Outline

- **Device Drivers**
  - Level 0, Level 1
  - MicroBlaze Processor: Interrupts
  - Integration in EDK
- Libraries
- BSP
  - Boot Files and Sequence

Device Drivers

- The Xilinx device drivers are designed to meet the following objectives:
  - Provide maximum portability
    - The device drivers are provided as ANSI C source code
  - Support FPGA configurability
    - Supports multiple instances of the device without code duplication for each instance, while at the same time managing unique characteristics on a per-instance basis
  - Support simple and complex use cases
    - A layered device driver architecture provides both
      - Simple device drivers with minimal memory footprints
      - Full-featured device drivers with larger memory footprints
  - Ease of use and maintenance
    - Xilinx uses coding standards and provides well-documented source code for developers

Outline

- **Device Drivers**
  - **Level 0, Level 1**
  - MicroBlaze Processor: Interrupts
  - Integration in EDK
- Libraries
- BSP
  - Boot Files and Sequence

Drivers: Level 0/Level 1

- The layered architecture provides seamless integration with…
  - (Layer 2) RTOS application layer
  - (Layer 1) High-level device drivers that are full-featured and portable across operating systems and processors
  - (Layer 0) Low-level drivers for simple use cases

Layer 2, RTOS Adaptation
Layer 1, High-level Drivers
Layer 0, Low-level Drivers

Drivers: Level 0

- Consists of low-level device drivers
- Implemented as macros and functions that are designed to allow a developer to create a small system
- Characteristics:
  - Small memory footprint
  - Little to no error checking is performed
  - Supports primary device features only
  - No support of device configuration parameters
  - Supports multiple instances of a device with base address input to the API
  - Polled I/O only
  - Blocking function calls
  - Header files have "_l" in their names (for example, xuartlite_l.h)

Drivers: Level 1

- Consists of high-level device drivers
- Implemented as macros and functions and designed to allow a developer to utilize all of the features of a device
- Characteristics:
  - Abstract API that isolates the API from hardware device changes
  - Supports device configuration parameters
  - Supports multiple instances of a device
  - Polled and interrupt driven I/O
  - Non-blocking function calls to aid complex applications
  - May have a large memory footprint
  - Typically, provides buffer interfaces for data transfers as opposed to byte interfaces
  - Header files *do not* have "_l" in their names (for example, xuartlite.h)

Comparison Example

- **Uartlite Level 1**
  - **XStatus XUartLite_Initialize** (**XUartLite** *InstancePtr, **Xuint16** DeviceId)
  - void **XUartLite_ResetFifos** (**XUartLite** *InstancePtr)
  - unsigned int **XUartLite_Send** (**XUartLite** *InstancePtr, **Xuint8** *DataBufferPtr, unsigned int NumBytes)
  - unsigned int **XUartLite_Recv** (**XUartLite** *InstancePtr, **Xuint8** *DataBufferPtr, unsigned int NumBytes)
  - **Xboolean XUartLite_IsSending** (**XUartLite** *InstancePtr)
  - void **XUartLite_GetStats** (**XUartLite** *InstancePtr, **XUartLite_Stats** *StatsPtr)
  - void **XUartLite_ClearStats** (**XUartLite** *InstancePtr)
  - **XStatus XUartLite_SelfTest** (**XUartLite** *InstancePtr)
  - void **XUartLite_EnableInterrupt** (**XUartLite** *InstancePtr)
  - void **XUartLite_DisableInterrupt** (**XUartLite** *InstancePtr)
  - void **XUartLite_SetRecvHandler** (**XUartLite** *InstancePtr, **XUartLite_Handler** FuncPtr, void *CallBackRef)
  - void **XUartLite_SetSendHandler** (**XUartLite** *InstancePtr, **XUartLite_Handler** FuncPtr, void *CallBackRef)
  - void **XUartLite_InterruptHandler** (**XUartLite** *InstancePtr)
- **Uartlite Level 0**
  - void **XUartLite_SendByte** (**Xuint32** BaseAddress, **Xuint8** Data)
  - **Xuint8 XUartLite_RecvByte** (**Xuint32** BaseAddress)

Outline

- **Device Drivers**
  - Level 0, Level 1
  - **MicroBlaze Processor: Interrupts**
  - Integration in EDK
- Libraries
- BSP
  - Boot Files and Sequence

Interrupts

- An interrupt is an asynchronous signal from hardware indicating the need for attention, or a synchronous event in software indicating the need for a change in execution
  - Embedded processor peripheral (FIT, PIT, for example)
  - External bus peripheral (Uart, EMAC, for example)
  - External interrupts enter via hardware pin(s)
  - Multiple hardware interrupts can be ORed or utilize an interrupt controller (bus peripheral)

What Happens?

- Service both interrupts and exceptions
  - Current program execution is suspended after the current instruction
  - Context information is saved so that execution can return to the current program
  - Execution is transferred to an interrupt handler to service the interrupt
    - Interrupt Service Routine (ISR) must be registered
    - Interrupt handler calls an ISR
    - For simple situations, the handler and ISR can be combined operations
    - Each ISR is unique to the task at hand
      - Uart interrupt to process a character
      - Divide-by-zero exception to change program flow
  - When finished
    - Normal - returns to point in program where interrupt occurred
    - Exception - branches to error recovery

Interrupts Inclusion

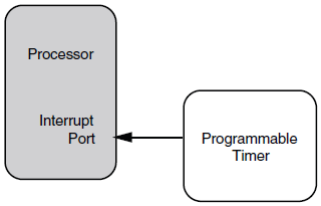
- Make hardware connections (next slides)
- Develop software routines
  - Write a void software function that services the interrupt
  - Register the interrupt handler by using an appropriate function
    - Single external interrupt registers with the processor function
    - Multiple external interrupts register with the interrupt controller
    - The call back registration argument is optional
  - Use the provided device IP routines to facilitate writing the ISR
    - Clear interrupt
    - Perform the interrupt function
    - Re-enable interrupt upon exit

Interrupt Management

Without Interrupt Controller

- The interrupt controller is not required when only one interrupting device is present
  - The interrupt signal of the peripheral (or the external interrupt signal) must be connected to the interrupt input of the MicroBlaze™ processor
    - Define an external requesting signal in the global ports section of the MHS file

      e.g., PORT interrupt_in1 = interrupt_in1, DIR = IN, LEVEL = low, SIGIS = Interrupt



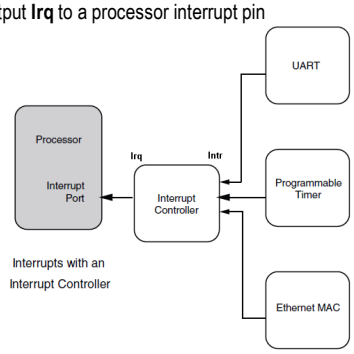
Interrupt Management

With Interrupt Controller

- The interrupt controller is required if more than one interrupting device is present
  - Connect peripheral's (or external) interrupt requesting signals to the **Intr** port of the interrupt controller

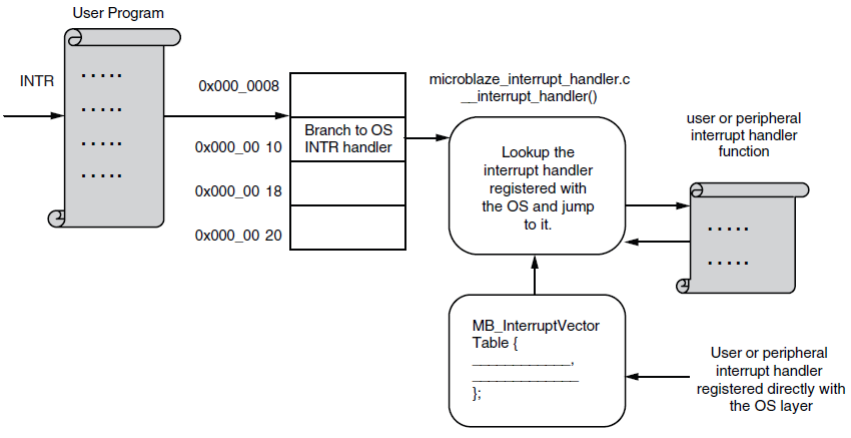
    e.g., PORT Intr = RS232_Interrupt & interrupt_push & interrupt_timer
  - Connect interrupt controller output **Irq** to a processor interrupt pin

    e.g., PORT Irq = interrupt_req

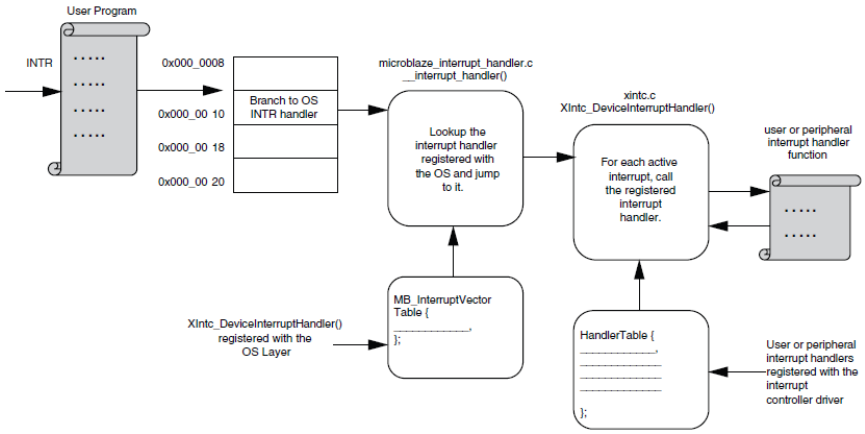


Interrupts with an Interrupt Controller

Interrupt Service Flow (Without INTC)



Interrupt Service Flow (With INTC)



MicroBlaze Interrupts

- MicroBlaze processor functions
  - void microblaze_enable_interrupts(void)
    - This function enables interrupts on the MicroBlaze processor
    - When the MicroBlaze processor starts up, interrupts are disabled. Interrupts must be explicitly turned on by using this function
  - void microblaze_disable_interrupts(void)
    - This function disables interrupts on the MicroBlaze processor. This function may be called when entering a critical section of code where a context switch is undesirable
  - void microblaze_register_handler(XInterruptHandler *Handler, void *DataPtr)*
    - The handler is invoked in turn, by the first level interrupt handler that is present in Standalone.
    - The first level interrupt handler saves and restores registers, as necessary for interrupt handling, so that the function you register with this handler can be dedicated to the other aspects of interrupt handling, without the overhead of saving and restoring registers

Outline

- **Device Drivers**
  - Level 0, Level 1
  - MicroBlaze Processor: Interrupts
  - **Integration in EDK**
- Libraries
- BSP
  - Boot Files and Sequence

Integration in EDK

- When the interrupt generating device is connected to the processor interrupt pin, either through an interrupt controller or directly, the interrupt handler function must be developed (You must explicitly write code to set up the interrupt mechanism)
- The interrupt handler must be registered explicitly in code

```
71 int main() {
72
73  int count_mod_3;
74
75   // Enable MicroBlaze Interrupts
76  microblaze_enable_interrupts();
77
78  /* Register the Timer interrupt handler in the vector table */
79  XIntc_RegisterHandler(XPAR_XPS_INTC_0_BASEADDR,
80                        XPAR_XPS_INTC_0_DELAY_INTERRUPT_INTR,
81                        (XInterruptHandler) timer_int_handler,
82                        (void *)XPAR_DELAY_BASEADDR);
83
84  /* Initialize and set the direction of the GPIO connected to LEDs */
85  XGpio_Initialize(&gpio, XPAR_LEDS_8BIT_DEVICE_ID);
86  XGpio_SetDataDirection(&gpio,LEDChan, 0);
```

Registering an Interrupt Handler