Writing Platforms and Modules in C User Guide

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

Author:	Imperas Software Limited
Version:	2.0.1
Filename:	Writing_Platforms_and_Modules_in_C_User_Guide.doc
Project:	Writing Platforms and Modules in C User Guide
Last Saved:	Monday, 13 January 2020

Copyright Notice

Right to Copy Documentation

Destination Control Statement

Disclaimer

Table of Contents

1 Preface

The Imperas simulators can use models described in C or C++ and the models can be exported to be used in simulators and platforms using C, C++, SystemC or SystemC TLM2.

This introductory document describes how to use the OVP C APIs to write simple platforms and modules and to control the simulator with test harnesses also written in C.

It describes how the OVP OP C API is used in C programs for use with Imperas and OVP virtual platform simulators.

1.1 Related Documentation

There are two related documents that focus on controlling the simulators using the OVP OP C API. If your interest is running simulations of existing platforms/models, more advanced test harnesses, or debugger/3rd party simulator integration then these documents should be your focus:

- Simulation Control of Platforms and Modules User Guide
- Advanced Simulation Control of Platforms and Modules User Guide

If you are creating platforms or subsystems/modules of components, then you need to look at using iGen. There are two documents that describe the use of iGen to create platforms and modules using its advanced, high level input script. With iGen script, larges, complex platforms and modules can be created with very few lines of code:

- iGen Model Generator Introduction
- iGen Platform and Module Creation User Guide

There are several other relevant documents available:

Getting Started

- Imperas Installation and Getting Started Guide

Interface, API, and iGen related

- OVP Peripheral Modeling Guide
- OVPsim Using OVP Models in SystemC TLM2.0 Platforms
- Imperas Peripheral Generator Guide (using iGen)

1.2 Notation

Code	Text representing code, a command or output.
keyword	A word with special meaning.

1.3 Glossary / Terminology

OP API - OVP **P**latforms API - C API used for creating and controlling virtual platforms. 2nd generation API, replaces ICM API. iGen creates modules/platforms in C using this API.

iGen - Imperas productivity tool that has a powerful script based function API that is used to create C/C++/SystemC models and templates. Described in the iGen Model Generator Introduction, and for platforms/modules, in the iGen Platform and Module Generator User Guide.

OVPsim - Simulator for Open Virtual Platforms that executes platforms and models coded in the OVP APIs

CpuManager - Imperas commercial simulator that fully implements the APIs defined by OVP (OVPsim implements a subset)

Platform / Module – a collection of components connected together into a level of hierarchy in a system to be simulated. This is a program in C/C++ making calls into OP API and normally compiled into a shared object/dynamically linked library and loaded by the simulator at run time.

Testbench / Harness – (used interchangeably) – A program in C/C++ making calls into the OP API to connect and control OVP components. It is normally linked to the simulator to provide an .exe binary that can be executed. Used to instance one or more platforms/modules and controls their execution. The main difference, from a platform/module, is that a testbench or harness includes a definition of the function main(), may include a command line parser and is linked to create an executable binary (.exe) file.

Root Module - used to describe the initial platform/module that instances one or more platforms/modules and controls their execution. Used in the testbench / harness.

2 Introduction

Imperas simulation technology enables very high performance simulation, debug and analysis of platforms containing multiple processors and behavioral peripheral models. The technology is designed to be extensible: you can create your own platforms, new models of processors, and other platform components using interfaces and libraries supplied by Imperas. Platform models developed using this technology can be used both with Imperas simulation products and the freely-available OVPsim platform simulator.

2.1 Platform construction & simulation approaches

There are two ways to construct and simulate designs:

- **Combined approach**: the platform and the test harness that controls it can be created in C as one compilation unit. This is a combined approach and is the same as the usage of the deprecated ICM platform modeling API.
- **Module approach**: the platform and the test harness that controls it can be created as separate compilable objects using OP API modules. (The modules can be written directly in C using OP API calls, or created from iGen script.)

If using a module based approach (separate platform and test harness), then there are two ways of making a test harness:

- **Bespoke test harness**: a test harness can be written in C using the OP API.
- ***harness.exe***: In the Imperas / OVP release packages, there is a program provided, *harness.exe*, which removes the need for writing a C test harness. *harness.exe* simulates modules directly.

This introductory document explains the basic usage of the OP API to write harnesses / test benches and simple modules. For all but simple platforms/modules, iGen should be used. Please consult the documents referred to above.

For a description and example of the **combined approach**, please see chapter 5.

For a description and example of the **module approach, with bespoke harness**, please see chapter 6.

For a description and example of the **module approach, using the provided *harness.exe*** program, please see chapter 7.

2.2 Prerequisites

Since harnesses and test benches for use with Imperas and OVP tools are written in C, an important prerequisite is that you must be proficient in the C language. If you want to use C++ then it is expected that you are proficient in the use of C++ and how it uses a C API.

2.3 Obtaining & Installing the OP API

The OP API is part of all Imperas / OVP installations and thus you should already have it installed and be ready for use.

2.4 Compiling Examples described in this Document

The examples use modules, processors, component models and tool chains, available to download from the www.OVPworld.org website or as part of an Imperas installation.

The compilation of the examples utilize a Makefile, the instructions for which indicate the use of the command *make*, on Windows systems the MinGW *mingw32-make* command should be used in its place.

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 as (for example) *example.sh*, this being the extension used on Linux or for Windows MSYS shells. On Windows the script would be called *example.bat*

2.5 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 have the suffix *.exe*

3 Imperas Simulation Overview

Before starting to create models for use with the Imperas simulation environment, you must understand how the components used in that environment interact. This section describes this in detail.

3.1 Simulation Environments

There are currently two simulation environments available that can be used with models and platforms that you create:

- *OVP* allows component models created using OVP modeling technology to be used in C harness, platform and testbench files to create executables, which represent real hardware. These include processor and other behavioral components. The executable can be used to run binary application files compiled and linked for the processors whose models are included. OVP can be used in 3rd party simulation environments (for example, SystemC). It can be used to create a test harness to help validate processor models under construction, or even to create custom simulation environments. OVP has less functionality than the Imperas Professional Simulator Products in some areas and has restricted commercial usage as stipulated in the OVP click-through license agreement.

- *Imperas Professional Simulator Products* enhance the basic capabilities provided by OVPsim, particularly in the areas of debugger integration, tool integration and multiprocessor simulation support (including QuantumLeap parallel simulation). Contact Imperas for more information.

3.2 What are OVPsim and CpuManager?

OVP provides the *OVPsim* simulator and the Imperas Professional product provides the *CpuManager* simulator as dynamic linked libraries (`.so` suffix on Linux, `.dll` suffix on Windows) implementing Imperas simulation technology. The shared objects contain implementations of the OP interface functions used in this document. The OP functions enable instantiation, interconnection and simulation of complex multiprocessor platforms using multicore processors, advanced peripheral devices and complex memory topologies.

Processor models for use with *CpuManager* and *OVPsim* are created using the *OVP Virtual Machine Interface* (VMI) API, and are available for download from the www.OVPworld.org website. This API enables the creation of processor models that run at very high simulation speeds (typically hundreds of millions of simulated instructions per second). The use of the API is described in the *OVP Processor Modeling Guide*, also available for download from the www.OVPworld.org website.

The CpuManager simulator is part of the commercial/professional product offering available from Imperas. OVPsim is the freely-available (for Non-Commercial usage) version of the simulator. The simulator can be selected at runtime by the

IMPERAS_RUNTIME environment variable. If it is not set or is set to OVPsim the OVPsim library (which requires an OVP license) will be used. If it is set to CpuManager the CpuManager library (which requires an Imperas license) will be used.

The legacy ICM API is supported by the same products, providing a subset of the functionality offered by OP. In fact, the ICM API is implemented using OP so will be supported for the foreseeable future.

A subset of OP functionality can be used in SystemC TLM2.0. The TLM2.0 C++ interface code is available as source for processor and peripheral models, allowing the use of these models in SystemC TLM2.0 platforms.

3.3 Use of OP with Imperas tools

A program using the OP or ICM APIs must be linked with the Imperas RuntimeLoader library to perform runtime dynamic loading of either the CpuManager or OVPsim dynamic linked libraries, to produce a stand-alone executable. This allows the runtime selection of the CpuManager simulator for any defined platform and so enabling the use of the Imperas tools.

3.4 C API and iGen

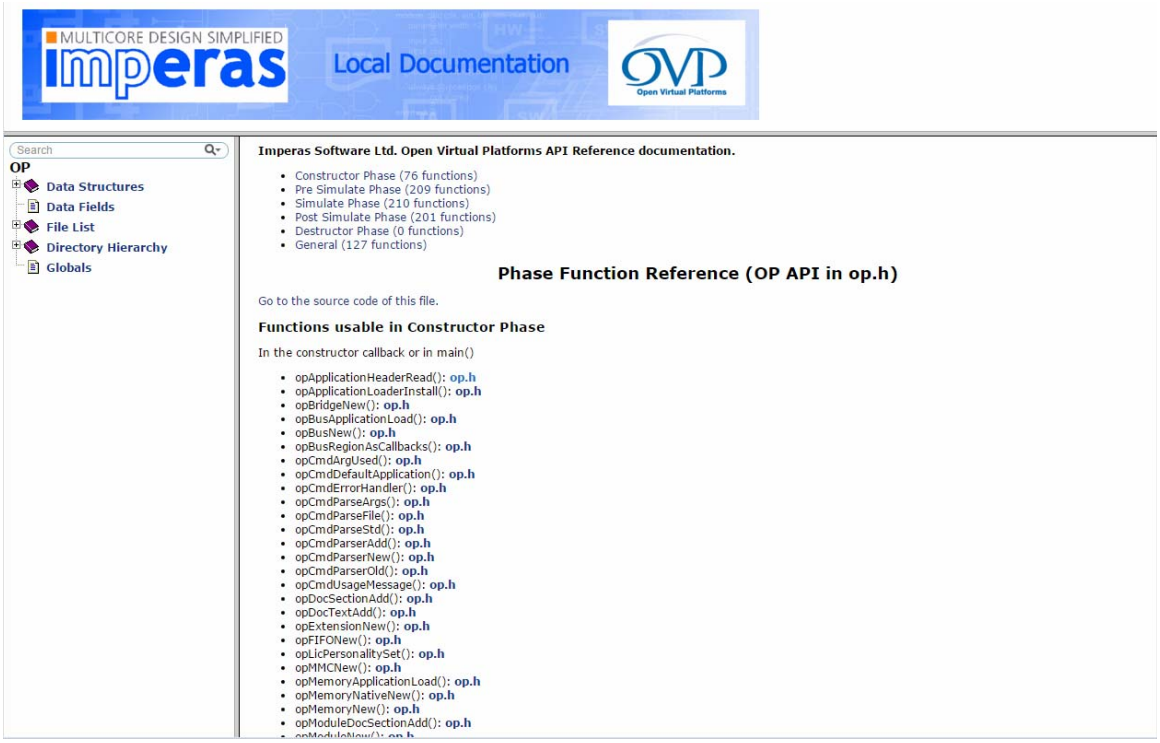
An OVP platform/module is written in C code and compiled and linked on the host computer to produce a shared object. An OVP testbench/harness, is written in C code, compiled and linked with the *RuntimeLoader* (*libRuntimeLoader.so*) link library on the host computer to produce an executable.

iGen is a program from Imperas that can create either outline or substantially complete C code for a module or testbench/harness using a description written using iGen function calls. Use of iGen is described elsewhere. This document is focusing on platforms/modules and test harness written directly in C using the OVP OP API.

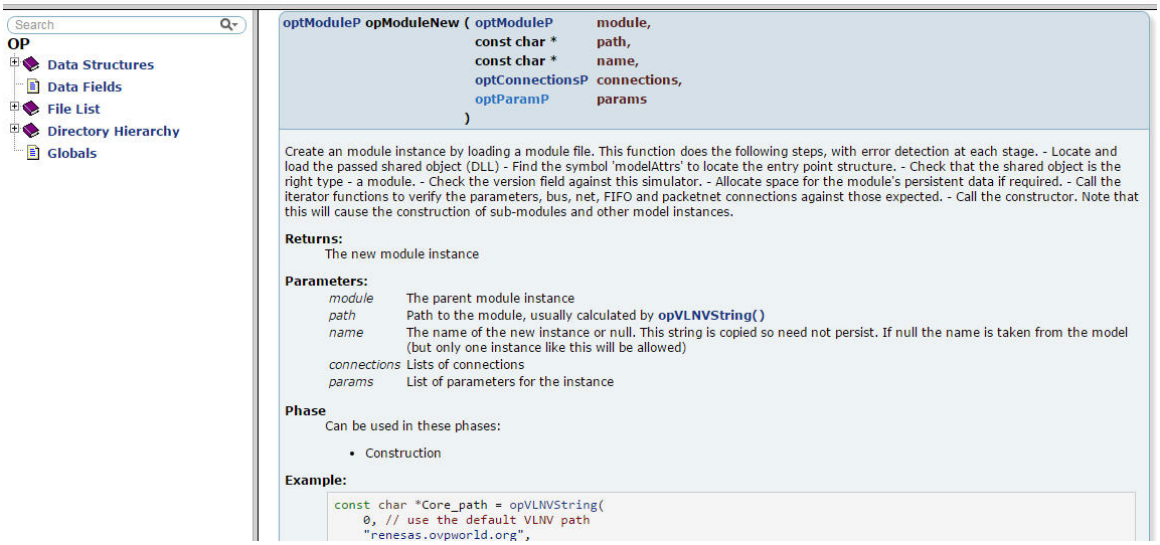
4 Documentation on the OP API

Provided in the installation is the online OP API Function Reference documentation. This is Doxygen API documentation available at:

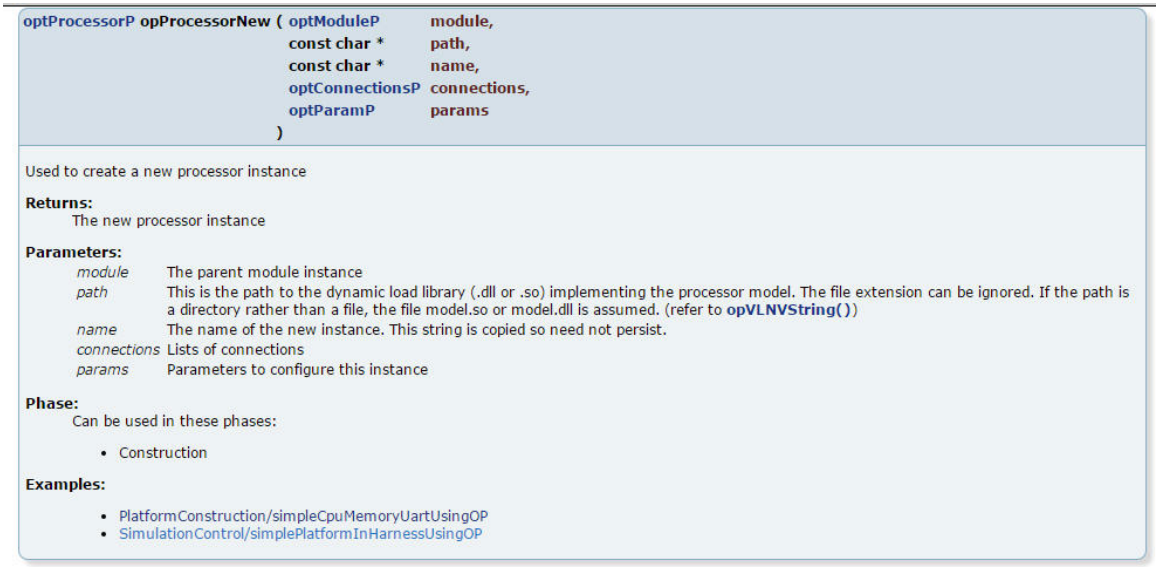
IMPERAS_HOME/doc/api/op/html/index.html



It includes per function references:



There are hyperlinks to many complete examples that illustrate usage of the function:



```
optProcessorP opProcessorNew ( optModuleP       module,
                               const char *     path,
                               const char *     name,
                               optConnectionsP  connections,
                               optParamP        params
                             )
```

Used to create a new processor instance

Returns:
The new processor instance

Parameters:

module	The parent module instance
path	This is the path to the dynamic load library (.dll or .so) implementing the processor model. The file extension can be ignored. If the path is a directory rather than a file, the file model.so or model.dll is assumed. (refer to **opVLNVString()**)
name	The name of the new instance. This string is copied so need not persist.
connections	Lists of connections
params	Parameters to configure this instance

Phase:
Can be used in these phases:

- Construction

Examples:

- PlatformConstruction/simpleCpuMemoryUartUsingOP
- SimulationControl/simplePlatformInHarnessUsingOP

5 C harness and platform in one file (Combined Approach)

This style is not the recommended approach for anything but the smallest simplest examples.

For most designs it is good practice to put the *structure* in a separate module from the *testbench*. The *structure* is code that creates instances of the components and connects them together. The *testbench* sets the initial conditions then runs the platform for the required length of time. Keeping the two separate allows the structure to be used in different test scenarios or as part of a larger design. For most designs the Imperas iGen tool should be used to create the C platform/modules from a concise tcl iGen input script. Please see the documents described above for more information.

You can develop your design as one C program combining the harness, the instance of the processors, peripherals buses etc, all in one compilable program.

```
> cp -r $IMPERAS_HOME/Examples/SimulationControl/simplePlatformInHarnessUsingOP .
> cd simplePlatformInHarnessUsingOP
> ls
application   harness    example.sh
```

The application is a simple hello world writing to a UART it has initialized:

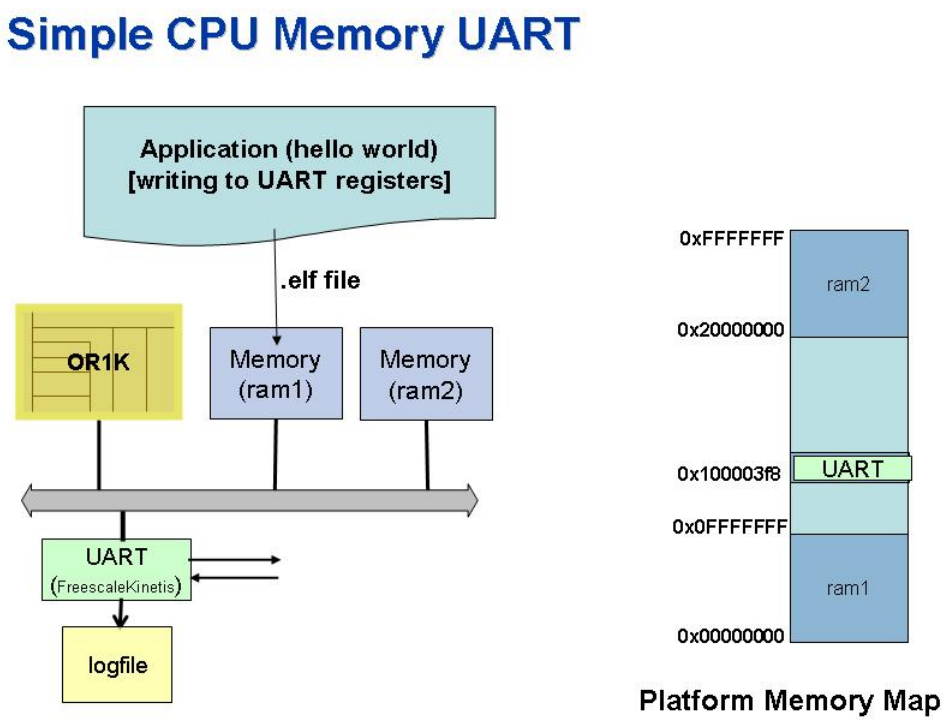
```
> cat application/application.c
...
int main(int argc, char **argv) {

    initFreeScaleKinetisUart(UART0_BASE);

    printf ("Writing to uart - see log file\n\n");

    writeMessFreescaleKinetisUart(UART0_BASE,
    "Hello UART0 world with design written in C using OP API instanced directly in the harness.\n\n");

    return 0;
}
```

Where *initFreeScaleKinetisUart* sets up the UART for writing and *writeMessFreescaleKinetisUart* writes the message.

It is cross compiled using make to create the binary *.elf* file:

```
> make -C application
# Compiling application.c
# Linking application.OR1K.elf
```

The platform is an OR1K cpu with memory and UART:



As this is an example of the harness and platform being combined, there is one C file that describes both the platform and the harness.

At the bottom of *harness/harness.c* is the call to *main*:

```
int main(int argc, const char *argv[]) {
    opSessionInit(OP_VERSION);
    opCmdParseStd (argv[0], OP_AC_ALL, argc, argv);

    optModuleP mi = opRootModuleNew(0, 0, 0);
    constructPlatform (mi);

    opRootModuleSimulate(mi);
    opSessionTerminate();
    return 0;
}
```

We can see we initialize the session, make a call to the standard command parser and then instance the root module of the design.

We then call our function (*constructPlatform*) that constructs the platform, call *opRootModuleSimulate* to simulate the root module (*mi*) and then terminate the session.

The platform is constructed in the call to *constructPlatform* declared at the top of *harness/harness.c*:

We first declare the bus and two nets:

```
static void constructPlatform (optModuleP mi) {

    optBusP mainBus_b = opBusNew(mi, "mainBus", 32, 0, 0);

    optNetP directWrite_n = opNetNew(mi, "directWrite", 0, 0);
    optNetP directRead_n = opNetNew(mi, "directRead", 0, 0);
```

The processor with VLNV ovpworld.org/processor/or1k/1.0 is selected from the library. It is given the name *"cpu1"*.
The *'generic'* variant is selected.

```
const char *cpu1_path = opVLNVString(
        0, // use the default VLNV path
        "ovpworld.org",
        "processor",
        "or1k",
        "1.0",
        OP_PROCESSOR,
        1   // report errors
    );

    optProcessorP cpu1_c = opProcessorNew(
        mi,
        cpu1_path,
        "cpu1",
        OP_CONNECTIONS(
            OP_BUS_CONNECTIONS(
                OP_BUS_CONNECT(mainBus_b, "INSTRUCTION"),
                OP_BUS_CONNECT(mainBus_b, "DATA")
            )
        ),
        OP_PARAMS(
            OP_PARAM_STRING_SET("variant", "generic")
        )
    );
```

We define its semihost library so that any calls to printf (for simulation monitoring) are displayed in the simulation console:

```
    const char *or1kNewlib_0_expath = opVLNVString(
        0, // use the default VLNV path
        0,
        0,
        "or1kNewlib",
        0,
        OP_EXTENSION,
        1   // report errors
    );
```

```
opProcessorExtensionNew(
    cpu1_c,
    or1kNewlib_0_expath,
    "or1kNewlib_0",

    0
);
```

We instance two memories:

```
// Memory ram1

opMemoryNew(
    mi,
    "ram1",
    OP_PRIV_RWX,
    (0x0fffffff) - (0x0),
    OP_CONNECTIONS(
        OP_BUS_CONNECTIONS(
            OP_BUS_CONNECT(mainBus_b, "sp1", .slave=1, .addrLo=0x0, .addrHi=0x0fffffff)
        )
    ),
    0
);

// Memory ram2

opMemoryNew(
    mi,
    "ram2",
    OP_PRIV_RWX,
    (0xffffffff) - (0x20000000),
    OP_CONNECTIONS(
        OP_BUS_CONNECTIONS(
            OP_BUS_CONNECT(mainBus_b, "sp1", .slave=1, .addrLo=0x20000000, .addrHi=0xffffffff)
        )
    ),
    0
);
```

The constructing function concludes with an instance of a Freescale Kinetis UART:

```
const char *periph0_path = opVLNVString(
    0, // use the default VLNV path
    "freescale.ovpworld.org",
    "peripheral",
    "KinetisUART",
    "1.0",
    OP_PERIPHERAL,
    1   // report errors
);

opPeripheralNew(
    mi,
```

```
        periph0_path,
        "periph0",
        OP_CONNECTIONS(
            OP_BUS_CONNECTIONS(
                OP_BUS_CONNECT(mainBus_b, "bport1",
                        .slave=1, .addrLo=0x100003f8, .addrHi=0x100013f7)
            ),
            OP_NET_CONNECTIONS(
                OP_NET_CONNECT(directWrite_n, "DirectWrite"),
                OP_NET_CONNECT(directRead_n, "DirectRead")
            )
        ),
        OP_PARAMS(
             OP_PARAM_STRING_SET("outfile", "uartTTY0.log")
        )
    );
}
```

The combined harness and platform is also compiled with the provided *harness/Makefile*:

```
> make -C harness
# Host Depending obj/Linux32/harness.d
# Host Compiling Harness obj/Linux32/harness.o
# Host Linking Harness harness.Linux32.exe
```

Which creates the compiled executable: *harness.Linux32.exe*.

The simulation is run with:

```
> harness/harness.Linux32.exe --program application/application.OR1K.elf
...
OVPsim started: Tue Feb  9 00:08:44 2016

Initializing KinetisUART
Writing to uart - see log file

OVPsim finished: Tue Feb  9 00:08:44 2016
```

The log file contains the characters written by the application to the UART:

```
> cat uartTTY0.log
Hello UART0 world with design written in C using OP API instanced directly in the harness.
```

The complete combined *harness/harness.c* is:

```
#include <string.h>
#include <stdlib.h>

#include "op/op.h"

#define HARNESS_NAME "harness"
```

```
static void constructPlatform (optModuleP mi) {

    // Bus mainBus

    optBusP mainBus_b = opBusNew(mi, "mainBus", 32, 0, 0);

    // nets

    optNetP directWrite_n = opNetNew(mi, "directWrite", 0, 0);
    optNetP directRead_n = opNetNew(mi, "directRead", 0, 0);

    // Processor cpu1

    const char *cpu1_path = opVLNVString(
        0, // use the default VLNV path
        "ovpworld.org",
        "processor",
        "or1k",
        "1.0",
        OP_PROCESSOR,
        1   // report errors
    );

    optProcessorP cpu1_c = opProcessorNew(
        mi,
        cpu1_path,
        "cpu1",
        OP_CONNECTIONS(
            OP_BUS_CONNECTIONS(
                OP_BUS_CONNECT(mainBus_b, "INSTRUCTION"),
                OP_BUS_CONNECT(mainBus_b, "DATA")
            )
        ),
        OP_PARAMS(
            OP_PARAM_STRING_SET("variant", "generic")
        )
    );

    // semihost library

    const char *or1kNewlib_0_expath = opVLNVString(
        0, // use the default VLNV path
        0,
        0,
        "or1kNewlib",
        0,
        OP_EXTENSION,
        1   // report errors
    );

    opProcessorExtensionNew(
        cpu1_c,
        or1kNewlib_0_expath,
        "or1kNewlib_0",

        0
```

```
);

// Memory ram1

opMemoryNew(
    mi,
    "ram1",
    OP_PRIV_RWX,
    (0x0fffffff) - (0x0),
    OP_CONNECTIONS(
        OP_BUS_CONNECTIONS(
            OP_BUS_CONNECT(mainBus_b, "sp1", .slave=1, .addrLo=0x0, .addrHi=0x0fffffff)
        )
    ),
    0
);

// Memory ram2

opMemoryNew(
    mi,
    "ram2",
    OP_PRIV_RWX,
    (0xffffffff) - (0x20000000),
    OP_CONNECTIONS(
        OP_BUS_CONNECTIONS(
            OP_BUS_CONNECT(mainBus_b, "sp1", .slave=1, .addrLo=0x20000000, .addrHi=0xffffffff)
        )
    ),
    0
);

// peripheral periph0

const char *periph0_path = opVLNVString(
    0, // use the default VLNV path
    "freescale.ovpworld.org",
    "peripheral",
    "KinetisUART",
    "1.0",
    OP_PERIPHERAL,
    1   // report errors
);

opPeripheralNew(
    mi,
    periph0_path,
    "periph0",
    OP_CONNECTIONS(
        OP_BUS_CONNECTIONS(
            OP_BUS_CONNECT(mainBus_b, "bport1",
                    .slave=1, .addrLo=0x100003f8, .addrHi=0x100013f7)
        ),
        OP_NET_CONNECTIONS(
            OP_NET_CONNECT(directWrite_n, "DirectWrite"),
            OP_NET_CONNECT(directRead_n, "DirectRead")
```

```
            )
        ),
        OP_PARAMS(
            OP_PARAM_STRING_SET("outfile", "uartTTY0.log")
        )
    );

}

int main(int argc, const char *argv[]) {
    opSessionInit(OP_VERSION);
    opCmdParseStd (argv[0], OP_AC_ALL, argc, argv);

    optModuleP mi = opRootModuleNew(0, 0, 0);
    constructPlatform (mi);

    opRootModuleSimulate(mi);
    opSessionTerminate();
    return 0;
}
```

To compile both the application and harness, and run the example, a script is provided:

```
> ./example.sh
```

Note that you can pass in any appropriate simulation command line arguments, for example:

```
> ./example.sh --help
...
> ./example.sh --showbuses
...
> ./example.sh --showmodule
...
> harness/harness.Linux32.exe --program application/application.OR1K.elf  --trace
...
```

To see data written to the UART, use the *--modeldiags* command line argument:

```
> harness/harness.Linux32.exe --program application/application.OR1K.elf --modeldiags 0x3
...
OVPsim started: Tue Feb  9 00:20:32 2016

Info (UART_UIS) periph0: Uart initialized in serial channel mode
Initializing KinetisUART
Info (UART_BRC) periph0: Baud rate changed to 19921
Info (UART_BRC) periph0: Baud rate changed to 19577
Info (UART_TFT) periph0: Transmitter fifo threshold set to 1
Info (UART_RFT) periph0: Receiver fifo threshold set to 1
Info (UART_UW) periph0: Write to Data register: data=0x0d ('
')
Writing to uart - see log file

Info (UART_UW) periph0: Write to Data register: data=0x48 ('H')
```

```
Info (UART_UW) periph0: Write to Data register: data=0x65 ('e')
Info (UART_UW) periph0: Write to Data register: data=0x6c ('l')
Info (UART_UW) periph0: Write to Data register: data=0x6c ('l')
Info (UART_UW) periph0: Write to Data register: data=0x6f ('o')
Info (UART_UW) periph0: Write to Data register: data=0x20 (' ')
...
Info (UART_UW) periph0: Write to Data register: data=0x65 ('e')
Info (UART_UW) periph0: Write to Data register: data=0x73 ('s')
Info (UART_UW) periph0: Write to Data register: data=0x73 ('s')
Info (UART_UW) periph0: Write to Data register: data=0x2e ('.')
Info (UART_UW) periph0: Write to Data register: data=0x0a ('
')
Info (UART_UW) periph0: Write to Data register: data=0x0a ('
')

OVPsim finished: Tue Feb  9 00:20:32 2016
```

6 C harness and separate platform (Module Approach)

You can develop models of hierarchical systems where you have many different sub-systems (modules), each written as a separate model and separately compiled as module shared objects. The harness instances the top level (root) module, which in turn instances the other (potentially hierarchical) modules.

In this example, there are three separate sub-directories:

```
> cp -r $IMPERAS_HOME/Examples/PlatformConstruction/simpleCpuMemoryUartUsingOP .
> cd simpleCpuMemoryUartUsingOP
> ls
application/   harness/    module/    example.sh
```

The application is very similar to the application in the previous example and is compiled using the provided Makefile:

```
> make -C application
# Compiling application.c
# Linking application.OR1K.elf
```

In this example, the harness and module are two separate compilation units. Basically they have very similar content to the previous combined example, they are just split into a harness and a module that is instanced in it.

6.1 Module written in C using OP

A module is a separately compilation unit of the design hierarchy. It provides an interface to the caller using a *modelAttrs* table. In this example, at the bottom of the *module/module.c* file, we use the bare minimum *modelAttrs* table:

```
> cat module/module.c
...
optModuleAttr modelAttrs = {
    .versionString      = OP_VERSION,
    .type               = OP_MODULE,
    .name               = MODULE_NAME,
    .releaseStatus      = OP_UNSET,
    .purpose            = OP_PP_BAREMETAL,
    .visibility         = OP_VISIBLE,
    .constructCB        = moduleConstructor,
};
```

Where the interesting line is the declaration of our module constructor function:

```
    .constructCB        = moduleConstructor,
```

Please refer to the Simulation Control documents introduced at the beginning of this document for more information on the *modelAttrs* table.

It is the module constructor function that instances the components that make up our module. It is the same as that in the example in the previous chapter apart from now we declare it with a macro *OP_CONSTRUCT_FN* (and instead of calling it in a *main* function, we define it as an entry point in our *modelAttrs* table). The complete *module/module.c* looks like:

```
> cat module/module.c
...
#include "op/op.h"

#define MODULE_NAME "simpleCpuMemoryUart"

static OP_CONSTRUCT_FN(moduleConstructor) {

    // Bus mainBus

    optBusP mainBus_b = opBusNew(mi, "mainBus", 32, 0, 0);

    // nets

    optNetP directWrite_n = opNetNew(mi, "directWrite", 0, 0);
    optNetP directRead_n = opNetNew(mi, "directRead", 0, 0);

    // Processor cpu1

    const char *cpu1_path = opVLNVString(
        0, // use the default VLNV path
        "ovpworld.org",
        "processor",
        "or1k",
        "1.0",
        OP_PROCESSOR,
        1   // report errors
    );

    optProcessorP cpu1_c = opProcessorNew(
        mi,
        cpu1_path,
        "cpu1",
        OP_CONNECTIONS(
            OP_BUS_CONNECTIONS(
                OP_BUS_CONNECT(mainBus_b, "INSTRUCTION"),
                OP_BUS_CONNECT(mainBus_b, "DATA")
            )
        ),
        OP_PARAMS(
             OP_PARAM_STRING_SET("variant", "generic")
        )
    );

    // processor semihosting library

    const char *or1kNewlib_0_expath = opVLNVString(
        0, // use the default VLNV path
        0,
```

```
    0,
    "or1kNewlib",
    0,
    OP_EXTENSION,
    1   // report errors
);

opProcessorExtensionNew(
    cpu1_c,
    or1kNewlib_0_expath,
    "or1kNewlib_0",

    0
);

// Memory ram1

opMemoryNew(
    mi,
    "ram1",
    OP_PRIV_RWX,
    (0x0fffffff) - (0x0),
    OP_CONNECTIONS(
        OP_BUS_CONNECTIONS(
            OP_BUS_CONNECT(mainBus_b, "sp1", .slave=1, .addrLo=0x0, .addrHi=0x0fffffff)
        )
    ),
    0
);

// Memory ram2

opMemoryNew(
    mi,
    "ram2",
    OP_PRIV_RWX,
    (0xffffffff) - (0x20000000),
    OP_CONNECTIONS(
        OP_BUS_CONNECTIONS(
            OP_BUS_CONNECT(mainBus_b, "sp1", .slave=1, .addrLo=0x20000000, .addrHi=0xffffffff)
        )
    ),
    0
);

// peripheral periph0

const char *periph0_path = opVLNVString(
    0, // use the default VLNV path
    "freescale.ovpworld.org",
    "peripheral",
    "KinetisUART",
    "1.0",
    OP_PERIPHERAL,
    1   // report errors
);
```

```
    opPeripheralNew(
        mi,
        periph0_path,
        "periph0",
        OP_CONNECTIONS(
            OP_BUS_CONNECTIONS(
                OP_BUS_CONNECT(mainBus_b, "bport1",
                        .slave=1, .addrLo=0x100003f8, .addrHi=0x100013f7)
            ),
            OP_NET_CONNECTIONS(
                OP_NET_CONNECT(directWrite_n, "DirectWrite"),
                OP_NET_CONNECT(directRead_n, "DirectRead")
            )
        ),
        OP_PARAMS(
             OP_PARAM_STRING_SET("outfile", "uartTTY0.log")
        )
    );

}

optModuleAttr modelAttrs = {
    .versionString      = OP_VERSION,
    .type               = OP_MODULE,
    .name               = MODULE_NAME,
    .releaseStatus      = OP_UNSET,
    .purpose            = OP_PP_BAREMETAL,
    .visibility         = OP_VISIBLE,
    .constructCB        = moduleConstructor,
};
```

A Makefile is provided that will take as input the hand written *module.c* and will create the *model.so/.dll* shared objects.

```
> make -C module
# Host Depending obj/Linux32/module.d
# Host Compiling Module obj/Linux32/module.o
# Host Linking Module object model.so
```

6.2 Harness in C using OP instancing a separate module

In this example, which is very similar to the harness part of the combined example in the previous chapter, we don't instance the individual components in our harness, we 'instance' the previously defined/compiled module as a sub component.

We define an empty root module:

```
    optModuleP mi = opRootModuleNew(0, 0, 0);
```

We then add an instance of a module (from the directory '*module*'):

```
    opModuleNew(mi, "module", "u1", 0, 0);
```

The complete harness looks like:

```
> cat harness/harness.c
...
#include <stdlib.h>

#include "op/op.h"

int main(int argc, const char *argv[]) {
    opSessionInit(OP_VERSION);

    opCmdParseStd (argv[0], OP_AC_ALL, argc, argv);

    optModuleP mi = opRootModuleNew(0, 0, 0);
    opModuleNew(mi, "module", "u1", 0, 0);

    opRootModuleSimulate(mi);

    opSessionTerminate();
    return 0;
}
```

Which we compile with the provided Makefile:

```
> make -C harness
# Host Depending obj/Linux32/harness.d
# Host Compiling Harness obj/Linux32/harness.o
# Host Linking Harness harness.Linux32.exe
```

Then it is run, giving the application as the program command line argument:

```
> harness/harness.Linux32.exe --program application/application.OR1K.elf
...
OVPsim started: Tue Feb  9 06:12:29 2016

Initializing KinetisUART
Writing to uart - see log file

OVPsim finished: Tue Feb  9 06:12:29 2016
```

Look at the UART log to see what was written to the UART:

```
> cat uartTTY0.log
Hello UART0 world with separate harness and module in C using OP API.
```

We have provided a script with the above compile and run commands:

```
> ./example.sh
...
```

For convenience this can be used with command line arguments.

7 Using harness.exe to simulate modules

In the first example in this document we introduced a combined harness and design as a single C program.

In the second example we split the harness and design into a harness program and a separately compiled module.

In this example we will use just the design/module and use the *harness.exe* program that is included with Imperas / OVP installations to control the simulation of the design. There is often no need to create a bespoke testbench.

```
> cp -r  $IMPERAS_HOME/Examples/PlatformConstruction/simpleCpuMemoryUartUsingOPandHarnessExe .
> cd simpleCpuMemoryUartUsingOPandHarnessExe
> ls
application/    module/     example.sh
```

If you look at the *module/module.c* it is exactly the same as in the *simpleCpuMemoryUartUsingOP* example. The application writes out slightly different text to the UART.

Again a script is provided to compile and run the example, and display the UART log file:

```
> ./example.sh
# Compiling application.c
# Linking application.OR1K.elf
...
# Host Depending obj/Linux32/module.d
# Host Compiling Module obj/Linux32/module.o
# Host Linking Module object model.so
...
OVPsim started: Tue Feb  9 06:34:09 2016

Initializing KinetisUART
Writing to uart - see log file

OVPsim finished: Tue Feb  9 06:34:09 2016

Hello UART0 world with module in C using OP API and harness.exe.
```

To see the command line arguments that you can use with the *harness.exe*, use the *--help* command:

```
> harness.exe --help
...
```