
The guide to Xillybus Lite

Xillybus Ltd.
www.xillybus.com

Version 3.0

1	Introduction	3
1.1	General	3
1.2	Obtaining Xillybus Lite	4
2	Usage	5
2.1	Sample design	5
2.2	Interface with Host application	6
2.3	Interface with logic design	7
2.3.1	Register related signals	7
2.3.2	Module hierarchy	8
2.3.3	32-bit aligned register access	9
2.3.4	Unaligned register access	12
2.4	Interrupts	16
3	Xillybus Lite on projects not using Xilinx	17
3.1	Applying the IP core	17
3.2	Modifying the device tree	21
3.3	Compilation of the Linux driver	23
3.4	Installing the driver	24

3.5 Loading and unloading the driver 25

1

Introduction

1.1 General

Xillybus Lite is a simple kit for easy access of registers in the logic fabric (PL) by a user space program running under Linux. It presents an illusion of a bare-metal environment to the software, and a trivial interface of address, data and read/write-enable signals to the logic design.

Using this kit frees the development team from dealing with the AXI bus interface as well as Linux kernel programming, and allows a straightforward memory-like control of the peripheral without the need for any knowledge on the the operating system or the bus protocol.

The kit consists of an IP core and a Linux driver. These are included in the Xilinx distribution for the Zedboard (versions 1.1 and up), and are also available for download separately for inclusion in projects.

Xillybus Lite does not involve any DMA functionality. The maximal data rate is around 28 MB/s (7 Million 32-bit accesses of reads or write per second, with the processor clock at 666 MHz).

The Xillybus Lite IP core is released for any use at no cost. In particular, it may be downloaded, copied and included in binaries used and sold for commercial purposes with no limitation and without any additional consent nor specific licensing.

The Xillybus Lite driver for Linux is released under GPLv2, which makes it free for distribution under the same terms as the Linux kernel itself.

1.2 Obtaining Xillybus Lite

For learning about Xillybus Lite and trying it out, it's recommended to download and install Xilinx (version 1.1 and above).

The distribution has everything set up for trying out Xillybus Lite on sample logic that can be easily modified in `xillydemo.v/vhd`. The Linux side has the driver already installed and a couple of sample programs to start off with.

Xilinx can be downloaded at <http://xillybus.com/xilinx>

To verify that an existing installation is recent enough, the following check can be run at the shell prompt on Xilinx:

```
# uname -r
3.3.0-xilinx-1.1+
```

The suffix ("1.1" in the example above) shows the Xilinx version (which is OK in this case).

Xillybus Lite can be included in any Zynq-7000 design, regardless of Xilinx. This requires the inclusion of the IP core in the XPS project, some wiring, compilation and installation of the driver, as described in section 3.

The Xillybus Lite bundle can be downloaded at <http://xillybus.com/xillybus-lite>

2

Usage

2.1 Sample design

A sample design, consisting of a connected IP core, a preinstalled Linux driver and a couple of simple demo user space programs is included in Xillinux (versions 1.1 and later).

On the logic side, the `xillydemo.v(hd)` module source file contains an implementation of a 32x32 bit RAM, which is inferred from an array. This RAM is accessed in the host's sample program. Its compilation and execution can be done directly on Xillinux as follows:

```
# make
gcc -g -Wall -I. -O3 -c -o uiotest.o uiotest.c
gcc -g -Wall -I. -O3 uiotest.o -o uiotest
gcc -g -Wall -I. -O3 -c -o intdemo.o intdemo.c
gcc -g -Wall -I. -O3 intdemo.o -o intdemo
# ./uiotest /dev/uis0 4096
0 1 2 3
```

The C sources can be found in Xillinux' file systems at `/usr/src/xillinux/xillybus-lite/` (version 1.1 and up).

The "uiotest" program merely writes four values to the first 32-bit elements in the register array, and then reads back and prints their values, but it's easily changed into something more useful.

The "intdemo" program shows how interrupts are handled. Since the sample logic doesn't trigger any interrupts, there's no point running it as is. Nevertheless, it shows how interrupts are waited for.

2.2 Interface with Host application

Xillybus Lite is based upon Linux' User I/O interface (UIO), which represents a peripheral as a device file which is primarily accessed by its memory mapping. To obtain access, the following code applies:

```
#include <sys/mman.h>

int fd;
void *map_addr;
int size = ...;

fd = open("/dev/uio0", O_RDWR);

if (fd < 0) {
    perror("Failed to open devfile");
    exit(1);
}

map_addr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED,
                fd, 0);

if (map_addr == MAP_FAILED) {
    perror("Failed to mmap");
    exit(1);
}
```

Except for error checking, this code snippet performs two operations:

- Calling the function `open()` to open the device file (obtaining a file handle).
- Calling the function `mmap()` to obtain an address for accessing the device. The second parameter ("size") is the number of bytes that are mapped. It must not exceed the number of bytes allocated for the peripheral, according to the device tree (4096 on unmodified Xilinx).

`map_addr` is an address in the virtual memory space of the process, but for all purposes it can be treated as if it was the physical address to which the peripheral is mapped in a bare-metal environment (i.e. with no operating system).

The allowed access range goes from `mem_addr` to `mem_addr + size - 1`, where "size" is the second argument given to `mmap()`. Attempting to access memory beyond this range may cause a segmentation fault.

With the address at hand, writing and reading a 32-bit word to the register at the peripheral's base address (offset zero) is just:

```
volatile unsigned int *pointer = map_addr;

*pointer = the_value_to_write;
the_value_read_from_register = *pointer;
```

On the specific memory region, memory caching is disabled by the Linux driver, and the pointer is flagged volatile. Hence each read and write operation in the program triggers a bus operation, and consequently an access cycle on the Xillybus Lite's logic interface signals.

IMPORTANT:

The pointer must be flagged as volatile with the “volatile” keyword, as shown in the example above. The lack of this flag will allow the C compiler to reorder and possibly optimize out I/O operations.

It is also fine to access the peripheral with an 8-bit volatile char pointer or a 16-bit volatile short int pointer, provided that the logic supports byte granularity access.

In the example above, it's assumed that only one Xillybus Lite peripheral is present. The first instance, “/dev/uio0” is therefore opened. If additional UIO devices are present (e.g. there's more than one Xillybus Lite instance), they are represented as /dev/uio1, /dev/uio2, etc.

In order to know which device file belongs to which logic element, the application should obtain the information in /sys/class/uio/ (e.g. /sys/class/uio/uio0/name or /sys/class/uio/uio0/maps/map0/addr). The udev framework is recommended for consistent naming of the device files when several UIO devices are created.

2.3 Interface with logic design

2.3.1 Register related signals

The Xillybus Lite IP core presents seven signals to the application logic, given here in Verilog format:

```
output        user_clk;

output [31:0] user_addr;

output        user_wren;
output [3:0]  user_wstrb;
output [31:0] user_wr_data;

output        user_rden;
input  [31:0] user_rd_data;
```

The interface is synchronous, and based upon `user_clk`, which is provided by Xillybus Lite (it's wired to the AXI Lite clock of the processor).

The signal names above are those appearing in the `Xillydemo` module (part of the Xilinx bundle). The signals' names in the processor's module are slightly different, e.g. `user_wren` may appear as `xillybus_lite_0_user_wren_pin`.

These signals can be connected directly to a standard block RAM, in which case the host gets direct access to that RAM (which can be used as a "mailbox" if a dual-port RAM is chosen). They can also be connected to registers defined in logic, as detailed below.

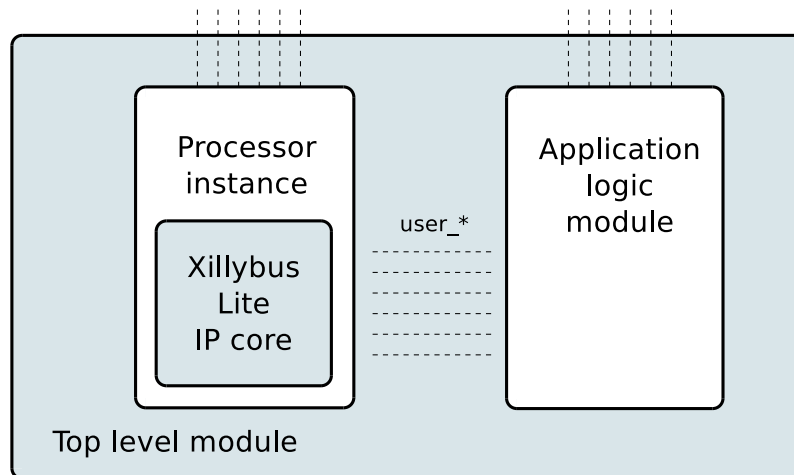
2.3.2 Module hierarchy

When a Xilinx logic design involves an embedded processor, there is a module representing it, typically instantiated in the top level module. Usually, the ports exposed by this module are all connected directly to physical pins, following the paradigm that the processor is the center of things, and that any logic around it is some kind of peripheral.

Xillybus Lite is intended for interfacing with a substantial piece of application logic, and therefore breaks this common structure somewhat: Its `user_*` signals are intended for routing to the top level module, so custom logic is instantiated in that top level module as well. The overall project's structure ends up in two large chunks: An instantiated module that contains a processor and its IP cores (including the Xillybus Lite IP core) and a second module with the application logic. The `user_*` signals connect between the two.

So even though the Xillybus Lite IP core itself is instantiated by Xilinx' tools somewhere deep inside the processor's hierarchy, it is interfaced with from the top level module.

This is the chosen layout in the demo bundle for Xilinx (shown in the drawing below) and also what this guide assumes. It's possible to connect Xillybus Lite's signals internally within the processor's hierarchy, but it's not necessarily going to make things simpler.



2.3.3 32-bit aligned register access

To access a 32x32 bit array in the logic ("litearray" below), code like the following can be used. This works fine only if the host sticks to 32-bit word access (using pointers to e.g. unsigned int only):

In Verilog:

```
always @(posedge user_clk)
begin
  if (user_wren)
    litearray[user_addr[6:2]] <= user_wr_data;

  if (user_rden)
    user_rd_data <= litearray[user_addr[6:2]];
end
```

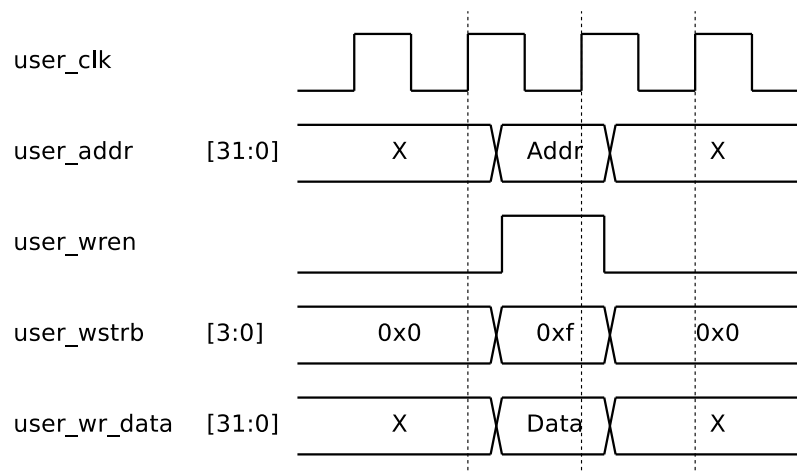
Or in VHDL:

```
lite_addr <= conv_integer(user_addr(6 DOWNT0 2));

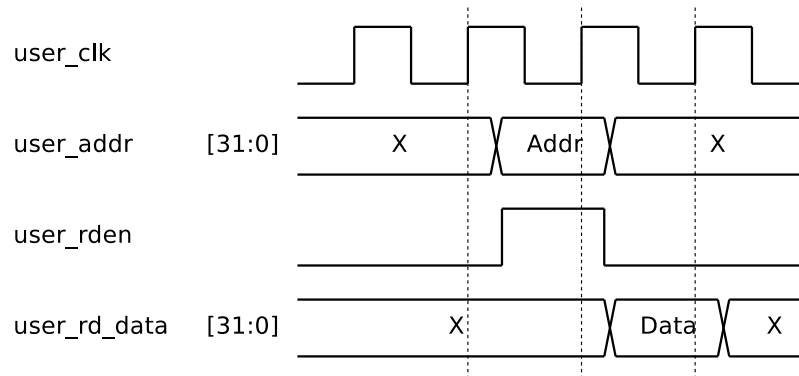
process (user_clk)
begin
  if (user_clk'event and user_clk = '1') then
    if (user_wren = '1') then
      litearray(lite_addr) <= user_wr_data;
    end if;

    if (user_rden = '1') then
      user_rd_data <= litearray(lite_addr);
    end if;
  end if;
end process;
```

The waveforms for an aligned write cycle and any read cycle are:



Waveform 1: Write cycle for aligned access



Waveform 2: Read cycle

Notes:

- Any bus operation on the address region allocated in XPS to the Xillybus Lite peripheral always results in `user_wren` or `user_rden` being high for exactly one clock cycle.
- The `user_rd_data` is sensed by the Xillybus Lite core only one clock cycle after `user_rden` is high. There is hence no practical need to monitor `user_rden`: It's also fine to always update `user_rd_data` depending on `user_addr` (with one clock's latency), for instance,

```
always @(posedge user_clk)
    user_rd_data <= litearray[user_addr[6:2]];
```

- The code above demonstrates access of a 32-bit wide array of 32 elements. A more common setting is accessing registers e.g. in Verilog

```
always @(posedge user_clk)
    if ((user_wren) && (user_addr[6:2] == 5))
        myregister <= user_wr_data;
```

for mapping "myregister" at address offset 0x14.

- Likewise, a case statement that depends on `user_addr` is the common implementation of `user_rd_data`'s value assignment, such as

```
always @(posedge user_clk)
  case (user_addr[6:2])
    5: user_rd_data <= myregister;
    6: user_rd_data <= hisregister;
    7: user_rd_data <= herregister;
    default: user_rd_data <= 0;
  endcase
```

- user_addr is 32 bit wide, and holds the full physical address being accessed. Since the enable signals are high only when the address is within the allocated range, there is no need to verify the address' MSBs.
- Always ignore user_addr[1:0]. These two LSBs are always zero on 32-bit aligned bus accesses, and as explained below, they should be ignored even for unaligned access.

2.3.4 Unaligned register access

When there's a possibility that the host will access the register space in a 32-bit unaligned manner, each byte needs to be handled separately in the logic.

Note that accessing a byte and a 32-bit word on the bus take the same time, so unaligned access is bandwidth inefficient by four times.

Suppose that litearray3, litearray2, litearray1 and litearray0 are memory arrays of 32 elements with 8 bits each. The following code snippets demonstrate how the exam-

ples in 2.3.3 are rewritten to support unaligned access. In Verilog:

```
always @(posedge user_clk)
begin
  if (user_wstrb[0])
    litearray0[user_addr[6:2]] <= user_wr_data[7:0];

  if (user_wstrb[1])
    litearray1[user_addr[6:2]] <= user_wr_data[15:8];

  if (user_wstrb[2])
    litearray2[user_addr[6:2]] <= user_wr_data[23:16];

  if (user_wstrb[3])
    litearray3[user_addr[6:2]] <= user_wr_data[31:24];

  if (user_rden)
    user_rd_data <= { litearray3[user_addr[6:2]],
                     litearray2[user_addr[6:2]],
                     litearray1[user_addr[6:2]],
                     litearray0[user_addr[6:2]] };
end
```

Or in VHDL:

```
lite_addr <= conv_integer(user_addr(6 DOWNTO 2));

process (user_clk)
begin
  if (user_clk'event and user_clk = '1') then
    if (user_wstrb(0) = '1') then
      litearray0(lite_addr) <= user_wr_data(7 DOWNTO 0);
    end if;

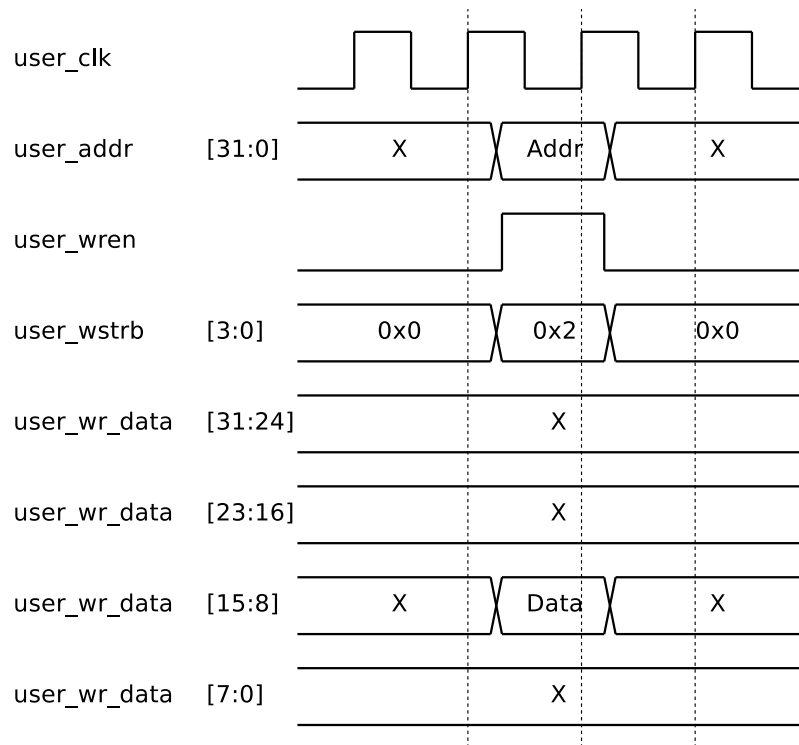
    if (user_wstrb(1) = '1') then
      litearray1(lite_addr) <= user_wr_data(15 DOWNTO 8);
    end if;

    if (user_wstrb(2) = '1') then
      litearray2(lite_addr) <= user_wr_data(23 DOWNTO 16);
    end if;

    if (user_wstrb(3) = '1') then
      litearray3(lite_addr) <= user_wr_data(31 DOWNTO 24);
    end if;

    if (user_rden = '1') then
      user_rd_data <= litearray3(lite_addr) & litearray2(lite_addr) &
        litearray1(lite_addr) & litearray0(lite_addr);
    end if;
  end if;
end process;
```

The waveform for an unaligned write cycle on a single byte with 0x01 offset from the base address follows.



Waveform 3: Write cycle for unaligned access (byte offset 0x01 shown)

Notes:

- A write bus operation on the allocated address region always results in user_wren and at least one of user_wstrb's bits being high simultaneously for one clock cycle. As shown above, if the value assignment depends on user_wstrb, there is no need to check user_wren.
- Unaligned read accesses are handled the same by the logic as aligned ones. For example, when the program running on the processor reads a byte, the whole 32-bit word is read on the bus, and the processor picks the required portion from the word.
- user_addr[1:0] may be non-zero when the address required by the processor is unaligned. This has no significance, since the logic's correct behavior on write cycles depends on user_wstrb only. These two bits are therefore best ignored even for unaligned access.

2.4 Interrupts

The Xillybus Lite IP core exposes an input signal, `user_irq`, which allows the application logic to send hardware interrupts to the processor. It is treated as a synchronous positive edge-triggered interrupt request signal, i.e. an interrupt is generated when this signal changes from low to high from one clock cycle to the next.

This signal is held zero in the `xillydemo.v(hd)` module.

Xillybus Lite adopts UIO's method of handling interrupts: The user space program sleeps as it attempts to read data from the device file. When the interrupt arrives, four bytes of data is read, waking up the process. These four bytes should be treated as an unsigned int, having the value of the total number of interrupts that have been triggered since the driver was loaded. The program may ignore this value, or use it to check if interrupts have been missed, by verifying that the value is one plus the value previously read.

Note that in a normal system's operation, this interrupt counter is never zeroed.

For example, assuming that "fd" is the file handle to `/dev/ui0`:

```
unsigned int interrupt_count;
int rc;

while (1) {
    rc = read(fd, &interrupt_count, sizeof(interrupt_count));

    if ((rc < 0) && (errno == EINTR))
        continue;

    if (rc < 0) {
        perror("read");
        exit(1);
    }

    printf("Received interrupt, count is %d\n", interrupt_count);
}
```

Note that the `read()` function call must require 4 bytes. Any other length argument will return an error. The interrupt file descriptor may be used in `select()` function calls.

Also note that the part checking for `EINTR` handles software interrupts properly (e.g. the process being stopped and restarted) and has nothing to do with the hardware interrupt.

3

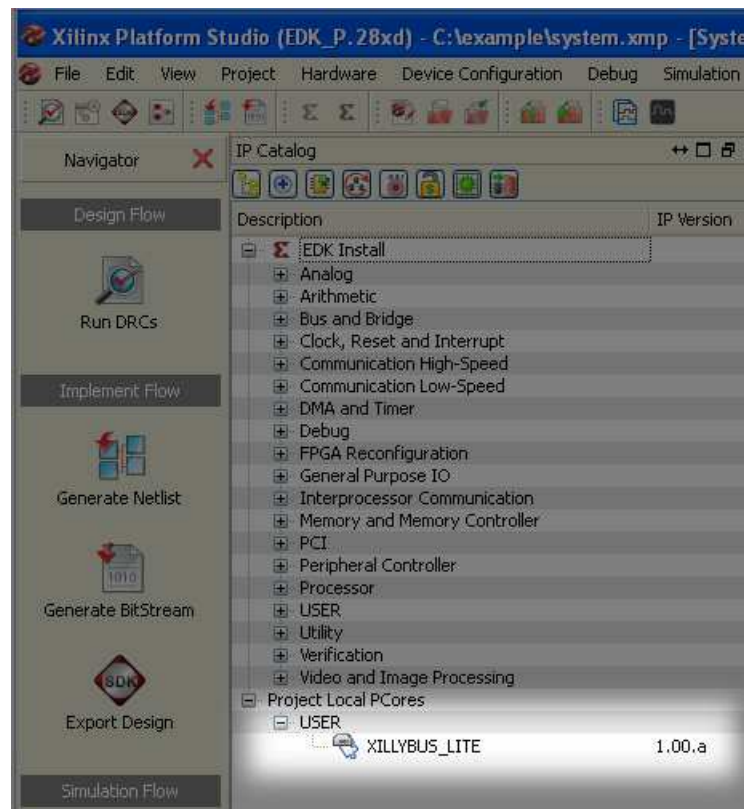
Xillybus Lite on projects not using Xilinx

3.1 Applying the IP core

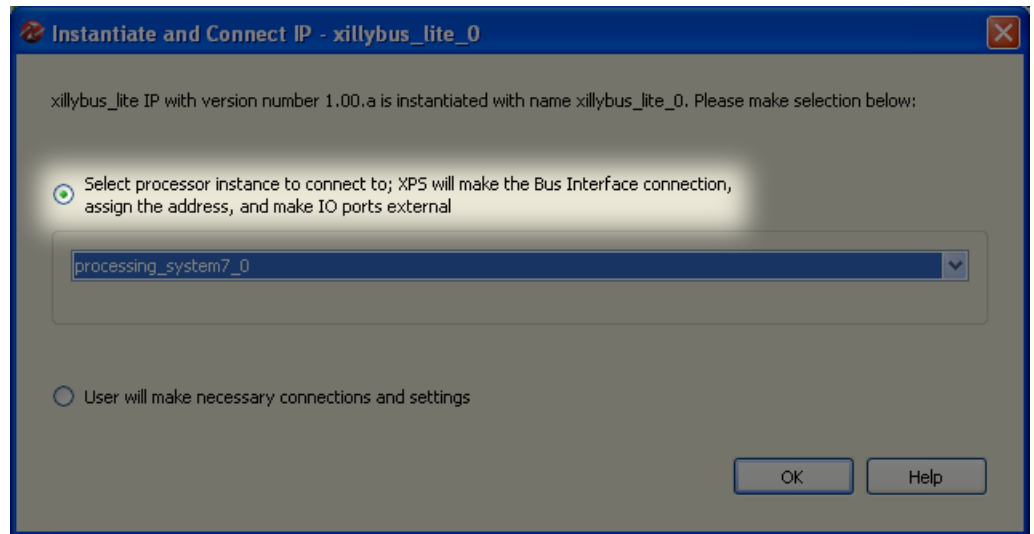
The procedure described next is based upon Xilinx Platform Studio 14.2 (XPS), but later versions are expected to behave much the same.

To add Xillybus Lite to an existing project, follow the steps below:

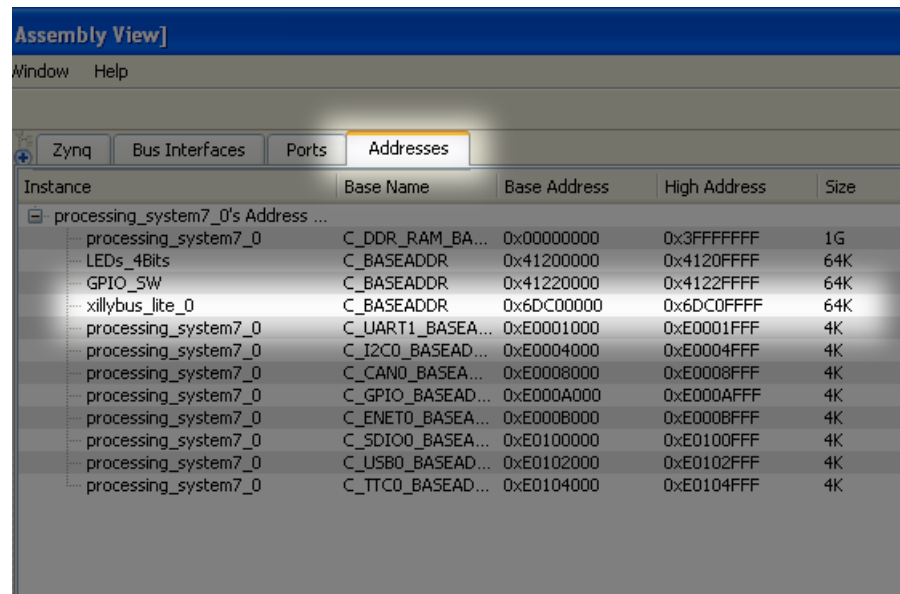
- Download the Xillybus Lite bundle from <http://xillybus.com/xillybus-lite>.
- Copy the `xillybus_lite.v1_00.a` folder from Xillybus Lite's bundle's "pcores" folder into your XPS project's "pcores" folder. The latter may be empty if the XPS project was just generated.
- With the relevant project open in XPS, click Project > Rescan User Repositories. As a result, a "USER" entry will appear in the IP Catalog to the left, under "Project Local PCores". Expand this entry, and find the XILLYBUS.LITE core.



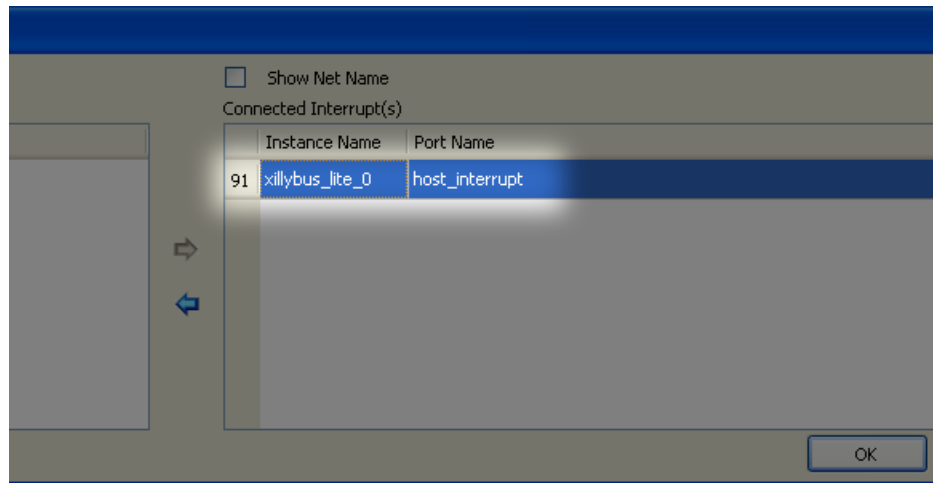
- Double-click XILLYBUS.LITE. Confirm the popup window asking whether the IP core should be added to the design.
- An XPS Core Config window appears next. Just click “OK”. There is no need for changes.
- On the following window, “Instantiate and Connect IP” allow XPS to make the bus interface connections by choosing the upper radio button.



- Pick the “Addresses” tab and take note of the address range allocated for xillybus_lite_0 for future reference (Base address and Size). It’s also possible to change this range as necessary.

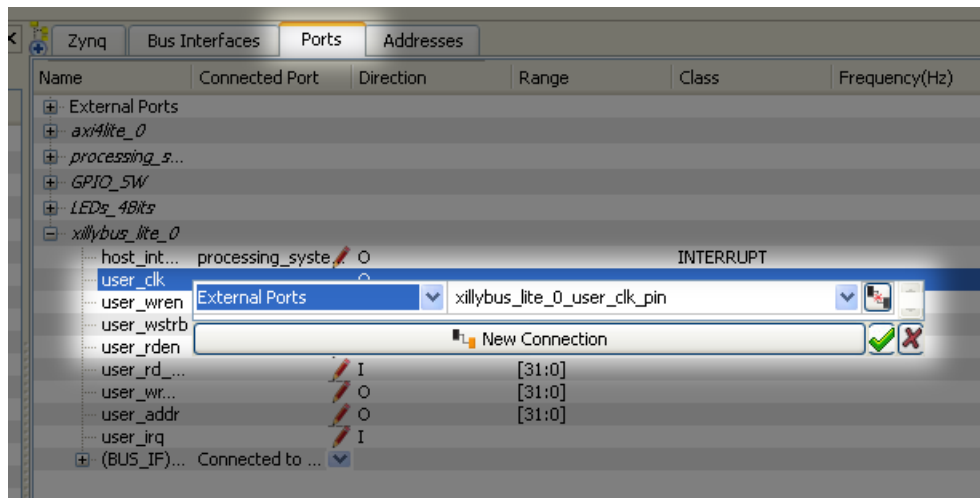


- Pick the “Zynq” tab and click on the IRQ box. Select the “host_interrupt” port in the list of unconnected interrupts, and click on the arrow in the middle to move it to the list of connected interrupts. Take a note of the interrupt number (91 in the sample screenshot).



- Pick the “Ports” tab and expand the “xillybus_lite_0” entry. For each of the eight signals with “user_” prefix, click on the empty space to the right of the port’s name, and make the port external:

Pick “External Port” on the drop-down selection box, and accept the default wire name given (or possibly change it). Confirm by clicking the check mark or pressing ENTER for each port.



IMPORTANT:

The host_interrupt port is set to EDGE_RISING. Don't change it to level triggered, or an interrupt may lock up the system.

The XPS project is now ready for build (with e.g. “Create Netlist”).

The processor’s module (usually named “system”) will have 8 additional ports. These should be added in this module’s instantiation (and architecture description in VHDL).

For example, in Verilog

```
...
wire      user_clk;
wire      user_wren;
wire [3:0] user_wstrb;
wire      user_rden;
wire [31:0] user_rd_data;
wire [31:0] user_wr_data;
wire [31:0] user_addr;
wire      user_irq;

...

system
  system_i (
    ...

    .xillybus_lite_0_user_clk_pin ( user_clk ),
    .xillybus_lite_0_user_wren_pin ( user_wren ),
    .xillybus_lite_0_user_wstrb_pin ( user_wstrb ),
    .xillybus_lite_0_user_rden_pin ( user_rden ),
    .xillybus_lite_0_user_rd_data_pin ( user_rd_data ),
    .xillybus_lite_0_user_wr_data_pin ( user_wr_data ),
    .xillybus_lite_0_user_addr_pin ( user_addr ),
    .xillybus_lite_0_user_irq_pin ( user_irq )
  );
```

All signals except `user_rd_data` and `user_irq` are outputs from the processor.

3.2 Modifying the device tree

The device tree for the existing system must be obtained, so that an entry for Xillybus Lite can be added. It’s important to start from the device tree in effect, or the system’s configuration may change or possibly even fail in the boot process.

For Xillinux 2.0 and later, the device tree sources in use are part of the kernel source,

which can be downloaded from Github. Please refer to section 6 in [Getting started with Xilinx for Zynq-7000](#) on how to obtain this.

In earlier revisions of Xilinx, the device tree source is in the /boot directory as e.g. devicetree-3.3.0-xilinx-1.1.dts.

If the device tree sources isn't available, it can be reconstructed from its binary, which is available in the same directory which boot.bin is loaded from on powerup. A tutorial explaining this issue (and others related to the device tree) can be found at

<http://xillybus.com/tutorials/device-tree-zynq-1>

An entry as follows should be added to the DTS file, in the segment containing bus peripherals (within the curly brackets enclosing `axi@0`):

```
xillybus_lite@6dc00000 {
    compatible = "xillybus_lite_of-1.00.a";
    reg = < 0x6dc00000 0x10000 >;
    interrupts = < 0 59 1 >;
    interrupt-parent = <&gic>;
} ;
```

IMPORTANT:

This sample entry matches the screenshots shown above, and not the settings used in Xilinx.

The following changes may be needed in the DTS entry to match your instance of the peripheral in the XPS project:

- Set the base address, taken from the address map of XPS, in the “reg” assignment and in the node’s name (0x6dc00000 above, without the “0x” prefix in the node’s name).
- Set the second argument of “reg” to the number of bytes allocated from the base address (0x10000 above)
- Set the second argument of “interrupts” to the interrupt number given in XPS minus 32. In the example, XPS allocated interrupt 91 to the peripheral, and consequently $91 - 32 = 59$.
- If the device tree was been obtained by virtue of a reverse compilation from a binary or /proc/device-tree/, it will not have the “gic” label defined. Since all

devices in the system use the same interrupt controller, it's fine to copy the numeric value from other "interrupt-parent" assignment in the device tree. In almost all cases, this simply means replacing `&gic` with the value `0x1`.

After editing the device tree source file, run a compilation to turn it into a binary blob (DTB) with the Device Tree Compiler (DTC). This compiler is part of the Linux kernel tree.

On a running Xillinux system, this can be done as follows. The directory of the kernel tree may vary, depending on the Xillinux distribution used.

```
# cd /usr/src/kernels/3.3.0-xillinux-1.1+/  
# scripts/dtc/dtc -I dts -O dtb -o devicetree.dtb my_device_tree.dts
```

Then overwrite the existing `devicetree.dtb` file on the media used for boot, with the file just generated.

Note that Xillinux can be used for compilation of a device tree that is intended for a system other than Xillinux.

3.3 Compilation of the Linux driver

The driver can be found in the Xillybus Lite bundle, in the "linuxdriver" directory.

There are two options for its compilation:

- cross compilation for the ARM processor against the intended kernel's source code (or at least its headers).
- Direct compilation on the Zynq board itself is possible, if the intended distribution has the GNU tools and kernel headers installed.

Xillybus lite relies on several kernel options, in particular the UIO option (`CONFIG_UIO`) being enabled at least as a module.

In what follows, a direct compilation on the board is assumed. This can be done in the Xillinux distribution, but isn't necessary for running Xillybus Lite on Xillinux (since the driver is already installed). On the other hand, if compilation of the driver has been done on Xillinux (and hence against Xillinux' kernel headers), the binary may not work on other kernels.

First change directory:

```
$ cd /path/to/linuxdriver
```

and type “make” to for a compilation of the driver. The session should look something like this:

```
$ make
make -C /lib/modules/3.3.0/build SUBDIRS=/tmp/lite/linuxdriver modules
make[1]: Entering directory `/usr/src/kernels/3.3.0'
  CC [M]  /tmp/lite/linuxdriver/xillybus_lite_of.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /tmp/lite/linuxdriver/xillybus_lite_of.mod.o
  LD [M]  /tmp/lite/linuxdriver/xillybus_lite_of.ko
make[1]: Leaving directory `/usr/src/kernels/3.3.0'
```

Note that the compilation of the kernel module is done specifically for the kernel running during the compilation. If another kernel is used, type “make TARGET=kernel-version” where “kernel-version” is the intended kernel version, as it appears in `/lib/modules/`.

The session’s output may vary slightly, but no errors or warnings should appear.

In particular, if these warnings appear,

```
WARNING: "__uio_register_device" [xillybus_lite_of.ko] undefined!
WARNING: "uio_unregister_device" [xillybus_lite_of.ko] undefined!
```

it means that the intended kernel lacks the UIO option, and inserting the driver into the kernel will most likely fail.

3.4 Installing the driver

Copy the `xillybus_lite_of.ko` directory to some existing driver subdirectory, and run `depmod` as follows (assuming that the intended kernel is currently running):

```
# cp xillybus_lite_of.ko /lib/modules/$(uname -r)/kernel/drivers/char/
# depmod -a
```

The installation does not load the driver into the kernel immediately. It will do so on the next boot of the system if a Xillybus Lite peripheral is discovered. How to load the driver manually is shown next.

3.5 Loading and unloading the driver

In order to load the driver (and start working with Xillybus Lite), type as root:

```
# modprobe xillybus_lite_of
```

This will make the Xillybus Lite device file (/dev/uio0) appear.

Note that this should not be necessary if a Xillybus Lite peripheral was present when the system carried out boot and the driver was already installed as described above.

To see a list of modules in the kernel, type “lsmod”. To remove the driver from the kernel, go:

```
# rmmmod xillybus_lite_of
```

This will make the device file vanish.

If something seems to have gone wrong, please check up the /var/log/syslog log file for messages containing the word “xillybus”. Valuable clues are often found in this log file.

If no /var/log/syslog log file exists, it's probably /var/log/messages instead.

If an “unknown symbol” error regarding `_uio_register_device` or similar appears in the log files, it indicates that the running kernel lacks the UIO configuration option.