XAPP1129 (v1.0) May 5, 2009

Integrating an EDK Custom Peripheral with a LocalLink Interface into Linux

Author: Brian Hill

Abstract

This application note discusses the usage of a Local Link DMA peripheral with the Linux operating system. A reference system with a Local Link DMA Loopback peripheral is included, as well as an example driver.

The integration of the provided driver into the Linux kernel is discussed, as well as relevant design and operational information.

Included Systems

Included with this application note is one reference system built for the Xilinx ML507 Rev A board. The reference system is available in the following ZIP file available at:

https://secure.xilinx.com/webreg/clickthrough.do?cid=115119

Introduction

This application note provides the steps and methodology needed to utilize a custom LocalLink Scatter-Gather DMA (SGDMA) core with Linux.

An example Local Link loopback core is provided with a Linux driver. The driver design and operation are discussed at length.

Hardware Requirements

The hardware requirements for this reference system are:

- Xilinx ML507 Rev A board
- Xilinx Platform USB or Parallel IV programming cable
- RS232 serial cable and serial communication utility (HyperTerminal)
- Xilinx Platform Studio 11.1
- Xilinx Integrated Software Environment (ISE®) 11.1
- MontaVista Linux 5.0 (Linux kernel 2.6.24)

Reference System Specifics

This system uses the PowerPC® 440 processor block with a processor frequency of 400 MHz and the Memory Interface Block (MIB) frequency of 266 MHz. The processor block crossbar is set to 266 MHz. In addition, the MPLB and the first HDMA port of the crossbar are set to 133 MHz.

The reference system includes PPC440MC DDR2, XPS LL EXAMPLE, XPS BRAM, XPS UART16550, XPS GPIO, and XPS INTC.

The PPC440MC DDR2 is connected to the MIB of the processor block with a frequency of 266 MHz.

The XPS BRAM, XPS GPIO, and XPS UART16550 cores are connected as slaves to the PLB v4.6 instance that is connected to the MPLB.

The XPS LL EXAMPLE core which contains a LocalLink interface is connected to the first HDMA port. The core's PLB v4.6 slave interface of the core is connected to MPLB.

See Figure 1 for the block diagram and Table 1 for the address map of the system.

Block Diagram

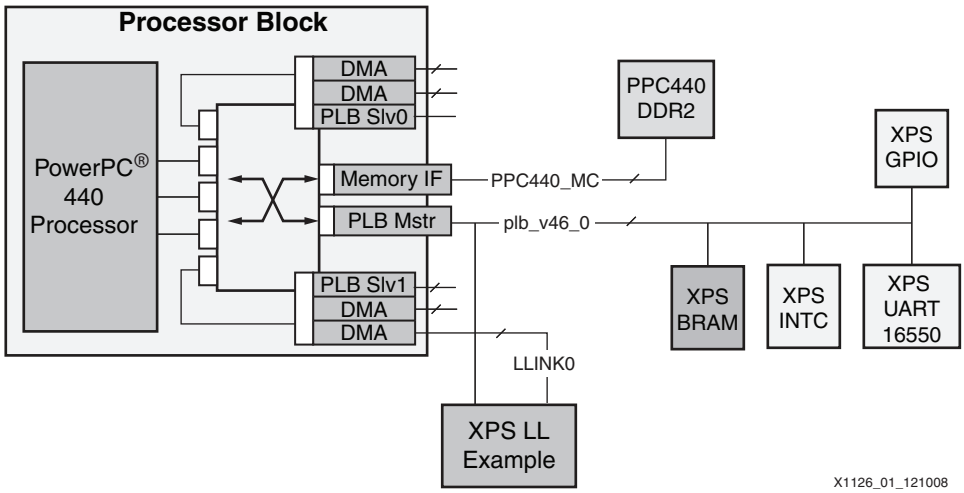


Figure 1: **Reference System Block Diagram**

Address Map

Table 1: **Reference System Address Map**

Peripheral	Instance	Base Address	High Address
ppc440mc_ddr2	ppc440_mc_ddr2_0	0x00000000	0x0FFFFFFF
xps_uart16550	xps_uart16550_0	0x40400000	0x4040FFFF
xps_gpio	LEDs_8Bit	0x40000000	0x4000FFFF
xps_bram_if_cntlr	xps_bram_if_cntlr_1	0xFFFF0000	0xFFFFFFFF
xps_intc	xps_intc_0	0x41200000	0x4120FFFF
xps_ll_example	xps_ll_example_0	0x60000000	0x6000FFFF

Overview of XPS LL EXAMPLE core

The XPS LL EXAMPLE core implements Local Link loopback. All data received are mirrored back to the sender. This application note will show data in memory DMAed to the loopback core, which redirects it back to the HDMA engine. The HDMA engine then DMAs this received data to a different memory location.

The reference system included with this application note and the XPS LL EXAMPLE core are discussed in XAPP1126 “Designing an EDK Custom Peripheral with a LocalLink Interface”.

Generate the Linux BSP

The user will generate a Linux BSP within the Xilinx EDK, and then perform necessary modifications to build the LL Example driver in this kernel.

Export the system to SDK

Generation of Board Support Packages in XPS is deprecated in EDK 11.1. The system is exported to SDK, where the Linux BSP is generated.

1. In the XPS project, choose **Project->Export Hardware Design to SDK**.

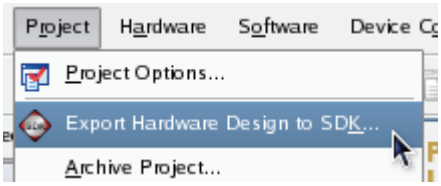


Figure 2: **Export Hardware Design to SDK**

2. Click **Export & Launch SDK**.

Figure 3: **Export and Launch SDK**

Note: This operation will take some time while XPS generates the bitstream.

3. Once SDK has launched, choose **File->New->Project**

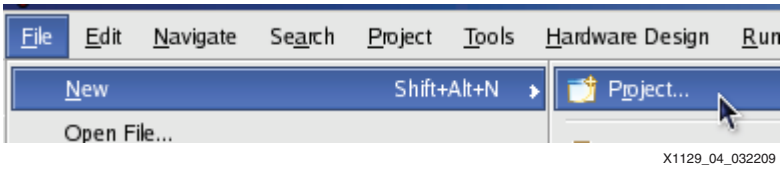


Figure 4: **New SDK Project**

4. In the Wizard window, select **Board Support Package**.

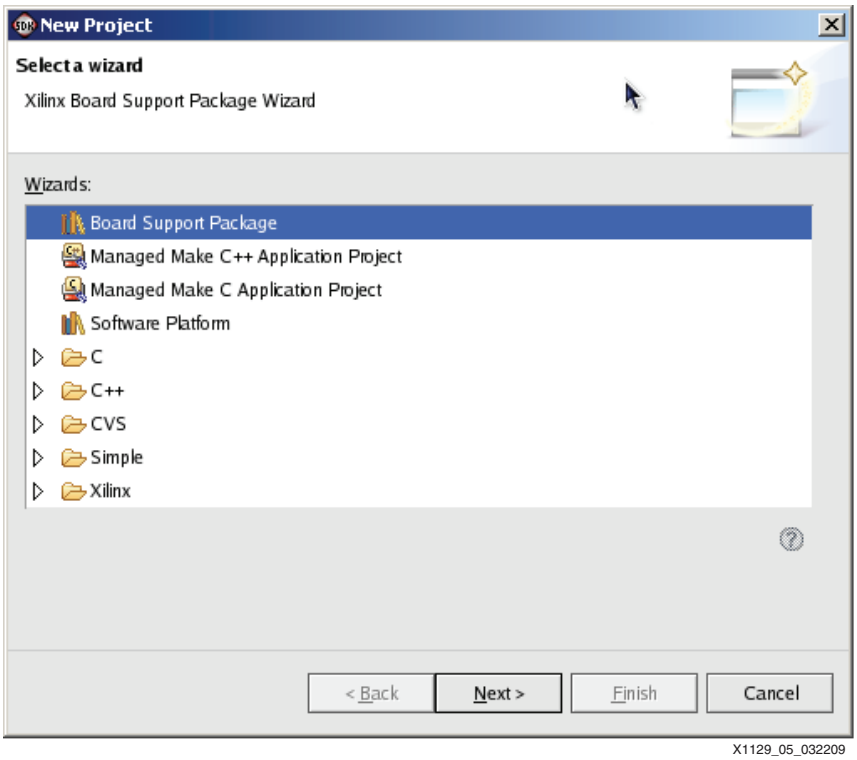


Figure 5: **New Board Support Package**

5. In the **Project name** field select a new project name. In the **Board Support Package Type** field select **linux_2_6**.

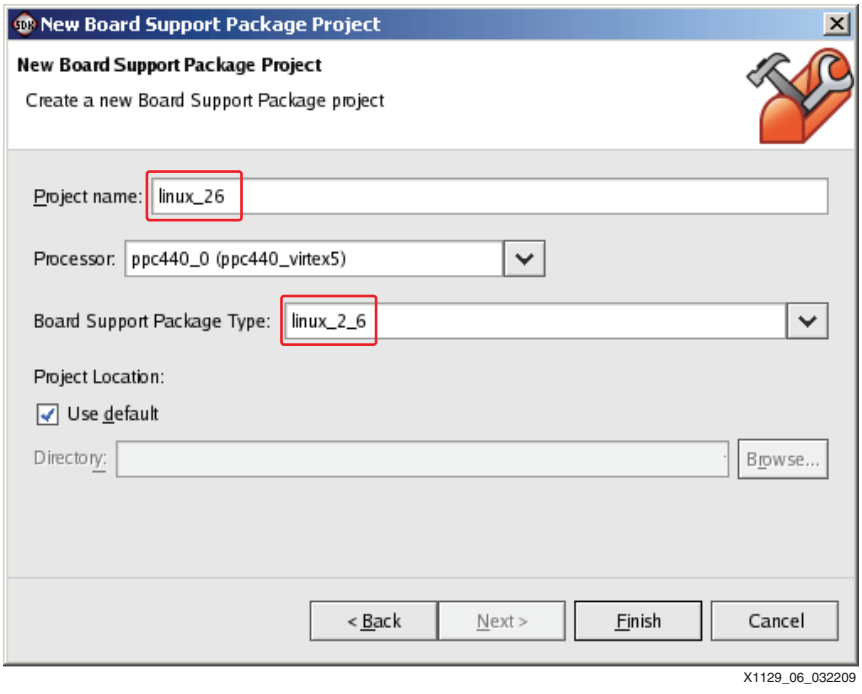


Figure 6: **linux_2_6**

6. Fill in the pertinent Board Support Package settings

a. Set the Linux Distribution to **"MontaVista 5.0"**
b. Set Memory Size to `0x10000000`
c. Target directory is set to the location of the users Linux kernel
d. Set Rootfs type to **ramdisk**
e. Click **OK**. The BSP is generated.

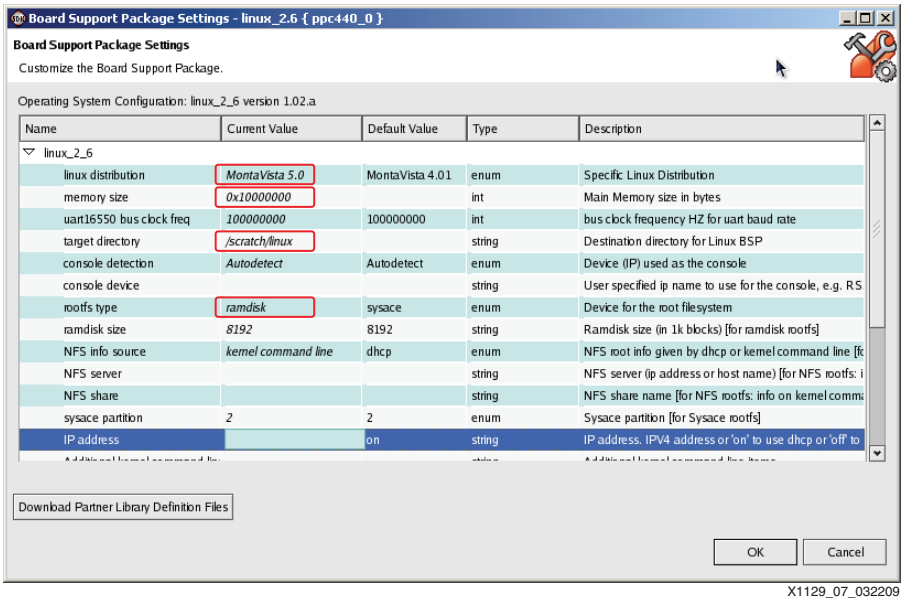


Figure 7: **Board Support Package Settings**

7. Edit the generated `<target directory>/arch/powerpc/boot/dts/ml507.dts` to reflect the modifications shown in **red**:

```
xps_ll_example_0: xps-ll-example@60000000 {
            compatible = "xlnx,xps-ll-example-1.00.a";
            reg = < 0x60000000 0x10000 >;
            llink-connected = <&DMA0>;
            xlnx,family = "virtex5";
            xlnx,include-dphase-timer = <0x0>;
        } ;

xps_intc_0: interrupt-controller@41200000 {
            #interrupt-cells = <0x2>;
            compatible = "xlnx,xps-intc-2.00.a", "xlnx,xps-intc-1.00.a";
            interrupt-controller ;
            reg = < 0x41200000 0x10000 >;
            xlnx,num-intr-inputs = <0x3>;
        } ;
```

Note: step 7 will not be required in a future release of EDK. The text added in this step is specific to the system included with this application note and may require adjustment for any other EDK system.

This MontaVista 5.0 BSP which is generated primarily consists of a single text file which describes the hardware system. It is placed under the configured target directory at `arch/powerpc/boot/dts/ml507.dts`. The file used to generate the provided executables is available in the `ready_for_download` area of the EDK project. The user may choose to copy this file to their kernel rather then generate a BSP with SDK.

Copy the driver

The provided driver is copied from the EDK project to the appropriate location in the kernel tree. From an EDK shell issue the following commands:

```
$ cd <linux tree>
$ mkdir drivers/char/xilinx_ll_example
$ cp -a <EDK project>/drivers/xps_ll_example_v1_00_a/src/*
drivers/char/xilinx_ll_example
```

Add the driver to the kernel configuration

1. Edit <linux tree>/drivers/char/Kconfig, adding the driver to those configurable to the kernel as shown:

```
config XILINX_LL_EXAMPLE
        bool "Xilinx Local Link example"
        depends on XILINX_DRIVERS
        select XILINX_EDK
        select NEED_XILINX_LLDMA
        select NEED_XILINX_PPC_DCR
        help
          Example driver for Xilinx Local Link Loopback core.
```

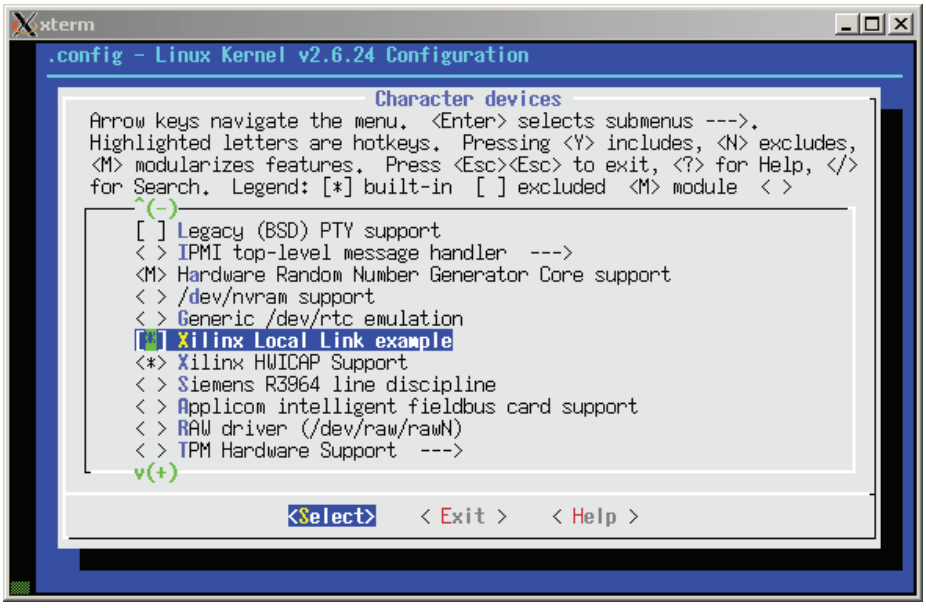
2. Edit `<linux tree>/drivers/char/Makefile`, adding the driver directory to the kernel character driver build:

```
obj-$(CONFIG_XILINX_LL_EXAMPLE) += xilinx_ll_example/
```

3. Edit the kernel configuration to enable the Local Link example driver with the commands shown:

```
$ cd <linux tree>
$ make ARCH=powerpc menuconfig
```

4. Choose **Device Drivers→Character devices→Xilinx Local Link example**



X1129_08_032209

Figure 8: **Enable Local Link Example Driver in Kernel Build**

5. Copy the provided ramdisk image to the kernel by using the command:

```
$ cp <EDK project>/ready_for_download/ramdisk.image.gz <linux
tree>/arch/powerpc/boot/
```

6. Build the bootable kernel image

```
$ cd <linux tree>
$ make ARCH=powerpc zImage.initrd
```

Executing the Reference System

Using HyperTerminal or a similar serial communications utility, map the operation of the utility to the physical COM port to be used. Then connect the UART of the board to this COM port. Set the HyperTerminal to the Bits per second to **9600**, Data Bits to **8**, Parity to **None,** and Flow Control to **None**. The settings are shown in Figure 9. This is necessary to see the results from the software application.

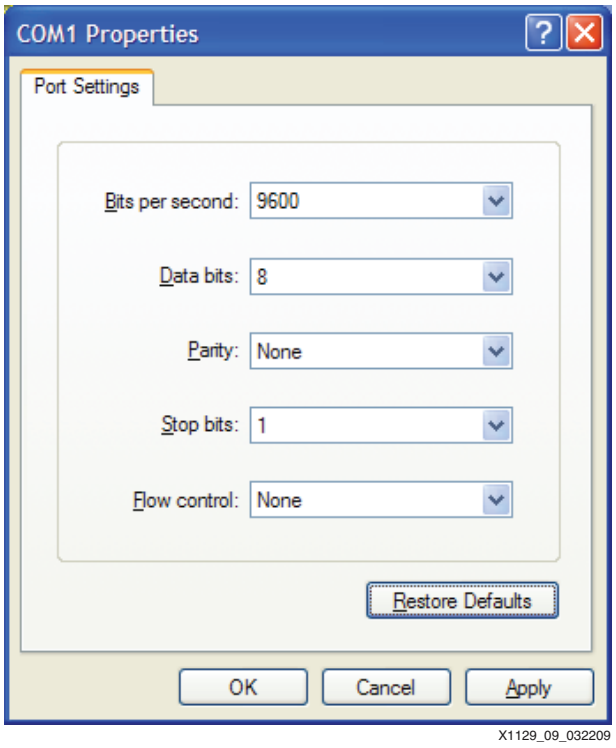


Figure 9: **HyperTerminal Settings**

Executing the Reference System using the Pre-Built Bitstream and the Compiled Software Application

To execute the system using files in the `ready_for_download/` directory in the project root directory, follow these steps:

1. Change directories to the `ready_for_download` directory.
2. Use iMPACT to download the bitstream by using the following command:

   ```
   impact -batch xapp1129.cmd
   ```

3. Invoke XMD and connect to the PowerPC 440 processor by using the following command:

   ```
   xmd -opt xapp1129.opt
   ```

4. Download the executable by using the following command depending on the software application:
   - **`dow zImage.initrd`**
5. Enter in the `run` command to run the software application.
6. Login as ‘root’. There is no password.

Executing the Reference System from XPS for Hardware

To execute the system for hardware using XPS, follow these steps:

1. Open `system.xmp` in XPS.
2. Select **Hardware→Generate Bitstream** to generate a bitstream for the system.
3. Select **Device Configuration→Download Bitstream** to download the bitstream.
4. Select **Debug→Launch XMD to i**nvoke XMD.
5. Download the executable file by using the following command depending on the software application:
   - **`dow zImage.initrd`**
6. Enter in the `run` command to run the software application.

Testing the Local Link Example driver

All data DMAd to the Local Link loopback core is mirrored back across the LocalLink, resulting in the received data being DMAd to another memory location. Operation will ultimately resemble a memory-to-memory DMA as perceived by software. The Local Link Example driver presents this to the user as a loopback device. Data written to the device can be read back at a later time (after the DMA is completed).

EXAMPLE USAGE:

```
# cat /etc/issue > /dev/llex0
# cat /dev/llex0
MontaVista(R) Linux(R) Professional Edition 5.0.24 (0502020)
```

XLL Example Driver

The Local Link example driver is provided in the `<EDK project>/drivers/xps_ll_example_v1_00_a/src` directory. There are three files:

```
Makefile
xll_example.c
xll_example.h
```

The LL Example driver utilizes the Xilinx lldma driver to assist with DMA activities. The lldma driver creates and manipulates descriptor rings. It can not exist as a standalone driver, but rather must be used by another driver (the driver of the core connected to the Local Link interface) which provides all of the front end hooks into the Linux kernel. Users of the lldma driver are still responsible for DMA interrupts and DMA related memory management. In many respects the lldma driver is the functional equivalent of a library. A description of the lldma driver API may can be found in `<Linux tree>/drivers/xilinx_common/xlldma.h`.

The LL Example driver utilizes the following Linux kernel functions:

`dma_unmap_single`	Destroy single use DMA memory mapping
`dma_map_sungle`	Create single use DMA memory mapping
`virt_to_phys`	Provide the physical (bus) address for the virtual address
`mutex_lock`	Lock the indicated mutex
`mutex_unlock`	Unlock indicated mutex
`printk`	Emit text out the console. Kernel equivalent of printf.
`tasklet_disable`	Prevent the tasklet from being scheduled
`tasklet_enable`	Re-enable scheduling of indicated tasklet
`tasklet_schedule`	One-time schedule the indicated tasklet

`list_entry`	Convert list pointer to user structure pointer.
`list_empty`	Is the indicated list empty?
`list_add_tail`	Add provided item to list tail
`copy_to_user`	Copy kernel memory to user space
`copy_from_user`	Copy user application memory to kernel
`kmalloc`	Allocate contiguous block of memory. Kernel equivalent of malloc.
`create_proc_entry`	Create a file in the /proc filesystem.
`register_chrdev_region`	Assign a device number to a driver
`ioremap`	Memory map a peripheral into the kernel's virtual address space
`request_irq`	Assign an IRQ handler
`cdev_add`	Add character device file operation handlers
`of_register_platform_driver`	Register probe function and hardware compatable with this driver
`of_address_to_resource`	Obtain hardware address of peripheral from device tree
`of_irq_to_resource`	Obtain virtual IRQ of peripheral in the device tree
`of_get_cpu_node`	Find the processor node in the device tree

Linux Device Tree

The linux kernel provides two separate architectures which support the PowerPC processor. The original, ARCH=ppc, is still present in the kernel tree but has been deprecated. All current development occurs on the ARCH=powerpc architecture. Both are present in the `arch` directory of the kernel tree. The example driver provided with this application note is designed to function with the ARCH=powerpc architecture and **will not function** with ARCH=ppc without modification.

One of the primary differences between the two architectures is that ARCH=powerpc uses a device tree to describe the hardware system. The older ARCH=ppc has platform hardware settings bound at kernel compile time, using mechanisms such as the Xilinx `xparameters.h` file. When building a kernel with ARCH=powerpc, `xparameters.h` can no longer be used; hardware configuration **must** be retrieved from the device tree by the driver at runtime.

The MontaVista 5.0 BSP that is generated in the "Generate the Linux BSP" section is nothing more than a text file. It is a device tree which is lumped together with the kernel for bootable ramdisk images.

Refer to `<linux tree>/Documentation/powerpc/booting-without-of.txt`.

Driver initialization

The entry point of the driver is defined with the macro `module_init()`. It is seen that the function `xll_example_init()` is responsible for hooking the driver into the kernel:

```
module_init(xll_example_init);
```

The `xll_example_init()` will register the driver as an Open Firmware platform device driver. Open Firmware is generally associated with drivers which will utilize a device tree. In this instance, there is no ROM monitor providing the device tree to the kernel; the device tree is compiled into the bootable ramdisk image. The source of the device tree is of no consequence to the driver.

The driver registers itself as compatable with “xps-ll-example-1.00.a”. Any instance of this in the device tree will cause the drivers probe function to be called. The probe function will ultimately lead to a driver setup function which will map this driver to a unique major device number, provide file system hooks so that the driver services can be accessed from an application in user space, memory map the LL EXAMPLE registers into kernel virtual address space, and perform any necessary memory allocations.

Major Device Number

All Linux drivers have a unique major device number. Drivers also have a minor device number which is locally significant to that driver only. The driver may use the minor device number to differentiate between multiple instances of the same device, or for any other purpose the driver author chooses.

A PC running Linux will often utilize drivers which obtain a major device number dynamically. There are a limited number of device numbers available, and for a PC with potentially a large number of devices present and generally available as kernel modules this is a sensible mode of operation. An embedded device is typically of a fixed configuration, and a statically assigned number is a more appropriate (and simpler) choice. Some well-established assignments may be found in `<linux tree>/include/linux/major.h`.

The LL Example driver uses a static major device number of 253. This mapping is performed with `register_chrdev_region()`, which also assigns the name “ll_example” to the device:

```
err = register_chrdev_region(devno, 1, "ll_example");
```

The device is now visible in `/proc/devices`

```
# cat /proc/devices
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  4 ttyS
259 icap
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 29 fb
 89 i2c
 90 mtd
128 ptm
136 pts
204 ttyUL
253 ll_example

Block devices:
  1 ramdisk
  7 loop
 31 mtdblock
254 xsysace
#
```

User space accesses a particular driver by device number with the use of special files. The driver author will add a special file for the new device with the `mknod` command, typically placing the special file in the `/dev` filesystem:

```
# mknod /dev/llex0 c 253 0
```

This command creates the special file `/dev/llex0`, which is specified as a character device with major number 253, minor number 0. This file is already present in the ramdisk provided with this application note.

File operations for the driver are registered with the kernel with the `cdev_add()` function. These vectors are called whenever an application performs file operations (`open, close, read, write, seek, ioctl`) on the drivers special file.

Register Access

It is expected that a driver will need to access peripheral registers. To provide access to these memory mapped registers the `ioremap()` function is used. Ioremap will provide a kernel virtual address mapping for the physical (bus) address it is given.

```
ll_ex_dev->mapaddr = ioremap(ll_ex_dev->physaddr, ll_ex_dev->addrsize);
```

After this call, the LL Example core registers may be accessed with the virtual address `mapaddr`.

Note: The example driver uses the kernel function `in_be32()` to perform register reads.

Interrupt Handlers

Interrupt vectors are registered using the `request_irq()` procedure. The LL Example driver registers handlers for DMA TX and DMA RX interrupts:

```
err = request_irq(ll_ex_dev->dma_tx_irq,
                  &xllex_dma_tx_interrupt, 0, "xilinx_dma_tx_int", ll_ex_dev);
err = request_irq(ll_ex_dev->dma_rx_irq,
                  &xllex_dma_rx_interrupt, 0, "xilinx_dma_rx_int", ll_ex_dev);
```

Interrupt handlers are eligible for execution **immediately** after `request_irq()` has been called; the driver writer must be certain that the driver is sufficiently initialized before registering these handlers.

The interrupt sources and their statistics counts are now visible in `/proc/interrupts`

```
# cat /proc/interrupts
           CPU0
 16:          2  Xilinx INTC Edge      xilinx_dma_rx_int
 17:          2  Xilinx INTC Edge      xilinx_dma_tx_int
 18:       1424  Xilinx INTC Edge      serial
BAD:          0
#
```

The Bottom Half

Hardware interrupts handlers execute with processor interrupts disabled. Interrupt handlers must perform their work very quickly for the system to equitably share processor resources and maintain real-time performance. This is accomplished by dividing the interrupt handler into two halves - the hardware interrupt handler, and the Bottom Half. The LL Example driver hardware interrupt handlers will acquiesce the DMA peripheral, and schedule a **tasklet** to service the DMA ring. Tasklets are run with interrupts enabled, so that scheduling still occurs and hardware interrupts from other devices are still serviced.

DMA Memory Management

The lldma driver provides all the functions needed to manage descriptor rings. The task of memory management is still left to the primary device driver. A Linux driver which performs DMA has several memory management considerations.

DMA requires blocks of contiguous physical memory. Kernel memory allocation functions such as `kmalloc()` and `get_free_pages()` are suitable for this purpose. Note that `vmalloc()` can not be used for this purpose. Vmalloc allocations are contiguous in virutal address space

because the TLB has mapped discontiguous physical blocks together into a contiguous virtual block.

DMA operates with bus (physical) addresses. All addresses provided by kernel memory allocators provide a kernel virtual address. The driver must map these virtual addresses to the physical address needed by hardware. The LL Example driver uses the `virt_to_phys()` function to obtain the physical address of buffer descriptors which the driver allocated with `kmalloc()`.

The DMA buffers are allocated by the LL Example driver with `kmalloc()`. These allocations are in cached memory. In addition to the virtual to physical address mapping required to perform DMA, the driver must manage the processor cache for these buffers. These tasks are performed by the LL Example driver with the `dma_map_single()` and `dma_unmap_single()` kernel functions.

This buffer was written by software (a transmit buffer, as indicated by DMA_TO_DEVICE). `dma_map_single()` will flush this buffer from the data cache and return the physical address needed by hardware.

```
phy_addr = (u32) dma_map_single(NULL, buff, buff_len, DMA_TO_DEVICE);
```

This buffer is waiting to be written by the DMA core (a receive buffer, as indicated by DMA_FROM_DEVICE). `dma_map_single()` will invalidate the data cache entries pertaining to this buffer and return the physical address needed by hardware.

```
new_buff_physaddr = (u32) dma_map_single(NULL, rx_buff->data,
                                                LL_EX_BUF_SIZE,
                                                DMA_FROM_DEVICE);
```

For any call to `dma_map_single()` there must be a corresponding call to `dma_unmap_single()` after DMA completion (excerpts shown). Software **must not** access the buffer in any way after `dma_map_single()` until `dma_unmap_single()` has been performed.

```
dma_unmap_single(NULL, buff_phys_addr,
                                LL_EX_BUF_SIZE,
                                DMA_FROM_DEVICE);
dma_unmap_single(NULL, dma_phys_addr, len,
                        DMA_TO_DEVICE);
```

File Operations

User applications interact with the LL Example driver by performing operations on the file system. Open, close, read, and write system calls on the drivers special file `/dev/llex0` result in the functions which the driver has registered being executed in kernel space.

```
int ll_ex_open(struct inode *inode, struct file *filp)
```

This function handles a user `open()` of the device file. A maximum of one reader and one writer is enforced here.

```
static int ll_ex_release(struct inode *inode, struct file *filp)
```

This function handles a user `close()` of the device file.

```
ssize_t ll_ex_write(struct file *filp, const char __user *buf,
                    size_t count, loff_t *f_pos)
```

This function handles a user `write()` to the device file. The user data is copied into buffers and added to the DMA TX ring. The Local Link example core will loop these data back, so it is expected that the data will soon be found on the DMA RX ring.

```
ssize_t ll_ex_read(struct file *filp, char __user *buf, size_t count,
                   loff_t *f_pos)
```

DMA receive operations place received buffers on the drivers `rx_buffer_list`. The `ll_ex_read()` function handles a user `read()` on the device by copying the buffer at the head of the receive list to user space. The buffer is then removed from the list and freed. Refer to "DMA receive operation".

DMA receive operation

The lldma driver manages DMA descriptor manipulations performed by software. The user's driver is responsible for certain descriptor fields to varying degrees. The descriptors themselves, 0x40 bytes each, are allocated by the main driver. The physical buffer address, provided to hardware in descriptor word 1, is allocated by the main driver. For a receive operation, the buffer length field (descriptor word 2) is provided by the driver to indicate the maximum contiguous space available at the buffer address. **The Local Link user core must provide the valid byte count, as this is not provided by the DMA engine.** The Local Link Example core included with this application note provides the valid byte count in descriptor word 7 "APP4". The APP fields are available for any data the core wished to provide to its driver using the Local Link footer words.

Several words are available in the descriptor for software use (hardware does not use or modify these fields). The LL Example driver uses word 8 as an ID field. For receive operations, this field contains the virtual address of an `llex_rx_buff` structure corresponding to this DMA buffer.

This structure is the mechanism used to maintain the drivers `rx_buffer_list` - received buffers awaiting consumption by a user `read()` on `/dev/llex0`. See Figure 10.

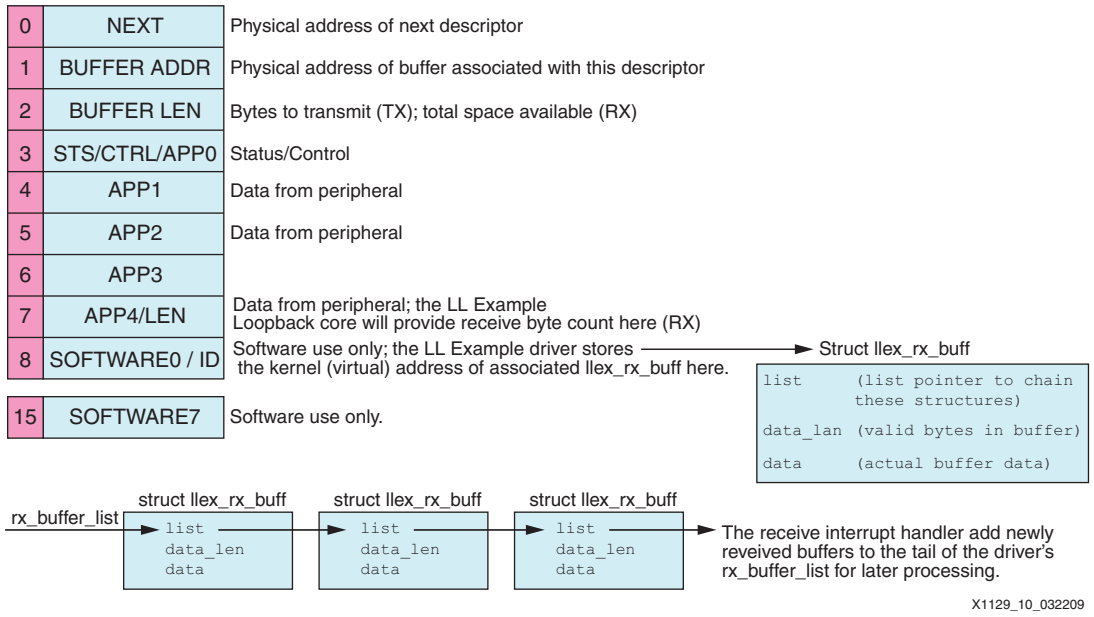


Figure 10: **DMA Receive Descriptor Usage**

DRIVER EXAMPLE USAGE (REVISITED):

```
# cat /etc/issue > /dev/llex0
```

The standard Linux '**`cat`**' command writes the text file `/etc/issue` to the device special file. The driver code copies this data from user memory to kernel memory. The kernel buffers are then added to the DMA TX ring, which will DMA this data from memory to the Loopback core. The loopback core forwards this data back across the Local Link to the DMA engine, which will DMA the data from the loopback core to a receive buffer. The DMA RX interrupt handler will place this used buffer in the receive buffer list for later usage.

```
# cat /dev/llex0
MontaVista(R) Linux(R) Professional Edition 5.0.24 (0802884)
```

The '**cat**' command is used to read the device special file. Data from the head of the receive buffer list `rx_buffer_list` is copied to user space. The buffer is then removed from `rx_buffer_list` and freed.

/proc File System

The LL Example driver creates an entry in the /proc file system at initialization:

```
proc_entry = create_proc_entry("driver/ll_example_drvr", 0, NULL);
```

This entry is used to provide driver status and statistics to the user:

```
# cat /proc/driver/ll_example_drvr

MAPADDR:  0xd1060000
Reads:    13
Writes:   5
Opens:    9
Closes:   9
TX IRQ:   4
TX Buff:  5
TX Bytes: 1798
RX IRQ:   4
RX Buff:  5
RX Bytes: 1798
Errors:   0

REGISTERS:
CTL:      0x00000000
STS:      0x00000001
TXFRM:    5
RXFRM:    5
TXBYTE:   1798
RXBYTE:   1798

Buffer Descriptors: 0x01148800 (Virtual 0xc1148800)
```

Exercises for the User

The example driver provided with this application note is operational as described in the "XLL Example Driver" section. This driver a simple example, made so to facilitate ease of understanding. Various operations and driver services which are not provided are presented here as an exercise for the user. These modifications may be desired in any driver written by the user for a custom core, depending upon the requirements of the project. It is expected that the user will require additional reference material to implement these modifications.

A Word on Driver Types

The example driver discussed is a character device. This is the simplest of all Linux kernel drivers. Character device drivers are best suited to streaming interfaces, and as such, Local Link DMA devices are a good fit. Linux also provides the concept of a block device. These are typically mass storage devices. Lastly, Linux provides the infrastructure for Network devices, such as the Xilinx XPS Local Link Tri-Mode Ethernet MAC. The appropriate choice depends on the User's application and system design.

Blocking Writes

User writes to the device file result in the driver adding a buffer descriptor with the written data to the LLDMA TX ring. In the vast majority of instances, this will always succeed. It is possible that in a high load situation that the TX ring will be full at this instant. The finite maximum number of descriptors awaiting transmission may all be in use. In that instance, the software must try again at a later time. The applicable driver function `ll_ex_write()` will return -

ERESTARTSYS in the present form, which may cause the user space application to receive an EINTR error for the write request. Consult the man page for sigaction(2) SA_RESTART for details.

If the driver were to provide blocking IO, the driver would operate in a more resource friendly way. When the TX ring is full, the driver would put the writing thread to sleep. The TX interrupt handler DmaSendHandlerBH() would then wake the sleeping process after DMA TX completion (a time when there is guaranteed to be at least one free TX Ring entry available).

Blocking Reads

A read of the device file from user space will result in the application reading the buffer at the head of the rx_buffer_list. If there are no buffers present, userspace will see an end-of-file. If no receive buffers are available, it may be desirable for the user application to wait until data has been received. This waiting is accomplished with a blocking read -- if there are no buffers in the rx_buffer_list, the reading process is put to sleep. Later, when received buffers are added to the list in DmaRecvHandlerBH() this sleeping process is awakened.

Non-blocking IO

The use of blocking IO, described in "Blocking Writes" and "Blocking Reads", is best suited for use in multi-threaded applications and very simple applications with a single thread of execution. The requirements of the user application may not be met by the use of blocking IO. In such cases Non-blocking IO is required. In such cases, the user application will typically wish to use select() on several file descriptors (one being the device). The use of select() allows the application to be aware of when there is data available to be read from a descriptor, and when data may be written to a descriptor. See the man page for select(2).

To enable the use of select() on the device file from user space, the poll() file operation must be implemented in the driver. This function will be added to the ll_ex_fops structure. Consult <Linux tree>/Documentation/filesystems/vfs.txt

References

1. UG200 *Embedded Processor Block in Virtex-5 FPGAs Reference Guide*
2. XAPP1126 *Designing an EDK Custom Peripheral with a LocalLink Interface*
3. Johnathan Corbet, Allesandro Runini, Greg Kroah-Hartman. 2005. *LINUX DEVICE DRIVERS*. O'Reilly Media, Inc.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
5/5/09	1.0	Initial Xilinx release.

Notice of Disclaimer

Xilinx is disclosing this Application Note to you “AS-IS” with no warranty of any kind. This Application Note is one possible implementation of this feature, application, or standard, and is subject to change without further notice from Xilinx. You are responsible for obtaining any rights you may require in connection with your use or implementation of this Application Note. XILINX MAKES NO REPRESENTATIONS OR WARRANTIES, WHETHER EXPRESS OR IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, IMPLIED WARRANTIES OF MERCHANTABILITY, NONINFRINGEMENT, OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT WILL XILINX BE LIABLE FOR ANY LOSS OF DATA, LOST PROFITS, OR FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR INDIRECT DAMAGES ARISING FROM YOUR USE OF THIS APPLICATION NOTE.