# 

# 

# 

# 

prplHypervisor Deployment on PIC32MZ2048EF Platform

## 

October 2016

**Contents**

[**1. Introduction**](#_if1spktb6ol0)[**3**](#_if1spktb6ol0)

[**2. Prerequisites**](#_dzfyuzkuwllu)[**3**](#_dzfyuzkuwllu)

[**3. Linux Host Packages Requirements**](#_sqqhhz1f1ek8)[**3**](#_sqqhhz1f1ek8)

[3.1 Remove ModemManager Software Package](#_kp989mtc52vp) [3](#_kp989mtc52vp)

[**4. Third Party Software and Tools**](#_xrk868jk8gnn)[**4**](#_xrk868jk8gnn)

[**5. Hypervisor Deployment**](#_eagiuvj72ceh)[**4**](#_eagiuvj72ceh)

[5.1 Obtaining the prplHypervisor sources](#_hlkf7wctm0e3) [5](#_hlkf7wctm0e3)

[5.2 Flashing the bootloader](#_cd7ub3ivbq8d) [5](#_cd7ub3ivbq8d)

[5.3 Compiling the prplHypervisor](#_wnlvglxbf01i) [5](#_wnlvglxbf01i)

[5.4 Uploading the firmware](#_wbtydqe0l23n) [7](#_wbtydqe0l23n)

[5.5 Serial output](#_2wav4ir8ydak) [8](#_2wav4ir8ydak)

[**6. Hypervisor configuration**](#_8rshuxga74mw)[**9**](#_8rshuxga74mw)

[6.1 system group](#_deqqa6bxbggv) [9](#_deqqa6bxbggv)

[6.1.1 Example of a valid system configuration](#_kjkj0pnikqi5) [9](#_kjkj0pnikqi5)

[6.2 virtual\_machines group](#_hb68ft9nen01) [10](#_hb68ft9nen01)

[6.2.1 Example of a valid virtual\_machine configuration](#_gidqyu2ntlag) [11](#_gidqyu2ntlag)

[6.3 memory\_maps versus device\_mapping](#_jytyb16jn0rj) [12](#_jytyb16jn0rj)

[6.4 interrupt\_redirect property](#_7taml36s4gxq) [12](#_7taml36s4gxq)

[**7. Bare-metal application development**](#_ugn4h1ij89v3)[**12**](#_ugn4h1ij89v3)

[7.1 Timer](#_441u6390st2r) [13](#_441u6390st2r)

[7.2 Inter-VM Communication](#_5knfv4hk6nop) [13](#_5knfv4hk6nop)

[7.3 Unix-like API](#_qg4gc3vtmo1o) [14](#_qg4gc3vtmo1o)

[7.4 UART access](#_c3x3twrmt18z) [15](#_c3x3twrmt18z)

[7.5 Virtualized I/O](#_uptf5p9mgbjk) [15](#_uptf5p9mgbjk)

[7.6 Implementing device drivers at guest level](#_rjhbri8pqyo8) [15](#_rjhbri8pqyo8)

[7.7 Available hypercalls](#_xqb9uz1yj25p) [16](#_xqb9uz1yj25p)

## 

## 

## 1. Introduction

This document is intended to show how to configure your host environment and deploy the prplHypervisor with virtualized bare-metal applications to the Microchip PIC32MZ processor and supported boards. The table below shows the supported boards.

|  |  |  |
| --- | --- | --- |
| **Microchip PIC32MZ Embedded**  **Connectivity Starter Kit** | **Digilent PIC32MZ Chipkit**  **Wi-Fire** | **Microchip PIC32MZ Curiosity**  **Board** |

**Current PI32MZ supported boards.**

## 2. Prerequisites

* Ubuntu 14.04 LTS or higher. It should work on Debian as well.
* A PIC32MZ supported board.
* Cables for the corresponding board.

## 3. Linux Host Packages Requirements

Perform the following commands to install the required packages on Ubuntu 14.04 LTS.

|  |
| --- |
| sudo dpkg --add-architecture i386  sudo apt-get update  sudo apt-get install gcc-multilib  sudo apt-get install libc6:i386 libncurses5:i386 libstdc++6:i386  sudo apt-get install libexpat1-dev:i386  sudo apt-get install libX11-dev:i386  sudo apt-get install libXext-dev:i386 |

### 3.1 Remove ModemManager Software Package

The ModemManager software package scans for modems on individual serial ports when they are plugged, e.g. ttyACMx ports. This keeps the port busy for several seconds. To avoid this inconvenience, uninstall it from your system:

|  |
| --- |
| sudo apt-get purge modemmanager |

## 4. Third Party Software and Tools

* MPLABX IPE already installed (used to flash the bootloader for the first time). You don't need the MPLABX IDE installed, just the IPE;

|  |
| --- |
| Installation Instructions:<http://microchip.wikidot.com/ipe:installation> |

* MIPS-MTI toolchain already installed;

|  |
| --- |
| Download page:  <https://community.imgtec.com/developers/mips/tools/codescape-mips-sdk/download-codescape-mips-sdk-essentials/>; |

Note: Do not forget to add the toolchain to your system PATH variable.

* git client installed;

|  |
| --- |
| *sudo apt-get install git* |

* Installed srecord software package;

|  |
| --- |
| *sudo apt-get install srecord* |

* libconfig used to read the configuration files;

|  |
| --- |
| *sudo apt-get install libconfig-dev* |

## 5. Hypervisor Deployment

This section shows how to obtain the prplHypervisor and deploy it on a board.

### 5.1 Obtaining the prplHypervisor sources

Clone the git repository of the prplHypervisor and make sure you are at the master branch:

|  |
| --- |
| git clone https://github.com/prplfoundation/prpl-hypervisor  git checkout master |

### 5.2 Flashing the bootloader

Use the MPLABX IPE to flash the corresponding bootloader to your board. The bootloader can be found in the *prplHypervisor/bin* directory. The available bootloaders are:

* Microchip\_Curiosity\_UART.hex (Microchip PIC32MZ Curiosity board);
* Microchip\_Eth\_Starter\_Kit\_UART.hex (Microchip PIC32MZ Embedded Connectivity Starter Kit);
* chipKIT-WiFire-EF.hex (Digilent PIC32MZ Chipkit Wi-Fire Board).

These bootloaders were generated from the project <https://github.com/chipKIT32/PIC32-avrdude-bootloader>.

Connect the board to your host computer using a standard A to mini B cable. Use the mini B debugger connector on the board. After flashed, the **LED1** will keep blinking showing the that the bootloader is ready to upload the firmware.

### 5.3 Compiling the prplHypervisor

From the directory prplHypervisor created by the git clone on step 5, go to the platform directory platform corresponding to your board. The available platform (directories) are:

* platform/pic32mz\_starter\_kit;
* platform/pic32mz\_chipkit\_Wifire, and;
* platform/pic32mz\_curiosity.

On the chosen directory, use the make command to compile the hypervisor with its default configuration using the MIPS-MTI toolchain. This process will generate the *firmware.hex* file. This file contains the hypervisor and guests. Example:

|  |
| --- |
| ~/prplHypervisor/platform/pic32mz\_starter\_kit$: make  gcc -o genconf cfg\_reader/genconf.c -lconfig  #execute first and exit in case of errors  ./genconf samples\_cfg/sample-1VM.cfg || (exit 1)  blink  #execute and export to a makefile variable the ouput of the script  mips-mti-elf-gcc -EL -O2 -mtune=m14k -mips32r2 -Wa,-mvirt -mno-check-zero-division -msoft-float -fshort-double -c -ffreestanding -nostdlib -fomit-frame-pointer -G 0 -DETHERNET\_SUPPORT -DUSB\_SUPPORT -DCPU\_SPEED=200000000 -DHYPVERSION="v0.11.27 (g9100f1a)" -I../../arch/mips/pic32mz\_starter\_kit/include -I../../arch/mips/common/include -I../../sys/lib/include -I../../sys/kernel/include -I../../platform/pic32mz\_starter\_kit/include \  ../../sys/lib/libc.c \  ../../sys/lib/linkedlist.c \  ../../sys/lib/malloc.c  ...  mips-mti-elf-size blink.elf  text data bss dec hex filename  12248 16 1424 13688 3578 blink.elf  make[1]: Saindo do diretório `/home/moratelli/hyper/prpl-hypervisor/bare-metal-apps/apps/blink'  ../../scripts/genhex.sh blink  0+1 registros de entrada  1+0 registros de saída  65536 bytes (66 kB) copiados, 0,000635281 s, 103 MB/s  0+1 registros de entrada  1+0 registros de saída  32768 bytes (33 kB) copiados, 0,000245427 s, 134 MB/s |

**Note:** See the main Makefile in your platform directory. You may want to modify the input configuration file (CFG\_FILE), CROSS\_COMPILER or the optional features. Example:

|  |
| --- |
| #Input CFG file for VMs configuration  CFG\_FILE = ../samples\_cfg/sample-1VM.cfg  # CROSS Compiler  CROSS\_COMPILER = mips-mti-elf-  # Optional Device Drivers  CONFIG\_INTERVMCOMM\_DRV = yes  CONFIG\_PIC32MZ\_USB\_DRV = yes  CONFIG\_PIC32MZ\_ETHERNET\_DRV = yes  CONFIG\_VIRTUAL\_IO\_DRV = yes  CONFIG\_INTERRUPT\_REDIRECT\_DRV = yes |

### 5.4 Uploading the firmware

Upload the resulting *firmware.hex* file to the board using the *make load* command. Make sure that you have a cable connected to the UART port on your board. The following table shows the required cable to the supported boards.

|  |  |  |
| --- | --- | --- |
| **Board** | **Required Cables** | **Connections** |
| Microchip PIC32MZ Connectivity Starter Kit | 2 x Standard A to mini B cables. | Connect the cables to connectors J3 (power) and J11 (UART). |
| Digilent PIC32MZ Chiptkit Wi-Fire | 1 x Standard A to mini B cables. | Connect the cable to connector J1 (power/UART). |
| Microchip PIC32MZ Curiosity board | 1 x Standard A to mini B cables.  1 x Serial to USB adapter. | Connect the mini B cable connector to J3 (power). Use the USB adapter on the UART pins on MikroBus 1. |

Expected *make load* command output.

|  |
| --- |
| ~/prplHypervisor/platform/pic32mz\_starter\_kit$: make load  stty 115200 raw cs8 -hupcl -parenb -crtscts clocal cread ignpar ignbrk -ixon -ixoff -ixany -brkint -icrnl -imaxbel -opost -onlcr -isig -icanon -iexten -echo -echoe -echok -echoctl -echoke -F /dev/ttyACM0  ./pic32prog -S -d /dev/ttyACM0 firmware.hex  Programmer for Microchip PIC32 microcontrollers, Version 2.0.186  Copyright: (C) 2011-2015 Serge Vakulenko  Adapter: STK500v2 Bootloader  Program area: 1d000000-1d1fffff  Processor: Bootloader  Flash memory: 2048 kbytes  Boot memory: 80 kbytes  Data: 131072 bytes  Erase: done  Program flash: ################################# done  Program rate: 6600 bytes per second |

During power on, the bootloader will wait for 3 seconds before automatically loading the application. During this time, it is possible to flash a new firmware. Thus, before try to upload the firmware, press the reset button on your board and wait the blinking led.

### 5.5 Serial output

You can make a *cat* command to the corresponding serial port see the application’s output. The expected output the default hypervisor configuration is listed below. The default configuration generates the ping-pong application.

|  |
| --- |
| ===========================================================  prplHypervsior v0.11.112 (ge35a934) [Oct 13 2016, 20:26:29]  Copyright (c) 2016, prpl Foundation  ===========================================================  CPU ID: M5150  ARCH: Microchip Starter Kit  SYSCLK: 200MHz  Heap Size: 29Kbytes  Scheduler 5ms  VMs: 2  Initializing Virtual Machines.  Configuring ping VM starting at 0x80008000 RAM address.  Configuring pong VM starting at 0x80010000 RAM address.  PIC32mz in Vectored Interrupt Mode.  Inter-VM communication hypercalls registered.  Software reset interrupt (SW1) registered at offset 0x220.  Initializing USB device in Host mode.  USB interrupt vector at 0x240  Device mapping hypercalls registered.  Ethernet PHY at 0  Ethernet interface en0: interrupt 153, MAC address d8:80:39:75:a6:1a  Ethernet device is a SMSC LAN8740A  CP0 Timer interrupt registered at 0x260.  ping VM ID 1  Target VCPU not  pong VM ID 2 initialized.  pong VM: message from VM ID 1: "ping? 5" (8 bytes)  ping VM: message from VM ID 2: "pong!" (6 bytes)  pong VM: message from VM ID 1: "ping? 20" (9 bytes)  ping VM: message from VM ID 2: "pong!" (6 bytes)  pong VM: message from VM ID 1: "ping? 35" (9 bytes)  ping VM: message from VM ID 2: "pong!" (6 bytes)  pong VM: message from VM ID 1: "ping? 50" (9 bytes)  ping VM: message from VM ID 2: "pong!" (6 bytes)  pong VM: message from VM ID 1: "ping? 65" (9 bytes)  ... |

## 6. Hypervisor configuration

The prplHypervisor configuration is made through readable configuration files (cfg). During the compilation, the cfg file is read and interpreted to generate the config.h header in the platform directory. The *prplHypervisor/platform/cfg* directory has some configuration file samples. The cfg files are structured in two main groups: *system* and *virtual\_machines*. The *system* group contains system wide configuration, while the *virtual\_machines* group contains the configuration of each VM. The properties of each group are detailed below.

### 6.1 system group

System group properties description.

|  |  |  |  |
| --- | --- | --- | --- |
| **Propriety** | **Type** | **Description** | **Valid Range** |
| debug | Array of strings | Optional list of defines for output messages for debug purposes. | WARNINGS, INFOS, and ERRORS. |
| platform | Property group | Specific platform information | Not applicable |
| platform.cpu | string | CPU description. | Any valid string |
| platform.platform\_str | string | Platform description | Any valid string. |
| platform.system\_clock | integer | Processor's clock in Hertz. | Positive integers |
| uart\_speed | integer | Console UART baudrate | 115200, 57600 and 9600. Other baud rates may be used. |
| scheduler\_quantum\_ms | integer | Timer interval in milliseconds for the hypervisor scheduler timer (QUANTUM) | Positive integers. |

#### 6.1.1 Example of a valid system configuration

|  |
| --- |
| system = {  debug = [ "WARNINGS", "INFOS", "ERRORS"];  platform = {  cpu = "M5150";  platform\_str = "Microchip Starter Kit";  system\_clock = 200000000;  };  uart\_speed = 115200;  scheduler\_quantum\_ms = 5;  }; |

### 6.2 virtual\_machines group

Description of the virtual\_machines group properties.

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Type** | **Description** | **Valid Range** |
| app\_name | string | Name of the bare-metal application directory. | Any valid bare-metal app directory name. |
| os\_type | string | Type of the virtualized OS. | BARE\_METAL |
| RAM\_size\_bytes | string | Size of the VM's SRAM. | MEM\_SIZE\_32KB MEM\_SIZE\_64KB MEM\_SIZE\_128KB  MEM\_SIZE\_256KB |
| flash\_size\_bytes | string | Size of the VM's FLASH. | MEM\_SIZE\_32KB MEM\_SIZE\_64KB MEM\_SIZE\_128KB MEM\_SIZE\_256KB MEM\_SIZE\_512KB MEM\_SIZE\_1MB |
| fast\_interrupts | Array of strings | List of associated interrupts. Allows for the hypervisor to fast delivery an interrupt to a VM. | IRQ\_ETH |
| device\_mapping | Array of strings | List of associated devices for virtualized I/O. | PORTA through PORTK, UART1 through UART6, SPI1 through SP6, PM and AD1. |
| interrupt\_redirect | Array of strings | List of interrupts to redirect to the guest. | All possible IRQs. For example, IRQ\_U2RX. See the complete list in pic32mz.h file. |
| memory\_maps | Array of properties | Indicate the VM's memory mapping. | Not applicable. |
| memory\_maps.base\_addr | integer | Memory mapping base address | Any virtual address |
| memory\_maps.page\_size | string | Memory mapping size | MEM\_SIZE\_4KB  MEM\_SIZE\_8KB MEM\_SIZE\_16KB MEM\_SIZE\_32KB MEM\_SIZE\_64KB MEM\_SIZE\_128KB MEM\_SIZE\_256KB MEM\_SIZE\_512KB MEM\_SIZE\_1MB |

#### 6.2.1 Example of a valid virtual\_machine configuration

The blink VM has 32KB of SRAM, 64KB of FLASH access to PORTH and UART2 through virtualized I/O. The pong VM has 32KB of SRAM and FLASH, access to UART2 through virtualized I/O and a 4KB page mapped for direct peripheral access on base address 0xbf860000.

|  |
| --- |
| virtual\_machines = (  {  app\_name = "blink";  os\_type = "BARE\_METAL";  RAM\_size\_bytes = "MEM\_SIZE\_32KB";  flash\_size\_bytes = "MEM\_SIZE\_64KB";  device\_mapping = [ "PORTH", "UART2"];  },  {  app\_name = "pong";  os\_type = "BARE\_METAL";  RAM\_size\_bytes = "MEM\_SIZE\_32KB";  flash\_size\_bytes = "MEM\_SIZE\_32KB";  device\_mapping = [ "UART2"];  memory\_maps = ({  base\_addr = 0xbf860000;  page\_size = "MEM\_SIZE\_4KB";  })  }); |

### 6.3 memory\_maps *versus* device\_mapping

The property *memory\_maps* is an array of memory addresses and sizes that allows for directly mapped memory areas. It is useful when low virtualization overhead is desired since a guest can have direct access to one or more peripherals. However, it is impossible to separate peripherals that are in the same virtual memory page, like the UARTs in the PIC32MZ. For example, using direct memory mapping is not possible to map UART1 to guest 1 and UART2 to guest 2 and keep them separated. For this case, virtualized I/O is preferable. Use the *device\_mapping* property to allow for guests to have access to specific peripherics. Using *device\_mapping* is possible to map UART1 to guest 1 and UART2 to guest 2 and still keep than completely separated. As the hypervisor knows the space address of each peripheral it can control guests accesses to the hardware. When using virtualized I/O, the guest must use read()/write() hypercalls to access the peripherals. See the Section 7.5 for more details.

### 6.4 interrupt\_redirect property

The PIC32MZ processor does not implement the *interrupt pass-through* feature making impossible for a guest to handle an interrupt without hypervisor intervention. The prplHypervisor implements interrupt redirection to the guests using the *interrupt\_redirect* property. The interrupts associated with the guest must be indicated in this property. The hypervisor enables the associated interrupts injecting virtual interrupts into the guests when necessary.

This feature is especially useful when implementing device drivers at guest level since it allows for the guest to receive events generated by hardware controllers. See Section 7.6 for more details.

## 7. Bare-metal application development

To develop new applications move to the *~/prplHypervisor/bare-metal-apps/apps* directory and see the blink, ping, pong directories among others examples. Each directory contains the implementation of a bare-metal application. These applications became guests during the hypervisor execution.

The hypervisor can perform up to 6 bare-metal applications (virtual machines). Each virtual machine has configurable SRAM and FLASH sizes, as explained above. To compile different bare-metal applications, modify the input configuration file in *platform/<platform\_directory>/Makefile*, as in the following example:

|  |
| --- |
| #Input CFG file for VMs configuration  CFG\_FILE = cfg/sample-1VM.cfg |

The easiest way to implement a new bare-metal app is to copy the directory of an existing application, for example blink, to a new directory and modify the implementation. Additionally, to compile different combinations of applications see the explanation in Section 6.

### 7.1 Timer

The hypervisor scheduler quantum can be configured in *scheduler\_quantum\_ms* property (see section 6.1). Thus, each VM will perform for *scheduler\_quantum\_ms* milliseconds before it is preempted. During its execution, the guest will receive timer interrupts each millisecond. To handle the timer interrupts, the guest must register an interrupt handler calling:

|  |
| --- |
| interrupt\_register(irq\_timer, GUEST\_TIMER\_INT); |

Where irq\_timer is the interrupt handler.

### 7.2 Inter-VM Communication

The prplHypervisor supports Inter-VM communication through message copies from origin to the target VM. The API is simple and consists of the following calls:

|  |
| --- |
| int32\_t ReceiveMessage(uint32\_t \*source, char\* buffer, uint32\_t bufsz, uint32\_t block);  uint32\_t SendMessage(uint32\_t target, char\* buffer, uint32\_t size); |

Each VM has an identification number (ID) generated by the hypervisor during its initialization. The ID is attributed following the VM's initialization order. The first initialized VM’s ID is 1, the second is 2 and so on. The ID is used to identify the VMs during communication. A VM can discover its own ID calling the hypercall *get\_guestid()*.

The ReiceveMessage() hypercall returns greater than 0 if a message was received. The data is copied to the buffer limited to bufsz bytes. Zero means no data was received. The SendMessage() hypercall returns the number of bytes sent. Zero or less means an error. The follow table shows the possible error codes.

|  |  |
| --- | --- |
| **Error Code** | **Meaning** |
| MESSAGE\_VCPU\_NOT\_FOUND | Destination not found. |
| MESSAGE\_FULL | Destination queue full. |
| MESSAGE\_TOO\_BIG | Message to big. |
| MESSAGE\_VCPU\_NOT\_INIT | The target VPCU is not ready for communication. |

If non-blocking receive is desired, use block equal to zero when calling ReceiveMessage(), otherwise, it will block the application until a message is received.

The application Ping-Pong is an example of inter-VM communication. A message is sent from ping (VM ID 1) to pong (VM ID 2) and back. A cfg file is provided with a configuration sample for inter-VM communication. The cfg file can be found at *prplHypervisor/platform/cfg/sample\_2VMs.cfg*. To compile the Ping-Pong example, it is needed to adjust the CFG\_FILE variable in the platform Makefile to:

|  |
| --- |
| #Input CFG file for VMs configuration  CFG\_FILE = ../samples\_cfg/sample-2VMs.cfg |

### 7.3 Unix-like API

The bare-metal application's execution environment supports a minimal Unix-like API. The available functions are listed below:

|  |
| --- |
| int8\_t \*strcpy(int8\_t \*dst, const int8\_t \*src);  int8\_t \*strncpy(int8\_t \*s1, int8\_t \*s2, int32\_t n);  int8\_t \*strcat(int8\_t \*dst, const int8\_t \*src);  int8\_t \*strncat(int8\_t \*s1, int8\_t \*s2, int32\_t n);  int32\_t strcmp(const int8\_t \*s1, const int8\_t \*s2);  int32\_t strncmp(int8\_t \*s1, int8\_t \*s2, int32\_t n);  int8\_t \*strstr(const int8\_t \*string, const int8\_t \*find);  int32\_t strlen(const int8\_t \*s);  int8\_t \*strchr(const int8\_t \*s, int32\_t c);  int8\_t \*strpbrk(int8\_t \*str, int8\_t \*set);  int8\_t \*strsep(int8\_t \*\*pp, int8\_t \*delim);  int8\_t \*strtok(int8\_t \*s, const int8\_t \*delim);  void \*memcpy(void \*dst, const void \*src, uint32\_t n);  void \*memmove(void \*dst, const void \*src, uint32\_t n);  int32\_t memcmp(const void \*cs, const void \*ct, uint32\_t n);  void \*memset(void \*s, int32\_t c, uint32\_t n);  int32\_t strtol(const int8\_t \*s, int8\_t \*\*end, int32\_t base);  int32\_t atoi(const int8\_t \*s);  int8\_t \*itoa(int32\_t i, int8\_t \*s, int32\_t base);  int32\_t puts(const int8\_t \*str);  int8\_t \*gets(int8\_t \*s);  int32\_t abs(int32\_t n);  int32\_t random(void);  void srand(uint32\_t seed);  int printf(const int8\_t \*fmt, ...);  int sprintf(int8\_t \*out, const int8\_t \*fmt, ...);  void udelay(uint32\_t usec);  void putchar(int32\_t value);  uint32\_t getchar(void); |

### 7.4 UART access

Some platforms may have a secondary UART port. To select between the available UART ports use the following call:

|  |
| --- |
| int32\_t serial\_select(uint32\_t serial\_number); |

Verify the available UART ports on your platform on the following file:

|  |
| --- |
| prplHypervisor/bare-metal-app/platform/<specific platform directory>/include/uart.h |

### 7.5 Virtualized I/O

For a complete guest memory space separation use the property *device\_mapping* to describe which peripherals the guest can access through virtualized I/O. On the guest, use *read()/write()* calls for peripheral access.

For example:

|  |
| --- |
| uint32\_t v;  write(TRISHCLR, 1);  v = read(LATD); |

### 7.6 Implementing device drivers at guest level

Some hypervisor features are useful when implementing device drivers at guest level. Combining *device\_mapping* and *interrupt\_redirect* properties it is possible to keep a complete isolation between guests but still allowing for them to have access to hardware controllers.

Use the *interrupt\_redirect* to redirect hardware interrupt events to guests. On the guest, it must to register an interrupt handler to deal with the event, also called virtual interrupt. Once an interrupt is handled by the hypervisor, it disables it (cleaning the IEC bit) and inject a virtual interrupt to the guest. In order to receive further interrupts, the guest must re-enable the interrupt calling the hypercall reenable\_interrupt().

The bare-metal application *int\_redirect* implements an example of an UART driver that receives interrupts when characters are ready to be read from the input. Additionally, this driver uses *device\_mapping* property for a complete isolation between guests. The redirection.cfg file has the setup configuration and the redirection.c on the int\_redirection application directory has the instructions to use it.

### 7.7 Available hypercalls

This is the list of all hypercalls available to the guests.

|  |
| --- |
| /\* Read from privileged address \*/  int32\_t read(uint32\_t addr);  /\* Write to privileged address \*/  int32\_t write(uint32\_t reg, uint32\_t value);  /\* InterVM send message \*/  int32\_t ipc\_send(uint32\_t targed\_id, uint8\_t\* msg, uint32\_t size);    /\* interVM recv message \*/  uint32\_t ipc\_recv(uint32\_t source\_id, uint8\_t msg);    /\* Get own guest ID \*/  uint32\_t get\_guestid();    /\* Ethernet link checker \*/  uint32\_t eth\_watch();  /\* Ethernet get mac \*/  eth\_mac(uint8\_t \*msg)    /\* Ethernet send message \*/  int32\_t eth\_send\_frame(uint8\_t \*msg, uint32\_t size);    /\* Ethernet recv message \*/  int32\_t eth\_recv\_frame(uint8\_t \*msg);    /\* USB get device descriptor \*/  int32\_t usb\_device\_descriptor(uint8\_t \*descriptor, uint32\_t size);    /\* USB polling. Updates the USB state machines. \*/  int32\_t usb\_polling()    /\* USB send data \*/  int32\_t usb\_send(uint8\_t \*msg, uint32\_t size)  /\* Re-enable an interrupt \*/  int32\_t reenable\_interrupt(uint32\_t irq); |