Visualization for Platforms, Modules, Peripherals and Intercept Libraries

This document shows how the Imperas simulators can visualize different aspects of components in a virtual platform.

Imperas Software Limited

Imperas Buildings, North Weston,
Thame, Oxfordshire, OX9 2HA, UK
docs@imperas.com

Author:	Imperas Software Limited
Version:	1.3
Filename:	Visualization_for_Platforms_Modules_Peripherals_and_Intercept_Libraries.doc
Last Saved:	Monday, 13 January 2020
Keywords:	Visualization

Copyright Notice

Right to Copy Documentation

Destination Control Statement

Disclaimer

Table of Contents

1 Preface 5
1.1 Notation 5
1.2 Related Documentation 5
1.3 Glossary / Terminology 5
2 Introduction 6
2.1 Prerequisites 6
2.2 Obtaining & installing necessary files 6
2.3 Compiling Examples described in this Document 7
2.4 Shared Objects and executables 7
3 An Introduction to Virtual Platform Visualization 8
3.1 How it works 8
3.2 Connection overview 9
3.2.1 peripheral, module and harness 9
3.2.2 Binary intercept library 10
3.3 Turning visualization on 10
3.3.1 Command line 10
3.3.2 Environment variable 10
3.4 Connecting a browser to a components visualization HTTP port 11
3.5 Many component visualization images 11
4 A first example: Examples/PlatformVisualization/moduleNet 12
4.1 The structure in index.html 12
4.2 Compiling the C and running 13
4.3 Code in the module 14
5 Adding Visualization to Virtual Platform components 16
5.1 Creating visualization in peripheral model 16
5.2 Creating visualization in a harness 16
5.3 Creating visualization in a module 16
5.4 Creating visualization in an binary intercept library 16
5.5 Creating visualization for a processor model 16
6 Imperas Visualization using Binary Intercept Extension Libraries 17
6.1 Introduction 17
6.2 Using existing plugins at run time 17
7 Visualization using just HTML and Javascript 18
7.1 Radar Display Example 18
7.1.1 Overview 18
7.1.2 Running the example 19
8 Examples 20
9 Reference section 21
9.1 Overview of Visualization infrastructure 21
9.2 Joining the dots... (how C code values are displayed as images) 23
9.2.1 Imperas visualization 'classes' 23
9.3 Component index.html file 24
9.4 Peripheral C code 25

9.4.1 iGen Definition 26
9.4.2 Constructor 26
9.4.3 BHM_HTTP_GET_FN get callback function 27
9.4.4 the redraw function 27
9.4.5 BHM_HTTP_POST_FN post callback function 28
9.5 Module and Harness C code 28
9.5.1 Pre-Simulation initialization 29
9.5.2 get callback 29
9.5.3 post callback 30
9.6 Intercept Library C code 30
9.6.1 Constructor 30
9.6.2 get callback 31
9.6.3 post callback 32
9.7 Top Level visualization.html file 32
9.8 C API functions for peripherals 33
9.9 C API functions for modules and harnesses 34
9.10 C API functions for intercept libraries 34
9.11 Using iGen to help assist in creating visualization 35
9.11.1 iGen can create templates for peripherals 35
9.11.2 iGen can create templates for modules 36
9.12 Imperas visualization classes: class, images, values 36
9.12.1 ovpled 37
9.12.2 ovpbuttonled 37
9.12.3 ovplcd7seg 37
9.12.4 ovplcd2x16 38
9.12.5 ovpswitchtoggle 38
9.12.6 ovpswitchdip 38
9.12.7 ovpbar 38
9.12.8 ovpvertbar 39
9.12.9 ovppower 39
9.12.10 ovpreset 39
9.12.11 ovpgauge 40
9.12.12 ovphistogram 40
9.13 Adding your own visualization classes and items 40
10 Common Problems 43
10.1 Fixed HTTP port numbers 43
10.2 No index.html for component 43
10.3 Browser can not connect to HTTP port 43
10.4 Browser pop up: runtime.js sendCommand has class name not recognized 44
10.5 Browser pop up: runtime.js sendCommand error. Your C code function has an element which is undefined in the html 44
10.6 Tracing HTTP port data 44
10.7 Other errors 44
10.8 How to run regression test models with visualization 44

1 Preface

The Imperas simulators can open an HTTP port and display different aspects of various components in an HTML web browser such as Chrome, Firefox, Safari..

This document describes the usage of the Imperas visualization and walks through several examples of platforms, modules, peripherals and intercept libraries.

This document introduces the infrastructure that provides the visualization and what is required in the models to enable it.

1.1 Notation

`code`	Text representing code, a command or output from *iGen* or other program.
keyword	A word with special meaning.

1.2 Related Documentation

This document assumes you have already created the basics of the model that you want to add visualization to and thus you need to be aware of the relevant documents for that task.

For example to model a peripheral you need to be aware of the iGen and BHM documents. For the intercept libraries you will need to be aware of the VMI documents etc.

1.3 Glossary / Terminology

OCL API – is a C API that provides common functionality between platforms, modules and intercept libraries. It can be used with OP, and also with VMI.

BHM API – is a C API that provides functionality in peripheral models.

HTTP - standard protocol which is the foundation of data communication for the World Wide Web.

HTTP port - in the context of Imperas simulation - is a socket that the simulator opens and writes/reads HTTP messages (get, post) which you can connect to a browser in which to see the Imperas model visualization.

Browser, or HTML Browser - Imperas supports the standard internet web HTML browsers: Chrome, Firefox, Safari.

HTML, Javascript, cascading style sheets (CSS) - normal www terms.

2 Introduction

The Imperas simulators can open HTTP ports to allow an web browser to communicate with the simulator to visualize different aspects of the simulatable models in the virtual platform.

There are several components involved in the generation of the visualization:

- the platform, module, peripheral model, or intercept library
    - calls an API to open an HTTP port
    - responds to a 'get' callback on the HTTP port and calls an API to construct messages for the simulator to write over the HTTP port
    - responds to a 'post' callback on the HTTP port to read browser generated values
- the simulator
    - processes requests from the browser, loads files for the visualization items and sends to the browser over an HTTP port
- HTML browser
    - uses .html files to organize the visualization
    - runs Javascript (.js scripts) to select images to display to represent data via the HTTP port
    - formats using cascading style sheets (.css)
    - posts button clicks (e.g. mouse events) back to the simulator via the HTTP port

2.1 Prerequisites

Since models and intercept libraries for use with Imperas and OVP tools are written in C, an important prerequisite is that you must be proficient in the C language.

You need to have experience with the component type and API(s) that you want to add visualization to. For example to add an image to the visualization of a module, you need to be aware of how to modify the C code of the module.

Some knowledge of HTML code is also required as you will need to edit the HTML files that provide the containers for the visualization.

It is assumed you have experience with running the Imperas `harness.exe` and `iss.exe` programs.

You can use the provided visualizations without any prior knowledge related to Imperas visualization.

2.2 Obtaining & installing necessary files

Visualization is a standard part of the OVP OVPsim, Imperas DEV and SDK packages. So it is assumed you have downloaded one of these from the OVPworld.org or Imperas.com websites and have installed it on the host machine.

2.3 Compiling Examples described in this Document

The examples use models, intercept libraries and toolchains, available to download from the www.OVPworld.org website or as part of an Imperas installation.

The compilation of the examples makes use of Makefiles and GNU make. The instructions indicate the use of the command *make* on Linux systems and MinGW *mingw32-make* command on Windows systems.

The Makefiles referred to in this document are written for GNU make. Standard Makefiles supplied by Imperas support compilation and linking using GNU tools on both Windows and Linux.

Example scripts will be referred to, for example, as *example.sh.* The shell (extension *sh)* script files may be used on Linux and in Windows MSYS shells. The batch (extension *bat*) files may be used in Windows Explorer or in a Windows command shell.

2.4 Shared Objects and executables

The shared objects referred to in this document are either Linux shared objects, with suffix *.so,* or Windows dynamic link libraries, with suffix *.dll*.

The executables referred to in this document are either Linux or Windows programs and both have the suffix *.exe.*

3 An Introduction to Virtual Platform Visualization

3.1 How it works

When running a simulation it is often desirable to see a representation of the values of certain variables. The Imperas Platform Visualization provides this. Also you might want to interactively input some information to a running simulation - for example toggle a switch, cause an interrupt, or hold a processor in reset. The Imperas Platform Visualization also provides a simple input mechanism.

In the C code of a peripheral, module, harness or binary intercept library, calls can be made into API functions that open HTTP ports and send/receive formatted data. You can then connect up a standard web HTML browser to access the visualization.

Each model instance can open one HTTP port.

Your web browser will need to connect to this port, and if there are several model instances and thus several ports, then you need to either connect a different browser page instance to each port, or use HTML frames to have one page with each port connected to its own frame.

If you only have one HTTP port being used in the visualization you can connect the browser directly to the port with no need to create any HTML with frames, etc.

The different provided examples illustrate these different approaches.

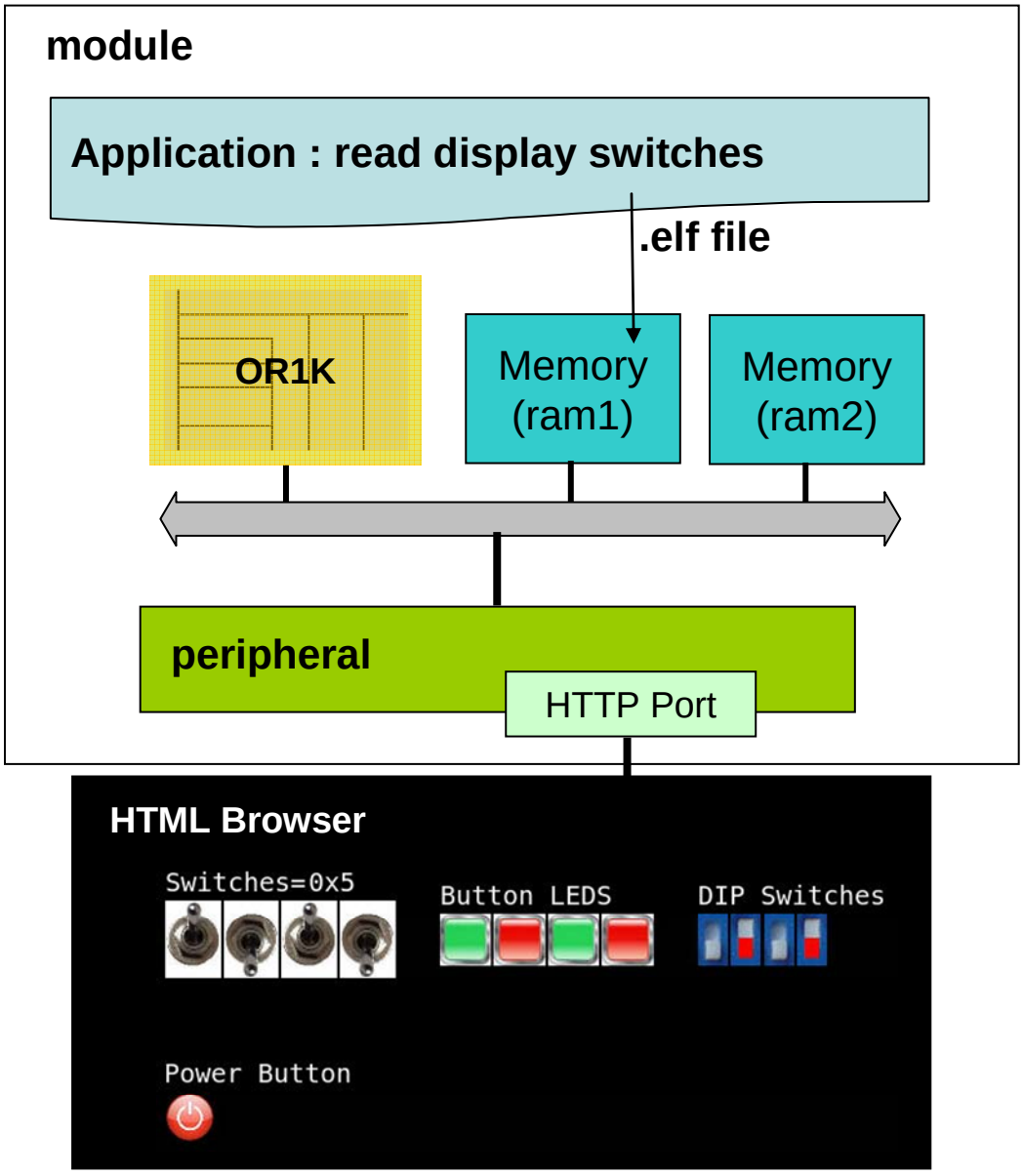
The basic operational mechanism is that when the simulator starts and visualization is being used, the simulator monitors requests on its HTTP ports. If no browser connects, then there is no HTTP activity. When a browser connects to the HTTP port, the simulator sends HTML pages to the browser and then at the chosen refresh interval the HTML page requests data so that it can redraw the page.

Thus the browser is sampling the data that is being written from the simulator's models.

3.2 Connection overview

3.2.1 peripheral, module and harness

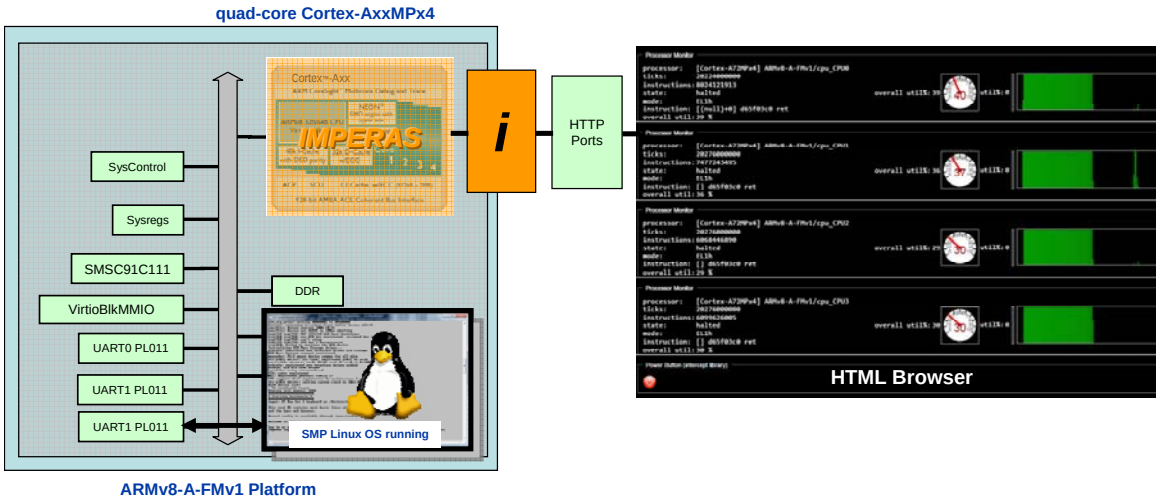
Below is a picture of a visualization of one peripheral. This is provided as an example in `Examples/Models/Peripherals/visualization/switches`.



A module or harness would be connected in a similar way and can provide visualization for any values it has access to. An example of a harness visualization is provided in: `Examples/PlatformVisualization/processorReset`.

3.2.2 Binary intercept library

There is a processor monitor intercept library provided with the Imperas products that can be used with single, SMP, or AMP processors to monitor the status of the processor. This can be seen in the provided demonstration: `Demo/Platforms/Linux_ARMv8-A-FMv1/visualization` and is shown below:



3.3 Turning visualization on

If you run the simulation, by default, no visualization will be enabled.

If a component has visualization, then it must be enabled with either a command line option or with an environment variable.

If components have visualization and it is either not enabled, or enabled and not connected to a browser, then there is no overhead in the simulation run time.

When the visualization is enabled and connected to a browser, there is simulation overhead that is dependent upon the amount of data transferred (though this is usually minimal).

3.3.1 Command line

The command line option to use is

```
    --httpvis
```

This enables all components' HTTP ports.

3.3.2 Environment variable

The environment variable to use is

```
    IMPERAS_HTTP=1
```

This enables all components' HTTP ports.

3.4 Connecting a browser to a components visualization HTTP port

In the component's C code that opens an HTTP port there will be a default port number. This is either provided explicitly by the user, or built into the simulator code. You can use the simulator command line argument

```
    --showoverrides
```

to see the name of the parameter, and its default value. You can then use the --override command to set it at run time. You can, of course, also set a peripheral or module's parameters during platform construction.

To connect a browser to an HTTP port on the same computer, use the url:

```
    http://localhost:8000
```

And use the required port number, for example port 8000 shown above.

You can open different browsers or browser tabs to different HTTP ports in your platform.

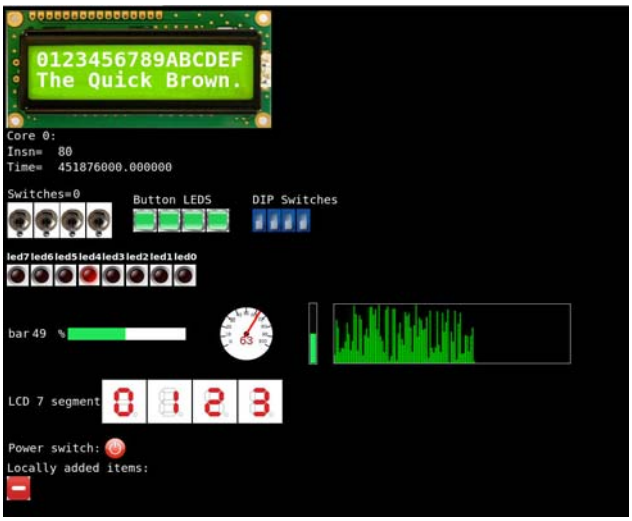
If your simulation is on another host computer (not your local machine), then use the IP address or hostname in the url, for example *machine* in the following:

```
    http://machine:8002
```

If you want to see several components HTTP ports (visualizations) in one browser window, see the section (9.7) describing the use of a top level `visualization.html` file.

3.5 Many component visualization images

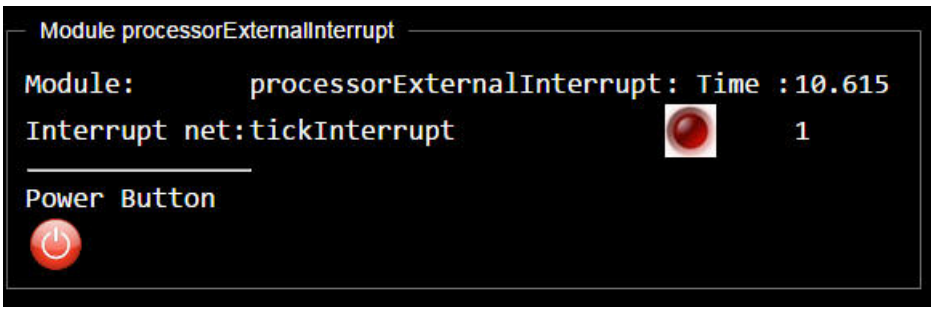
There are many different items provided as part of the simulator that can be displayed. Below is a simple illustration of some of those available:



4 A first example: Examples/PlatformVisualization/moduleNet

This section introduces an example of adding visualization to a module. For details and reference please see later sections in this document.

In this example we are going to monitor a net and add a button so we can terminate the simulation from the connected browser. The visualization in the browser will look like:



4.1 The structure in index.html

In the `module` directory is also the `httpvis` directory that includes the `index.html` file which defines the structure of what will be displayed.

The index.html file is listed here. It includes:

the header calling up the .js and .css files:

```
<html lang="en">
<link rel="icon" type="image/icon" href="/imperas.ico">

<head>
    <title>Imperas Simulation</title>
    <meta   charset="utf-8"/>
    <link   rel="stylesheet"        href="normal.css"   type="text/css"/>
    <script language="JavaScript"   src="runtime.js"        ></script>
</head>
```

(See section 9.1 for info on the normal.css and runtime.js files)

The periodic refresh:

```
<body onload="startRefresh('/', 500);">
```

A block (fieldset) including a table with rows and data to structure the page:

```
    <fieldset><legend>Module processorExternalInterrupt</legend>
        <table>
```

A row with text:

```
            <tr>
                <td class='ovplabel'>Module:</td>
                <td class='ovptext' id="moduleInstName0" ></td>
```

```
                <td class='ovplabel'>: Time :</td>
                <td class='ovptext' id="moduleSimTime0" ></td>
            </tr>
```

Another row with text:

```
            <tr>
                <td class='ovplabel'>Interrupt net:</td>
                <td class='ovptext' id="netName0" ></td>
```

And the LED placeholder (id=led0):

```
                <td class='ovpled' id="led0" ></td>
```

And the value placeholder (id=netValue0):

```
                <td class='ovptext' id="netValue0" ></td>
            </tr>
            <tr><td><hr></td></tr>
```

And the power button placeholder (id=power0):

```
            <tr><td class='ovplabel'>Power Button </td></tr>
            <tr><td><div class='ovppower' id="power0"
onmousedown="ovpswitchclicked(event)" title='click to quit
simulation'>SIMULATION NOT RUNNING</div></td></tr>
        </table>
    </fieldset>

<div class='console' id='console'></div>
</body>
</html>
```

4.2 Compiling the C and running

We will need the or1k.toolchain and OVPpse.toolchain packages installed.

First copy the example to a local directory:

```
> cp /Examples/PlatformVisualization/moduleNet .
> cd moduleNet
> ls
application example.sh module peripheral
```

And look at the `example.sh` script which will compile up the application, module, and peripheral:

```
CROSS=OR1K
make -C application  CROSS=${CROSS}
make -C module       NOVLNV=1
make -C peripheral   NOVLNV=1
```

Then there is code to ask the user if they want to start a browser to see the visualization, and then it runs the module and application using `harness.exe`:

```
harness.exe --modulefile module \
    --objfilenoentry application/asmtest.OR1K.elf \
    --override processorExternalInterrupt/timerPeripheral/diagnosticlevel=3 \
    --override processorExternalInterrupt/httpvisportnum=8000 \
    --verbose --wallclock --output imperas.log \
    --httpvis \
```

```
    $*
```

Note the enabling of the HTTP port with `--httpvis`, and the override of the module's `httpvisportnum`.

The application is written in assembler for the OR1K and after resetting the processor and initializing a peripheral timer, it loops and responds to interrupts. The timer interrupts the processor using a net, `tickInterrupt`. It is `tickInterrupt` in the module that we are going to monitor.

4.3 Code in the module

If we look at the end of the module/module.op.tcl there is the declaration of the parameter for the port number:

```
    ihwaddformalparameter -name httpvisportnum -type Uns32 -defaultValue 8000
```

The code to open and write to the HTTP port is in `module/module.c`

We need to include the headers:

```
#include "op/op.h"
#include "ocl/oclhttp.h"
```

and declare static storage to be used by the callbacks:

```
typedef struct optModuleObjectS {
    // insert module persistent data here
    optModuleP  mi;
    optNetP     intNet;
    Uns32       intNetValue;
    const char  *intNetName;
} optModuleObject;
```

In the pre simulate phase we need to open the HTTP port and declare our HTTP get and post callbacks and allow the port number to be set by the module parameter:

```
static OP_PRE_SIMULATE_FN(modulePreSimulate) {
    Uns32 httpvisportnum  = opObjectParamUns32Value (mi,"httpvisportnum",0);
    octHTTPMethods m = { .get=get, .post=post, .userData=object };
    opModuleHTTPOpen(mi, &m, httpvisportnum, "httpvis");
    object->mi = mi;
    monitorNets (object);
}
```

The local function, `monitorNets` sets up the monitors on the net in the module:

```
static void monitorNets(optModuleObjectP object) {
    optNetP net = opObjectByName (object->mi, "tickInterrupt", OP_NET_EN).Net;
    if (!net) {
        opMessage ("F", "NNF", "monitorNets (net not found)");
    } else {
        opPrintf ("monitorNets(%s)\n", opObjectHierName(net));
        opNetWriteMonitorAdd(net, netCallback, object);
        object->intNet = net;
        object->intNetName = opObjectName(net);
    }
}
```

and saves in the static object structure a handle to the net and its name.

The get callback is called when the browser needs updating and the callback reads the appropriate data and sends it to the HTTP port.

```
static OCL_HTTP_GET_FN(get) {
    optModuleObjectP object = userData;
    oclHTTPElementOpen(ch, "ovpelement");

    oclHTTPKeyPrintf(ch, "moduleInstName0", opObjectName(object->mi));
    oclHTTPKeyPrintf(ch, "moduleSimTime0","%g",
    (double)opModuleCurrentTime(opObjectRootModule(object->intNet)) );
    oclHTTPKeyPrintf(ch, "netName0", object->intNetName);
    oclHTTPKeyPrintf(ch, "netValue0", "%d", object->intNetValue);
    oclHTTPKeyPrintf(ch, "led0", "%d", object->intNetValue);

    oclHTTPKeyPrintf(ch, "power0", "0");
    oclHTTPElementClose(ch, "ovpelement");
    oclHTTPSend(ch);
}
```

We are also getting simulation time as well as object variable values.

The `oclHTTPKeyPrintf` sends a value associated with an item's instance name, as specified in the browser's HTML file which may be found in `module/httpvis/index.html`.

The post callback is called when a button/switch is clicked in the browser:

```
static OCL_HTTP_POST_FN(post) {
    optModuleObjectP object = userData;
    if (strstr(body, "power0=clicked")) {
        opMessage("I", PREFIX "_SW", "Power Switch pushed - terminating
simulation.");
        opModuleFinish(object->mi, 0);
    }
}
```

It decodes what was pushed and in this case will terminate the simulation.

5 Adding Visualization to Virtual Platform components

Visualization can be added to all the different Imperas/OVP component types. Below is a list of the available examples. Each one shows the usage of one or more visualization items.

Have a look at the reference section (9) below and then browse the examples.

5.1 Creating visualization in peripheral model

Examples/Models/Peripherals/visualization/power_button
Examples/Models/Peripherals/visualization/switch_led
Examples/Models/Peripherals/visualization/switches
Examples/Models/Peripherals/visualization/lcd_7segment
Examples/Models/Peripherals/visualization/lcd_2x16
Examples/Models/Peripherals/visualization/bars
Examples/Models/Peripherals/visualization/dial
Examples/Models/Peripherals/visualization/time_histogram
Examples/Models/Peripherals/visualization/local_item

5.2 Creating visualization in a harness

Examples/PlatformVisualization/processorReset

5.3 Creating visualization in a module

Examples/PlatformVisualization/moduleNet

5.4 Creating visualization in an binary intercept library

Examples/BinaryInterception/Visualization/1.powerButton
Examples/BinaryInterception/Visualization/2.monitorProcessor

5.5 Creating visualization for a processor model

Examples/Models/Processor/Visualization/monitorProcessorSMP
(Actually this is an example of how to use an intercept library which is the best approach as opposed to modifying the source of the model itself.)

6 Imperas Visualization using Binary Intercept Extension Libraries

6.1 Introduction

Imperas professional products provide binary interception capabilities. This can be used to load additional visualization features onto components in a hardware definition without having to make any change to that definition.

See binary interception user guide for the full set of feature available; this document deals only with the usage for visualization.

6.2 Using existing plugins at run time

In the Imperas DEV and SDK packages are two binary intercept libraries that can be used by adding them using the command line. There is no need to modify harnesses or platforms. They can be found in ImperasLib/imperas.com/intercept.

powerButtonHttpvis - this adds a power button to a processor
processorMonitorHttpvis - this adds processor monitoring to a processor
Note: you need to add one monitor to each processor of an SMP core.

These are used in the same way as any other binary intercept library.

For example (from `/Demo/Platforms/Linux_ARMv8-A-FMv1/visualization`):

```
harness.exe ^
    --extlib mod0/cpu_CPU0/pm=processorMonitorHttpvis ^
    --override mod0/cpu_CPU0/pm/httpvisportnum=8000 ^
    --extlib mod0/cpu_CPU0/pb=powerButtonHttpvis ^
    --override mod0/cpu_CPU0/pb/httpvisportnum=8030 ^
    ...
```

7 Visualization using just HTML and Javascript

Most of the examples in the this document make use of the Imperas provided Javascript, OVP class items and .jpg images.

You can create your own visualization, directly augmenting or replacing the Imperas provided infrastructure.

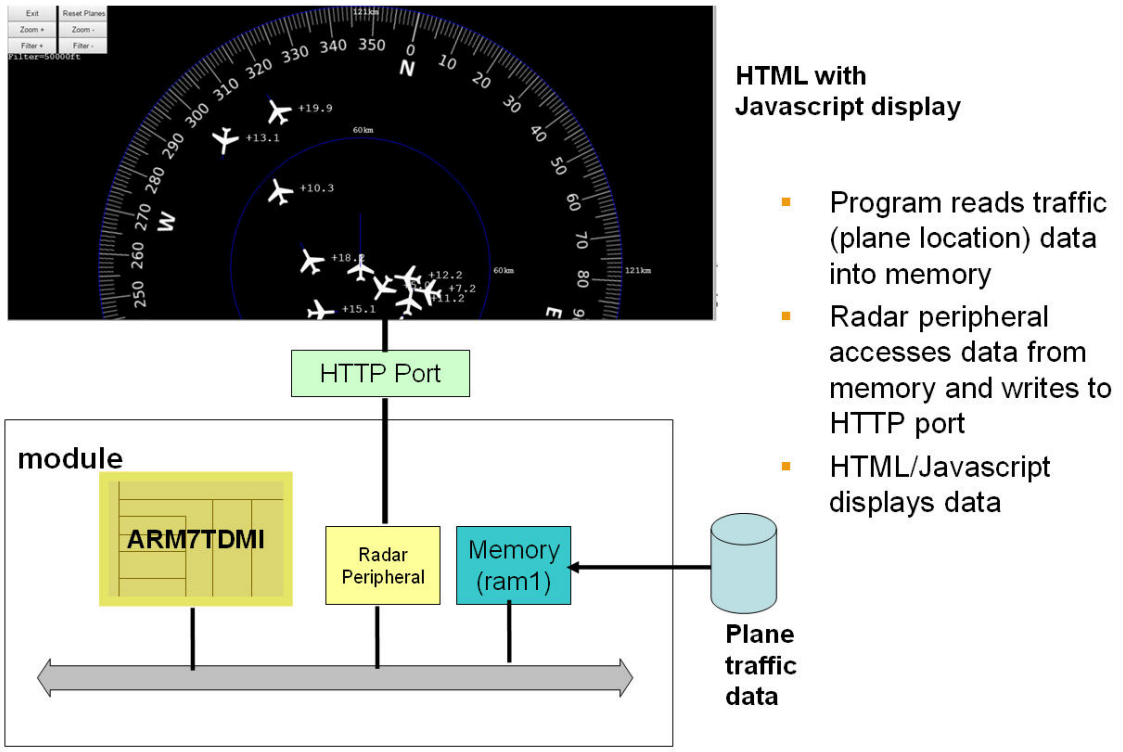
The example below - the Radar display example, does just this - it makes use of the HTTP port and the writing of the data from the peripheral C code, but all the display and interactivity is provided directly in the HTML/Javascript.

7.1 Radar Display Example

7.1.1 Overview

This example is available:

Examples/Models/Peripherals/visualization/radar



The example is an ARM7TDMI processor that has a memory and a peripheral on its bus.

The application program (**`application/application.c`**) sits in a loop with a delay loading data from the plane traffic data file into an area of shared memory. This is representing the loading of data from external radio sensors.

The Javascript (in file **peripheral/httpvis/index.html**) running in the browser has a redraw function which is called every 1,000 mSecs which makes an HTTP request to the peripheral for the data.

The peripheral user code (in **peripheral/radar.user.c**) responds to the **BHM_HTTP_GET_FN(peripheralHTTPGet)** request and gets the data from the processor's shared memory (i.e. the traffic plane positional data), and sends it to the port (using the **bhmHTTPSend** function).

The Javascript in the browser processes the data (in the **get_traffic** function called in the **redraw** function) and draws the plane and other objects onto the canvas.

The Javascript responds to the local zooming and filtering buttons within its code, and only posts the reset and exit button presses back to the peripheral code

7.1.2 Running the example

As in the other examples there is a shell script (**example.sh/.bat**) to compile and run the example. It compiles the application, then uses iGen to generate the module code, compiles the module code, and then compiles the code for the peripheral. It then prompts if you want to see the visualization and runs the simulation.

The example uses the **harness.exe** to run the program:

```
harness.exe \
    --verbose --output imperas.log \
    --modulefile module/model.${IMPERAS_SHRSUF} \
    --program application/application.${CROSS}.elf \
    --override simpleMonitor/radar/diagnosticlevel=1 \
    \
    --httpvis \
    --override simpleMonitor/radar/httpvisportnum=8000 \
    $*
```

It specifies the port number and also turns on the built-in peripheral diagnostic tracing.

8 Examples

This is a list of current examples that show how to use the Imperas visualization. They are self contained and have an example.bat/.sh script that can be run to compile the platforms, models, and applications and run them while launching a browser to allow you to easily see what is available.

There is usually a .jpg that shows a block diagram of the simulated platform and what the visualization looks like.

Demo/Platforms/Linux_ARMv8-A-FMv1/visualization
Examples/PlatformVisualization/processorReset
Examples/PlatformVisualization/moduleNet
Examples/Models/Processor/Visualization/monitorProcessorSMP
Examples/BinaryInterception/Visualization/1.powerButton
Examples/BinaryInterception/Visualization/2.monitorProcessor
Examples/Models/Peripherals/visualization/bars
Examples/Models/Peripherals/visualization/dial
Examples/Models/Peripherals/visualization/lcd_2x16
Examples/Models/Peripherals/visualization/lcd_7segment
Examples/Models/Peripherals/visualization/local_item
Examples/Models/Peripherals/visualization/power_button
Examples/Models/Peripherals/visualization/switches
Examples/Models/Peripherals/visualization/switch_led
Examples/Models/Peripherals/visualization/time_histogram

9 Reference section

9.1 Overview of Visualization infrastructure

The Imperas HTTP visualization uses the HTML5 standards of HTML, CSS, Javascript and JSON. To create peripheral, module, platform or intercept visualization you will need to know the basics of HTML and CSS. You should not need to know Javascript or JSON.

There are several files that make up the infrastructure used by the Imperas HTTP visualization.

These are in three layers/levels:

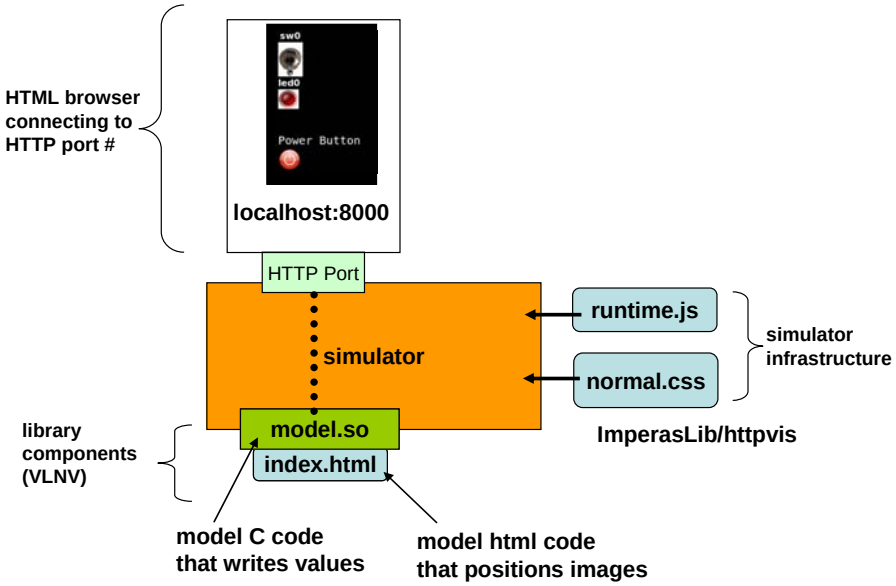
1) At the top is the HTML browser displaying the top level platform/harness file
<runDir>/visualization.html file
This file is not necessary for single component visualization. For displaying multiple components, HTML frames are used to position the different component displays in the browser screen.

2) In the middle is the simulator visualization internals
ImperasLib/httpvis/{runtime.js, normal.css, *.jpg, *.png}
These are files provided by Imperas as part of the simulator infrastructure. They are the image files that will be displayed (*.jpg, *.png), the cascading style sheet (normal.css) that formats the images and text, and the Javascript file (runtime,js) that takes the data written by the models to the HTTP port and displays the appropriate images.

3) The lowest layer is the individual component visualization.
<modelDir>/httpvis/index.html
<modelDir>/user.c
This comprises two parts, first the HTML file that defines which items are to be displayed and where they are to be located on the screen/frame. The second is the C code in the component (e.g. the peripheral) that writes the values to the HTTP port.

To display a single component the browser is connected to the HTTP port opened by that single component as shown here:

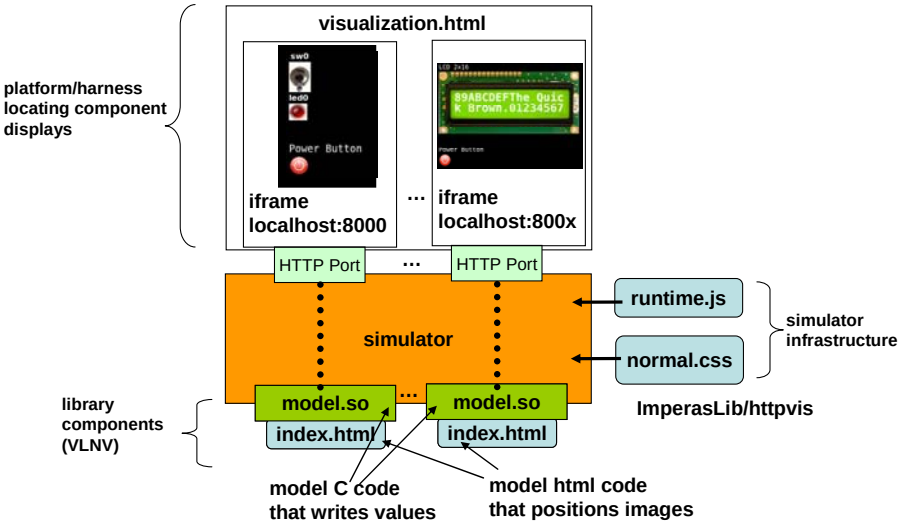
Single component visualization



If you have several components that are opening HTTP ports, you could use several HTML browser windows and connect to each HTTP in a different browser window.

Alternately, you can use the HTML frame concept and using one HTML file (by convention called visualization.html) in one browser window, you can connect to multiple components/HTTP ports, as shown here:

Multiple component visualization



9.2 Joining the dots... (how C code values are displayed as images)

The way the visualization works is very simple and straightforward.

The C code for a component opens an HTTP port.

The user starts a web browser and connects to that HTTP port.

The simulator sends an HTML file which defines items to be displayed.

During simulation the C code for the component will update its internal current values/variables based on normal simulation operation/updating (e.g. writes to the component etc.).

The HTML file includes a periodic timeout that requests data (using 'get's) to be displayed from the HTTP port. Thus the browser is periodically requesting data for it to display.

The C code in the component responds to these periodic requests for data from the HTTP port and sends current values of the items to be displayed. (The browser is thus sampling (at its periodic refresh rate) the current values that the component is maintaining.)

Also some of the component HTML pages have buttons/switches and other inputs that when clicked in the browser send HTTP commands ('post's) to the C code in the component allowing it to respond and update values in the component.

9.2.1 Imperas visualization 'classes'

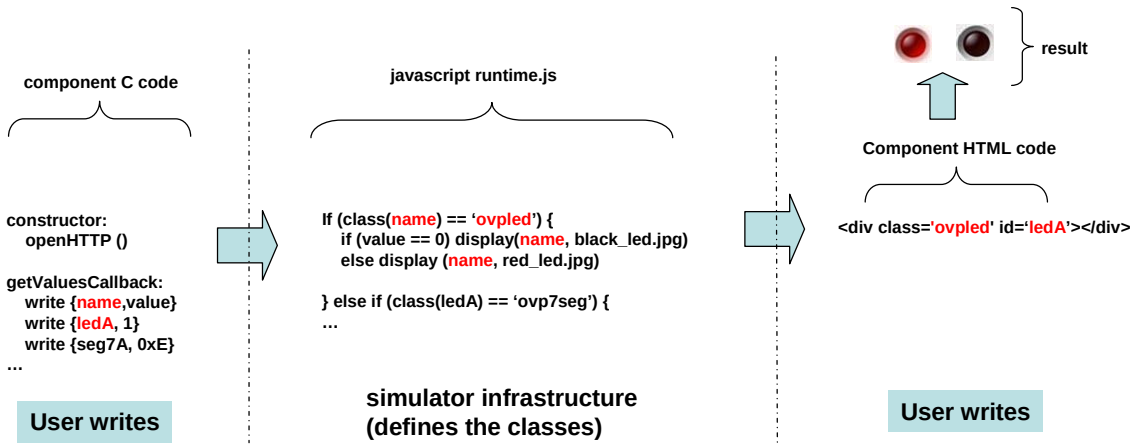
The link between the component C code writing a value and an image being displayed in the browser is handled via the Imperas visualization 'classes' and the instances of them.

The component `index.html` file creates object placeholders that include a class type and an instance name.

The component C code writes key value pairs, {instance name, value} to the HTTP port (in JSON format, but that is hidden from the user).

The Javascript `runtime.js`, running in the browser looks up the HTML for that instance name and determines its class. Based on its class and the current value, the Javascript writes to the HTML different appropriate values/images (that it finds in the `ImperasLib/httpvis` directory).

So the connection between the HTML placeholder and the Javascript is the **class name** and the connection between the component C code, the HTML placeholder is the **instance name**.



The only items that can be displayed by writing values from the component C code are instances of the Imperas visualization classes. These are defined in the `ImperasLib/httpvis/runtime.js` file.

The section (9.12) below lists the available classes, shows the value range that are allowed, and shows various visualization displays and images.

Section (9.13) below explains how to extend the system and add your own user defined classes.

9.3 Component index.html file

This is the same for peripherals, modules, intercept libraries, and harnesses.

This file is in the Imperas ImperasLib VLNV library tree next to the model binary, in a directory `httpvis/` and must be called `index.html`. It provides the structure for this component to be displayed in an the HTML browser.

This file defines which items are to be displayed and where they are to be located on the screen/frame. It includes the Javascript file (`runtime.js`) to perform the displaying and the cascading style sheet file (`normal.css`) to format the output.

The mandatory parts of this file are shown here:

```
<html lang="en">
<link rel="icon" type="image/icon" href="/imperas.ico">
<head>
    <title>Imperas Simulation</title>
    <meta   charset="utf-8"/>
    <link   rel="stylesheet"        href="normal.css"   type="text/css"/>
    <script language="JavaScript"   src="runtime.js"    ></script>
</head>
<body onload="startRefresh('/', 500);">
```

```
<!-- your class instances here in <div> blocks -->

<div class='console' id='console'></div>
</body>
</html>
```

An example class instance for a simple LED would be:

```
    <div class='ovpled' id='led0'></div>
```

this has a class of `ovpled`, and an instance name (id) of `led0`.

For a power button it could be:

```
    <div class='ovppower' id="power0"
onmousedown="ovpswitchclicked(event)"></div>
```

Note that for input items, there is the `class` to be displayed, the instance `id` name, and the `onmousedown` Javascript function that is to be called. (it must be `'ovpswitchclicked(event)'`).

For an item to be displayed it must have an instance name in the HTML file with a known class. (And the component C code must match and write appropriate values.)

The line:

```
    <body onload="startRefresh('/', 500);">
```

indicates that every 500 msecs the browser should call the `'startRefresh'` Javascript function (which updates the screen value by getting data from the HTTP port and thus the component C code call back).

The line:

```
<div class='console' id='console'></div>
```

is used for tracing if needed, so it is a good idea to include.

9.4 Peripheral C code

To add visualization to a peripheral you need to add code in three places; In the constructor to open the HTTP and declare your call backs, in the callback for the `'get'` function to send display data, and in the `'post'` function to process switch clicks, button pushes etc.

The code below is extracted from the example `/Examples/Models/Peripherals/visualization/switch_led`.

9.4.1 iGen Definition

The peripheral model is created using the iGen model generation tool.

This file defines the peripheral model

```
set vendor  ovpworld.org
set library peripheral
set name    SwitchLed
set version 1.0

imodelnewperipheral -name $name -imagefile pse.pse \
    -library $library -vendor $vendor -version $version \
    -constructor constructor \
    -destructor destructor

iadddocumentation  \
    -name Description  \
    -text "Simple LED and Switch"

iadddocumentation  \
    -name Licensing  \
    -text "Open Source Apache 2.0"
```

The interface for connectivity of the peripheral

```
imodeladdbusslaveport -name bport -mustbeconnected -size 0x8
imodeladdaddressblock -name reg -port bport -size 0x8 -offset 0 -width 32

imodeladdmmregister -addressblock bport/reg -name sw -width 32 \
    -offset 0 -access r -readfunction ReadSwitch
imodeladdmmregister -addressblock bport/reg -name led -width 32 \
    -offset 4 -access w -writefunction WriteLed
```

It must also include the following formal macros to define the configuration parameters that are included with the visualization.

```
#
# Formal attributes
#
imodeladdformalmacro -name BHM_HTTP_FORMALS
imodeladdformalmacro -name BHM_RECORD_REPLAY_FORMALS
```

9.4.2 Constructor

For the peripheral constructor you need to declare a static buffer (which will be used to transfer the data from the peripheral to the HTTP port) and then you need to declare your callback methods etc. and open the HTTP port:

```
#define BUFSIZE 128000
static char space[BUFSIZE];

PPM_CONSTRUCTOR_CB(constructor) {
```

```
    ...
    bhmHTTPMethods m = { .get=get, .post=post, .message=space,
    .length=BUFSIZE, .userData=0 };
    bhmHTTPOpen(&m, "httpvis");
}
```

The `get` and the `post` are the names of the callbacks discussed below. `space` is a static buffer to hold the data being sent.

`httpvis` in the the `bhmHTTPopen` is the directory to find the `index.html` to use to display the structure.

9.4.3 BHM_HTTP_GET_FN get callback function

The get callback is called when the HTTP port receives a `get` command from the browser which is initiated by the periodic timeout.

The get callback just calls a function `redraw ()`, that reads the data from C model data structures, formats them, and then initiates an HTTP send message.

```
static BHM_HTTP_GET_FN(get) {
    redraw (ch);
}
```

9.4.4 the redraw function

The redraw function reads the local variables in the peripheral (that will have been updated in the normal operation of the peripheral) and sends them to the HTTP port as shown:

```
static void redraw (bhmHTTPChannel ch) {

    Uns32 led = bport_reg_data.led.value;
    Uns32 sw = bport_reg_data.sw.value;

    bhmHTTPElementOpen(ch, "ovpelement");
    bhmHTTPKeyPrintf(ch, "led0", "%d", led);

    bhmHTTPKeyPrintf(ch, "power0", "");

    bhmHTTPElementClose(ch, "ovpelement");
    bhmHTTPSend(ch);
}
```

With `bhmHTTPElementOpen` you are constructing the start of a JSON message. With `bhmHTTPElementClose` you are ending the message and with `bhmHTTPSend` you send it.

Between the `bhmHTTPElementOpen` and `bhmHTTPElementClose` you use `bhmHTTPKeyPrintf` to define the instances you are displaying and their values. (NOTE these instance names must be the '`id`'s defined in the `index.html` file.)

The section (9.12) below defines the visualization classes and the value ranges they take.

Note that we create the redraw code as a separate function, so that we can call it in the get callback but also after a button has been pressed (i.e. whenever a button has been pressed we do an immediate update of the display).

9.4.5 BHM_HTTP_POST_FN post callback function

The post callback is called when the HTTP port receives a `post` command from the browser which is initiated by a button push or switch click etc. in the browser.

The post callback compares the value of the string body variable that is passed to it, and performs the user defined appropriate action, for example setting a value of a variable in the peripheral, or affecting the simulation such as calling termination.

If the post callback changes a value in the peripheral state, it does not explicitly change any of the displays, but expects that the next get callback will send updated values when it is called, updating the browser visualization.

```
static BHM_HTTP_POST_FN(post) {
    if (strstr(body, "sw0=clicked")) {
        bport_reg_data.sw.value = (~bport_reg_data.sw.value) & 0x01;
    } else if (strstr(body, "power0=clicked")) {
        bhmMessage("I", PREFIX "_SW",
        "Power Switch pushed - terminating simulation.");
        bhmFinish();
    }
    redraw (ch);
}
```

Currently the Imperas visualization classes only include simple push buttons and switches and so they all return a string of the form `'<instance name>=clicked'`.

The section (9.12) below defines the visualization classes that can initiate a call to the post callback.

Note the call to the `redraw ()` function so that any changes made by the button click are immediately updated in the display (as opposed to awaiting the next browser initiated refresh).

9.5 Module and Harness C code

To add visualization to either a module or harness you need to add code in three places; In the pre-simulation phase to open the HTTP and declare your call backs, in the callbacks for the `'get'` function to send display data, and in the `'post'` function to process switch clicks, button pushes etc.

This is very similar to the code discussed above for peripherals.

The code below is extracted from the example `/Examples/PlatformVisualization/moduleNet`.

9.5.1 Pre-Simulation initialization

In the pre-simulate function (`OP_PRE_SIMULATE_FN`) you need to declare your callback functions etc. and open the HTTP port:

```
static OP_PRE_SIMULATE_FN(modulePreSimulate) {
    Uns32 httpvisportnum  =
    opObjectParamUns32Value (mi,   "httpvisportnum" , 0);
    octHTTPMethods m = { .get=get, .post=post, .userData=object };
    opModuleHTTPOpen(mi, &m, httpvisportnum, "httpvis");
    ...
}
```

The `get` and the `post` are the names of the callback functions discussed below with `object` and `userData` being passed in (to allow the callbacks to find out what module they are in etc.).

`httpvis` in the the `opModuleHTTPOpen` is the directory to find the `index.html` to use to display the structure.

In this example, there is parameter `httpvisportnum` that can be set at run time with an override that sets the port number for this component instance. This `httpvisportnum` is passed into the `opModuleHTTPOpen` call.

9.5.2 get callback

The get callback is called when the HTTP port receives a `get` command from the browser which is initiated by the periodic timeout.

The get callback reads the local variables in the component (that will have been updated in the normal operation of the component) and sends them to the HTTP port as shown:

```
static OCL_HTTP_GET_FN(get) {
    optModuleObjectP object = userData;

    oclHTTPElementOpen(ch, "ovpelement");

    oclHTTPKeyPrintf(ch, "moduleInstName0", opObjectName(object->mi));
    oclHTTPKeyPrintf(ch, "moduleSimTime0","%g",
        (double)opModuleCurrentTime(opObjectRootModule(object->intNet)) );
    oclHTTPKeyPrintf(ch, "netName0", object->intNetName);
    oclHTTPKeyPrintf(ch, "netValue0", "%d", object->intNetValue);
    oclHTTPKeyPrintf(ch, "led0", "%d", object->intNetValue);
    oclHTTPKeyPrintf(ch, "power0", "0");

    oclHTTPElementClose(ch, "ovpelement");
    oclHTTPSend(ch);
}
```

Using `oclHTTPElementOpen` you can construct the start of a JSON message. Using `oclHTTPElementClose` you end the message and Using `oclHTTPSend` you send it.

Between the `oclHTTPElementOpen` and `oclHTTPElementClose` you use `oclHTTPKeyPrintf` to define the instances you are displaying and their values. (NOTE these instance names must be the '`id`'s in the `index.html` file.)

The section (9.12) below defines the visualization classes and the value ranges they take.

9.5.3 post callback

The post callback is called when the HTTP port receives a `post` command from the browser which is initiated by a button push or switch click etc. in the browser.

The post callback compares the value of the string body variable that is passed to it, and performs the user defined appropriate action, for example setting a value of a variable in the component, or affecting the simulation such as calling termination.

If the post callback changes a value in the component state, it does not explicitly change any of the displays, but expects that the next get callback will send updated values when it is called, updating the browser visualization.

```
static OCL_HTTP_POST_FN(post) {
    optModuleObjectP object = userData;
    if (strstr(body, "power0=clicked")) {
        opMessage("I", PREFIX "_SW",
        "Power Switch pushed - terminating simulation.");
        opModuleFinish(object->mi, 0);
    }
}
```

Currently the Imperas visualization classes only include simple push buttons and switches and so they all return a string of the form `'<instance name>=clicked'`.

The section (9.12) below defines the visualization classes that can initiate a call to the post callback.

As there is no state change on any button click (except simulation termination), we do not need to do a redraw after a button click.

9.6 Intercept Library C code

To add visualization to an intercept library you need to add code in three places; In the constructor to open the HTTP and declare your call backs, in the callback for the `'get'` function to send display data, and in the `'post'` function to process switch clicks, button pushes etc.

The code below is extracted from the examples in `/Examples/BinaryInterception/Visualization`.

9.6.1 Constructor

For the intercept library constructor you need to declare your callback methods etc. and open the HTTP port:

```
static VMIOS_CONSTRUCTOR_FN(constructor) {
    vmiPrintf("\n" INAME " VMIOS_CONSTRUCTOR_FN(constructor): (%s)\n\n",
    vmirtProcessorName(processor));
```

```
    paramValuesP params = parameterValues;

    Uns32 port = params->httpvisportnum;

    octHTTPMethods m = { .get=get, .post=post, .userData=object };
    vmihttpOpen(object, &m, port, "httpvis");
}
```

The `get` and the `post` functions implement the behavior discussed below with `object` and `userdata` variables being provided that the callbacks can use to access the processor they are in etc.

`httpvis` in the `vmihttpOpen` is the directory to find the `index.html` used to display the structure.

Note that for intercept libraries, the open is a call to the VMI API not the OP (for modules/harnesses) or BHM (peripherals) APIs.

Note there is also a parameter, `httpvisportnum` allowing the port number to be set using overrides at runtime.

9.6.2 get callback

The get callback is called when the HTTP port receives a `get` command from the browser which is initiated by the periodic timeout.

The get callback reads the local variables in the intercept library (that will have been updated in the normal operation of the intercept library) and sends them to the HTTP port as shown:

```
static OCL_HTTP_GET_FN(get) {
    vmiosObjectP object = userData;
    vmiProcessorP p = object->cpu;
    oclHTTPElementOpen  (ch, "ovpelement");
    oclHTTPKeyPrintf    (ch, "name0",      vmirtProcessorName(p));
    oclHTTPKeyPrintf    (ch, "power0",      "0");
    oclHTTPElementClose (ch, "ovpelement");
    oclHTTPSend         (ch);
}
```

Using `oclHTTPElementOpen` you can construct the start of a JSON message. Using `oclHTTPElementClose` you end the message and with `oclHTTPSend` you send it.

Between the `oclHTTPElementOpen` and `oclHTTPElementClose` you use `oclHTTPKeyPrintf` to define the instances you are displaying and their values. (NOTE these instance names must be the '`id`'s in the `index.html` file.)

The section (9.12) below defines the visualization classes and the value ranges they take.

9.6.3 post callback

The post callback is called when the HTTP port receives a `post` command from the browser which is initiated by a button push or switch click etc. in the browser.

The post callback compares the value of the string body variable that is passed to it, and performs the user defined appropriate action, for example setting a value of a variable in the peripheral, or affecting the simulation such as calling termination.

If the post callback changes a value in the intercept libraries state, it does not explicitly change any of the displays, but expects that the next get callback will send updated values when it is called, updating the browser visualization.

```
static OCL_HTTP_POST_FN(post) {
    if(!strcmp(body, "power0=clicked")) {
        vmiPrintf("\n"  INAME " OCL_HTTP_POST_FN(post): %s\n\n", body);
        vmirtStop();
    }
}
```

Currently the Imperas visualization classes only include simple push buttons and switches and so they all return a string of the form `'<instance name>=clicked'`.

The section (9.12) below defines the visualization classes that can initiate a call to the post callback.

Note that if want the display to be updated on a button click, put the code inside the OCL_HTTP_GET_FN(get) function in a separate redraw function, and call it there and also at the end of the OCL_HTTP_POST_FN(post) call. See the example in section 9.4.4 above.

9.7 Top Level visualization.html file

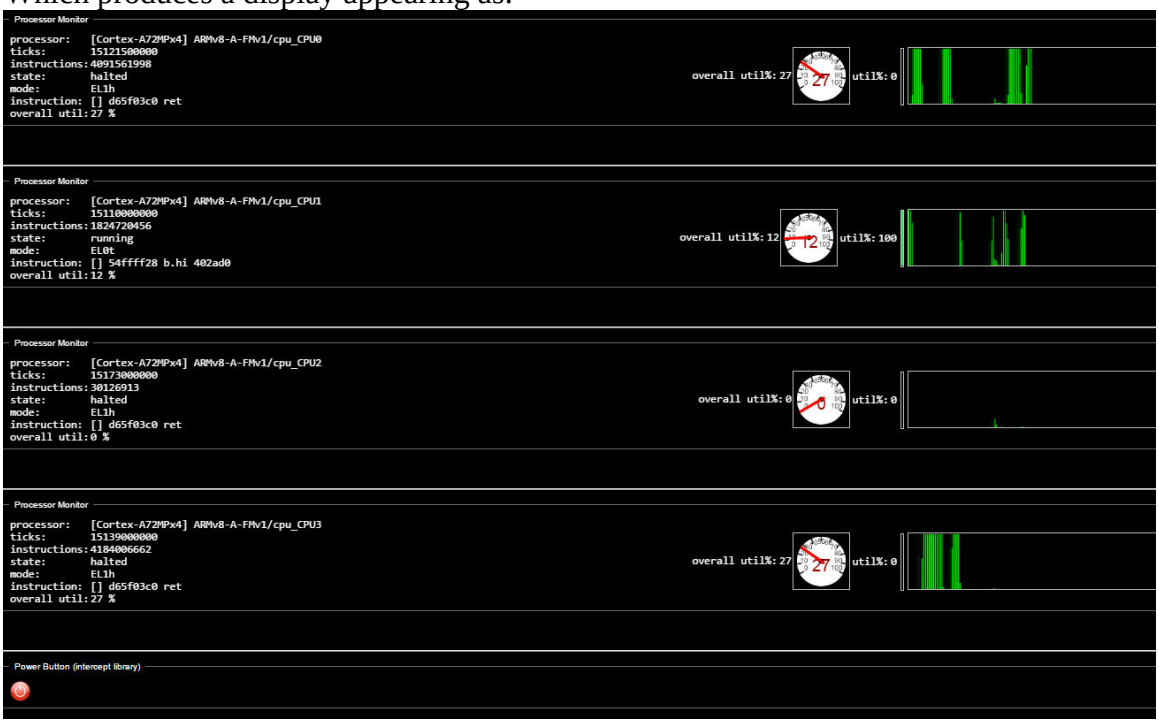
If you have several components that have visualization, you can use the HTML 5 frame's concept to make a single html page open up several different HTTP ports in different areas of the page, i.e. in different frames.

Uses of this are; if you have several peripherals you want to display in a harness or module, or if you have several processors you are monitoring.

The example, `Demo/Platforms/Linux_ARMv8-A-FMv1/visualization` in the `Demo_Linux_ARMv8-A-FMv1` package has this file `visualization_ARMv8-A-FMv1_arm_Cortex_A72MPx4.html`:

```
<html>
<head>
    <title>Processor Monitoring</title>
</head>
<body>
    <iframe id="cpu0" class="monitor0" style="width:100%; height:22%" src="http://localhost:8000"></iframe>
    <iframe id="cpu1" class="monitor1" style="width:100%; height:22%" src="http://localhost:8001"></iframe>
    <iframe id="cpu2" class="monitor2" style="width:100%; height:22%" src="http://localhost:8002"></iframe>
    <iframe id="cpu3" class="monitor3" style="width:100%; height:22%" src="http://localhost:8003"></iframe>
    <iframe id="pwr" class="power"     style="width:100%; height:12%" src="http://localhost:8030"></iframe>
</body>
</html>
```

Which produces a display appearing as:



9.8 C API functions for peripherals

The header file used to define the functions for the visualization in peripherals is part of the BHM API, this file can be found here
`ImpPublic/include/target/peripheral/bhmHttp.h`

There is doxygen documentation of the API in your installation at:
`doc/api/peripheral/html/index.html`

The functions are:

```
    BHM_HTTP_GET_FN((*bhmHTTPGetFn))
    BHM_HTTP_POST_FN((*bhmHTTPPostFn))

    bhmHTTPMethods m = { .get=get, .post=post, .message=space,
                         .length=BUFSIZE, .userData=0 };
    bhmHTTPOpen (bhmHTTPMethodsP methods, const char *fileRoot)
    bhmHTTPSend (bhmHTTPChannel ch)
    bhmHTTPClose (bhmHTTPChannel ch)

    bhmHTTPElementOpen (bhmHTTPChannel ch, const char *key)
    bhmHTTPKeyPrintf (bhmHTTPChannel ch, const char *key,
                      const char *fmt,...)
    bhmHTTPElementClose (bhmHTTPChannel ch, const char *key)
```

9.9 C API functions for modules and harnesses

Most of the functions used for visualization in the C code of modules and harnesses is in common with those used in intercept libraries and this is defined within the OCL API.

There is doxygen documentation at: `doc/api/ocl/html/index.html`

This header can be found here: `ImpPublic/include/host/ocl/oclhttp.h`

The functions are:

```
    OCL_HTTP_GET_FN(_name)
    OCL_HTTP_POST_FN(_name)

    octHTTPMethods m = { .get=get, .post=post, .userData=object };
    oclHTTPSend (octHTTPChannelP channel)
    oclHTTPClose (octHTTPChannelP channel)

    oclHTTPElementOpen (octHTTPChannelP channel, const char *key)
    oclHTTPKeyPrintf (octHTTPChannelP channel, const char *key,
                      const char *fmt,...)
    oclHTTPElementClose (octHTTPChannelP channel, const char *key)
```

The function to open the HTTP is not defined within the OCL API but is specific to modules/harnesses and is defined within the OP API.

```
    opModuleHTTPOpen (void *module, octHTTPMethodsP methods,
    Uns32 portNum, const char *fileRoot)
```

There is doxygen documentation at: `doc/api/op/html/index.html`

This header can be found here: `ImpPublic/include/host/op/op.h`

9.10 C API functions for intercept libraries

Most of the functions used for visualization in the C code of intercept libraries is in common with those used in modules/harnesses and this is defined within the OCL API.

There is doxygen documentation at: `doc/api/ocl/html/index.html`

This header can be found here: `ImpPublic/include/host/ocl/oclhttp.h`

The callbacks are:

```
    OCL_HTTP_GET_FN(_name)
    OCL_HTTP_POST_FN(_name)
```

The functions are:

```
    oclHTTPSend (octHTTPChannelP channel)
    oclHTTPClose (octHTTPChannelP channel)

    oclHTTPElementOpen (octHTTPChannelP channel, const char *key)
    oclHTTPKeyPrintf (octHTTPChannelP channel, const char *key,
                      const char *fmt,...)
    oclHTTPElementClose (octHTTPChannelP channel, const char *key)
```

The function to open the HTTP is not defined within the OCL API but is specific to intercept libraries and is defined within the VMI API.

```
       octHTTPMethods methods = { .get=get, .post=post, .userData=object };

    vmihttpOpen (vmiosObjectP object, octHTTPMethodsP methods,
        Uns32 portNum, const char *fileRoot)
```

There is doxygen documentation at: `doc/api/vmi/html/index.html`

This header can be found here: `ImpPublic/include/host/vmi/vmiHTTP.h`

9.11 Using iGen to help assist in creating visualization

9.11.1 iGen can create templates for peripherals

When you create peripherals with `iGen`, you get templates for the most useful peripheral API calls and callbacks. If you use the `-httpvis` argument `iGen` will create the appropriate code. For example, in your `peripheral.tcl`:

```
imodelnewperipheral \
    -name      pseWithHTTP \
    -version   1.0 \
    -httpvis
```

Will add to the `pse.c.igen.stubs` templates for the `get` and `post` callbacks:

```
BHM_HTTP_GET_FN(peripheralHTTPGet) {
    bhmHTTPElementOpen(ch, "ovpelement");
    // insert your code to add key/value pairs here. e.g.
    //
    // bhmHTTPKeyPrintf(ch, "yourkey", "%u", yourValue);

    bhmHTTPElementClose(ch, "ovpelement");
    bhmHTTPSend(ch);
}

BHM_HTTP_POST_FN(peripheralHTTPPost) {
    // insert your code to process the string 'body'
    //
}
```

And the `pse.igen.c` will get the code to open the HTTP port:

```
int main(int argc, char *argv[]) {

    diagnosticLevel = 0;
    bhmInstallDiagCB(setDiagLevel);
    periphConstructor();

    char space[1024];
    bhmHTTPMethods methods = { .get=peripheralHTTPGet, .post=peripheralHTTPPost,
                               .message=space, .length=1024, .userData=0 };
    bhmHTTPOpen(&methods, "httpvis");

    bhmWaitEvent(bhmGetSystemEvent(BHM_SE_END_OF_SIMULATION));
    return 0;
}
```

9.11.2 iGen can create templates for modules

When you create OP modules with `iGen`, you get templates for the most useful module OP API calls and callbacks. If you use the `-httpvis` argument `iGen` will create the appropriate code. For example:

```
    igen.exe --batch module.tcl -writec mod -op
```

with `module.tcl`:

```
ihwnew \
    -name      moduleWithHTTP \
    -version   1.0 \
    -httpvis
```

gets the code to open the HTTP port in your constructor in `module.igen.h`:

```
static OP_CONSTRUCT_FN(moduleConstructor) {
    // igen passes the current module (mi) as user data
    octHTTPMethods methods = { .get=moduleHTTPGet, .post=moduleHTTPPost,
                               .userData=mi };
    opModuleHTTPOpen(mi, &methods, 0, "httpvis");
}
```

and the template callbacks in `module.c.igen.stubs`:

```
OCL_HTTP_GET_FN(moduleHTTPGet) {
    oclHTTPElementOpen(ch, "ovpelement");

    // insert your code to add key/value pairs here. e.g.
    //
    // oclHTTPKeyPrintf(ch, "yourkey", "%u", yourValue);

    oclHTTPElementClose(ch, "ovpelement");
    oclHTTPSend(ch);
}

OCL_HTTP_POST_FN(moduleHTTPPost) {
    // insert your code to process the string 'body' here
    //
}
```

9.12 Imperas visualization classes: class, images, values

This section lists the Imperas visualization classes that are provided with the simulator. These will be added to in future releases.

See the section below describing how to add your own visualization classes without the need to edit the Imperas provided files (you don't want to edit these as each new release will overwrite the files you edit).

It is the file `ImperasLib/httpvis/runtime.js` that defines classes and defines how they are displayed.

When the browser needs to display data, it issues an HTTP `get` in the Javascript `sendCommand` function. It then processes the returned data (in JSON format) and there is a `switch` statement that performs the appropriate action for each of the different classes.

For example, the `ovpled` class displays a red or black led .jpg depending on the value of that specific `ovpled` instance. (It uses the normal Javascript approach of just replacing the `.innerHTML` of the appropriate element in the html.)

```
    switch (classname) {
        case 'ovpled':
            if ((value & 0x01) == 1) {
               document.getElementById(data.ovpelement[i].idname).innerHTML=
            "<img src=led-red.jpg>";
            } else {
               document.getElementById(data.ovpelement[i].idname).innerHTML=
            "<img src=led-black.jpg>";
            }
            break;
```

9.12.1 ovpled

Class name: ovpled
Value range: 0 (black), 1 (red)
HTML instance code:

```
<div class='ovpled' id="led0"></div>
```

Example: Examples/Models/Peripherals/visualization/switch_led

9.12.2 ovpbuttonled

Class name: ovpbuttonled
Value range: 0 (green), 1 (red)
Click returns: '<instance id>=clicked'
HTML instance code:

```
<div class='ovpbuttonled' id="bled3"
onmousedown="ovpswitchclicked(event)"></div>
```

Example: Examples/Models/Peripherals/visualization/switches

9.12.3 ovplcd7seg

Class name: ovplcd7seg
Value range: 0 - 15 (0-F)
HTML instance code:

```
<div class='ovplcd7seg' id="lcd0"></div>
```

Example: Examples/Models/Peripherals/visualization/lcd_7segment

9.12.4 ovplcd2x16

Class name: ovplcd2x16
Value range: displays the ASCII of a 32 character string. 16 chars on each line.
HTML instance code:

```
<div class='ovplcd2x16' id="lcd2x16_0"></div>
```

Example: Examples/Models/Peripherals/visualization/lcd_2x16

9.12.5 ovpswitchtoggle

Class name: ovpswitchtoggle
Value range: 0 (up), 1 (down)
Click returns: '<instance id>=clicked'
HTML instance code:

```
<div class='ovpswitchtoggle' id="sw3"
onmousedown="ovpswitchclicked(event)"></div>
```

Example: Examples/Models/Peripherals/visualization/switches

9.12.6 ovpswitchdip

Class name: ovpswitchdip
Value range: 0 (up), 1 (down)
Click returns: '<instance id>=clicked'
HTML instance code:

```
<div class='ovpswitchdip' id="swt0"
onmousedown="ovpswitchclicked(event)"></div>
```

Example: Examples/Models/Peripherals/visualization/switches

9.12.7 ovpbar

Class name: ovpbar
Value range: 0 - 100
HTML instance code:

```
    <table>
        <tr>
            <td class='ovplabel'>Horizontal Bar</td>
            <td><div class='ovpbar' id="bar0"></div></td>
        </tr>
    </table>
```

Example: Examples/Models/Peripherals/visualization/bars

9.12.8 ovpvertbar

Class name: ovpvertbar
Value range: 0 - 100
HTML instance code:

```
<div class='ovpvertbar' id="bar1" height='100' width='20'
    linewidth='3' style='border: 2px solid; margin: 4px;'></div>
```

Example: Examples/Models/Peripherals/visualization/bars

9.12.9 ovppower

Class name: ovppower
Click returns: '<instance id>=clicked'
HTML instance code:

```
<div class='ovppower' id="power0" onmousedown="ovpswitchclicked(event)"
    title='click to quit simulation'>SIMULATION NOT RUNNING</div>
```

Example: Examples/Models/Peripherals/visualization/power_button

9.12.10 ovpreset

Class name: ovpreset
Value range: 0 (red), 1 (blue)

Click returns: '<instance id>=clicked'
HTML instance code:

```
<div class='ovpreset' id="reset1" onmousedown="ovpswitchclicked(event)"
    title='click to toggle reset'></div>
```

Example: Examples/PlatformVisualization/processorReset

9.12.11 ovpgauge

Class name: ovpgauge
File included in index.html:

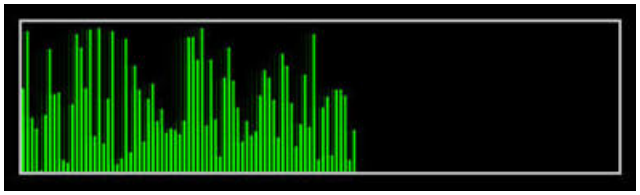
```
 <script language="JavaScript" src="canvasgauge.js"></script>
```

HTML instance code:

```
<div>
    <canvas class='ovpgauge' id="gauge0"> </canvas>
    <script>
        var gauge0 = new Gauge("gauge0", {'mode':'needle', 'range': {'min':0, 'max':100 } });
    </script>
</div>
```

Value range: min - max (as defined in the html instance)
Example: Examples/Models/Peripherals/visualization/dial

9.12.12 ovphistogram



Class name: ovphistogram
Value range: 0 - 100
HTML instance code:

```
<div>
    <canvas class='ovphistogram' id='histo0' height='100' width='400' linewidth='3'
        style='border: 2px solid;'></canvas>
</div>
```

Example: Examples/Models/Peripherals/visualization/time_histogram

9.13 Adding your own visualization classes and items

To add your own visualization classes and images you need to add several files to your component's `httpvis/` directory and then make use of them.

The example:

```
    Examples/Models/Peripherals/visualization/local_item
```

shows adding a new class that will toggle an image of a plus/minus when clicked.

It has the directory:

```
    peripheral/httpvis
```

which shows these files and how to add your own classes.

You need a Javascript file (`local.js`) to add the display and button/switch click processing, a css file (`local.css`) to style the displayed items and the images (`minus.jpg`, `plus.jpg`) to display (though these might not be needed if are just drawing in Javascript).

First, include the new files into the `peripheral/httpvis/index.html`:

```
<head>
    <title>Imperas Simulation</title>
    <meta   charset="utf-8"/>
    <link   rel="stylesheet"        href="normal.css"   type="text/css"/>
    <link   rel="stylesheet"        href="local.css"    type="text/css"/>
    <script language="JavaScript"   src="runtime.js"        ></script>
    <script language="JavaScript"   src="local.js"></script>
</head>
```

Note the local files must be listed following the built in files.

Then instance the new item:

```
   <div class='localitem_a' id="local0" onmousedown="ovpswitchclicked(event)" title='clickable'></div>
```

In this example, the `local.css` simply sizes the image:

```
.localitem_a {
height: 40px;
width: 40px;
}
```

The `local.js` will have a function for displaying, and a function (if needed) for the switch/button clicking.

The names of these functions is fixed (if they exist, and they are called from the built in `runtime.js` to extend its code).

For the display, we are extending the class decoding:

```
function localClassDecoder (classname, idname, value) {
    //trace ("in localClassDecoder ("+classname+", "+idname+", "+value+")");
    switch (classname) {
        case 'localitem_a':
            if (value & 0x01 == 1) {
                document.getElementById(idname).innerHTML="<img
src=minus.jpg>";
            } else {
                document.getElementById(idname).innerHTML="<img src=plus.jpg>";
            }
            return true; // good return, processed
            break;
        default:
```

```
            return false; // error return, not found
            break;
    }
}
```

It is important to `return true` if we have locally taken care of the class type, and `false` if not - so the appropriate error can be reported.

To extend the switch/button processing:

```
// process local item, return true if ok, else false if not found etc
function localSwitchDecoder (classname, idname) {
    //trace ("in localSwitchDecoder ("+classname+", "+idname+")");
    switch (classname) {
        case 'localitem_a':
            var form = document.createElement("form");
            form.setAttribute("method", "post");
            form.setAttribute("action", ""); // path
            var hiddenField = document.createElement("input");
            hiddenField.setAttribute("type", "hidden");
            hiddenField.setAttribute("name", idname);
            hiddenField.setAttribute("value", "clicked");
            form.appendChild(hiddenField);
            document.body.appendChild(form);
            form.submit();
            return true; // good return, processed
            break;
        default:
            return false; // error return, not found
            break;
    }
}
```

The mechanism is that if it is our switch/button that is clicked (in this example the instance name is 'localitem_a'), we use the Javascript to add a dynamic form to the HTML page and then submit it -with the data value set to 'clicked' - so that the C code in our component's post callback will receive '<idname>=clicked'.

It is important to `return true` if we have locally taken care of the class type, and `false` if not - so the appropriate error can be reported.

10 Common Problems

10.1 Fixed HTTP port numbers

When components are defined they will have a default HTTP port number (typically 8000). You can normally override this from the command line or when instancing the component. If you get a port clash - i.e. two or more ports trying to be opened with the same number, you will see the following form of error message:

```
Fatal (HTTP_PNIU) 'ARMv8-A-FMv1/cpu_CPU1/pm' is trying to open HTTP port number
8000 which is already being used by 'ARMv8-A-FMv1/cpu_CPU0/pm'
Info Exiting
```

You need to change the port numbers so they are all unique.

If you have several sessions simultaneously using the same machine/port numbers you will get errors. Within Imperas we have solutions to this, for example during regression testing, so please contact Imperas support for advice.

10.2 No index.html for component

The message:

```
Error (HTTP_FNF) Model 'test/pse0' cannot find file './index.html'
```

means that when looking for the `httpvis/index.html` for a component it could not be found - each component needs one. It should be in the same path as the binary of the component.

10.3 Browser can not connect to HTTP port

If you get an error in your browser similar to:

```
    This site can't be reached
    localhost refused to connect.
```

Then either you are trying to connect to the wrong hostname/port number, or you have not enabled the HTTP ports in the simulation run.

Did you see this in the simulation console?:

```
    Info (HTTP_PORT) 'test/pse0' listening on port 8000
```

If not - then you need to enable the simulators HTTP ports.

To enable HTTP ports either use the command line option:

```
    --httpvis
```

Or set the environment variable, for example:

```
    export IMPERAS_HTTP=1
```

or

```
    set IMPERAS_HTTP=1
```

And ensure some of the components in your simulation are opening HTTP ports.

10.4 Browser pop up: runtime.js sendCommand has class name not recognized

This message pop up indicates that you have used a class type in your `index.html` that is not recognized in the `runtime.js` or in any `local.js` decoder that you have specified. Check your html instances to ensure that all class names are correct.

10.5 Browser pop up: runtime.js sendCommand error. Your C code function has an element which is undefined in the html

This message pop up indicates that you are trying to write a value in your c code (with `bhmHTTPKeyPrintf` or `oclHTTPKeyPrintf`) to an instance id/name in your `index.html` that is not found. Check your html instance names to ensure that they are all the same as those written to in the C.

You can also get this message if you run one simulation that uses an HTTP port with certain data, and then leave that browser open (listening) and then run a different simulation with different data being sent - with the same port number. You then have two browsers listening to the same port, which is OK, but the problem is that the early one has different HTML/Javascript data requirements and thus is incompatible as it can not find the data it requires in the HTTP data received. To resolve this, always make sure you cancel all listening browsers before starting a new simulation.

10.6 Tracing HTTP port data

To see what is happening across the HTTP ports, set the environment variable:

```
    IMPERAS_HTTP_TRACE=1
```

10.7 Other errors

Often it is informative to see what errors are reported from the browsers perspective. Each browser is different, but they all tend to have an 'error console' or 'console output' that you can view.

Chrome has Developer Tools->Console.

Safari has Develop->Show Error Console.

Firefox has Web Developer Tools->Web Console.

10.8 How to run regression test models with visualization

Within Imperas we have a methodology that addresses this and provides complete batch testing of the visualization displays and also the clicking of buttons/switches. Please contact Imperas support for more information.

#