GNATemulator Documentation

Release 27.0w (20260427)

AdaCore

Apr 28, 2026

CONTENTS

1 Introduction **3**
- 1.1 About GNATemulator . . . 3
- 1.2 Product Content . . . 3
- 1.3 Note on the Documentation . . . 3

2 Getting Started **5**
- 2.1 Installation . . . 5
- 2.2 Setting your environment . . . 5
- 2.3 Running the examples . . . 6

3 Using GNATemulator **9**
- 3.1 Launching GNATemulator . . . 9
- 3.2 Displaying the help . . . 9
- 3.3 GNAT Project File . . . 10
- 3.4 Debugging . . . 11
- 3.5 Redirecting serial port(s) . . . 13
- 3.6 Connecting to GNAT Bus devices . . . 14
- 3.7 Board selection . . . 14
- 3.8 Access to host file system . . . 14
- 3.9 Guest reset behavior . . . 15
- 3.10 Snapshot usage (beta feature) . . . 15

4 Extending GNATemulator **17**
- 4.1 Introduction . . . 17
- 4.2 **GNAT Bus** . . . 17

5 Indices and tables **31**

Index **33**

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. 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 three 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.
- **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
```

Always replace `<target>` by the relevant target prefix in your context. As a reminder here is the list of target supported by **GNATemulator** along with the expected tool name:

Platform	Tool name
AARCH64 ELF	aarch64-elf-gnatemu
ARM ELF	arm-eabi-gnatemu
LEON 3 ELF	leon3-elf-gnatemu
MORELLO ELF	morello-elf-gnatemu
PowerPC ELF	powerpc-elf-gnatemu
RISCV32 ELF	riscv32-elf-gnatemu
RISCV64 ELF	riscv64-elf-gnatemu
X86_64 ELF	x86_64-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`).

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 GPR_PROJECT_PATH=%GNATEMULATOR_INSTALL_DIR%\share\gpr;%GPR_PROJECT_PATH%
```

And on Linux:

```
export GPR_PROJECT_PATH=$GNATEMULATOR_INSTALL_DIR/share/gpr:$GPR_PROJECT_PATH
```

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

2.3 Running the examples

In the following subsections, small examples are described for each supported platforms. The examples sources are located in `%GNATEMULATOR_INSTALL_DIR%/share/examples/gnatemu/`.

2.3.1 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:
```

(continues on next page)

(continued from previous page)

```
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.

2.3.2 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=embedded-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.

2.3.3 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=embedded-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.

2.3.4 The RISC-V64 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=riscv64-elf -P hello.gpr --RTS=embedded-polarfiresoc
```

Running the example

To launch the example just run:

```
riscv64-elf-gnatemu --board=polarfiresoc 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:

```
$ <target>-gnatemu hello
```

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

To stop, press `Control-a x`.

3.2 Displaying the help

All the available options can be listed with the –help arguments:

```
$ <target>-gnatemu --help
Usage: <target>-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: .)
  --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
  --auto-reboot        Don't exit the emulator when the guest reboots or resets.
                       The emulator will run until stopped by the user.
  --gnatbus=HOST:PORT[,HOST:PORT]
                       Connect a GNATBus device
  --gnatbus-timeout=timeout
                       Specify the connection timeout for GNATBus devices
```

(continues on next page)

(continued from previous page)

```
  --add-memory=name=MEM_NAME,size=MEM_SIZE,addr=MEM_ADDR[,read-only=on|off]
                       Add a memory bank to the emulated address space.
                       Address can be in hexadecimal (0xF0000000) and
                       size accepts (K)ilo, (M)ega, (G)iga, (T)era postfix.
  --show-memory-map    Display the emulated address space
  --emulator-help      Display available Qemu options
  --eargs              Start a group of Qemu options
  --eargs-end          End a group of Qemu options
  --board=BOARD_NAME
  --enable-qmp=tcp[,port=PORT]
                       Make QMP available through TCP on port PORT (default: 1235)
  --enable-qmp=unix,path=PATH
                       Make QMP available through a Unix socket file located at PATH
  --snapshot-load      (beta) Load a migration state file: When this option is
                       used, the last argument of the command line specifies the
                       file containing the saved VM state, replacing the usual
                       executable/kernel.
```

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

For example:

```
package Emulator is
   for Board use "BOARD_NAME";
   for Debug_Port use "1234";
   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:

```
$ <target>-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.

-g

This is a shortcut for *--gdb --freeze-on-startup*.

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:

```
$ <target>-gnatemu -g hello
```

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

```
$ <target>-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\0305-036\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\0305-036\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 GNAT Studio

To debug with GNAT Studio, you should:

1. Read the GNAT Studio documentation, section "Working in a Cross Environment -> Debugger Issues" and set the following project attributes::

```
package IDE is
   for Program_Host use ":1234";
   for Communication_Protocol use "remote";
end IDE;
```

2. Once you have configured your project for cross-debugging in GNAT Studio, just click on the *Debug with Emulator* toolbar button: this will build the executable, launch GNATemulator with the right options, and start a debugging session in GNAT Studio, automatically connecting the debugger to the port specified via the *Emulator'Debug_Port* and *IDE'Program_Host* project attributes (the two port values should match).

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:

```
$ <target>-gnatemu --serial=stdio hello
```

2. Redirection to null:

```
$ <target>-gnatemu --serial=null hello
```

3. Redirection to a file (write only):

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

4. To a TCP port:

```
$ <target>-gnatemu --serial=tcp:hostname:1234 hello
```

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

```
$ <target>-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:

```
$ <target>-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 <target>-gnatemu —help to get the list of boards.

3.8 Access to host file system

(only for aarch64-elf, arm-elf, leon3-elf, morello-elf, ppc-elf, riscv64-elf and x86_64-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
```

(continues on next page)

(continued from previous page)

```
   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:

```
$ # Add GNATBus's project files directory in GPR_PROJECT_PATH
$ export GPR_PROJECT_PATH=<PATH_TO_GNATEMULATOR>/share/gpr:$GPR_PROJECT_PATH
$ # And run gprbuild
$ gprbuild --target=leon3-elf -XGNATEMU_BOARD=leon3-elf test
```

3.9 Guest reset behavior

By default, GNATemulator exits when the guest is reset. This behavior can be changed using `--auto-reboot` which will allow the guest to reset without exiting GNATemulator. Note that in this case the emulator will continue to run until it is explicitly stopped or exited by the user. The parts of the emulator state that are preserved over the reset (such as the contents of RAM) is highly dependent on the emulated target board and the application.

3.10 Snapshot usage (beta feature)

(only for aarch64-elf)

To save the state of a virtual machine, it is first necessary to launch GNATemulator with the QMP communication protocol enabled using the `--enable-qmp` option. Once the emulator is running, a 'migrate' command can be sent via the QMP interface to save the complete state of the virtual machine, including the CPU and memory, into a file. This generated file, commonly named "snapshot", can then be used to start one or several new GNATemulator instances. By using the `--snapshot-load` option, each new instance will load the VM state from the file, which allows for efficiently cloning and replicating a specific machine state.

CHAPTER

FOUR

EXTENDING GNATEMULATOR

4.1 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.

4.2 GNAT Bus

4.2.1 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**

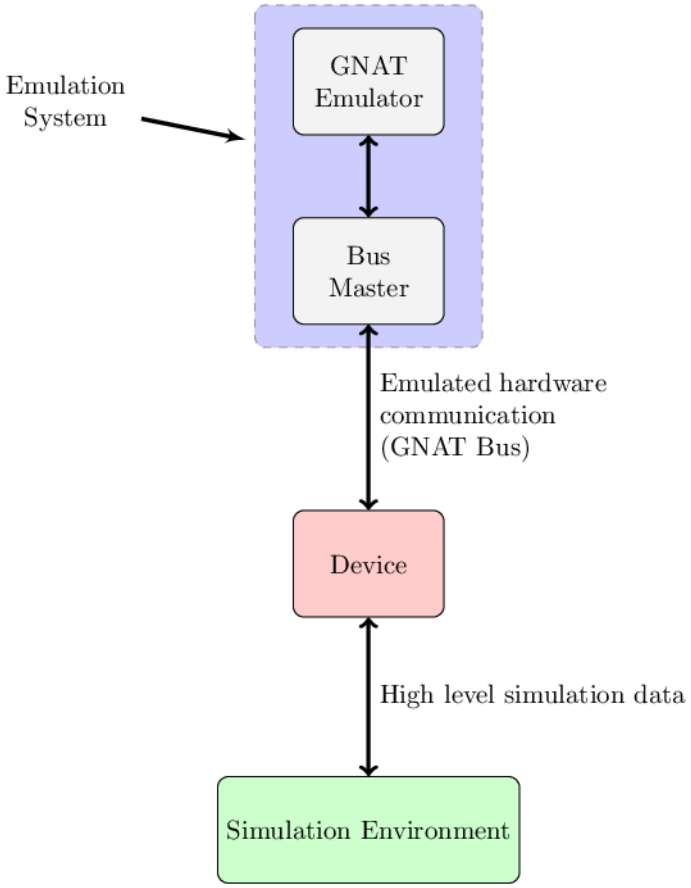
   From the device you can trigger interrupts on the emulated system.

   - 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.



4.2.2 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
```

4.2.3 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 1: UART registers

Register	Offset
UART Control	0x0
UART Data	0x4

The following tables describe the fields of each register:

Table 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 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

In order to explicit the separation, it's easier to split the project into two directory:

- *native/* containing the project for GNATBus device, run on the host.
- *guest_code/* containing the project for the guest part, run inside GNATEmulator.

Our project directory tree will look like

```
uart/
 |-- guest_code/
      |-- src/
 `-- native/
      |-- src/
```

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

```
with "gnatbus_ada.gpr";

project UART is

  for Languages    use ("Ada");
  for Source_Dirs  use ("src", "../../helpers");
  for Object_Dir   use "obj";
  for Exec_Dir     use ".";
  for Main         use ("main.adb");

  package Naming is
     for Spec("Board_Parameters") use
       "board_parameters-" & external ("GNATEMU_BOARD") & ".ads";
  end Naming;

  package Compiler is
     for Default_Switches ("Ada") use ("-gnaty", "-gnatwa");
  end Compiler;

  package Builder is
     for Executable ("main.adb") use "gnatbus_uart";
  end Builder;

  package Linker is
     for Required_Switches use GnatBus_Ada.Required_Linker_Switches;
  end Linker;
end UART;
```

`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.

`helpers/`, available under GNATBus example directory, contains a set of default values in order to ease the portability across boards.

`GNATEMU_BOARD` is the board you wish to target. This will fetch the correct helper files stored above.

package UART_Controller

```
uart/native/src/uart_controller.ads
uart/native/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/native`).

package UART_Device

```
uart/native/src/uart.ads
uart/native/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
   Connected : Boolean := False;
   --  A boolean storing the status of the connection

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, when the emulator have been connected. Therefore, we can use it to update our `Connected` field.

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");
   Self.Connected := True;
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/native/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
```

(continues on next page)

(continued from previous page)

```
   My_UART : UART.UART_Ref;
begin
   My_UART := new UART.Uart_Device
     (16#ffff_ffff#,                -- Vendor_Id
      16#aaaa_aaaa#,                -- Device_Id
      Board_Parameters.Addr,        -- Base Address
      8032);                        -- TCP Port

   -- Start the Device loop

   My_UART.Start;

   -- Wait for the connection

   while not My_Uart.Connected loop
       delay 1.0;
   end loop;

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

   -- Send three messages

   For Cnt in 1 .. 3 loop
       My_UART.UC.Put ("Send Message: " & Cnt'Img & ASCII.LF);
       delay 1.0;
   end loop;

   -- Request the target to shutdown

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

   My_Uart.Shutdown_Request;

   -- Stop the device and exit

   My_Uart.Wait_Termination;

end Main;
```

The device's TCP port is hardcoded to 8032 while the base address is retrieved inside the helper files.

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 GPR_PROJECT_PATH
$ export GPR_PROJECT_PATH=<PATH_TO_GNATEMULATOR>/share/gpr:$GPR_PROJECT_PATH
# And run gprbuild
$ cd native
$ gprbuild -Puart.gpr -XGNATEMU_BOARD=<board>
```

We also have to build the guest executable.

```
$ cd guest_code
$ gprbuild --target=<target> --RTS=<runtime>
    -Pguest_uart.gpr -XGNATEMU_BOARD=<board>
```

Device connection and execution

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

```
$ ./gnatbus_uart
```

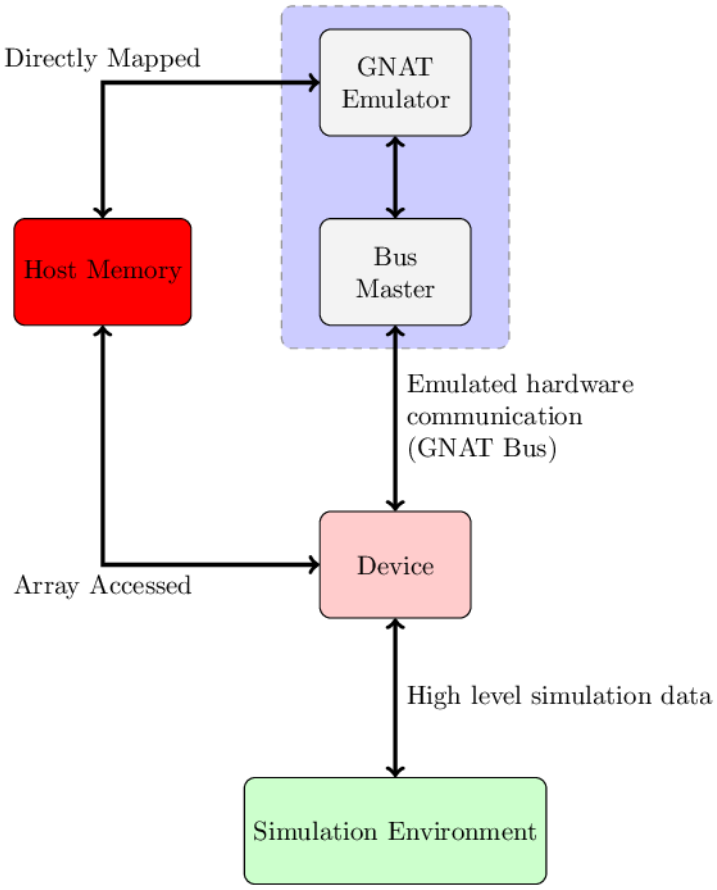
and then in another terminal, start **GNAT Emulator** with the **GNAT Bus** switch and a comma-separated list of "host-name:port" items.

Our device uses port 8032.

```
$ <target>-gnatemu --gnatbus=localhost:8032 guest_uart
```

4.2.4 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
```

(continues on next page)

(continued from previous page)

```
   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

FIVE

INDICES AND TABLES

- genindex
- search

INDEX

Symbols

---board
    gnatemu command line option, 14
--freeze-on-startup
    gnatemu command line option, 11
--gdb
    gnatemu command line option, 11
--gnatbus
    gnatemu command line option, 14
--serial
    gnatemu command line option, 13
--serial1
    gnatemu command line option, 13
--serial2
    gnatemu command line option, 13
--serialN
    gnatemu command line option, 13
-g
    gnatemu command line option, 11

E

environment variable
    GNATEMULATOR_INSTALL_DIR, 5
    GPR_PROJECT_PATH, 5

G

gnatemu command line option
    ---board, 14
    --freeze-on-startup, 11
    --gdb, 11
    --gnatbus, 14
    --serial, 13
    --serial1, 13
    --serial2, 13
    --serialN, 13
    -g, 11
GNATEMULATOR_INSTALL_DIR, 5
GPR_PROJECT_PATH, 5