# GNATemulator Documentation

***Release 2018 (20180524)***

**AdaCore**

May 25, 2018

Contents:

CHAPTER

ONE

# INTRODUCTION

## 1.1 About GNATemulator

Simulators are useful tools in many respects. Installation, setup and deployment to a development team will prove to be easier and quicker than with a real target. Simulator usage on a native platform will also bring more flexibility in the development of the application.

GNATemulator is particularly suited for functional testing and unit testing. The product comes with a complete simulation environment and related BSP for non bareboard systems such as VxWorks 653. It allows for efficient target code execution. It simulates a simple board and does not aim at complete board simulation. This should be mostly transparent to the application developer as the differences between the real target and the simulator should be handled and hidden by the platform provider.

## 1.2 Product Content

The product contains four components:

- **The Simulator**: This is the heart of the product. The main tool is called `gnatemu` and is prefixed by the relevant target name. Under the hood it relies on the **QEMU** processor and board simulator. **GNATemulator** efficiently runs the executable: the target code is translated on the fly, one basic block at a time, to the host processor. The translated code is then kept in a cache so that no new translation is required when the execution passes through a basic block that has been executed. An important point to keep in mind is that **GNATemulator** is translating and executing instructions as fast as possible and thus won't be cycle accurate. The clock used by **GNATemulator** is the host system clock. Timing events are thus dependent on that clock and will not reflect the timing characteristics of a real target board.
- **The BSP**: On non-bareboard systems, a BSP is provided when necessary. This is the case for example on **VxWorks 653**. On that target we provide a **BSP** called `qemu` that should be installed in the **VxWorks 653** installation. The BSP simulates a board with one UART (serial link), one ethernet controller, a timer, ROM and 128MB of RAM.
- **GNAT Bus**: A native framework to emulate devices (see *Extending GNATemulator* chapter).
- **The examples**: For each supported platform there are examples that can be used as a reference for further development with **GNATemulator**

## 1.3 Note on the Documentation

As mentioned in the previous section the main tool name is:

```
<target>-gnatemu
```

Along this documentation, in the sections that are not target specific we will use `gnatemu` for the tool name, skipping the target prefix. In that cases always replace `gnatemu` by `<target>-gnatemu` where `<target>` is the relevant target prefix for your context.

As a reminder here is the list of target supported by **GNATemulator** along with the expected tool name:

| Platform | Tool name |
|---|---|
| PowerPC e500v2 VxWorks 6.x | e500v2-wrs-vxworks-gnatemu |
| PowerPC VxWorks 6.x | powerpc-wrs-vxworks-gnatemu |
| PowerPC VxWorks 653 | powerpc-wrs-vxworksae-gnatemu |
| PowerPC ELF | powerpc-elf-gnatemu |
| PowerPC 55xx and e500v2 ELF | powerpc-eabispe-gnatemu |
| LEON 2 ELF | leon-elf-gnatemu |
| LEON 3 ELF | leon3-elf-gnatemu |
| ARM ELF | arm-eabi-gnatemu |
| AARCH64 ELF | aarch64-elf-gnatemu |

CHAPTER

TWO

# GETTING STARTED

## 2.1 Installation

On Windows host, installation is performed automatically by **InstallShield**. On Linux host, you will need first to unpack the package using `tar` utility and then launch the `doinstall` script located at the toplevel directory. In both cases, you will be prompted for an installation directory for the simulator and the example (later referred to as `GNATEMULATOR_INSTALL_DIR`).

If you are using **GNATemulator** for **VxWorks 653** then once the product is installed, you should install the BSP in the **VxWorks 653** installation. In order to do that you need to copy the `%GNATEMULATOR_INSTALL_DIR%\share\gnatemulator\ppc-vx653\bsp\qemu` directory as `%WIND_BASE%\target\config\qemu`.

Note that the PATH to the target BSP directory that you are copying to may be slighty different depending on your version of **VxWorks 653**. Note in particular that this BSP uses the PPC604gnu toolchain, which is not available on 653 2.4; so this version of 653 is not supported. On 653 2.5, you will also need to add ne2000End.o to the library, for instance by doing the following:

```
cd %WIND_BASE%\target\lib
arppc x libPPC604gnuvx.a ne2000End.o
cp ne2000End.o objPPC604gnuvx\ne2000End.o
```

## 2.2 Setting your environment

In order to set your environment for **GNATemulator** you need to do on Windows:

```
set PATH=%GNATEMULATOR_INSTALL_DIR%\bin;%PATH%
```

And on Linux:

```
export PATH=$GNATEMULATOR_INSTALL_DIR/bin:$PATH
```

Where `GNATEMULATOR_INSTALL_DIR` is the root directory of your installation.

If you need to build your own devices using **GNATBus**, then you should also do on Windows:

```
set ADA_PROJECT_PATH=%GNATEMULATOR_INSTALL_DIR%\lib\gnat;%ADA_PROJECT_PATH%
```

And on Linux:

```
export ADA_PROJECT_PATH=$GNATEMULATOR_INSTALL_DIR/lib/gnat:$ADA_PROJECT_PATH
```

Note that if **GNATemulator** has been installed in the same location as your native compiler then you don't need to modify `ADA_PROJECT_PATH` environment variable.

## 2.3 Running the examples

In the following subsections, small examples are described for each supported platforms except **VxWorks 6.x**. The examples sources are located in `%GNATEMULATOR_INSTALL_DIR%/share/examples/gnatemu/`. To get started on **GNATemulator** for **VxWorks 6.x**, you should go directly to *VxWorks 6 Topics*.

### 2.3.1 The VxWorks 653 example

#### First run

The example can be built using the **`make`** tool from the VxWorks Development Shell. Open the VxWorks Development Shell, go to the directory where the example has been installed ( `%GNATEMULATOR_INSTALL_DIR%/share/examples/gnatemu/ppc-vx653/sys653`) and then type:

```
$ make
```

This command configures and builds the system. It generates a ROM image of the VxWorks 653 system in `int/demo/hello.flash`.

To run the example in the simulation environment, type:

```
$ make run
```

To stop the simulator, press `Control-Ax`.

#### Structure of the Example

The example comprises 4 components, following the typical organization of a VxWorks 653 system:

- The **Module OS** is the kernel of the system and contains the BSP. It is configured and built in the `mos` subdirectory. In the example the `mos` directory does not contain any source. There is just a Makefile whose purpose is to build and configure the **Module OS** using existing VxWorks 653 libraries and **GNATemulator** BSP sources. The **Module OS** is configured and compiled for a specific BSP/board, in this case the one described by the **qemu** BSP.

  To include tools useful during the development of your application (Target Tools), you should call make with the following parameter:

  ```
  $ make DEBUG=1
  ```

  Should you need to reconfigure the Core OS (to include/exclude Target Tools for example), you should clean the @file{mos} subdirectory first by calling:

  ```
  $ make clean
  ```

in the `mos` directory. Calling **`make clean`** in the top-level example directory will clean the complete VxWorks 653 system. It removes the generated code and keeps the example infrastructure so the example can be regenerated from scratch.

- The **Partition OS** is a passive partition (i.e. it will not execute code by itself) that contains the OS code to be executed in the application partitions.

  Like the `mos` directory, the `pos` directory in the example does not contain any source. There is just a Makefile that is called to configure the partition (to define the size, the base virtual address and the functions that can be called).

  The compilation will generate a partition and stubs. The stubs will be linked with the application, allowing access to POS services (see below).

- The **Application**. This example contains one application located in the `app` directory. Configuration related to the application is minimal: only memory constraints are defined. To allow users to easily replace the example code, a wrapper (`appwrapper.c`) is provided as the entry point of the application. The wrapper then calls function `hello` in file `hello.c`.

- The **Integration**. The components described above can be developed separately. The purpose of the integration component is to agregate these components into the final system. The logic for the integration is in directory `int`. The directory contains makefiles and configuration files to build the final ROM image that will be run by the simulator.

### 2.3.2 The LEON 2 ELF Bareboard Example

This tutorial shows how to build an interactive example and run it with **GNATemulator**.

#### Compiling the example

The example comprises 3 small ada units:

- hello.adb, the main subprogram.
- uart.ads and uart.adb which perform I/O using the UART.

To build, simply invoke gprbuild in the example directory:

```
gprbuild --target=leon-elf --RTS=ravenscar-sfp-leon
```

the option '–RTS=ravenscar-sfp-leon' selects the small foot print ravenscar profile.

#### Running the example

To launch the example just run:

```
leon-elf-gnatemu hello
```

**GNATemulator** will automatically load the ELF file (`hello`) and start execution at the entry point.

Here is a quick run scenario:

```
Menu:
1) Hello
2) Bye
3) Quit
```

```
You choice:
1
hello
Menu:
1) Hello
2) Bye
3) Quit
You choice:
2
bye
Menu:
1) Hello
2) Bye
3) Quit
You choice:
3
qemu: fatal: Trap 0x80 while interrupts disabled, Error state
```

(The double trap which is an error is used to stop the simulator).

#### Redirecting the uart

It is possible to redirect the UART to a TCP port:

```
leon-elf-gnatemu --serial=tcp::1234,server hello
```

This will redirect UART1 to the TCP port 1234 of localhost. With the 'server' option, **GNATemulator** will wait for the TCP connection.

### 2.3.3 The LEON 3 ELF Bareboard Example

This tutorial shows how to build an interactive example and run it with **GNATemulator**.

#### Compiling the example

The example comprises 3 small ada units:

- hello.adb, the main subprogram.
- uart.ads and uart.adb which perform I/O using the UART.

To build, simply invoke gprbuild in the example directory:

```
gprbuild --target=leon3-elf --RTS=ravenscar-sfp-leon3
```

the option '–RTS=ravenscar-sfp-leon3' selects the small foot print ravenscar profile.

#### Running the example

To launch the example just run:

```
leon3-elf-gnatemu hello
```

**GNATemulator** will automatically load the ELF file (`hello`) and start execution at the entry point.

Here is a quick run scenario:

```
Menu:
1) Hello
2) Bye
3) Quit
You choice:
1
hello
Menu:
1) Hello
2) Bye
3) Quit
You choice:
2
bye
Menu:
1) Hello
2) Bye
3) Quit
You choice:
3
qemu: fatal: Trap 0x80 while interrupts disabled, Error state
```

(The double trap which is an error is used to stop the simulator).

#### Redirecting the uart

It is possible to redirect the UART to a TCP port:

```
leon3-elf-gnatemu --serial=tcp::1234,server hello
```

This will redirect UART1 to the TCP port 1234 of localhost. With the 'server' option, **GNATemulator** will wait for the TCP connection.

### 2.3.4 The PowerPC 55xx and e500v2 ELF Bareboard Example

This tutorial shows how to build an example and run it with **GNATemulator**.

#### Compiling the example

The example comprises 2 files:

- hello.adb, the main subprogram.
- hello.gpr, the project to build the program

To build, simply invoke gprbuild:

```
gprbuild --target=powerpc-eabispe -P hello.gpr --RTS=xravenscar-full-p2020
```

#### Running the example

To launch the example just run:

```
powerpc-eabispe-gnatemu hello
```

**GNATemulator** will automatically load the ELF file (`hello`) and start execution at the entry point.

### 2.3.5 The PowerPC ELF Bareboard Example

This tutorial shows how to build an example and run it with **GNATemulator**.

#### Compiling the example

The example comprises 2 files:

- hello.adb, the main subprogram.
- hello.gpr, the project to build the program

To build, simply invoke gprbuild:

```
gprbuild --target=powerpc-elf -P hello.gpr --RTS=ravenscar-full-prep
```

#### Running the example

To launch the example just run:

```
powerpc-elf-gnatemu hello
```

**GNATemulator** will automatically load the ELF file (`hello`) and start execution at the entry point.

### 2.3.6 The ARM ELF Bareboard Example

This tutorial shows how to build an example and run it with **GNATemulator**.

#### Compiling the example

The example comprises 2 files:

- hello.adb, the main subprogram.
- hello.gpr, the project to build the program

To build, simply invoke gprbuild:

```
gprbuild --target=arm-eabi -P hello.gpr --RTS=ravenscar-full-stm32f4
```

### Running the example

To launch the example just run:

```
arm-eabi-gnatemu --board=stm32f4 hello
```

**GNATemulator** will automatically load the ELF file (`hello`) and start execution at the entry point.

CHAPTER

THREE

# USING GNATEMULATOR

## 3.1 Launching GNATemulator

To launch a guest application just run:

```
gnatemu hello
```

GNATemulator will automatically load the ELF file (`hello`) and start execution at the entry point. Note that in the **VxWorks 653** and **VxWorks 6.x** case, the file passed to `gnatemu` is a full **VxWorks** kernel image.

To stop, press `Control-a x`.

## 3.2 Displaying the help

In order to list the available options you should run:

```
$ gnatemu --help
Usage: gnatemu [OPTIONS] FILE
Options are:
  -v, --verbose        be verbose
  -h, --help           display this help
  -Pproj or -P proj    Use GNAT Project File proj
  -Xnm=val             Specify an external reference for Project Files
  --version            display version
  --serial=null        redirect 1st serial port to null file
  --serial=stdio       redirect 1st serial port to stdio
  --serial=file:FILENAME
                       redirect 1st serial port to a file (write only)
  --serial=tcp:HOST:PORT[,server]
                       redirect 1st serial port to HOST:PORT via tcp.
  --serialN            idem as --serial for the Nth serial port
  --tftp-root=path     Set root directory of tftp server (default: .)
  --wdb[=HOST_PORT]    Set WDB redirection listen UDP port (default: 17185)
  --gdb[=PORT]         allow gdb connection on port PORT (default port is 1234)
  -g                   allow default debug (i.e --wdb or --gdb --freeze-on-startup
  --freeze-on-startup  freeze emulation on startup
  --gnatbus=HOST:PORT[,HOST:PORT]
                       connect a GNATBus device
  --emulator-help      display available Qemu options
  --eargs              start a group of Qemu options
  --eargs-end          end a group of Qemu options
```

## 3.3 GNAT Project File

Project attributes for GNATemulator are specified in package "Emulator":

```
project Prj is
   package Emulator is
      [...]
   end Emulator;
end Prj;
```

Supported attributes:

- Board: equivalent of switch `--board=`
- Switches: A list of switches processed before the command line switches

For example:

```
package Emulator is
   for Board use "BOARD_NAME";
   for Switches use ("Sw1", "Sw2");
end Emulator;
```

## 3.4 Debugging

This section explains how to set up a debugging session with GNATemulator.

### 3.4.1 Debugging options in GNATemulator

GNATemulator provides various switches to ease debugging of guest applications. Here are the options that control debugging

**--gdb, --gdb**=<PORT>
: This flags will initiate the gdbserver and wait for gdb connection on port PORT. If not port is specified then the default port 1234 is used. For example:

```
$ gnatemu --gdb=2048 hello
```

**--freeze-on-startup**
: Freeze the CPU at startup. When used GNATemulator will freeze simulation and wait for a **continue** command from gdb.

**--wdb, --wdb**[=HOST_PORT]
: Allow debugging with WindRiver Workbench debugger. If HOST_PORT is not passed then 17185 is used.

**-g**
: This is a shortcut for `--gdb --freeze-on-startup` on non VxWorks platforms. In VxWorks context this is a shortcut for `--wdb`.

### 3.4.2 Debugging with GDB

To debug with gdb:

#. Invoke GNATemulator with the `-g` flag so that the emulator will stop and await a connection from gdb:

```
$ gnatemu -g hello
```

1. Invoke gdb and connect to GNATemulator with the GDB **`target`** command:

```
$ gdb hello
(gdb) target remote localhost:1234
(gdb)
```

We can use a bare-board target to illustrate these steps, in this case a LEON3, working with a simple "hello world" program.

In one console we start the LEON3 version of the emulator:

```
$ leon3-elf-gnatemu -g hello
```

The emulator is now waiting for the debugger to connect.

In another console we start the LEON3 version of the debugger:

```
$ leon3-elf-gdb hello
...
Reading symbols from hello...done.
(gdb)
```

In response gdb emits several lines of information, loads the image, and then prompts for another command.

Next, we instruct gdb to connect to the simulator over TCP:

```
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0x40000000 in trap_table ()
(gdb)
```

Again gdb responds and prompts for another command. We can have it show the source code, for example:

```
(gdb) list
1       with Ada.Text_IO; use Ada.Text_IO;
2
3       procedure hello is
4       begin
5          Put_Line ("hello world");
6       end;
7
(gdb)
```

We set a breakpoint on line five:

```
(gdb) break hello.adb:5
```

As a result gdb will respond with an indication of the breakpoint being successfully set:

```
Breakpoint 1 at 0x40001004: file C:\temp\O305-036\hello.adb, line 5.
(gdb)
```

We can then command gdb to continue execution of the application:

```
(gdb) cont
Continuing.

Breakpoint 1, hello () at C:\temp\O305-036\hello.adb:5
5          Put_Line ("hello world");
(gdb)
```

In response, once the breakpoint is hit, gdb displays the breakpoint line and prompts us again. At this point the call to Put_Line has not yet occurred. If we then issue the "step" command the call will occur:

```
(gdb) step
6       end;
```

Now gdb is ready to execute from line six.

In the other console we will see the message printed as part of the emulator execution:

```
$ leon3-elf-gnatemu -g hello
hello world
```

We can continue debugging if there is more of the program, or simply tell gdb to quit. Quitting will break the connection to the emulator and will cause it to terminate as well.

### 3.4.3 Debugging with GPS

To debug with GPS:

1. Run GNATemulator with `-g` flags:

```
$ gnatemu -g hello
```

2. Read the GPS documentation, section "Working in a Cross Environment -> Debugger Issues" and use the following configuration:

```
targetname: "localhost:1234"
protocol: "remote"
```

## 3.5 Redirecting serial port(s)

In GNATemulator it is possible to redirect the serial communication ports. In order to do that you need to use the switch:

**--serial**=file:<FILE>, **--serial**=tcp:<HOST>:<PORT>[,server], **--serial**=stdio, **--serial**=null

Serial can be redirected to null, standard output, file (write only) or TCP port. Note that only one serial port can be redirected to standard output. By default if you have several ports then the first one will be redirected to standard output and the others to null.

1. Redirection to standard output:

```
$ gnatemu --serial=stdio hello
```

2. Redirection to null:

```
$ gnatemu --serial=null hello
```

3. Redirection to a file (write only):

```
$ gnatemu --serial=file:/tmp/serial_output.txt hello
```

4. To a TCP port:

```
$ gnatemu --serial=tcp:hostname:1234 hello
```

   With the 'server' option, GNATemulator will wait for the tcp connection:

```
$ gnatemu --serial=tcp:hostname:1234,server hello
```

The switch `--serial` will redirect the first serial port. If there is more than one serial port on the target, you can redirect them using:

**`--serial1, --serial2, --serialN`**

where N is the number of your serial port. `--serial` is equivalent to `--serial1`

For example, the first serial to a file and the second to a TCP port:

```
$ gnatemu --serial=file:/tmp/serial1.txt  --serial=tcp:localhost:1234 hello
```

As mentioned before, only one serial port can be redirected to the standard output. By default the first serial port is redirected to that output except if another port has been assigned explicitly to it. In that case the first serial port is redirected to null by default.

## 3.6 Connecting to GNAT Bus devices

In order to connect additional devices developed with **GNAT Bus** to your board you should use the following option:

**`--gnatbus`**`=<HOST>:<PORT>[,<HOST2>:<PORT2>...]`
For more in depth introduction to that feature please see *Extending GNATemulator* chapter.

## 3.7 Board selection

If GNATemulator provides multiple emulations for the target platform, use the following option to select a specific board:

**`---board`**`=<BOARD_NAME>`

Use gnatemu —help to get the list of boards.

## 3.8 Access to host file system

(only for ppc-elf, p55-elf, leon-elf, leon3-elf, arm-elf, aarch64-elf)

GNATemulator provides a simplified version of the GNAT.OS_Lib package that allows access to the host file system from the simulated program.

To use this package you have to include "hostfs_bareboard.gpr" into your project file:

```
with "hostfs_bareboard.gpr";

project Test is
...
end Test;
```

You can then use the package in your project:

```
with GNAT.OS_Lib; use GNAT.OS_Lib;
with GNAT.IO; use GNAT.IO;

procedure Test is
   FD : File_Descriptor;
   File_Name : constant String := "new_file.txt";
   Text : String (1 .. 12) := "Hello World" & ASCII.LF;
begin

   --  Create a new file on the host file system
   FD := Create_New_File (File_Name, GNAT.OS_Lib.Text);

   if FD /= Invalid_FD then

      if Write (FD, Text'Address, 12) /= 12 then
         Put_Line ("Cannot Write '" & File_Name & "'");
      end if;

      Close (FD);

   else
      Put_Line ("Cannot create new file '" & File_Name & "'");
   end if;
end Test;
```

Finally, when compiling your project you have to specify the board like so:

```
$ gprbuild --target=leon3-elf -XGNATEMU_BOARD=leon3-elf test
```

CHAPTER

FOUR

# WORKBENCH/VXWORKS 653 TOPICS

## 4.1 Integration of the Simulation Environment in Workbench

### 4.1.1 Building a System

A **VxWorks 653** system can be configured and built in the usual way, either from the command line or from within Workbench. See the **VxWorks 653** user manuals for general instructions, or the **GNATbench** tutorial available in **Workbench** via *Help –> Help Contents –> GNATbench Ada Development User Guide –> Tutorial: Creating a VxWorks 653 Integration Project*. If using the latter, follow the instructions given for a **QEMU** BSP.

### 4.1.2 Creating a ROM Image

The **VxWorks 653** build system generates separate image files for each component. These images have to be gathered into a single ROM image in order to be executed by **GNATemulator**. To simplify this unification procedure, we provide the **`powerpc-wrs-vxworksae-romgen`** tool. It automatically extracts the list of partitions and their respective locations from the `configRecord.xml` file and uses this data to automatically build the final ROM image.

To use generate a ROM image from a **VxWorks 653** system under **Workbench**, add a build target, e.g. `qemu_system.flash` to the integration project. Let's call this build target `KERNEL_IMAGE` in further discussion.

In the `Makefile` for the integration project, append `KERNEL_IMAGE` to the prerequites of the **`all`** target, e.g.

```
all: <other targets> KERNEL_IMAGE
```

Then add a target for the image:

```
KERNEL_IMAGE: rom
     powerpc-wrs-vxworksae-romgen -o KERNEL_IMAGE
```

Note that the second line starts with a <tab> character, not spaces.

If using a `Makefile` for the integration project from the command line, you can omit the step of adding the build target, and still make the changes to the `Makefile`.

### 4.1.3 Running the System

The simulator can easily be run from the **Workbench** interface. To do so:

1. Click on *External Tools Configuration...* in the menu entry *Run → External Tools*.

2. Select *Program* and click on the *New* button.
3. In *Location* enter **powerpc-wrs-vxworksae-gnatemu** with its full path (e.g. `%GNATEMULATOR_INSTALL_DIR%\bin\powerpc-wrs-vxworksae-gnatemu.exe`).
4. In the *Working Directory* browse to the directory containing *KERNEL_IMAGE* and select it.
5. In the *Arguments* section enter the arguments to pass to **powerpc-wrs-vxworksae-gnatemu**. See *Using GNATemulator* for the available options. You should add `--wdb` to enable target server connections.
6. Under the *Build* tab, uncheck *Build before launch* in order to avoid systematic recompilation when you start the simulation environment.

Clicking on *Run* will save the configuration and run the system. The simulation platform can then be easily accessed from the **Workbench** toolbar in the Advanced Development perspective via the green *External Tools* icon.

### 4.1.4 Creating a Target Server

The Wind River Debug Server (DFW) can connect to **GNATemulator** for VxWorks 653 using the `wdbrpc` backend that will use the ethernet connection available in **GNATemulator**.

To create a new target server for **GNATemulator** for VxWorks 653:

1. Click on *New Connection...* in the *Target* menu of Workbench.
2. Select *Wind River VxWorks 653 Target Server Connection* as the system type.
3. In the *Target Server Options* page, select *wdbrpc* as the Backend, set the IP address to `127.0.0.1` and the port to `17185`.
4. Set the Kernel image using the `boot.txt` file.

#. To complete the setup of the target server you can keep the other fields to their default values.

## 4.2 Adapting QEMU for VxWorks 653 to other Contexts

### 4.2.1 Starting QEMU for VxWorks Manually

You can start manually **powerpc-wrs-vxworksae-gnatemu** using these following options:

```
$ powerpc-wrs-vxworksae-gnatemu [--wdb] KERNEL_IMAGE
```

Where `KERNEL_IMAGE` is the path to your kernel image.

## 4.3 Limitations

The following elements should be taken into account when using **GNAT Emulator** for VxWorks 653:

1. The time reported to the guest operating system (here VxWorks 653) is based on the host OS time. The host OS is a time sharing OS (and not a real-time OS), which can lead to situations where QEMU is rescheduled or preempted by another host process, which in turn makes the VxWorks 653 OS miss certain deadlines. That may trigger error handlers in VxWorks 653. This means that you should avoid loading your host machine when running QEMU.

   See in *Health Monitor Configuration* an example of health monitor configuration that avoid unexpected simulator halts.

2. **GNAT Emulator** is translating and executing instructions as fast as possible and thus won't be cycle accurate. This means for example that within a frame defined by the active VxWorks 653 schedule, processing can be faster than on the final target.

3. When **GNAT Emulator** is executing the VxWorks 653 operating system it will preempt all CPU available and allowed by its priority. The rest of the Windows system may become hardly usable because of a lack of CPU resources allocated to the rest of the system. This is particularly true on monocore host machines.

   If you don't have a multicore system, it is advised to start **GNAT Emulator** with the following command:

```
> start /LOW powerpc-wrs-vxworksae-gnatemu
```

## 4.4 Health Monitor Configuration

Here is an example of health monitor configuration that can be used to avoid a simulator stop in case of tick loss:

```
<HealthMonitor>
  <SystemHMTable Name="systemHm">
    <SystemState SystemState="HM_PARTITION_MODE">
      <ErrorIDLevel ErrorIdentifier="HME_DEFAULT"
                    ErrorLevel="HM_MODULE_LVL"/>
    </SystemState>
    <SystemState SystemState="HM_MODULE_MODE">
      <ErrorIDLevel ErrorIdentifier="HME_DEFAULT"
                    ErrorLevel="HM_MODULE_LVL"/>
    </SystemState>
    <SystemState SystemState="HM_PROCESS_MODE">
      <ErrorIDLevel ErrorIdentifier="HME_APPLICATION_ERROR"
                    ErrorLevel="HM_MODULE_LVL"/>
    </SystemState>
  </SystemHMTable>
  <ModuleHMTable Name="moduleHm">
    <SystemState>
      <ErrorIDAction ErrorIdentifier="HME_DEFAULT"
                     ErrorAction="hmDbg_DH_EventShow"/>
    </SystemState>
    <SystemState>
      <ErrorIDAction ErrorIdentifier="HME_APPLICATION_ERROR"
                     ErrorAction="hmDbg_DH_EventShow"/>
    </SystemState>
    <SystemState>
      <ErrorIDAction ErrorIdentifier="HME_LOST_TICKS"
                     ErrorAction=""/>
    </SystemState>
    <Settings stackSize="0x0400" maxQueueDepth="2" />
  </ModuleHMTable>
  <PartitionHMTable Name="helloHm">
    <SystemState>
      <ErrorIDAction ErrorIdentifier="HME_DEFAULT" ErrorAction=""/>
    </SystemState>
    <Settings stackSize="0x0400" maxQueueDepth="2" />
  </PartitionHMTable>
</HealthMonitor>
```

CHAPTER

FIVE

# VXWORKS 6 TOPICS

## 5.1 Building a Kernel

**GNATemulator** supports a specific BSP for VxWorks 6, the first step is to build a kernel with this BSP.

To create a VxWorks 6 kernel:

1. In WorkBench: File -> New -> VxWorks Image Project
2. Enter project name (e.g. GNATemu_kernel), click *Next*
3. On **GNATemulator** for **PowerPC e500v2 VxWorks 6.x**, select the **wrSbc8548** BSP, and on **GNATemulator** for **PowerPC VxWorks 6.x** select the **wrSbc834x** BSP.
4. Select the *gnu* tool chain.
5. Click *Next* twice
6. Select Profile: *PROFILE_DEVELOPMENT* and click *Finish*
7. In the kernel configuration window for BSP wrSbc8548 profile:
   (a) Exclude DRV_TIMER_OPENPIC
   (b) Include INCLUDE_STANDALONE_SYM_TBL
   (c) Include FOLDER_DOSFS2
   (d) Set LOCAL_MEM_SIZE to 0x8000000 # Note 0x800_0000
   (e) Set WIND_JOBS_MAX to 256
8. In the kernel configuration window for BSP wrSbc834x profile:
   (a) Exclude INCLUDE_NET_DRV
   (b) Include INCLUDE_NETSTAT
   (c) Include INCLUDE_STANDALONE_SYM_TBL
   (d) Include FOLDER_DOSFS2
   (e) Set LOCAL_MEM_SIZE to 0x8000000 # Note 0x800_0000
   (f) Set WIND_JOBS_MAX to 256
   (g) Change IP addresses of DEFAULT_BOOT_LINE to h=192.168.0.1 e=192.168.0.2
9. On **GNATemulator** for **PowerPC VxWorks 6.x**
   (a) Click on *File -> Import...*
   (b) Select *General -> File System* and press *Next*

(c) In the *File System* dialog, browse to following directory

```
<GNATEMULATOR_INSTALL_DIR>/share/gnatemu/<TARGET>/autoexec
```

(d) Press *OK*

(e) Press *Select All* then *Finish*

(f) Answer *Yes To All* in the *Overwrite...* dialog box

10. To build the kernel run: Project -> Build Project

You can find the kernel image at `<PROJECT_ROOT_DIR>/Default/VxWorks`.

## 5.2 Running GNATemulator

### 5.2.1 Downloadable Kernel Module

**GNATemulator takes the DKM (.out) as last command line argument,** automatically loads and runs it and then stop execution.

```
$ powerpc-wrs-vxworks-gnatemu --kernel=<PROJECT_ROOT_DIR>/Default/VxWorks print.out
Target Name: vxTarget

Adding 6942 symbols for standalone.


                VxWorks

Copyright 1984-2012  Wind River Systems, Inc.

             CPU: Wind River SBC8349E
    Runtime Name: VxWorks
 Runtime Version: 6.9
     BSP version: 6.9/0
         Created: Dec  1 2012, 19:28:15
ED&R Policy Mode: Deployed
   WDB Comm Type: WDB_COMM_END
             WDB: Agent Disabled.


cmdline: kernel print.out
Instantiating /tmp as rawFs,  device = 0x1
Formatting /tmp for DOSFS
Instantiating /tmp as rawFs, device = 0x1
Formatting...-> OK.
Mount ramdisk in /tmp...OK
copy host:print.out -> local:print.out
708798 byte(s) transferred
<runkernel print.out at addr=0x17567e0 id=24468064>
Hello world!
</runkernel>
```

### 5.2.2 Interactive Mode

When there is no DKM on the command line, **GNATemulator** starts vxWorks in interactive mode.

```
$ powerpc-wrs-vxworks-gnatemu --kernel=<PROJECT_ROOT_DIR>/Default/VxWorks
Target Name: vxTarget

Adding 6942 symbols for standalone.


                VxWorks

Copyright 1984-2012

             CPU: Wind River SBC8349E
    Runtime Name: VxWorks
 Runtime Version: 6.9
     BSP version: 6.9/0
         Created: Dec  1 2012, 19:28:15
ED&R Policy Mode: Deployed
   WDB Comm Type: WDB_COMM_END
             WDB: Agent Disabled.

cmdline:
Instantiating /tmp as rawFs,  device = 0x1
Formatting /tmp for DOSFS

Instantiating /tmp as rawFs, device = 0x1
Formatting...-> OK.
Mount ramdisk in /tmp...OK
Run_shell
->
```

## 5.3 Connecting Workbench to GNATemulator

The Wind River Debug Server (DFW) can connect to **GNATemulator** using the `wdbrpc` back-end that will use the Ethernet connection available in **GNATemulator**.

To create a new target server for **GNATemulator** and VxWorks 6:

1. Click on *New Connection...* in the *Target* menu of Workbench.
2. Select *Wind River VxWorks 6.x Target Server Connection* as the system type.
3. In the *Target Server Options* page, select *wdbrpc* as the Backend, set the IP address to `127.0.0.1` and the port to `17185`.
4. Set the Kernel image
5. To complete the setup of the target server you can keep the other fields to their default values
6. Start **GNATemulator** with the `--wdb` option
7. Note: INCLUDE_WDB_ALWAYS_ENABLED must be in kernel configuration.

   ```
   $ powerpc-wrs-vxworks-gnatemu --wdb --kernel=<PROJECT_ROOT_DIR>/Default/VxWorks
   ```

8. Click on *Connect '...'* in the *Target* menu.

## 5.4 Using the internal TFTP server

**GNATemulator**'s internal TFTP server can be used to load file and executable on the target.

1. Open the kernel project created in *Building a Kernel*
2. In the kernel configuration window:
   (a) Include INCLUDE_TFTP_CLIENT
   (b) Include INCLUDE_IPTFTPC
   (c) Include INCLUDE_IPTFTP_CLIENT_CMD
   (d) Include INCLUDE_RAM_DISK
   (e) Set RAM_DISK_SIZE to 16000000
   (f) Set RAM_DISK_DEV_NAME to "/tmp"
3. **In addition for the wrSbc834x BSP:**
   (a) Set DEFAULT_BOOT_LINE to "mottsec(0,0)host:target/config/wrSbc834x/vxWorks h=192.168.0.1 e=192.168.0.2 g=192.168.0.1 u=vxworks tn=wrSbc834x"
4. Re-build the kernel: Project -> Build Project

To use the TFTP in VxWorks:

1. Start **GNATemulator**

   ```
   $ powerpc-wrs-vxworks-gnatemu --wdb --kernel=<PROJECT_ROOT_DIR>/default/vxWorks
   ```

2. In the VxWorks prompt, format the ram disk

   ```
   -> dosFsVolFormat ("/tmp", 0, 0)
   ```

3. Change current directory

   ```
   -> cd "/tmp"
   ```

4. Enter command mode

   ```
   -> cmd
   ```

5. Use the *tftp* command to download a file

   ```
   [vxWorks *]# tftp 192.168.0.1 get test.txt
   ```

Here is a complete example of TFTP session:

```
$ echo Test TFTP server > test.txt
$ powerpc-wrs-vxworks-gnatemu --wdb --kernel=<PROJECT_ROOT_DIR>/default/vxWorks
Instantiating /tmp as rawFs,  device = 0x1
Target Name: vxTarget
Instantiating /ram as rawFs,  device = 0x10001
Formatting /ram for DOSFS
Instantiating /ram as rawFs, device = 0x10001
Formatting...Retrieved old volume params with %38 confidence:
Volume Parameters: FAT type: FAT32, sectors per cluster 0
  0 FAT copies, 0 clusters, 0 sectors per FAT
  Sectors reserved 0, hidden 0, FAT sectors 0
  Root dir entries 0, sysId (null)  , serial number 20000
  Label:"           " ...
Disk with 64 sectors of 512 bytes will be formatted with:
Volume Parameters: FAT type: FAT12, sectors per cluster 1
```

```
  2 FAT copies, 54 clusters, 1 sectors per FAT
  Sectors reserved 1, hidden 0, FAT sectors 2
  Root dir entries 112, sysId VXDOS12 , serial number 20000
  Label:"           " ...
OK.

Adding 6717 symbols for standalone.


                VxWorks

Copyright 1984-2012  Wind River Systems, Inc.

             CPU: Wind River SBC8349E
    Runtime Name: VxWorks
 Runtime Version: 6.9
     BSP version: 6.9/0
         Created: Dec  1 2012, 19:28:15
ED&R Policy Mode: Deployed
   WDB Comm Type: WDB_COMM_END
             WDB: Agent Disabled.


-> dosFsVolFormat ("/tmp", 0, 0)
Formatting /tmp for DOSFS
Instantiating /tmp as rawFs, device = 0x1
Formatting...Retrieved old volume params with %38 confidence:
Volume Parameters: FAT type: FAT32, sectors per cluster 0
  0 FAT copies, 0 clusters, 0 sectors per FAT
  Sectors reserved 0, hidden 0, FAT sectors 0
  Root dir entries 0, sysId (null)  , serial number 11690000
  Label:"           " ...
Disk with 31250 sectors of 512 bytes will be formatted with:
Volume Parameters: FAT type: FAT16, sectors per cluster 2
  2 FAT copies, 15546 clusters, 62 sectors per FAT
  Sectors reserved 1, hidden 0, FAT sectors 124
  Root dir entries 512, sysId VXDOS16 , serial number 11690000
  Label:"           " ...
OK.
value = 0 = 0x0
-> cd "/tmp"
value = 0 = 0x0
-> cmd
[vxWorks *]# tftp 192.168.0.1 get test.txt
Transfer completed: 17 bytes in 0.0 seconds.
[vxWorks *]# cat test.txt
Test TFTP server
[vxWorks *]#
```

CHAPTER

SIX

# VXWORKS 6 CERT TOPICS

## 6.1 Building a Kernel

**GNATemulator** supports a specific BSP for VxWorks 6 cert, the first step is to build a kernel with this BSP.

To create a VxWorks 6 cert kernel:

1. In WorkBench: File -> New -> VxWorks Cert Image Project
2. Enter project name (e.g. GNATemu_kernel), click *Next*
3. On **GNATemulator** for **PowerPC e500v2 VxWorks 6.x**, select the **wrSbc8548** BSP, and on **GNATemulator** for **PowerPC VxWorks 6.x** select the **wrSbc834x** BSP.
4. Select the *gnu* tool chain.
5. Click *Next*
6. Select option with RTP support and click *Next*
7. Select a Profile and click *Finish*
8. In the kernel configuration window:
   (a) Set LOCAL_MEM_SIZE to 0x8000000 # Note 0x800_0000
9. To build the kernel run: Project -> Build Project

You can find the kernel image at `<PROJECT_ROOT_DIR>/Default/VxWorks`.

## 6.2 Running GNATemulator

```
$ powerpc-wrs-vxworks-gnatemu --kernel=<PROJECT_ROOT_DIR>/Default/VxWorks


  Copyright 1984-2011 Wind River Systems, Inc.
  VxWorks Cert 6.6.4.1
  Created: Jun 14 2013, 17:48:22
  ...
```

## 6.3 Running tests on vxWorks Cert

To run a test on vxWorks Cert it has to be included in the kernel during build. SKMs linked to the kernel binary, RTPs in a ROMFS. Then usrAppInit() must be modified to load and/or run the test. You will find procedures to do so in WorkBench documentation. Note that theses procedures can be scripted using the *vxprj* command line tool.

CHAPTER

SEVEN

# EXTENDING GNATEMULATOR

## 7.1 Introduction

**GNAT Emulator** provides a powerful interface to emulate your own devices and create a rich simulation environment.

With native simulation code communicating with the target through a socket, you will be able to emulate any piece of hardware to make **GNAT Emulator** an exact representation of your target platform.

## 7.2 GNAT Bus

### 7.2.1 Overview

**GNAT Bus** is the link between your simulation environment and the emulator. You can regard **GNAT Bus** as the simulation of an internal bus (such as AMBA or PCI) connected to the emulated platform through a bridge.

From the guest-executable point of view, the **GNAT Bus** devices are just like any other emulated peripheral.

**GNAT Bus** provides four main features:

1. **Memory mapped IO**

   Devices can register memory mapped IO areas in the emulated address space. Each load/store instruction executed by the CPU in an IO area will result in a call to the read/write callback of the corresponding device.

   This can be used to share data structure between guest executable and the host environment.

2. **Direct Memory Access**

   With Direct Memory Access (DMA) a device can read/write directly from/to the emulated memory.

   This is useful to transfer large amount of data from/to the guest program.

3. **Host Shared Memory (Available on Linux only)**

   Devices can register a shared host memory and map it in the emulated address space. Each load/store instruction executed by the CPU in that area will be directly written in the shared memory. The device is able to use those data. This allows faster communications between the virtual machine and the device.

4. **Interrupt**

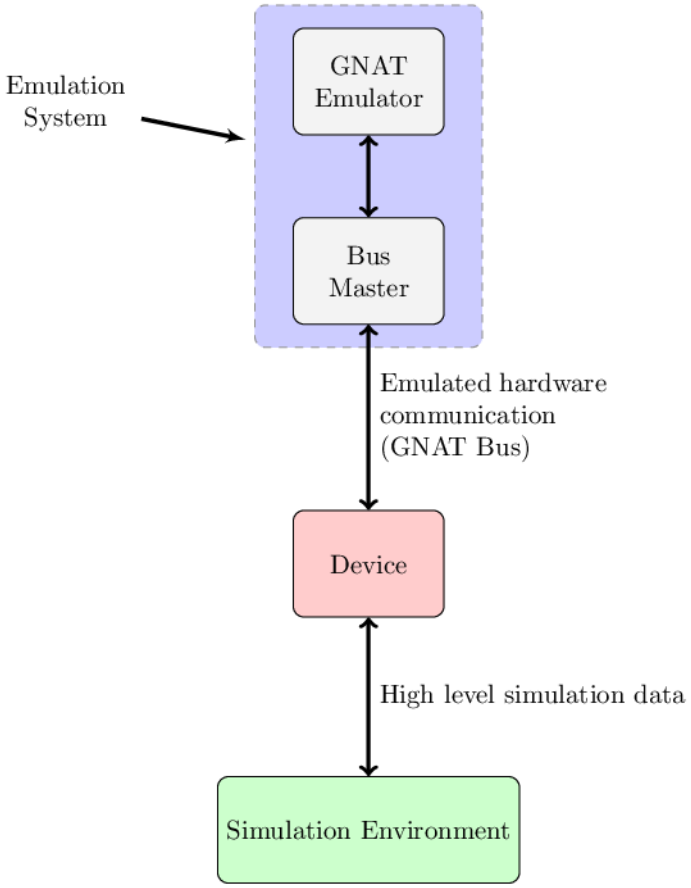
   From the device you can trigger interrupts on the emulated system.

   - Raise interrupt line
   - Lower interrupt line
   - Pulse i.e. quickly raise and lower interrupt line

Using the interrupt is thread safe, which means that the device can trigger an asynchronous IRQ at any time.

5. **Event**

   You can also create a timer running in the emulation time. When the timer expires, the emulation stops and an callback is executed in the device code.




### 7.2.2 GNAT Bus connection

There are two ways to connect a device to **GNAT Emulator**

1. **Named connection**

   In this mode the communication between the device and **GNAT Emulator** is done through a named connection (Unix Domain socket on Linux and Named pipe on Windows).

   On the device side use:

```
register_device_named(dev, "@my_device");
```

   On command line:

```
$ gnatemu --gnatbus=@my_device guest_uart
```

2. **TCP connection**

   In this mode the communication between the device and **GNAT Emulator** is done trough a TCP socket.

On the device side use:

```
register_device_tcp(dev, 8032);
```

On command line:

```
$ gnatemu --gnatbus=localhost:8032 guest_uart
```

### 7.2.3 Tutorial: Create A GNAT Bus Device

To show how to use **GNAT Bus**, we will define and emulate a UART controller. For simplicity, the controller will only be able to receive data.

You can write device code in C or Ada. This tutorial uses an Ada example but you can find the equivalent C example in <PATH_TO_GNATEMULATOR>/share/examples/gnatemu/gnatbus/).

#### Interface definition

First, we have to define the interface of our device.

The registers implemented in the UART controller are listed in the following table. The address of each register is defined as an offset to the base address:

Table 7.1: UART registers

| Register | Offset |
|---|---|
| UART Control | 0x0 |
| UART Data | 0x4 |

The following tables describe the fields of each register:

Table 7.2: UART Control register

| Bit number(s) | Field name | Reset state | Access | Description |
|---|---|---|---|---|
| 0 | Enable_Interrupt | 0 | R/W | If set an interrupt will be triggered for each character received |
| 1 | Data_To_Read | 0 | R | Set if there is at least one character to read |
| 2 - 31 | Reserved | undefined | | |

Table 7.3: UART Data register

| Bit number(s) | Field name | Reset state | Access | Description |
|---|---|---|---|---|
| 0 - 7 | Data | 0 | R | Read received character when Data_To_Read is set, 0 otherwise. |
| 8 - 31 | Reserved | undefined | | |

#### Project environment setup

Next, we have to create our project directory tree:

```
uart/
 |-- obj/
 `-- src/
```

Then we create a project file `uart/uart.gpr`, with the following content (see the GPRBuild documentation for detailed information on project files):

`gnatbus_ada.gpr` is a project distributed with **GNAT Emulator**, it contains the low-level circuitery (connection and communication with **GNAT Emulator**) and provides an abstaction layer so you just have to focus on the simulation code.

### package UART_Controller

```
uart/src/uart_controller.ads
uart/src/uart_controller.adb
```

This package implements a `UART_Control` protected object that contains the logic of our device (receive characters, manage the FIFO list, set the Data_To_Read flag, trigger interrupt when needed).

We will not go through the details of the `UART_Controller` since those are outside the scope of this tutorial. But you can find sources of this package in **GNAT Emulator**'s examples directory (<PATH_TO_GNATEMULATOR>/share/examples/gnatemu/gnatbus/uart).

### package UART_Device

```
uart/src/uart.ads
uart/src/uart.adb
```

To implement our UART device we create a class that inherits from the *Bus_Device* abstract class.

```
type UART_Device (Vendor_Id, Device_Id : Id;
                  Base_Address         : Bus_Address;
                  Port                 : Integer)
   is new Bus_Device (Vendor_Id, Device_Id, Port, Native_Endian) with record

   UC : UART_Control;
   --  The UART_Control protected object described earlier

end record;
```

The Vendor_Id, Device_Id and Port discriminants are required by the Bus_Device abstract type. Base_Address will be used latter as the address of our I/O area.

The device will have to implement six subprograms to provide the required interface:

- Device_Setup
- Device_Init
- Device_Reset
- Device_Exit
- IO_Read

- IO_Write

Let's look in detail how these are used by **GNAT Bus** and how they are implemented in our UART example.

#### Device_Setup

```
overriding procedure Device_Setup (Self : in out UART_Device);
```

This subprogram has to register the I/O area(s) and perform any other initialization needed before the device is started.

Body of `Device_Setup` procedure for `UART_Device`:

```
------------------
-- Device_Setup --
------------------

procedure Device_Setup (Self : in out UART_Device) is
begin
   Ada.Text_IO.Put_Line ("Device_Setup");

   --  Register the only I/O area: 8 bytes at base address to match the two
   --  registers.

   Self.Register_IO_Memory (Self.Base_Address, 8);

   --  Set UART_Device access in the UART_Control protected object

   Self.UC.Set_Device (Self'Unchecked_Access);
end Device_Setup;
```

#### Device_Init

```
overriding procedure Device_Init (Self : in out UART_Device);
```

As implied by its name, this subprogram has to perform device initialization. It will be called only once, at the beginning of emulation.

In our example there is nothing to do.

Body of `Device_Init` procedure for `UART_Device`:

```
-----------------
-- Device_Init --
-----------------

procedure Device_Init (Self : in out UART_Device) is
   pragma Unreferenced (Self);
begin
   Ada.Text_IO.Put_Line ("Device_Init");
end Device_Init;
```

### Device_Reset

```
overriding procedure Device_Reset (Self : in out UART_Device);
```

This procedure will be called each time a CPU reset occurs in the emulator. A reset is also triggered at the beginning of emulation (after `Device_Init`).

In our example, we have to flush the FIFO queue and set the registers to their reset value (this is handled by `UART_Control`).

Body of `Device_Reset` procedure for `UART_Device`:

```
------------------
-- Device_Reset --
------------------

procedure Device_Reset (Self : in out UART_Device) is
begin
   Ada.Text_IO.Put_Line ("Device_Reset");

   --  Send the reset signal to the UART_Control

   Self.UC.Reset;
end Device_Reset;
```

### Device_Exit

```
overriding procedure Device_Exit (Self : in out UART_Device);
```

`Device_Exit` is called one time, at the end of emulation.

In our example there is nothing to do.

Body of `Device_Exit` procedure for `UART_Device`:

```
-----------------
-- Device_Exit --
-----------------

procedure Device_Exit (Self : in out UART_Device) is
   pragma Unreferenced (Self);
begin
   Ada.Text_IO.Put_Line ("Device_Exit");
end Device_Exit;
```

### IO_Read

```
overriding procedure IO_Read (Self    : in out UART_Device;
                              Address : Bus_Address;
                              Length  : Bus_Address;
                              Value   : out Bus_Data);
--  Address : Bus_Address
```

```
--     Absolute address of the first byte targeted by this read operation.
--
--  Length : Bus_Address
--     Number of bytes targeted by this read operation (1, 2 or 4).
```

This procedure will be called when the CPU executes a load instruction in any of the I/O areas registered by the device. The procedure must set `Value` according to the specification of the emulated device.

The procedure is usually implemented with a case statement with branches for each register.

Body of `IO_Read` procedure for `UART_Device`:

```
-------------
-- IO_Read --
-------------

procedure IO_Read (Self    : in out UART_Device;
                   Address : Bus_Address;
                   Length  : Bus_Address;
                   Value   : out Bus_Data) is

   pragma Unreferenced (Length);
begin
   Ada.Text_IO.Put_Line ("Read @ " & Address'Img);

   --  case statement on the relative address

   case Address - Self.Base_Address is
      when 0 =>
         --  Return value of the control register
         Value := Self.UC.Get_CTRL;

      when 4 =>
         --  Pop a byte from FIFO queue
         Self.UC.Pop_DATA (Value);

      when others =>
         Ada.Text_IO.Put_Line ("Read unknown register:" & Address'Img);
         Value := 0;
   end case;
end IO_Read;
```

#### IO_Write

```
overriding procedure IO_Write (Self    : in out UART_Device;
                               Address : Bus_Address;
                               Length  : Bus_Address;
                               Value   : Bus_Data);
--  Address : Bus_Address
--     Absolute address of the first byte targeted by this write operation.
--
--  Length : Bus_Address
--     Number of bytes targeted by this write operation (1, 2 or 4).
```

This procedure is the equivalent of `Read_IO` when store instructions are executed.

Body of `IO_Write` procedure for `UART_Device`:

```
--------------
-- IO_Write --
--------------

procedure IO_Write (Self    : in out UART_Device;
                    Address : Bus_Address;
                    Length  : Bus_Address;
                    Value   : Bus_Data) is

   pragma Unreferenced (Length);
begin
   Ada.Text_IO.Put_Line ("Write @ " & Address'Img);

   --  case statement on the relative address

   case Address - Self.Base_Address is
      when 0 =>
         --  Set Control register value
         Self.UC.Set_CTRL (Value);

      when others =>
         Ada.Text_IO.Put_Line ("Write unknown register:" & Address'Img);
   end case;
end IO_Write;
```

### Main procedure

uart/src/main.adb

Finally, we need a main procedure to allocate and start our device. We also include a loop that sends a message to the UART every second.

```
with UART; use UART;
with Ada.Text_IO;

procedure Main is
   My_UART : UART.UART_Ref;
begin
   My_UART := new UART.UART_Device (16#ffff_ffff#, --  Vendor_Id
                                    16#aaaa_aaaa#, --  Device_Id
                                    16#8000_1000#, --  Base Address
                                    8032);         --  TCP Port

   --  Start the Device loop

   My_UART.Start;

   --  Now we are ready to receive connection from GNATemulator

   Ada.Text_IO.Put_Line ("Start Simulation");

   for Cnt in 1 .. 60 loop
      My_UART.UC.Put ("Send Message: " & Cnt'Img & ASCII.LF);
```

```
      delay 1.0;
   end loop;

   --  Abort the device loop

   My_UART.Kill;
end Main;
```

Note that the device's TCP port is 8032 and its base address is hexadecimal 80001000.

#### Compilation

With all the source files prepared (`main.adb`, `uart.adb`, `uart.ads`, `uart_controller.adb` and `uart_controller.ads`) we can build build the UART device program.

```
# Add GNATBus's project files directory in ADA_PROJECT_PATH
$ export ADA_PROJECT_PATH=<PATH_TO_GNATEMULATOR>/lib/gnat:$ADA_PROJECT_PATH
# And run gprbuild
$ gprbuild -Puart.gpr
```

We also have to build the guest executable. To do so, follow the instruction in <PATH_TO_GNATEMULATOR>/share/examples/gnatemu/gnatbus/uart/guest_code/README.

#### Device connection and execution

To set up your simulation environment, you first have to start the device

```
$ ./gnatbus_uart
```

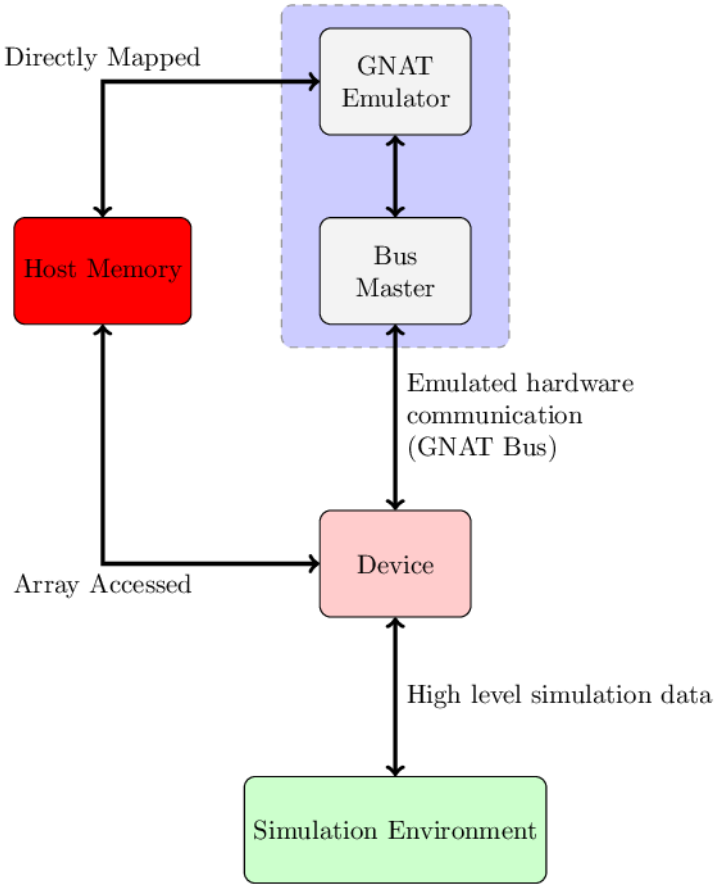
and then in another terminal, start **GNAT Emulator** with the **GNAT Bus** switch and a comma-separated list of "host-name:port" items.

Our device uses port 8032.

```
$ leon3-elf-gnatemu --gnatbus=localhost:8032 guest_uart
```

### 7.2.4 Sharing some host memory with the guest

Simulating some devices might require a lot of data to be loaded / stored. To improve performance GNATBus allow to map in the guest address space some host memory. Hence there are less operations to share large chunk of memory.




### Modifying the uart to output 1K of data in one shot

Mapping the memory is quite easy in the above GNATBus Device's Device_Setup it only requires to register a shared memory.

Body of `Device_Setup` procedure for `UART_Device`:

```
------------------
-- Device_Setup --
------------------

procedure Device_Setup (Self : in out UART_Device) is
begin
   Ada.Text_IO.Put_Line ("Device_Setup");

   --  Register the only I/O area: 8 bytes at base address to match the two
   --  registers.

   Self.Register_IO_Memory (Self.Base_Address, 8);

   --  Register a 1K shared memory area called "/foo": it is directly
   --  mapped at 0x80002000 in the guest address space.

   Self.Register_Shared_Memory (16#80002000#, 1024, "foo");

   --  Set UART_Device access in the UART_Control protected object
```

```
   Self.UC.Set_Device (Self'Unchecked_Access);
end Device_Setup;
```

The data is accessible on the device side as well. for example on a register write:

Body of `IO_Write` procedure for `UART_Device`:

```
--------------
-- IO_Write --
--------------

procedure IO_Write (Self    : in out UART_Device;
                    Address : Bus_Address;
                    Length  : Bus_Address;
                    Value   : Bus_Data) is

   --  Reading / Writing to Shm write to the host memory directly.
   type Shm_Array is array (1 .. 1024) of aliased Interfaces.Unsigned_8;
   Shm : Shm_Array;
   pragma Suppress_Initialization (Shm);
   -- Calling this synchronize the Shm.
   for Shm'Address use Self.Shm_Map (Id => 0);

   pragma Unreferenced (Length);
begin
   Ada.Text_IO.Put_Line ("Write @ " & Address'Img);

   --  case statement on the relative address

   case Address - Self.Base_Address is
      when 0 =>
         --  Set Control register value
         Self.UC.Set_CTRL (Value);
      when 4 =>
         --  Synchronize the Shm to ensure that all the write from the
         --  simulator are synchronized. This is implemented as no-op on
         --  modern OS.. but is not guaranted to be optional by the
         --  documentation.
         Self.Shm_Sync(Id => 0);

         for U in Shm'First .. Shm'Last loop
             --  Do something with the data..
         end loop;
      when others =>
         Ada.Text_IO.Put_Line ("Write unknown register:" & Address'Img);
   end case;
end IO_Write;
```

CHAPTER

EIGHT

# INDICES AND TABLES

- *genindex*
- *search*

## Symbols



## A



## E



## G