OVP Debugging Applications with GDB User Guide

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

Author:	Imperas Software Limited
Version:	1.7
Filename:	OVPsim_Debugging_Applications_with_GDB_User_Guide.doc
Project:	OVP Debugging Applications with GDB User Guide
Last Saved:	Monday, 22 June 2020
Keywords:	

Copyright Notice

Right to Copy Documentation

Destination Control Statement

Disclaimer

Table of Contents

1 Preface

This document describes how to debug an application running on the OVP simulator using the Gnu debugger, GDB.

1.1 Notation

`Code` Code and command extracts

1.2 Related OVP Documents

- CpuManager and OVPsim User Guide

2 Introduction

The *CpuManager and OVPsim User Guide* describes how platforms containing any number of processor models can be constructed. This document describes how to debug an application running on *one* processor in such a platform while it is simulating using the freely-available OVPsim simulation environment. OVPsim supports single-processor debugging with the Gnu debugger (GDB) via the Remote Serial Protocol (RSP). Advanced multi-processor debug facilities are available in Imperas commercial products.

2.1 Prerequisites

This documentation is supported by C code samples in an `Examples` directory, available either to download from the www.ovpworld.org website or as part of an Imperas installation.

GCC Compiler Versions

Linux32	4.5.2	i686-nptl-linux-gnu (Crosstool-ng)
Linux64	4.4.3	x86_64-unknown-linux-gnu (Crosstool-ng)
Windows32	4.4.7	mingw-w32-bin_i686-mingw
Windows64	4.4.7	mingw-w64-bin_i686-mingw

For Windows environments, Imperas recommends using MinGW (www.mingw.org) and MSYS.

The example given in this document uses the opencores OR1K processor model and tool chain, also available to download from the www.ovpworld.org website or as part of an Imperas installation.

3 Debugging Example

3.1 Creating a Debuggable Platform

A suitable single-processor platform example is available in the directory:

```
$IMPERAS_HOME/Examples/SimulationControl/debugWithGDB
```

This uses the freely-available OR1K processor (see http://www.opencores.org/projects.cgi/web/or1k/architecture).

The test hardware definition source is in file `module/module.op.tcl`:

```
ihwnew -name debugWithGDB -stoponctrlc

ihwaddbus -instancename bus -addresswidth 32

#
# Add a processor to do some reading and writing
#
ihwaddprocessor -instancename cpu1 \
                -vendor ovpworld.org -library processor -type or1k -version 1.0
\
                -variant generic \
                -semihostname or1kNewlib
ihwconnect -bus bus -instancename cpu1 -busmasterport INSTRUCTION
ihwconnect -bus bus -instancename cpu1 -busmasterport DATA

#
# Memory on the main bus
#
ihwaddmemory -instancename ram -type ram
ihwconnect -bus bus -instancename ram -busslaveport sp1 -loaddress 0x00000000 -
hiaddress 0xffffffff
```

This creates a definition including an OR1K processor connected to a bus which also contains a memory.
The C OP API code generated by iGen is in the file *module.igen.h* and looks like

```
// instantiate module components
static OP_CONSTRUCT_FN(instantiateComponents) {

    // Bus bus
    optBusP bus_b = opBusNew(mi, "bus", 32, 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(bus_b, "INSTRUCTION"),
                OP_BUS_CONNECT(bus_b, "DATA")
            )
        ),
        OP_PARAMS(
             OP_PARAM_STRING_SET("variant", "generic")
        )
    );

    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 ram

    opMemoryNew(
        mi,
        "ram",
        OP_PRIV_RWX,
        (0xffffffffULL) - (0x0ULL),
        OP_CONNECTIONS(
            OP_BUS_CONNECTIONS(
                OP_BUS_CONNECT(bus_b, "sp1", .slave=1, .addrLo=0x0ULL,
.addrHi=0xffffffffULL)
            )
        ),
        0
    );

}
```

For a full explanation of OVPsim platform construction please see the *iGen Platform and Module Creation User Guide*. This section describes only those aspects of platform construction that relate to debugging.

3.1.1 Specify debug using the Command Line Parser

The platform in this example includes the standard Command Line Parser (CLP). This allows the debug control for the platform to be specified on the command line:

3.1.1.1 Specifying the debugger connection details

The debug port is enabled by specifying the argument *--port <port number>* on the command line. A specific port number may be specified or by setting *port number* to 0 the next available port is opened.

Alternatively, the argument *--gdbconsole* will open a port and connect the default GDB debugger automatically.

3.1.1.2 Nominating the processor for debug

In an OVPsim simulation only a single processor may be connected to a GDB debugger[1]. this requires that the processor is selected using the *--debugprocessor <processor name>*. In this case the processor name is the instance name in the platform, for example *platform/OR1K*

3.1.2 Specify debug using OP API

3.1.2.1 Specifying the debugger connection details

The OP kernel is initialized by calling `opModuleNew`:

```
    optModuleP opRootModuleNew (optModuleAttrP attrs,
                                const char *name,
                                optParamP params)
```

The params argument of opRootModuleNew is used to initialize the simulator. One of the options available, OP_FP_GDBCONSOLE, is to enable the automatic startup and connection of a GDB to a processor in the simulated platform.

```
    opRootModuleNew(0, 0, OP_PARAMS(OP_PARAM_BOOL_SET(OP_FP_GDBCONSOLE, 1)));
```

GDB Remote Serial Protocol (RSP) debugging as supported by OVPsim uses standard operating system sockets on the host running OVPsim and GDB.

3.1.2.2 Nominating the processor for debug

If the processor has one core, it is passed to `opProcessorDebug()`.

```
    opProcessorDebug(processor);
```

[1] The Imperas Professional products allow the ability to attach a GDB debugger to any or all the processors defined in a platform. Imperas also provide alternative debugging solutions.

If it is a multicore device the appropriate core must be located:

```
    optProcessorP sub = opObjectByName(root, MODULE_NAME "/CPU0_P0",
OP_PROCESSOR_EN).Processor; // for example
    opProcessorDebug(sub);
```

Giving an incorrect name causes an error message which lists all the legal names. This is a useful way to find the core names.

3.2 Building the Platform

The OVPsim examples are written to work with GCC and MAKE which are typically available on Linux and can be installed on Windows as part of MinGW and MSYS (see section 2.1.) The example commands below assume you are using a Bash shell on Linux or MSYS.

Take a copy of the debugging example:

```
    cp –r $IMPERAS_HOME/Examples/SimulationControl/debugWithGDB .
```

The test platform can be compiled to produce an executable, `platform.<IMPERAS_ARCH>.exe`, by using `make` in the example directory:

```
    cd debugWithGDB
    make –C module
```

Cross-compile a simple test application for the OR1K processor:

```
    make –C application
```

3.3 Starting Debugging 'gdbconsole'

3.3.1 Running the Platform

Start the OVP simulator with the example platform by running the native platform executable built earlier. This simple platform uses the command line parser to specify the start up of a console in which the correct GDB for the procesor type will be invoked and connected to the platform.

```
    harness.exe –modulefile module/model.${IMPERAS_SHRSUF} --gdbconsole \
             --program application/application.OR1K.elf
```

```
OVPsim (32-Bit) v20150205.0 Open Virtual Platform simulator from
www.OVPworld.org.
Copyright (C) 2005-2015 Imperas Ltd.  Contains Imperas Proprietary Information.
Licensed Software, All Rights Reserved.
Visit www.imperas.com for multicore debug, verification and analysis solutions.
OVPsim  started: Mon  Mar 9 12:28:15 2015


Info (GDBT_PORT) Host: <hostname>, Port: <portnumber>
```

```
Info (GDBT_WAIT) Waiting for remote debugger to connect...
Info (GDBT_CONNECTED) Client connected
```

Once the platform has made a call to `opRootModuleSimulate` (or `opProcessorSimulate`), OVPsim will wait for the debugger connection. The output above shows the host and portnumber being provided in the GDBT_PORT message which is used to connect the automatically invoked GDB.
The GDB displays the current execution location:

```
0x00000100 in start ()
```

3.4 Starting Debugging Manual Attachment

3.4.1 Running the Platform

Start the OVP simulator with the example platform by running the native platform executable built earlier. This simple platform uses the command line parser to specify the port number to use for the debugger connection.

```
    harness.exe –modulefile module/model.${IMPERAS_SHRSUF} --port 0 \
             --program application/application.OR1K.elf
```

A non zero numeric value opens a port on the specified port while the value zero allows OVPsim to choose any free host port.

```
OVPsim (32-Bit) v20150205.0 Open Virtual Platform simulator from
www.OVPworld.org.
Copyright (C) 2005-2015 Imperas Ltd.  Contains Imperas Proprietary Information.
Licensed Software, All Rights Reserved.
Visit www.imperas.com for multicore debug, verification and analysis solutions.
OVPsim  started: Mon  Mar 9 12:28:15 2015


Info (GDBT_PORT) Host: <hostname>, Port: <portnumber>
Info (GDBT_WAIT) Waiting for remote debugger to connect...
```

Once the platform has made a call to `opRootModuleSimulate` (or `opProcessorSimulate`), OVPsim will wait for the debugger connection. The output above shows the host and portnumber being provided in the GDBT_PORT message which will be used to manually connect GDB remote target to this port.

3.4.2 Running GDB

When the OVPsim platform is waiting for a debugger connection we can start the Gnu debugger. GDB executables for OR1K and other processor model architectures provided by OVP are included with the Gnu toolchains available for download from the www.ovpworld.org website.

Start GDB in another shell/terminal:

```
    cd debugWithGDB
    "$IMPERAS_HOME/lib/$IMPERAS_ARCH/CrossCompiler/or32-elf/bin/or32-elf-gdb"
```

The GDB startup banner and prompt will be displayed:

```
GNU gdb 5.3
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-cygwin --target=or32-elf".
(gdb)
```

Now load the simulated application file into GDB to provide symbolic debugging information:

```
(gdb) file application/application.OR1K.elf
Reading symbols from application/application.OR1K.elf...done.
(gdb)
```

3.4.2.1 Connecting GDB to OVPsim

The GDB target command is used to connect GDB to OVPsim:

```
(gdb) target remote localhost:1438
Remote debugging using localhost:1438
0x00000100 in start ()
(gdb)
```

The port number must match the port on which OVPsim is waiting for a connection. Once the connection is made, OVPsim shows a message:

```
Info (GDBT_CONNECTED) Client connected
```

and GDB displays the current execution location:

```
0x00000100 in start ()
```

3.5 An example debug session

We are now able to inspect and control the platform and processor state while simulating the application on OVPsim.

Display a disassembly of the next instruction each time execution stops:

```
(gdb) display /i $pc
1: x/i $pc  0x100 <start>:      l.addi r2,r0,0x0
(gdb)
```

Show processor register values:

```
(gdb) info registers
```

```
            R0       R1       R2       R3       R4       R5       R6       R7
      00000000 00000000 deadbeef deadbeef deadbeef deadbeef deadbeef deadbeef
      ...
(gdb)
```

Step one instruction:

```
(gdb) stepi
0x00000104 in start ()
1: x/i $pc  0x104 <start+4>:    l.addi r3,r0,0x0
(gdb)
```

Show register values again:

```
(gdb) info registers
            R0       R1       R2       R3       R4       R5       R6       R7
      00000000 00000000 00000000 deadbeef deadbeef deadbeef deadbeef deadbeef
      ...
(gdb)
```

Set a breakpoint on the application's main function:

```
(gdb) break main
Breakpoint 1 at 0xf3c: file application/application.c, line 4.
(gdb)
```

Run until we hit a breakpoint:

```
(gdb) continue
Continuing.

Breakpoint 1, main () at application/application.c:4
4           printf("Hello\n");
1: x/i $pc  0xf3c <main+16>:    l.movhi r3,0x0
(gdb)
```

Step over the C printf call

```
(gdb) next
5       }
```

(The printf output is shown in the OVPsim window.)

Finally, run the test application to completion

```
(gdb) continue
Continuing.

Program exited normally.
(gdb)
```

4 Further GDB Connection Information and Features

This section describes some of the other ways in which the simulation platform execution may be started and used.

4.1 RSP Interface

RSP is the gdb (Gnu debugger) Remote Serial Protocol. It allows a debugger to communicate with a simulator on the same host machine or over a network to a simulator on a different host machine. OVPsim and CpuManager support RSP as used by most versions of gdb. They automatically switch to an extended version of RSP to communicate with the Imperas stand-alone multi-core debugger.

4.2 Environment variables

Variable	Type	Purpose
IMPERAS_NO_WAIT	boolean	Do not wait for an RSP connection before starting simulation (but keep listening).
IMPERAS_RSP_PORT	integer	Listen on this port for a debugger (0 means choose a port from the pool)
IMPERAS_RSP_PORT_FILE	filename	If port is chosen from the pool, write the port number in this file
IMPERAS_RSP_WAIT_DISCONNECT	boolean	When disconnected, the simulator waits for a new connection, rather than continuing.
IMPERAS_RSP_FINISH_DISCONNECT	boolean	When disconnected, the simulator finishes rather than waiting.

4.3 Detaching and Reattaching

The stand-alone multi-core debugger can be detached from a simulation. When the detach is performed the simulator may perform one of two operations

1. finish the simulation
2. continue the execution of the software application until the application completes or makes no further progress. A debugger can then be reattached, causing simulation to stop immediately so that debugging can continue.

The default operation is dependent upon the simulator runtime, the OVPsim and CpuManager simulators will free run when the debugger is disconnected.

4.3.1 Modifying simulator behavior on detach

The default behavior of the simulator when a debugger is disconnected can be modified to wait for a further connection or to finish (continue the execution of) the simulation.

4.3.1.1 Wait for next connection

Wait is the suspension of the simulation when the debugger is detached. No further execution will take place and the simulator will wait for a further debugger connection.

Set the environment variable IMPERAS_RSP_WAIT_DISCONNECT before starting the simulation.

4.3.1.2 Finish simulation

The simulation continues until it finishes or a further debugger connection is made when the debugger is detached.

Set the environment variable IMPERAS_RSP_FINISH_DISCONNECT before starting the simulation.

4.4 Enabling a debug port without initial connection

This 'no wait' option allows a simulation platform to be started with a debug port enabled but without the need to connect the debugger prior to simulation starting.
A debugger can be connected at any time but the simulation will start executing immediately.

The debug port is enabled in the normal way and the no wait mode is enabled by using one of the following:

Set the environment variable IMPERAS_NO_WAIT.

Add --nowait into a control file.

Add OP_FP_RSPNOWAIT into the OP Parameters (opParams) of a call to opRootModuleNew.

4.5 Environment Variable Enables Debug Connection

If you have a platform executable it is not always convenient to re-compile the platform in order to enable debugging.
The opening of a debug port can also be accomplished using an environment variable

Set the environment variable IMPERAS_ RSP_PORT to either a port number or to 0 and the next available port will be selected.

4.6 Debugging RSP Connections

When there is an error in the RSP connection additional information can be obtained by enabling logging of the connection.

This log file should be provided to Imperas when reporting a problem with other information about the platform used.

Set the environment variable IMPERAS_RSP_LOG_FILE to a file into which transactions over the RSP connection will be written.

5 Creating a Debuggable SystemC/TLM2.0 Platform

When an OVP model is used within a SystemC TLM2.0 platform it may still be debugged using the RSP connection.

A suitable single-processor platform example is available in the directory:

```
$IMPERAS_HOME/Examples/SimulationControl/debugSystemC_TLM2.0WithGDB
```

This uses the freely-available OR1K processor (see http://www.opencores.org/projects.cgi/web/or1k/architecture).

The test platform source is in file `platform/platform.cpp`:

```
class TLM2Platform : public sc_core::sc_module {

  public:
    TLM2Platform (sc_core::sc_module_name name);

    tlmModule        Platform;
    tlmDecoder       bus1;
    tlmRam           ram1;
    tlmRam           ram2;
    or1k             cpu1;
    extension        semihostlib;

    params platformParams() {
        params p;
        p.set("remotedebugport", (Uns32)0);
        return p;
    }

}; /* TLM2Platform */

TLM2Platform::TLM2Platform ( sc_core::sc_module_name name)
    : sc_module (name)
    , Platform ("", platformParams())
    , bus1     (Platform, "bus1", 2, 2)
    , ram1     (Platform, "ram1", 0x000FFFFF)
    , ram2     (Platform, "ram2", 0x0000FFFF)
    , cpu1     (Platform, "cpu1")
    , semihostlib (cpu1, opVLNVString (NULL, "ovpworld.org", "semihosting",
"or1kNewlib", "1.0", OP_EXTENSION, 1), "semihostlib")
…
```

For a full explanation of OVPsim platform construction please see the *iGen Platform and Module Creation User Guide*. This section describes only those aspects of platform construction that relate to debugging.

5.1.1 Nominating the debugged processor

The processor object method debugThisProcessor is called from sc_main:

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

```
    session s;

…

    TLM2Platform top("top");             // instantiate example top module
…
    // Specify the debug processor.
    top.cpu1.debug();
…
```

###