and switches that represent the physical appearance of the computer front panel. Upon successful completion of the configuration file, *FrontPanel* will open up a graphics window containing the visual representation of the panel being emulated.

fp_quit - Quit

```
      void fp_quit(void);
```

Description:

Shuts down *FrontPanel*. The graphics window is closed and the rendering thread is terminated.

Binding Data

The simulation clock, runflag and data values in the application user space need to be 'bound' to the *FrontPanel*. The following programming interface functions accomplish the task of binding data values to the portions of the *FrontPanel* as required.

fp_bindSimclock - Bind simulation clock

```
void fp_bindSimclock( uint64 *addr);
```

Argument: addr – pointer to an unsigned 64-bit integer
Return Value: none

Description:

In order for the lights to appear realistic during simulation, a simulation clock is referred to which is used to compute on/off durations for the signals being minitored. This function is used to provide a memory address pointer to an unsigned 64-bit value that is to be the simulation clock. Each time fp_sampleData() is called, the simclock is sampled along with the bit the light is bound to. These values are used to compute the light intensities while the simulation is running.

fp_bindLight - Bind Light to Data

```
int fp_bindLight32(char *name, uint32 *bits, int bitnum);
int fp_bindLight16(char *name, uint16 *bits, int bitnum);
int fp_bindLight8(char *name, uint8 *bits, int bitnum);
```

Arguments:

```
      Name  - text string describing the name of the light.
              There is a short-hand naming convention described below.
      Bits  - address of the variable which this light will be
              bound to. The data type must reflect 8, 16, or 32 bit
              precision depending on the function being called.

      Bitnum – a signed integer specifying which bit within the data
               item being pointed to by 'bits'. The least significant
               Bit starts at one (1).
```

Return Value: 1 = success, 0 = failure

Description:

The fp_bindLightxx functions bind a light to bits within data values in your simulator. There are versions of bindLight for 8, 16 and 32 bit unsigned integers and you choose the proper one for the type of data the bit being monitored is a part of. The 'name' is a character string that must match the name of a light defined in the configuration file. For cases where multiple bits in a variable are bound to a set of lights (e.g. 16 lights each representing one bit of a 16 bit variable), there is a short-hand method of naming. A numerical range may be supplied between braces ("{" and "}") in the name. For example, if the configuration file had 16 address lights named LED_ADDRESS_00 through LED_ADDRESS_15, and those are to monitor 16 bits in a variable named 'address', the following naming can be used:

```
   fp_bindLight16( "LED_ADDRESS_{00-15}", &address, 1);
```

The above light names will expand to:

```
LED_ADDRESS_00
LED_ADDRESS_01
     …
LED_ADDRESS_15
```

The third argument (bitnum) defines which bit to start at, with the least significant bit being 1. In the above example, LED_ADDRESS_00 will be bound to the least significant bit within the variable 'address' points to. LED_ADDRESS_01 would be bound to the second least significant bit and so on. If the bitnum was -16, LED_ADDRESS_00 would be bound to bit 16, LED_ADDRESS_01 to bit 15 and so on.

Examples:

```
fp_bindLight16( "LED_ADDRESS_{15-00}", &address, 1);
```

fp_bindLight invert   - Bind Light to Data

```
int
fp_bindLight32invert(char *name,uint32 *bits,int bitnum,uint32 mask);

int
fp_bindLight16invert(char *name,uint16 *bits,int bitnum,uint16 mask);

int
fp_bindLight8invert(char *name, uint8 *bits, int bitnum, uint8 mask);
```

```
Arguments:
      Name  - text string describing the name of the light.
              There is a short-hand naming convention described below.

      Bits  - address of the variable which this light will be
              bound to. The data type must reflect 8, 16, or 32 bit
              precision depending on the function being called.

      Bitnum – a signed integer specifying which bit within the data
               item being pointed to by ‘bits’. The least significant
               Bit starts at one (1).
```

Mask - A bit mask. Any bit of a logic one causes the function of the bound light to be inverted. I.e. bits bound to a light which have a logic zero will cause the light to illuminate, and those having a value of one will cause the light to be off.

fp_bindRunFlag - Bind run flag

```
void fp_bindRunFlag(uint8 *addr);
```

Description:

The intensity of a light is only calculated if the value of a variable defined by the runflag is non zero. When the runflag is zero, lights will be drawn at full intensity if the bit it is monitoring is a logic one, and at minimum intensity if the bit is a zero.

fp_bindSwitch - Bind Switch to data

```
int fp_bindSwitch32(char *name, uint32 *down, uint32 *up, int bitnum);
int fp_bindSwitch16(char *name, uint16 *down, uint16 *up, int bitnum);
Int fp_bindSwitch8(char *name, uint8 *down, uint8 *up, int bitnum);
```

```
Arguments:

      name  - text string describing the name of the switch.
              There is a short-hand naming convention described below.

      down  - address of the variable which this switch will be
              bound to, whose bit value will be affected when the
              switch is clicked to the down position. The data type
              must reflect 8, 16, or 32 bit precision depending on the
              function being called.

      up    - address of the variable which this switch will be
              bound to, whose bit value will be affected when the
              switch is clicked to the up position. The data type must
              reflect 8, 16, or 32 bit precision depending on the
              function being called.


      bitnum – a signed integer specifying which bit within the data
               item being pointed to by ‘down’ and ‘up’ pointers. The
               least significant bit starts at one (1).
```

Return Value: 1 = success, 0 = failure

Description:

The fp_bindSwitch*xx* functions bind a switch to bits within data values in your simulator. There are versions of bindSwitch for 8, 16 and 32 bit unsigned integers and you choose the proper one for the type of data the bit being controlled is a part of. The 'name' is a character string that must match the name of a switch defined in the configuration file. For cases where multiple bits in a variable are bound to a set of switches (e.g. 16 switches each representing one bit of a 16 bit variable), there is a short-hand method for naming. A numerical range may be supplied between braces ({ and }) in the name. For example, if the configuration file had 16 address switches named SW_ADDRESS_00 through SW_ADDRESS_15, and those are to control 16 bits in a variable named 'address', the following naming can be used:

```
fp_BindSwitch16("SW_ADDRESS_{00-15}",&address,&address,1);
```

The above switch names will expand to:

```
SW_ADDRESS_00
SW_ADDRESS_01
     …
SW_ADDRESS_15
```

The *up* and *down* arguments are address pointers to the data to be affected by switch state changes. For toggle switches where there is only two positions (up and down), the same address should be specified for both the up and down arguments. For momentary switches, the up and down addresses should be different and point to the variables within the simulator that are to be controlled by the switch.

The *bitnum* argument specifies which bit to start with within the variable being pointed to by the *up* and *down* arguments. The least significant bit specified with a value of 1. In the above example, SW_ADDRESS_00 will be bound to the least significant bit within the variable that *address* points to. SW_ADDRESS_01 would be bound to the second least significant bit and so on. If the *bitnum* was -16, SW_ADDRESS_00 would be bound to bit 16, SW_ADDRESS_01 to bit 15 and so on.

Data Sampling

The data sampling functions provide a means for a simulation program to tell *FrontPanel* when data values are to be read and used for light intensity calculations, or when switch positions shall be updated to reflect the current values of a variable within the simulator.

fp_sampleData - Sample Data

```
void  fp_sampleData(void);
```

Description:

This function causes the simulation clock value, and all data values which are bound to lights to be read.

fp_sampleDataWarp - Sample Data with Clock Warp

```
void  fp_sampleDataWarp(int clockwarp);
```

Argument:

clockwarp	Signed integer which is added to the current simulation clock.

Description:

This function is used for retarding or advancing the simulation clock before any data are sampled. The value of the simulation clock within the simulator is not affected.

fp_sampleLightGroup   - Sample Light Group

```
void  fp_sampleLightGroup(int groupnum, int clocwarp);
```

Arguments:

groupnum	integer specifying the group number of the lights to be sampled.
clockwarp	a signed integer whose value is added to the current value of the simulation clock.

Description:

Certain lights such as those for the purpose of displaying CPU status signals may need to have their data sampled more frequently than others. When lights are defined in the configuration file, they can optionally be given a group number. The sampleLightGroup function allows for more efficient operation by only processing the data of a desired subset of the lights.

fp_sampleSwitches – Sample Switches

```
void fp_sampleSwitches(void);
```

Description:

Switches are typically used to affect data values within the simulator program when the user clicks on them. The fp_sampleSwitches function causes the state of all switches that are bound to data values to have their states reflect those values.

Callbacks

Front panel switches are used to affect the bits of variables to which they are bound. A callback may be added to a switch which will result in a function of your choosing to be called when the switch is clicked. The callback is initiated after the data value the switch is bound to has been updated to reflect the current switch state.

Note: The *FrontPanel* switch mechanism runs in a thread independent from your simulation program, and thus the callback function will also be running in the FrontPanel simulation thread. Your callback function should be used as a notification mechanism only and not for main simulation processing such as executing simulated instructions.

fp_addSwitchCallback - Add Switch Callback

```
int   fp_addSwitchCallback(char *name, void (*cbfunc)(int state, int
val), int userval);
```

Arguments:

name	Character string containing the name of the switch that is to have the callback be added to. The name must match a switch that was defined in the configuration file.
cbfunc	The function in your program which will be called when the switch is activated by a mouse click.
userval	An integer value that will be passed as an argument to the callback function.

The function in your simulation program to be called shall have the following calling sequence:

```
      void myCallbackFunc ( int state, int userval )
      {
            /* your code */
      }
```

When a switch is clicked on and a callback function is invoked, it will be passed the following arguments:

state	An integer containing the current state of the switch and will be one of FP_SW_DOWN, FP_SW_UP, or FP_SW_CENTER as defined in *frontpanel.h*.

Example:

```
      fp_addSwitchCallback(“Deposit_SW”, deposit_clicked, 1);
```

User function to be called when the switch named “Depost SW” is activated.

```
void deposit_clicked(int state, int val)
{
      switch(state)
      {      case FP_SWITCH_UP:
                   …
                   break;
            case FP_SWITCH_DOWN:
                   …
                   break;
            default:
                   break;
      }
}
```

fp_addQuitCallback   - Add Quit Callback

```
void  fp_addQuitCallback(void (*cbfunc)(void));
```

Description:

This function provides a method of having a function within your simulator be called when the user closes the graphics window. When a quit callback is defined, a window closure event will not result in the window being closed. The defined callback will be called and it is then up to the application to call fp_quit() to terminate the graphics window.

Programming Interface Function List Summary

`fp_addSwitchCallback`	Add Switch Callback
`fp_addQuitCallback`	Add quit Callback
`fp_bindLight8`	Bind Light to Data
`fp_bindLight16`	
`fp_bindLight32`	
`fp_bindLight8invert`	Bind Light to Data, option to invert logic
`fp_bindLight16invert`	
`fp_bindLight32invert`	
`fp_bindRunFlag`	Bind run flag
`fp_bindSimclock`	Bind simulation clock
`fp_bindSwitch8`	Bind Switch to data
`fp_bindSwitch16`	
`fp_bindSwitch32`	
`fp_framerate`	framerate control
`fp_ignoreBindErrors`	Ignore Binding Errors
`fp_init`	initialize frontpanel
`fp_quit`	Quit
`fp_sampleData`	Sample Data
`fp_sampleDataWarp`	Sample Data with Clock Warp
`fp_sampleLightGroup`	Sample Light Group
`fp_sampleSwitches`	Sample Switches

References

[1] simh – The Computer History Simulation Project (http://simh.trailing-edge.com/)