# A Study of Open Virtual Platforms for Simulating Embedded Systems

Mian Shahzeb Ihsan

*Abstract*—**In this paper we explore the use of Open Virtual Platforms (OVP) for simulating complete Embedded Systems Platforms. Open Virtual Platforms is a technology developed by Imperas to establish a common, open standard to facilitate the development of software in emulated or virtual platforms which can represent a complete Embedded System or a System-on-Chip (SoC). This paper discusses the components and theory of OVP, setup of an OVP simulation (OVPsim) environment and the complete development cycle of a Cortex-M3 based platform. Finally, some alternatives are briefly discussed.**

*Index Terms*—**BHM: Behavioral Hardware Modeling, ICM: Innovative CPU Manager, MP-SoC: Multi-Processor System-on-Chip, OVP: Open Virtual Platforms, PPM: Peripheral Programming Modeling, SoC: System-on-Chip, VMI: Virtual Machine Interface**

## I. Introduction

There have been many breakthroughs in System Level Virtual Prototyping (SLVP) and Electronic System Level (ESL) design and verification in the last few years. The technology has been on the verge of adoption but has not really got as much traction as would have been expected. The reason could be that Electronic Design Automation (EDA) platforms and models are expected to have a level of interoperability between models created by different vendors but this relatively simple seeming requirement has eluded the Electronic System Level domain so far. To sum up, ESL or SLVP lacks the desired ecosystem within which models from different vendors comply with a certain industry standard.

Over the last decade or so, several solutions have been floated by academic circles and industry leaders [1] for hardware-software co-verification e.g. Seamless from Mentor Graphics which used Instruction Set Simulator (ISS) models for processor and integrated them into conventional RTL simulation environments. But the problem with most of these solutions was performance. They were excellent tools for hardware engineers as they provided the required visibility into hardware design such as timing information, waveforms etc. along with being cycle accurate. Their timing accuracy was also beneficial for driver development. But being cycle accurate meant that they were too slow for application software development, which doesn't require a high level of timing accuracy but is more dependent on the order of execution. Many developers use FPGA or even physical prototypes for software verification but that is too late in the development cycle for any major hardware changes resulting in inelegant software workarounds.

With Embedded Computing becoming ubiquitous and Embedded Processors, either as COTS components or part of a SoC, becoming more powerful and more capable, software is becoming increasingly more important in the context of overall system design. As Embedded Systems become more complex, so does software as more and more system functionality becomes dependent on the software. With increased software complexity come issues of bugs, dead-line slips, and code maintenance and updates. This situation is amplified in the case of Systems-on-Chip (SoCs) that contain multiple processor cores (MP-SoCs) and becomes worse when the system consists of heterogeneous processors. Due to ever shortening time-to-market (TTM) requirements, especially in the consumer electronics industry, it is generally desirable to start software development in parallel with hardware development. The real problem is verification of the software; cycle-accurate RTL simulations can generally only simulate small sections of the software since even simulation of a few milliseconds can take a few hours. On the other hand, high-level behavioral models deliver better performance. If the platform is modeled in enough detail, the software developers can use the model as a virtual development environment or platform. This approach allows software to be comprehensively tested earlier in the development cycle. The software, which is compiled for the silicon and then run or simulated on a virtual platform, can be more rigorously tested than on the actual hardware. The virtual platform's observability and controllability also make it easier to debug the code. This means if software development and verification has to be done in parallel with hardware development; relatively detailed hardware models need to be available early in the development cycle, but this would be counter-productive if the development of any system required creation of new models. Open Virtual Platforms endeavors to solve this by establishing a common, open set of standards to create such platforms for Embedded Systems. It provides the specification of an interface using which models can communicate, a set of tools for creating models, APIs, an instruction accurate (as opposed to cycle accurate) simulator and open source processor and peripheral models to demonstrate capabilities of OVP and create an ecosystem. OVP can be used to model complete Virtual Embedded Platforms. These models can be written in C, C++ or SystemC. It also allows the use of

TLM2.0 (OSCI Transaction Level Modeling) interfaces when using SystemC; this means the pure TLM2.0 models can still be used with platforms created in Open Virtual Platforms [2]. The benefit of this approach is that pre-existing peripherals or models that have a TLM2.0 interface can be seamlessly integrated into the platform without re-writing those peripherals or models using modeling APIs provided by Open Virtual Platforms. The feature that sets OVP apart from other virtualization platforms is the performance of the simulation environment. On regular x86 machines, several hundred MIPS can be achieved.

## II. MAIN COMPONENTS OF OPEN VIRTUAL PLATFORMS

Open Virtual Platforms consist of three major components: a simulation engine, modeling APIs and open source models.

### A. *The Simulation Engine*

The simulation engine of Open Virtual Platforms is called OVPsim and can be downloaded from the OVP website [3]. It is available for both Windows and Linux. The license for non-commercial use of the simulation engine can be obtained free but commercial usage requires a license to be obtained from Imperas [4]. OVPsim is a processor emulator that supports multi-processor platforms. It emulates the target processer by using binary translation of the emulated instruction set. OVPsim is not cycle accurate but it is instruction accurate and is capable of running embedded software at hundreds of MIPS on standard x86 machines. The simulation engine also supports the GDB (GNU Debugger) remote serial port (RSP) protocol which allows debugging of embedded software running on the emulated processors. There are several OVPsim configuration flags that can be specified during platform initialization which allow configuration of various simulation parameters such as traceability and simulation schedule.

### B. *Model Application Programming Interfaces*

Open Virtual Platforms come with several C/C++ APIs that can be used to create platforms and models of processors and peripherals. There are three main modeling APIs; ICM, VMI and BHM/PPM. The relationship between these API is shown in Fig. 1. These APIs are explained below.

**Innovative CPU Manager (ICM**), is used to create models of platforms. Platform APIs can be called from C, C++ or SystemC. The ICM APIs can be used to design, create, configure and connect components. ICM provides the basic structures and classes which can be used to create instances of processors, memories, peripherals etc. The address mapping of all peripherals and memories attached to the processor and the software that is to be run by the processor are also specified using ICM. ICM can be used to easily specify very complex and complete platform configurations employing various processors, local and/or shared memories, different memory geometries (e.g. von Neumann or Harvard), caches, bus bridges, peripherals and all their complex address maps, interrupts and operating systems and application software. OVP buses are used as a communication medium but are not buses in the true sense, as they are not considered to be a shared resource, so no bus arbitration is required and all connections are essentially one-to-one communication channels. OVP platforms, once created using the ICM API, compile to executables which run natively on the host machine. These executables are the complete simulation for the targeted processor.

**Virtual Machine Interface (VMI**), is used to create processor models. VMI processor models are written in C. The virtual machine interface functions provide the ability to easily specify the behavior of the CPU. In the model, VMI decodes the target instruction to be simulated and translates this to native x86 instructions that are then executed natively. The VMI API also provides an intercept or virtualization mechanism which can be used to intercept system function calls (like open, read, write etc.), these can be used from within the simulated application to interact with the host machine running the simulation.

For peripheral modeling, OVP provides two types of APIs. **The Peripheral Programming Modeling (PPM)** API and the **Behavioral Hardware Modeling (BHM)** API. Behavioral components, peripherals, and the overall environment is modeled using C code and calls to these two APIs. Strictly speaking, PPM is used for creating peripherals where as BHM is used to create behavioral blocks. The difference is that PPM understands buses, ports and is similar in functionality to the Open SystemC Initiative (OSCI) TLM interface. BHM, on the other hand, has more in common with behavioral modeling languages and can handle more general forms of communication and is timing aware. Underlying these APIs is an event based scheduling mechanism to enable modeling of time, events, and concurrency. Peripheral models provide callbacks that are called when the application software, running on processors modeled in the platform, accesses memory locations where the peripheral is enabled. Peripheral models are run by a part of the simulation engine known as PSE (Peripheral Simulation Engine). Peripheral models run in their own protected address space and hence cannot crash a running simulation.

### C. *Open Source Models*

OVP provides a set of open source models, collectively known as OVPlib. The complete OVPlib source is available on the OVPlib project hosted on Google Code [5]. OVPlib includes models for a variety of processors like various ARM processors (such as ARM9, Cortex-M3), MIPS, ARC, TenSilica Diamond Core, Renesas v850 and OpenCores' openRISC OR1K. Besides processor models, there are a number of peripheral models available as well, including popular UART models such as the Intel 16450, ARM's Prime Cell PL011 and Atmel's USART model used in their AT91 ARM processors. Other available peripherals include vectored interrupt controllers, I/O ports, LCDs, timers/counters, keyboards, watchdog, USB etc. There are also several system models such as caches, RAM, ROM etc. And to demonstrate how these models are used, several example platforms are there as well. The sample platforms range from simple single processor or multi-processor configurations to complex platforms that can run complete operating systems such as Embedded Linux, ucLinux and uc/OS-II. The installers for object files, compiled libraries and executables for processor and peripheral models and platforms are available for download on the OVP download page. All these models or platforms can also be built from the source and the instructions are found on the OVP library page [3].

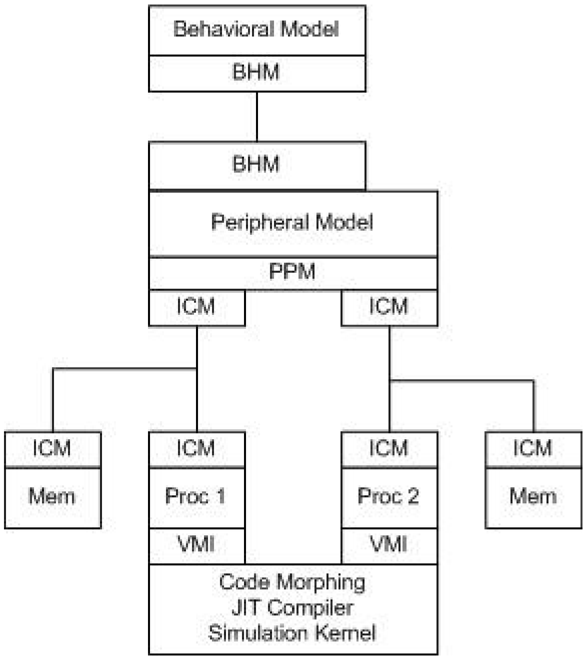


Fig. 1 OVP Interfaces [1].

## III. Setting up Open Virtual Platforms

There are several programs and tools that need to be installed for a complete OVP environment; these include OVPsim, host compiler, target (cross) compiler and required processor and peripheral models. Although OVP is available for both Linux and Windows hosts and mainly Linux was used in the development of the platform for this study, the process below describes the setup of an OVP environment for a Windows machine since the setup procedure for Linux is relatively simple whereas the Windows setup requires a few additional steps.

### *A. Installing OVPsim, the Open Virtual Platforms Simulation Engine*

OVPsim is the core component of OVP; it contains a set of shared runtime and compile-time libraries which the platform uses to execute the simulation. During the installation process, OVPsim sets the required environment variables that are required for the compilation process. Additionally, OVPsim is available as a free download from the OVP webpage [3], but registration is required to get access to the download. Additionally, a license is also required, which is available free of cost for non-commercial usage [4]. The default free license is valid for 90 days after which a new license can be requested again.

### *B. Peripheral and Processor Models and Sample Platforms*

A default set of the more commonly used peripheral models is installed by the OVPsim installation. Additional peripherals are available for download on the OVP library webpage [3]. Processor models need to be installed and are distributed as installable packages on the OVP download page [3]. The target processor used for this study was Cortex-M3; this processor model is available under the ARM section of processor model downloads. There are also a large number of sample platforms available for download. Several makefiles required for building the sample platforms are also installed though the compilers these makefiles require need to be installed separately and are covered in sections below. It will be worth noting that since the purpose of this study was to get a firm grasp of how Open Virtual Platforms work; the provided hierarchy of makefiles was not used for the platform implemented to demonstrate OVP usage. The build process was explored using a custom makefile and was later changed to use a WAF [6] script instead of a makefile for easier maintenance.

### *C. Host Compiler*

The host compiler is used to compile the platform into an executable platform simulator. GCC is required to be used as the host compiler and can be installed as part of Cygwin or via MinGW (Minimum GNU for Windows). OVP documentation recommends MinGW. Additionally, it is recommended that instead of the default Windows shell, the MSYS shell should be used to build the platforms.

### *D. Target (Cross) Compiler*

Depending on which target processor is being emulated, the appropriate cross compiler needs to be installed as well. For Cortex-M3, the default compiler recommended to compile the Cortex-M3 demo application is Code Sourcery's GCC-based ARM cross compiler "Sourcery G++ Lite Edition for ARM" [7]. This compiler is required by the sample platforms because the sample embedded applications use system calls to print output to the console, which requires ARM semi-hosting to be enabled, the Code Sourcery compiler comes with a default linker file that has semi-hosting pre-configured. The Cortex-M3 demo platform implemented as part of this study doesn't use semi-hosting and the ARM tool chain selected by the author based on personal preference was YAGARTO [8], but either tool chain can be used. If the default makefiles are used for compiling either the sample platforms or a custom platform, then the **ARM_CORTEX_M3_CROSSCOMPILER** variable in the following file should be set to the root folder of the cross compiler:

**$(IMPERAS_HOME)\lib\Windows\CrossCompiler\ARM_CORTEX_M3.makefile.include**

## IV. Sample Platforms

This section briefly discusses two sample Cortex-M3 platforms available for download from OVP world [3]. The first platform is a very simple configuration consisting of a single R/W memory attached to a Cortex-M3 processor core. The demo application is a Dhrystone benchmark algorithm implementation which uses semi-hosting to print the MIPS at the end of execution. The default build process will also be discussed. The second platform is slightly more complex and demonstrates a complete Cortex-M3 Embedded Platform which is executing uc/OS-II.

### *A. A Simple Cortex-M3 Based Single Processor Platform*

After the Cortex-M3 sample platforms' installation, the Cortex-M3 single processor platform will be available in the "OVPsim_single_arm_Cortex-M3" folder under the "Demo" folder in the OVP installation folder's root. The sample platform consists of just one Cortex-M3 processor core connected to a single memory.

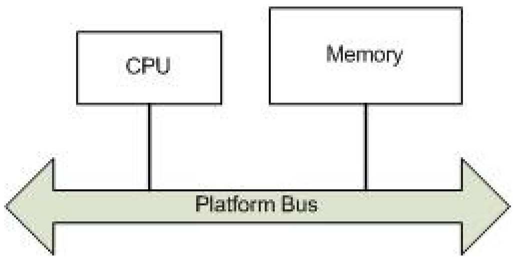


Fig. 2 Block Diagram of the Cortex-M3 Based Single Processor Platform.

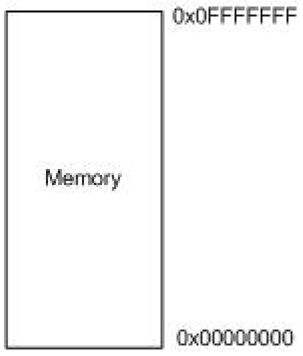
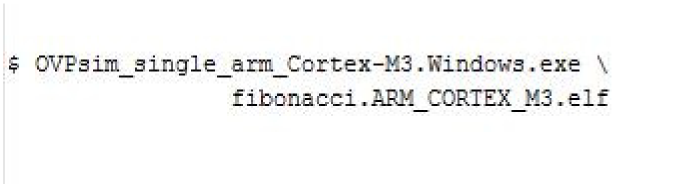


Fig. 3 Memory Map of the Cortex-M3 Based Single Processor Platform.

The compiled platform executable is named OVPsim_single_arm_Cortex-M3.Windows.exe. It takes two parameters, the first one is the application that the processor should run and is mandatory. The second parameter is optional and is the ARMM variant; by default the string "Cortex-M3" is used. For Cortex-M3, the application is required to be in ARM ELF format. As part of the installation, there are four sample applications available in this folder; Dhrystone benchmark, Fibonacci series, LINPACK benchmark and Peak Speed.

The Fibonacci series is calculated using recursion. The series is calculated for each number from 0 to 37 and the output is printed at the end of each iteration as fib(0), fib(1) ... fib(36), fib(37). The compiled Fibonacci application ELF file, fibonacci.ARM_CORTEX_M3.elf, can be executed by either executing the RUN_fibonacci.bat file or it can be executed by using the following command (Fig. 4):



```
$ OVPsim_single_arm_Cortex-M3.Windows.exe \
                fibonacci.ARM_CORTEX_M3.elf
```

Fig. 4 Executing the Fibonacci Series Application on the Cortex-M3 Based.

The sample single Cortex-M3 based platform can be compiled by executing the "make clean all" command in the platform's root folder. This will result in deletion of all object files and executables already present in the folder, and all source files will be compiled and linked again. If MinGW GCC and Cortex-M3 cross compiler are properly installed, the platform executable and the Cortex-M3 applications will be built. As mentioned earlier, for compiling the sample Cortex-M3 platforms, Code Sourcery "Sourcery G++ Lite Edition for ARM" [7] is required. The platform compilation also yields a dynamic library, OVPsim_single_arm_Cortex-M3.Windows.dll, which contains the complete platform compiled as a shared library which can be loaded into other C or SystemC applications. The platform makefile compiles the platform and the embedded application in two passes, in the first pass, the platform is compiled into an x86 executable. In the second pass, the application code is cross compiled into the ARM ELF format. The make processes for both platform and the application are briefly explained below; they will be disscussed in greater detail when the process of creating a custom platform is discussed. It should be noted that the brief explanation of the make process that follows is based on the default makefile hierarchy.

OVP platforms need to be linked against two libraries. One is a runtime shared library, libRuntimeLoader.dll and the other one is a static library icm.import.a. Both these libraries are present under the bin\Windows folder of the OVP installation. libRuntimeLoader.dll is a dynamic library which is loaded by the compiled platform for running the simulation, libRuntimeLoader.dll in turn runs OVPsim. The static library, icm.import.a, contains all the function stubs for ICM APIs. The rules for compiling the platform are defined in the Makefile.platform file and the platform makefile just includes that file from the OVP installation folder. Makefile.platform includes two more makefiles, Makefile.include and Makefile.common. Makefile.include sets up variables that define the suffixes for library and executable files (e.g. *so* for Linux, *dll* for Windows etc), paths for all platform include files (C header files) and libraries containing OVP API stubs (ICM, VMI etc.). Makefile.common sets up the work directory and if VLNV (Vendor Library Name and Version) is used, the path of the VLNV root folder. This platform doesn't use VLNV, Makefile.common checks three variables for VLNV, based on which configuration is selected, setting NOVLNV=1 disables VLNV, setting SYSTEMVLNV=1 compiles the platform and places it in the system VLNV root folder. If NOVLNV is not set to 1, either SYSTEMVLNV should be enabled or VLNVROOT should be set to the path where the platform is required to be compiled to. Besides including Makefile.include and Makefile.common, Makefile.platform also defines the rules for compiling the platform. Once all the include paths, library paths and the work directory path have been setup, Makefile.platform specifies linker and compiler options and builds the platform executable and library.

The rules for compiling the embedded application are all defined in the makefile in the platform folder itself. The cross compiler configuration, paths, target libraries and linker flags are all specified in the ARM_CORTEX_M3.makefile.include file, which is included by the platform makefile. The cross compiled object files are then linked in to their respective ELF files.

## *B. A Cortex-M3 Based Single Processor Platform running uc/OS-II*

This platform demonstrates the execution of an embedded operating system in an OVP simulation using Cortex-M3. The platform consists of one main memory, one stack memory and an LED peripheral model. The LED peripheral model has three simulated LEDs and on each write to its register, it prints out the status of the LEDs to the console.

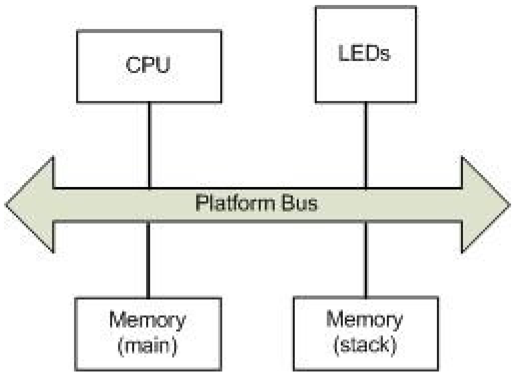


Fig. 5 Block Diagram of the Cortex-M3 Based Single Processor Platform running uc/OS-II.

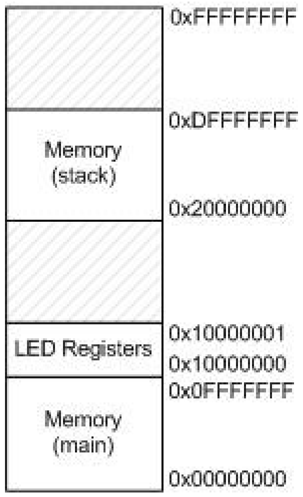


Fig. 6 Memory Map of the Cortex-M3 Based Single Processor Platform running uc/OS-II.

The application running on this platform creates an extended uc/OS-II task, which in turn creates two more tasks. These tasks toggle the LEDs in 1 second, 500 millisecond and 250 millisecond intervals. LED 1 toggles once every second, LED 2 once every 500 milliseconds and LED 3 once every 250 milliseconds. A sample run of this application is shown below with the timing information.

```
Info (LED) /led: -- -- -- at 0.002000
Info (LED) /led: ON ON ON at 0.002000
Info (LED) /led: ON ON ON at 0.002000
Info (LED) /led: ON -- ON at 0.002000
Info (LED) /led: -- -- ON at 0.002000
Info (LED) /led: -- -- -- at 0.002000
Info (LED) /led: -- ON -- at 0.002000
Info (LED) /led: ON ON -- at 0.252000
Info (LED) /led: -- ON -- at 0.502000
Info (LED) /led: -- -- -- at 0.502000
Info (LED) /led: ON -- -- at 0.752000
Info (LED) /led: ON ON ON at 1.002000
Info (LED) /led: -- ON ON at 1.002000
Info (LED) /led: ON ON ON at 1.002000
Info (LED) /led: ON -- ON at 1.002000
Info (LED) /led: -- -- ON at 1.252000
Info (LED) /led: ON -- ON at 1.502000
Info (LED) /led: ON ON ON at 1.502000
Info (LED) /led: -- ON ON at 1.752000
Info (LED) /led: ON ON ON at 2.002000
Info (LED) /led: ON -- -- at 2.002000
```

Fig. 7 Output of the uc/OS-II Application executing on the Cortex-M3 Based Single Processor Platform.

## V. Creating a custom platform using OVP

The demo platform created for the purpose of this study is a relatively simple configuration as well. It consists of a Cortex-M3 processor, a 16 kilobyte data memory (read and write), a 64 kilobyte instruction memory (read only), a two byte 'debug' memory location (write only) and an ARM Prime Cell PL011 UART [9]. The platform also supports debugging the application using GDB (GNU debugger) through RSP (Remote Serial Port) protocol. The debug memory can be used by the application to send debug messages to the simulation environment for printing to the console and the reason and principle of operation is explained later on in the text. The following sections will go through, in sequence, through the various steps of creating the simulation platform and the embedded application. OVP allows platforms to be written in C, C++ or SystemC. The demo platform has been developed in C and the platform, in essence, is a standard C program which invokes the ICM API to create the platform and is linked with OVP libraries.

### A. Introduction to Platform Concepts

As mentioned earlier, platforms are created using the Innovative CPU Manager (ICM) API, but before discussing how to proceed with creating a platform and instantianting a processor or any other models and using them in the platform, some information needs to be considered about how the platform locates models and how they are configured.

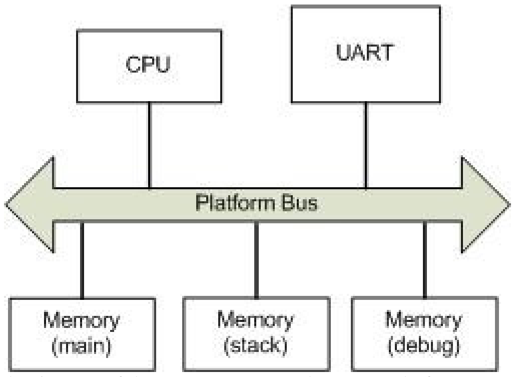


Fig. 8 Block Diagram of the Cortex-M3 Based Custom Platform.

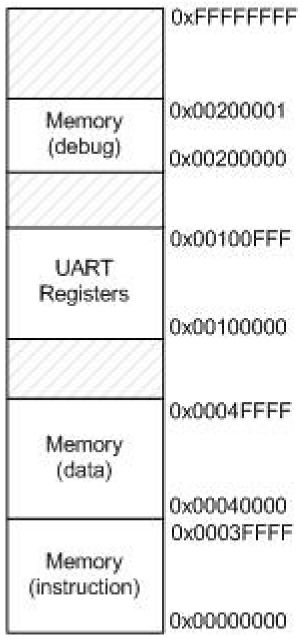


Fig. 9 Memory Map of the Cortex-M3 Based Custom Platform.

Processor and peripheral models can be specified by including the absolute path of the model directly at compile time but this makes the platform non-portable. It is recommended to use the VLNV (Vendor Library Name and Version) information and then at run-time reference the root of a library to use. The icmGetVlnvString(...) API constructs a path using the VLNV information based on the IMPERAS_VLNV environment variable to specify the root of the library and select a specific model. Each model has a set of attributes that can be either a string, double or an unsigned 64-bit integer. Each attribute is identified by a unique name specified as a string. At the time of writing this report, no formal documentation was available which explains the attributes of default OVP models. Each model has an XML file associated with it which contains some information about the model including a list of attributes, the contents of these XML files can also be seen on the library page of OVP world [3]. Due to the open source nature of default OVP models, the attributes can also be located in the source files. The process of initializing attributes involves getting an empty attribute list using the icmNewAttrList() API. This attribute list, along with the attribute name and value are then passed to either of icmAddDoubleAttr(...), icmAddStringAttr(...) or icmAddUns64Attr(...).

### B. Initializing the Platform

The ICM platform is initialized with a call to the icmInit(...) API call. This function takes three parameters, the first one is a list of attributes for the simulation such as the verbosity of the simulation and how the simulation is terminated (e.g. on a Ctrl-C). The last two parameters specify the GDB debugging options for the platform; with the second parameter being the GDB communication protocol (currently only "rsp" is supported) and the last parameter being the GDB port number. Platform initialization using the icmInit(...) API is shown below (Fig, 10):

```
// Initialize the platform.
icmInit(ICM_VERBOSE | ICM_STOP_ON_CTRLC,
        "rsp",
        9990);
```

Fig. 10 Initializing a Platform.

## C. *Instantiating a Processor*

For instantiating a processor, the model for the processor itself and the semi hosting model are required. The semi hosting model is used in the platform even if semi hosting is not used by the embedded application, this way the platform doesn't need to be changed if the application changes. The VLNV strings are initialized for the Cortex-M3 (also known as ARMM) model and semi-hosting using the icmGetVLNVString (...) API as follows (Fig,. 11):

```
// Initialize the processor model paths.
char * arm_model =
   icmGetVlnvString(NULL,
                    "arm.ovpworld.org",
                    "processor",
                    "armm",
                    "1.0",
                    "model");

char *arm_semihost =
   icmGetVlnvString(NULL,
                    "arm.ovpworld.org",
                    "semihosting",
                    "armNewlib",
                    "1.0",
                    "model");
```

Fig. 11 Initializing the Processor Model Paths.

A brief discourse on the parameters of icmGetVlnvString(...) is in order here. The first parameter specifies the path to the root folder of the VLNV database. If NULL is specified here, the default VLNV path of the OVP installation is used. The rest of the parameters specify the vendor name, library name, model name, model version and object type, respectively. The object type is mostly relevant to peripherals which have an intercept (semi hosting) model and a PSE (Peripheral Simulation Engine) model, both of which are distinctly identified by the object type. After initializing the path for both the processor model and the semi hosting model, the next step is to setup the processor attributes. The processor variant and endianness need to be selected. Since the platform is using Cortex, the variant is set to "Cortex-M3". At this time, the "armm" model only supports one variant, Cortex-M3, of the Cortex line of processors. The processor attributes are configured as shown below (Fig. 12):

```
// Initialize processor attributes.
icmAttrListP icm_attr = icmNewAttrList();

icmAddStringAttr(icm_attr,
                 "endian",
                 "little");


icmAddStringAttr(icm_attr,
                 "variant",
                 "Cortex-M3");
```

Fig. 12 Initializing the Processor's Attributes.

The next step is to instantiate the processor model using the icmNewProcessor() API as follows (Fig. 13):

```
// Instatiate the processor.
icmProcessorP cpu =
   icmNewProcessor("Cortex-M3",
                   "armm",
                   0,
                   0,
                   32,
                   arm_model,
                   "modelAttrs",
                   ICM_ATTR_RELAXED_SCHED,
                   icm_attr,
                   arm_semihost,
                   "modelAttrs");
```

Fig. 13 Instatiating a Processor.

Debugging via RSP using GDB is enabled by using the icmDebugThisProcessor(...) API, which takes an icmProcessorP variable (like the processor just created above) as its parameter. This is done as follows (Fig. 14):

```
// Make the processor debuggable.
icmDebugThisProcessor(cpu);
```

Fig. 14 Making the Processor Debuggable.

This makes the processor debuggable using GDB via the RSP port configured during platform initialization.

## D. *Creating Processor Memories*

There are three memories attached to the processor in this platform, a 64 kilobyte instruction memory (read only), a 16 kilobyte data memory (read/write) and a special two byte debug memory (write only). The debug memory location can be used by the embedded software executing on the platform to log messages to the simulator's output. It consists of only two bytes, the first byte is where data is written to and the second byte is used as a trigger of sorts, where writing 0x00 to this location indicates to the platform that a new write has started and writing 0xFF to it indicates that the log message write has ended. A callback is attached for writes to the debug memory address range, which buffers data written to the data byte when it sees 0x00 written to the trigger byte and dumps the data to the console when it sees 0xFF written to the trigger byte.

Using the default OVP memory model, memories are created using the icmNewMemory(...) API call. This API takes three parameters, the first one is a unique name for this memory, the second parameter indicates the read, write and execute permissions for this memory and the last parameter specifies the high address of the memory or size of the memory minus one. The instruction, data and debug memories are created as follows (Fig. 15):

```
// Create memories.
icmMemoryP instr_mem =
   icmNewMemory("instr_mem",
                ICM_PRIV_RX,
                0x0003FFFF);

icmMemoryP data_mem =
   icmNewMemory("data_mem",
                ICM_PRIV_RWX,
                0x0000FFFF);

icmMemoryP dbg_mem =
   icmNewMemory("dbg_mem",
                ICM_PRIV_W,
                0x00000001);
```

Fig. 15 Creating the Instruction, Data and Debug Memories.

The ICM_PRIV_RX specified as the second parameter of the instruction memory configures this as a read and execute only memory, similarly ICM_PRIV_RWX configures the data memory as read, write and execute and ICM_PRIV_W configures the special debug memory as write only. The callback for the debug memory is specified using the icmAddWriteCallback(...) as follows (Fig. 16):

```
// Add write callback for the debug
// memory region.
icmAddWriteCallback(cpu,
                    0x00200000,
                    0x00200001,
                    dbg_write_hook,
                    NULL);
```

Fig. 16 Adding a Write Callback for Debug Memory.

The first parameter is the processor created earlier. The second and the third parameters are the start and end addresses; the address mapping of the memory will be discussed in the section where the procedure for creating the platform's main bus and connecting all memories and peripherals to the processor is explained. The third parameter is the callback function and the last parameter is a pointer to any user data to be passed to the callback. The callback can be created using the callback prototype macro ICM_MEM_WRITE_FN(...) or the complete prototype can be used. Both methods are shown below (Fig. 17).

```
// Write callback using the macro.
ICM_MEM_WRITE_FN(dbg_write_hook)
{
   ...
}

...or...

// The complete write callback.
void dbg_write_hook(icmProcessorP cpu,
                    Addr start_addr,
                    uint32_t num_bytes,
                    const void * value,
                    void * user_data,
                    Addr end_addr)
{
   ...
}
```

Fig. 17 Adding Write Callbacks.

## E. *Instantiating the ARM Prime Cell PL011 UART Model*

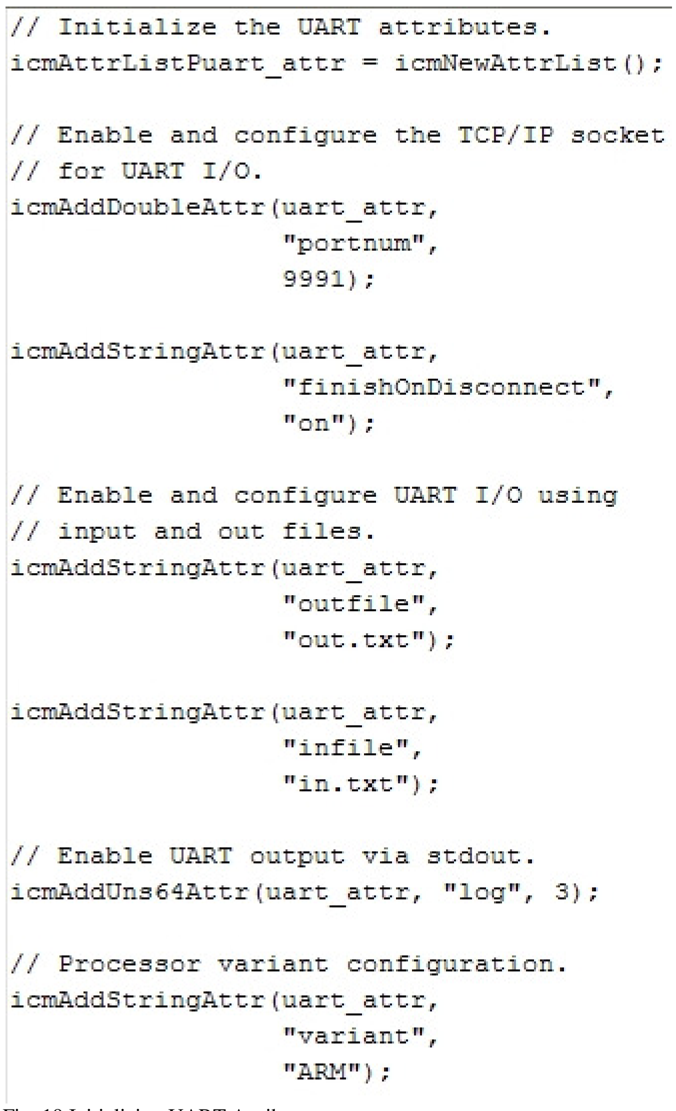
As mentioned earlier, peripherals, like processors have a semi hosting model as well, but it is referred to as an intercept model or intercept library. The intercept model is used by the peripheral model to interact with host operating system for file I/O, TCP/IP sockets etc. The peripheral model itself is known as the PSE (Peripheral Simulation Engine) model. The process of instantiating a peripheral is very similar to that of instantiating a processor. First the path for the PSE and intercept models is initialized using the VLNV method and then the attributes are configured for the PL011 UART peripheral (Fig. 18)).

```
// Initialize UART model paths.
char * uart_pse =
   icmGetVlnvString(0, 0,
                    0, "UartPL011",
                    0, "pse");

char * uart_model =
   icmGetVlnvString(0, 0,
                    0, "UartPL011",
                    0, "model");
```

Fig. 18 Initializing the UART Model Paths.

As can be seen from the last parameters of both function calls in Fig. 18, the first call to icmGetVlnvString(...) in the code snippet above initialises the path for the PSE model and the second call to icmGetVlnvString(...) does the same for the intercept model.

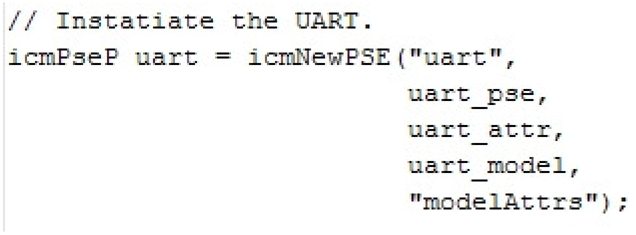
The PL011 UART model provides three modes of input and output. It can use the simulation console for output, files for both input and output and a server socket for both input and output. The server socket can be connected to from the host using any terminal, e.g. telnet. These modes of input and output can be configured via the peripheral's attributes. In addition to the input and out modes, another attribute for the PL011 UART model is the processor variant. These attributes are configured as shown in the code snippet below (Fig. 19):



```
// Initialize the UART attributes.
icmAttrListPuart_attr = icmNewAttrList();

// Enable and configure the TCP/IP socket
// for UART I/O.
icmAddDoubleAttr(uart_attr,
                 "portnum",
                 9991);

icmAddStringAttr(uart_attr,
                 "finishOnDisconnect",
                 "on");

// Enable and configure UART I/O using
// input and out files.
icmAddStringAttr(uart_attr,
                 "outfile",
                 "out.txt");

icmAddStringAttr(uart_attr,
                 "infile",
                 "in.txt");

// Enable UART output via stdout.
icmAddUns64Attr(uart_attr, "log", 3);

// Processor variant configuration.
icmAddStringAttr(uart_attr,
                 "variant",
                 "ARM");
```

Fig. 19 Initializing UART Attributes.

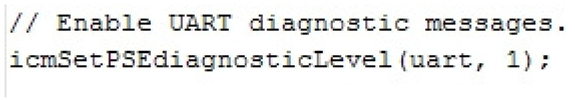
Now the UART PSE can be instantiated as follows (Fig. 20):



```
// Instatiate the UART.
icmPseP uart = icmNewPSE("uart",
                         uart_pse,
                         uart_attr,
                         uart_model,
                         "modelAttrs");
```

Fig. 20 Instatiating the UART.

For debugging purposes, UART diagnostics can be enabled by calling icmSetPSEdiagnosticLevel(...):

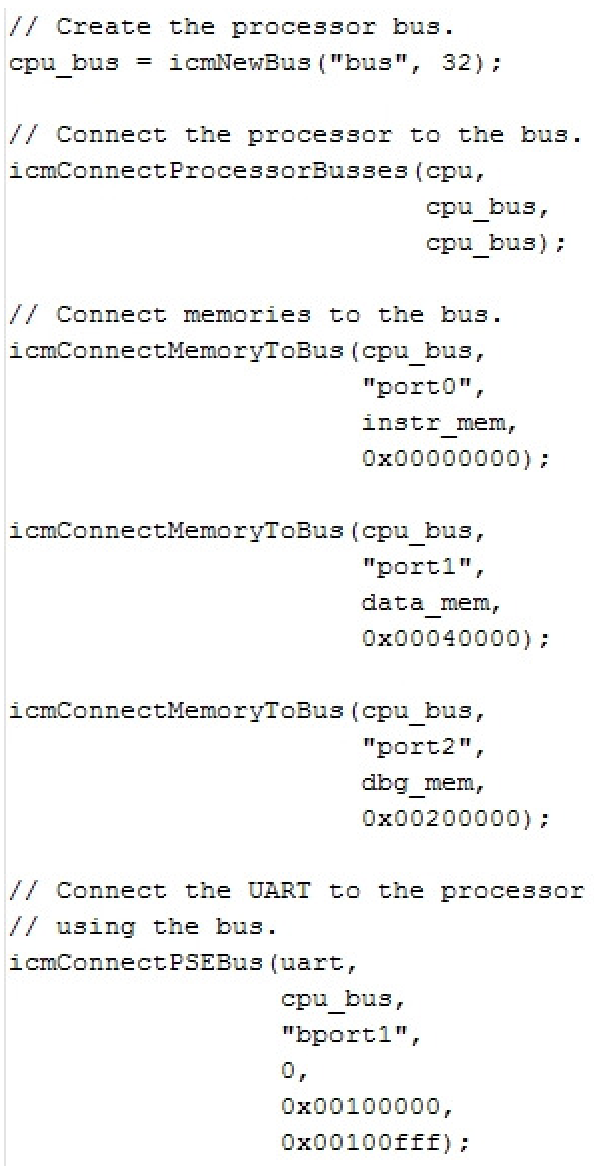


```
// Enable UART diagnostic messages.
icmSetPSEdiagnosticLevel(uart, 1);
```

Fig. 21 Enabling UART Diagnostics.

## *F. Creating a Bus and Connecting Memories and Peripherals to the CPU*

Now that the CPU, memories and peripherals have all been created and configured, they can be connected together to create the platform. First, the bus needs to be created using the icmNewBus(...) API call. It takes two parameters as inputs, the first one is a unique name for the bus and the second one is the width of the bus. After creating the bus, the icmConnectProcessorBusses(...) API function is used to connect the bus to the CPU. It takes an icmProcessorP variable (the platform processor) as its first parameter and two bus objects, instruction and data, as the second and third parameters, respectively. Since, in this platform, the same bus is used for both instructions and data, the same bus object can be specified for both buses. To connect memories to the bus, the icmConnectMemoryToBus(...) API call is used. This function takes the bus object, the unique port name for the memory, the memory object and the memory start address as parameters, respectively. Peripherals can be connected to the bus using the icmConnectPSEBus(...) function, which takes the peripheral (PSE) object, bus object, unique port name for the peripheral, peripheral start address and peripheral end address as it parameters, respectively, this is shown below (Fig. 20).

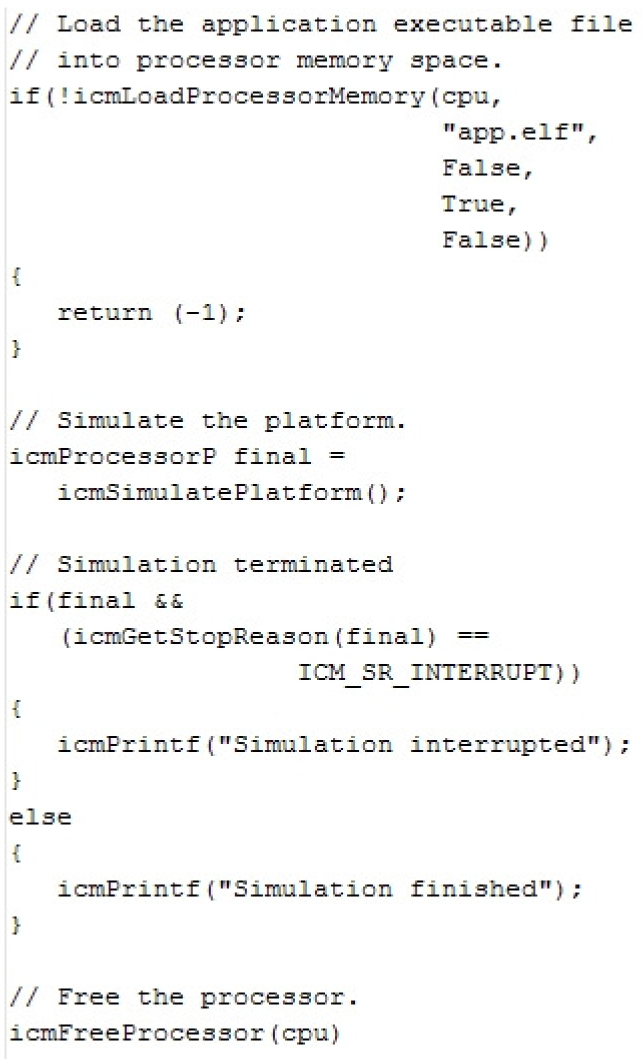


```
// Create the processor bus.
cpu_bus = icmNewBus("bus", 32);

// Connect the processor to the bus.
icmConnectProcessorBusses(cpu,
                          cpu_bus,
                          cpu_bus);

// Connect memories to the bus.
icmConnectMemoryToBus(cpu_bus,
                      "port0",
                      instr_mem,
                      0x00000000);

icmConnectMemoryToBus(cpu_bus,
                      "port1",
                      data_mem,
                      0x00040000);

icmConnectMemoryToBus(cpu_bus,
                      "port2",
                      dbg_mem,
                      0x00200000);

// Connect the UART to the processor
// using the bus.
icmConnectPSEBus(uart,
                 cpu_bus,
                 "bport1",
                 0,
                 0x00100000,
                 0x00100fff);
```

Fig. 22 Creating a New Bus and Connecting the Peripherals and Memories to the Processor.

### G. *Setting-up the Platform for Simulation*

After the processor, memories and peripherals have been created and connected, the embedded application can be loaded into the processor memory using the icmLoadProcessorMemory(...) which takes the CPU object, the path to the application file and three Boolean parameters related to the loading of the application, viz. loading the application using the virtual or physical addresses specified in the ELF (True = Physical), output information related to the applications sections (True = Verbose) and the program counter before running the application (True = Start address in the object file). The simulation can then be started with a call to icmSimulatePlatform(). One the simulation ends, icmGetStopReason(...) can be used to identify why the simulation ended. Once the simulation has ended, icmFreeProcessor(...) should be used to free the CPU object (Fig. 23).



```
// Load the application executable file
// into processor memory space.
if(!icmLoadProcessorMemory(cpu,
                           "app.elf",
                           False,
                           True,
                           False))
{
   return (-1);
}

// Simulate the platform.
icmProcessorP final =
   icmSimulatePlatform();

// Simulation terminated
if(final &&
   (icmGetStopReason(final) ==
                ICM_SR_INTERRUPT))
{
   icmPrintf("Simulation interrupted");
}
else
{
   icmPrintf("Simulation finished");
}

// Free the processor.
icmFreeProcessor(cpu)
```

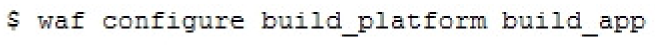
Fig. 23 Setting-up the Platform for Simulation.

### H. *The Embedded Application for this Platform*

The embedded application developed for this platform was designed such that it can be used directly on actual hardware with minimal changes, so no semi hosting was used and the stack setup, interrupt vector installation for Cortex-M3 and application initialization are done in an assembler startup file. The application itself is a serial port echo program, it initializes the UART and then goes into a loop where it reads data from the serial port and transmits the same back through the serial port. It keeps doing this till it sees a Ctrl-D (the break character) on the serial port. The debug memory region mentioned earlier is used to print out some execution information.

### I. *Compiling the Platform and the Embedded Application*

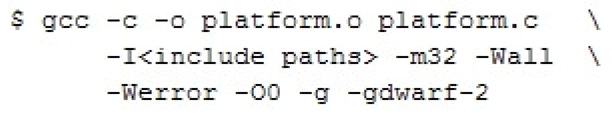
The build process for this platform and the embedded application is very similar to what was explained earlier. The only difference is that the embedded application has a startup assembler file which is built using the assembler. There is also a custom linker file. The linker file sets up the memory map for the application. Both the platform and the embedded application are compiled using a single WAF script [6]. Even though GNU 'make' [11] is more than adequate for building both the embedded application and the platform and 'make' was used in the earlier phase of development, a WAF script was used for compiling the final platform because of better readability, maintainability and easier management of the build process. The platform compiles to an executable and the embedded application to an ELF file. Regardless of which process is used to automate the build, the steps are the same. The WAF command line for building the platform and the application is (Fig. 24):



```
$ waf configure build_platform build_app
```

Fig. 24 Building the Custom Platform and Embedded Application.

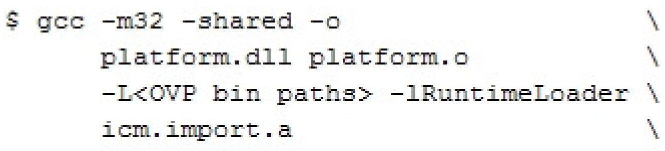
For the sake of completeness, the expanded build command lines for both the platform and the application are listed here as well. The build command line for the compiling the platform object file is (Fig. 25):



```
$ gcc -c -o platform.o platform.c   \
      -I<include paths> -m32 -Wall  \
      -Werror -O0 -g -gdwarf-2
```

Fig. 25 Compiling the Platform Object File.

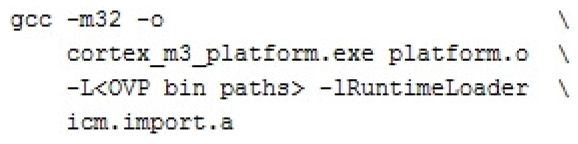
The required include paths for building the platform are the paths to the OVP common include files, OVP public host include files and OVP proprietary host include files. The build command line for the platform shared library is (Fig. 26):



```
$ gcc -m32 -shared -o                    \
      platform.dll platform.o            \
      -L<OVP bin paths> -lRuntimeLoader  \
      icm.import.a                       \
```

Fig. 26 Compiling the Platform Shared Library.

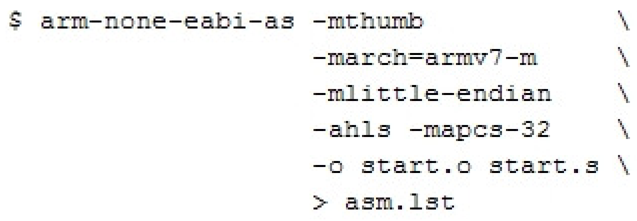
And, finally, the command line for compiling the platform executable (Fig. 27):



```
gcc -m32 -o                           \
    cortex_m3_platform.exe platform.o \
    -L<OVP bin paths> -lRuntimeLoader \
    icm.import.a
```

Fig. 27 Compiling the Platform Executable.

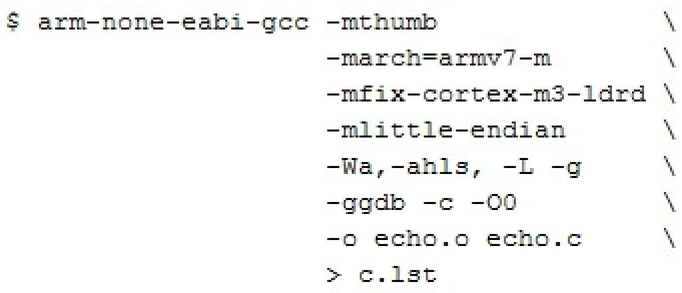
For building the embedded application, the startup file needs to be assembled, the application source file needs to be compiled and both the objects need to be linked together. The startup file can be assembled using the following command line (Fig. 28):



```
$ arm-none-eabi-as -mthumb          \
                   -march=armv7-m    \
                   -mlittle-endian   \
                   -ahls -mapcs-32   \
                   -o start.o start.s \
                   > asm.lst
```

Fig. 28 Compiling the Platform Executable.

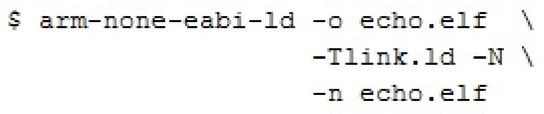
The application source file can be compiled using the following command line (Fig. 29):



```
$ arm-none-eabi-gcc -mthumb              \
                    -march=armv7-m        \
                    -mfix-cortex-m3-ldrd \
                    -mlittle-endian       \
                    -Wa,-ahls, -L -g      \
                    -ggdb -c -O0          \
                    -o echo.o echo.c      \
                    > c.lst
```

Fig. 29 Compiling the Application Source File.

Finally, the embedded application object files can be linked together as follows (Fig. 30):



```
$ arm-none-eabi-ld -o echo.elf  \
                   -Tlink.ld -N \
                   -n echo.elf
```

Fig. 30 Linking the Embedded Application.

The -Tlink.ld option specifies the link.ld linker file.

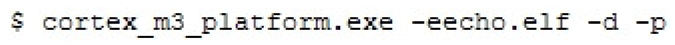
### J. *Executing the Simulation*

The platform executable takes a few command line parameters which can be used to configure options such as the embedded application that should be executed on the platform, the ARMM variant, if GDB debugging should be enabled and the RSP port, if standard out (stdout) should be used for UART output, if the TCP/IP socket should be used for UART input and output and the port number and if an output file should be used for UART output and the file name. The platform doesn't support input using a file; the example shown above (Fig. 19) was for demonstrative purposes only. The complete usage options for the platform are listed below (Table 1):

TABLE 1 COMMAND LINE OPTIONS FOR THE PLATFORM EXECUTABLE.

| | |
|---|---|
| **-e<application>** | The embedded application to run on the platform. |
| **-c<processor>** | Optional argument, the default variant is set to "Cortex-M3". |
| **-d<port (optional)>** | Debug the application using GDB. Port number is optional, default is set to "9999". |
| **-s** | Use stdout for UART output |
| **-p<port (optional)>** | Use a network connection for UART output. Port number is optional, default is set to "17321". |
| **-f<filename (optional)>** | Use a log file for UART output. File name is optional, by default "uart.log" is is created in the current directory. |

A sample command line for executing the platform simulation with GDB debugging enabled and TCP/IP connection for UART input/output enabled with default ports would be (Fig. 31):

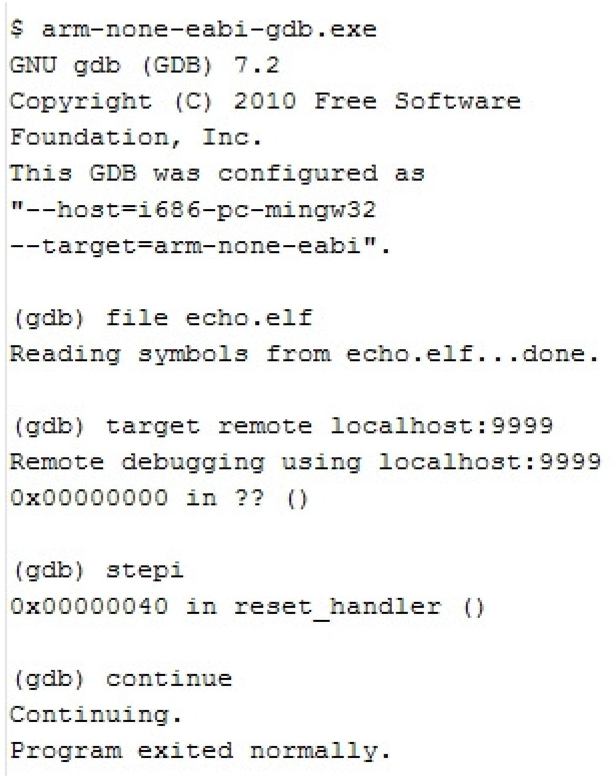


```
$ cortex_m3_platform.exe -eecho.elf -d -p
```

Fig. 31 Sample Command Line for the Platform.

At the start of each simulation OVP prints out the sections of the object file and their addresses. At the end of each simulation, the UART event log is printed out and OVP prints out some simulation statistics such as the MIPS, final program counter, number of simulated instructions, simulation time and real time.

The platform can be debugged using standard GDB commands [12]. A sample GDB debugging session for this platform is shown in Fig. 32. A graphical frontend for GDB, like DDD [13] or Insight [14], can be used as well. The Eclipse C Development Tools (CDT) [15] also support GDB debugging.



```
$ arm-none-eabi-gdb.exe
GNU gdb (GDB) 7.2
Copyright (C) 2010 Free Software
Foundation, Inc.
This GDB was configured as
"--host=i686-pc-mingw32
--target=arm-none-eabi".

(gdb) file echo.elf
Reading symbols from echo.elf...done.

(gdb) target remote localhost:9999
Remote debugging using localhost:9999
0x00000000 in ?? ()

(gdb) stepi
0x00000040 in reset_handler ()

(gdb) continue
Continuing.
Program exited normally.
```

Fig. 32 Debugging the Embedded Application using GDB.

## VI. ALTERNATIVES TO OPEN VIRTUAL PLATFORMS

There are many virtual platforms available for emulating processor based platforms. Some of the more popular examples include QEMU, SoCLib, GreenSoCs QEMU-SystemC and Wind River SimICs. An in-depth analysis and detailed comparison of these platforms with Open Virtual Platforms is beyond the scope of this study, so an overview of their advertised features and their comparison to Open Virtual Platfrms is given here.

**QEMU** is a generic, open source processor emulator and virtualizer. It can use the Xen Hypervisor or the KVM Kernel for fast virtualization of the processor instruction set. Additionally, complete platforms, such as the Nokia N800 tablet, the Beagle Board, Versatile Base Board, Real View Emulation Base Board or the ARM Integrator CP can be simulated as well, with models that emulate peripherals such as serial ports, graphical LCDs, network interfaces etc. QEMU is considered the "default" choice for emulating ARM based platforms and operating systems. Emulators for popular mobile operating systems like Android and MeeGo are, in fact, powered by QEMU. Besides ARM, QEMU also supports MIPS, x86 and ARM variants like Intel's XScale.

Developing new platforms or modeling new peripherals with QEMU, however, is not as straightforward as it is with Open Virtual platforms. For example, for creating a new processor model under QEMU, a developer would need to learn about technologies like Code Morphing. Similarly, creating a new peripheral or a platform requires a working knowledge of the way QEMU works and its source code. Additionally, only single processor platforms can be emulated using QEMU.

**GreenSoCs QEMU-SystemC** is a modification to QEMU to allow mixed HW/SW simulations. The use of QEMU allows simulating entire and complex systems (Intel-PC, ARM platforms, etc). With QEMU-SystemC, SystemC described modules can be simulated as if they were connected to the system bus (PCI in Intel platforms, AMBA in ARM platforms etc.). QEMU-SystemC solves the problem of the difficulty of adding new peripherals or creating virtual platforms using QEMU. In addition, standard SystemC backend and frontend tools can be used to analyze platform and simulation data. But since the processor virtualizer is still based on QEMU, the difficulty in creating new processor models and the limitation of single processor systems, are still very much there. It is also worth mentioning that GreenSocs also provides solutions based on Open Virtual Platforms.

**SoCLib** is an open platform for virtual prototyping of multi-processor or single processor SoCs. The core of SoCLib is a library of SystemC simulation models for virtual components. SystemC is used to create all simulation models, which can then be simulated with standard SystemC simulation environments. Two types of models are available for each virtual component (IP-Core); CABA (Cycle Accurate/Bit Accurate) and TLM-DT (Transaction Level Modeling with Distributed Time). SoCLib, not unlike Open Virtual Platforms, is still a relatively new project and was developed by a collaboration of thirteen French laboratories under the supervision of the French National Center for Scientific Research. Though the speed of execution would be a concern with cycle accurate modes, the TLM-DT models might provide better performance for embedded application development. Even though this technology seems to be aimed primarily at hardware model verification, it looks promising, especially due to the open nature of the platform and standards.

The **Wind River SimICs** platform is probably the most comprehensive virtual prototyping platform for embedded systems. There is a plethora of peripheral and processor models available for SimICs. But due to the proprietary nature of this platform and the models and the restrictive licensing, this platform has not seen widespread adoption, even though it has been around since 1994.

Other platforms worth mentioning are GXemul, Design Ware and Platform Architect. GXemul is a single processor open source platform emulator very similar to QEMU and supports ARM, MIPS, M88K, PowerPC, and SuperH processors. Design Ware and CoWare Platform Architect are proprietary tools developed by Synopsis (Platform Architect was initially developed by CoWare, which is now a part of Synopsis).

## VII. Conclusion

Open Virtual Platforms can potentially be the first true System Level Virtual Prototyping (SLVP) platform that has the capability to be not just a platform for hardware development, but also for software development. That means "it could be the first general purpose ESL modeling system that will form the cornerstone of complete ESL flows into both the hardware and software communities. While this has been done in specialized areas before, such as DSP design, it has never been solved in the more general case" [1]. OVP certainly fills a void in the spectrum of EDA/ESL tools which lacked an open standard. In addition, it has created a market for commercial prototypes which model complex platforms and peripherals. Imperas' OVP-based partnerships with major ESL/EDA/SoC industry stake holders such as Cadence, MIPS, Ten Silica and Forte is a proof of its potential. Additionally, most OVP processor models have been verified by the respective IP vendors. Besides the obvious industrial and industrial applications of Open Virtual Platforms, there are numerous academic and research areas where Open Virtual Platforms can be used. Embedded Systems Design Space Exploration and Hardware Software Co-verification and Design are prime candidates for research areas where OVP can be useful. Even as a general purpose teaching tool for introductory or advanced Computer Architecture courses, OVP can provide the required platform for experimentation with various basic concepts. Additionally, for Embedded Systems or Microprocessor courses, OVP can provide the required "hands-on" experience to students in the absence of actual hardware. Open Virtual Platforms are already being employed by researchers and teachers at various universities like University of Southampton, University of Washington and IIT Delhi.

## References

[1] Brian Bailey, "System Level Virtual Prototyping and Open Virtual Platforms," White Paper, June 2008, Brian Bailey Consulting

[2] Hybrid Simulation Framework for Virtual Prototyping Using OVP, SystemC & SCML - A Feasibility Study, Priya Agrawal, IIT Delhi, 2009.

[3] Open Virtual Platforms Webpage, http://www.ovpworld.org/

[4] OVPsim License Request Webpage, http://www.ovpworld.org/licensekey.php

[5] OVPlib Open Source Project, http://code.google.com/p/ovplib/

[6] WAF Meta Build System, http://code.google.com/p/waf/

[7] Sourcery G++ Lite Edition, http://www.codesourcery.com/sgpp/lite/arm

[8] YAGARTO, Yet Another GNU ARM Toolchain, http://www.yagarto.de/

[9] ARM Prime Cell UART PL011 Technical Reference Manual, http://infocenter.arm.com/help/topic/com.arm.doc.ddi0183g/

[10] Cortex-M3 Technical Reference Manual, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0337e/index.html

[11] GNU 'make', http://www.gnu.org/software/make/manual/make.html

[12] GNU Debugger Online Manual, http://www.delorie.com/gnu/docs/gdb/gdb_toc.html

[13] GNU DDD, http://www.gnu.org/software/ddd/

[14] Insight GDB UI, http://sourceware.org/insight/

[15] Eclipse C Development Tools (CDT), http://www.eclipse.org/cdt/