

Demystifying DDGEN

Under the hood of DDGEN

Explain the key concepts, algorithms and the manner in which DDGEN automatically generates device drivers.

- Provide a White box view of DDGEN
- Use USB device controller linux driver as the motivating example

USB device driver model on Linux

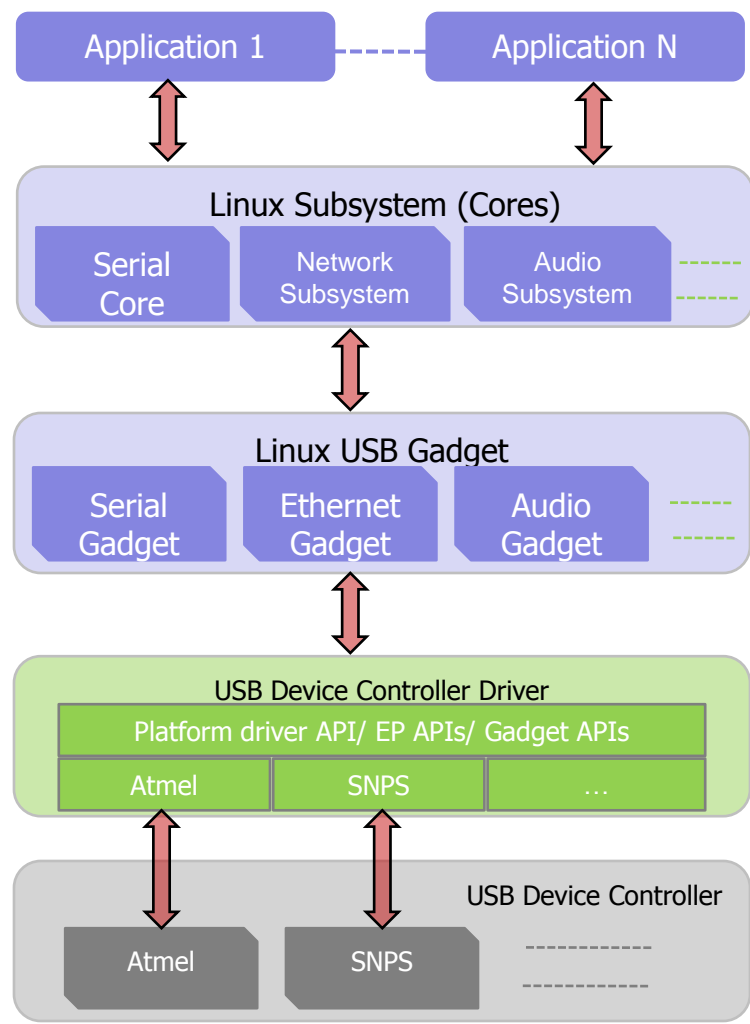
Key aspects in manually writing the USB device driver

Brief Introduction to DDGEN

DDGEN Demystified

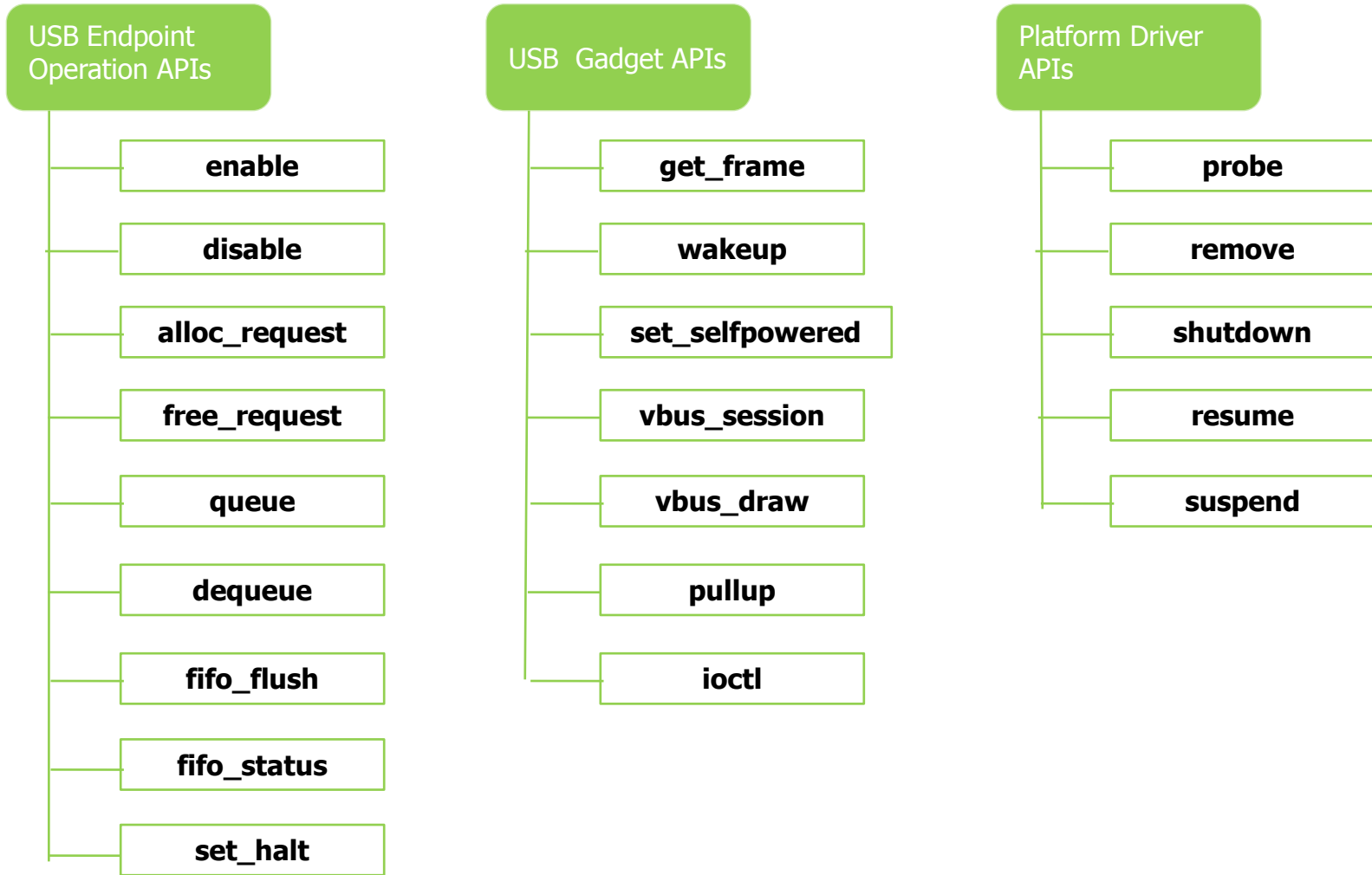
Summary

USB Device Driver Model on Linux



USB Stack

- A typical USB device controller driver consists of:
 - End point APIs corresponding to management and configuration of the USB Endpoints
 - Platform Driver APIs which are standard and common across all linux platform drivers
 - Gadget APIs which are essentially used by the class drivers in the USB stack



API	Description
<code>int (*probe)(struct platform_device *)</code>	This function registers usb device and irq handler, performs platform specific initialization if necessary
<code>int (*remove)(struct platform_device *)</code>	This function unregisters usb device and irq handler, performs platform specific deinitialization if necessary
<code>void (*shutdown)(struct platform_device*)</code>	This function shutdowns the device
<code>int (*suspend)(struct platform_device *)</code>	This function suspends the device
<code>int (*resume)(struct platform_device *)</code>	This function resumes the device

API	Description
<code>int (*enable) (struct usb_ep* , const struct usb_endpoint_descriptor*)</code>	Configures an endpoint making it usable
<code>int (*disable) (struct usb_ep *)</code>	Disables the specified endpoint. The endpoint will no longer be usable
<code>void (*free_request) (struct usb_ep*, struct usb_request*)</code>	Frees a request object
<code>struct usb_request* (*alloc_request)struct usb_ep*, gfp_t)</code>	Allocates request object to use with the specified endpoint
<code>int (*queue) (struct usb_ep*, struct usb_request*, gfp_t)</code>	Submits an I/O request to the specified endpoint
<code>int (*dequeue) (struct usb_ep*, struct usb_request*)</code>	Dequeues(cancels/unlinks) an I/O request from the specified endpoint
<code>int (*set_halt) (struct usb_ep*, int)</code>	Sets the endpoint halt feature
<code>int (*set_wedge) (struct usb_ep*)</code>	Sets the halt feature and ignores the clear requests
<code>void (*fifo_flush) (struct usb_ep*)</code>	Flushes the fifo contents of the specified endpoint
<code>int (*fifo_status) (struct usb_ep*)</code>	Returns the number of bytes in the fifo of the specified endpoint

API	Description
<code>int (*get_frame)(struct usb_gadget*)</code>	Returns the usb frame number, normally eleven bits from a SOF packet, or negative error number if this device doesn't support this capability.
<code>int (*wakeup)(struct usb_gadget*)</code>	Tries to wakeup the host connected to this gadget. Returns zero on success and negative error code on error
<code>int (*set_selfpowered)(struct usb_gadget*, int is_selfpowered)</code>	Sets the device selfpowered feature. This affects the device status reported by the hardware driver to reflect that it now has a local power supply.
<code>int (*vbus_session)(struct usb_gadget*, int is_active)</code>	Detects a VBUS power session starting incase of an external transceiver. Further it also enables the pullup in this case.
<code>int (*vbus_draw)(struct usb_gadget*, unsigned mA)</code>	This function is use to constrain controller's VBUS power usage. This call is used by gadget drivers during SET_CONFIGURATION calls, reporting how much power the device may consume.
<code>int (*pullup)(struct usb_gadget*, int is_on)</code>	This function is used by gadget driver to enable or disable the pullup on D+ or D-
<code>int (*ioctl)(struct usb_gadget*, unsigned code, unsigned long param)</code>	This function is used to perform device specific operations, read, write and I/O controller operations.

USB device driver model on Linux

Key aspects in manually writing the USB device driver

Brief Introduction to DDGEN

DDGEN Demystified

Summary

- The Driver Model imposes fixed APIs expected by the USB stack
 - All the hardware functionality can only be exposed through these APIs
 - Device Driver writer has to work within the confines of this driver architecture
- The Driver APIs need to be interfaced correctly with the USB stack
 - Driver architecture imposes usage of certain structures and functions from the USB stack
 - Different class drivers use different driver APIs
 - Some awareness of class drivers is required*
 - Other option is to implement all the APIs*

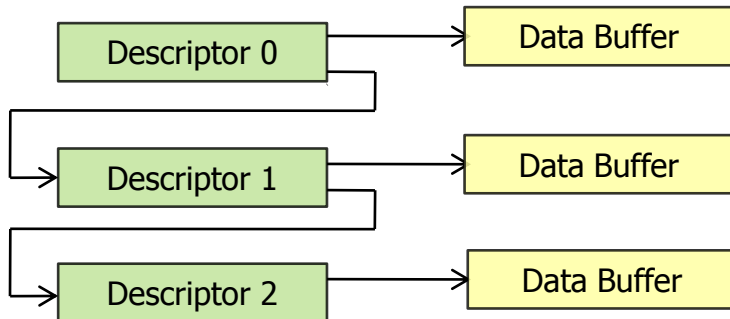
Buffer and Descriptor Management

- Buffer information typically provided to the device using *descriptors*. Descriptors may be a part of device registers or most typically system memory

Status	Terminate Bit
Total Bytes To Transfer	
Buffer Pointer	
Link Address	

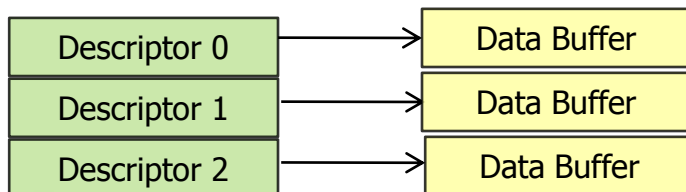
Example of a Descriptor Layout

- Figure on LHS shows typical descriptor Layout
- Buffer Pointer will hold the actual data buffer to be transmitted/received
- Link Address specifies the address of the next descriptor. This field is not applicable if descriptors are organized as an array



Descriptors in Linked List

- Multiple ways of organizing the descriptors
 - Lists, Arrays, Circular lists, Ring etc.
- Typical tasks while writing device driver:
 - Defining a C structure for descriptors
 - Laying out the descriptors in memory
 - Routines to iterate over descriptors based on the iterator type
 - Allocation, DeAllocation, initialization of descriptors, lot of book-keeping code
 - Often this descriptor management tasks are carried out across multiple driver APIs
 - Quite a lot of code that needs to be written
- All the above makes the driver writer's task tedious



Array of Descriptors

Generic aspects in ISR writing

- Interrupt Service Routine is one of the most important aspect of a device driver
- Typical considerations:
 - ✓ Ensure all interrupts are handled correctly. Appropriate enabling/disabling/clearing of interrupts
 - ✓ If multiple interrupts can occur at the same time, then it needs careful handling.
 - ✓ Check for interrupt occurrence in ISR body should be done using series of "if" instead of "if-else" or "switch" statements
 - ✓ Synchronization of ISR and main driver thread using semaphores, completion etc.
 - ✓ Should ISR be split?
 - ✓ In case it is split – should work be scheduled using tasklets or workqueues?
 - ✓ Does the device support nested interrupts?

USB device specific aspects in ISR writing

- USB device enumeration as well data transfer interrupt for each endpoint needs to be handled as a part of the ISR
- Typically `setup_data_status` interrupt will occur for endpoint 0 to perform the device enumeration
- Resume/Suspend Interrupt for the device controller will need to be serviced
- Transmit/Receive Interrupt for each endpoint will need to be serviced

USB device driver model on Linux

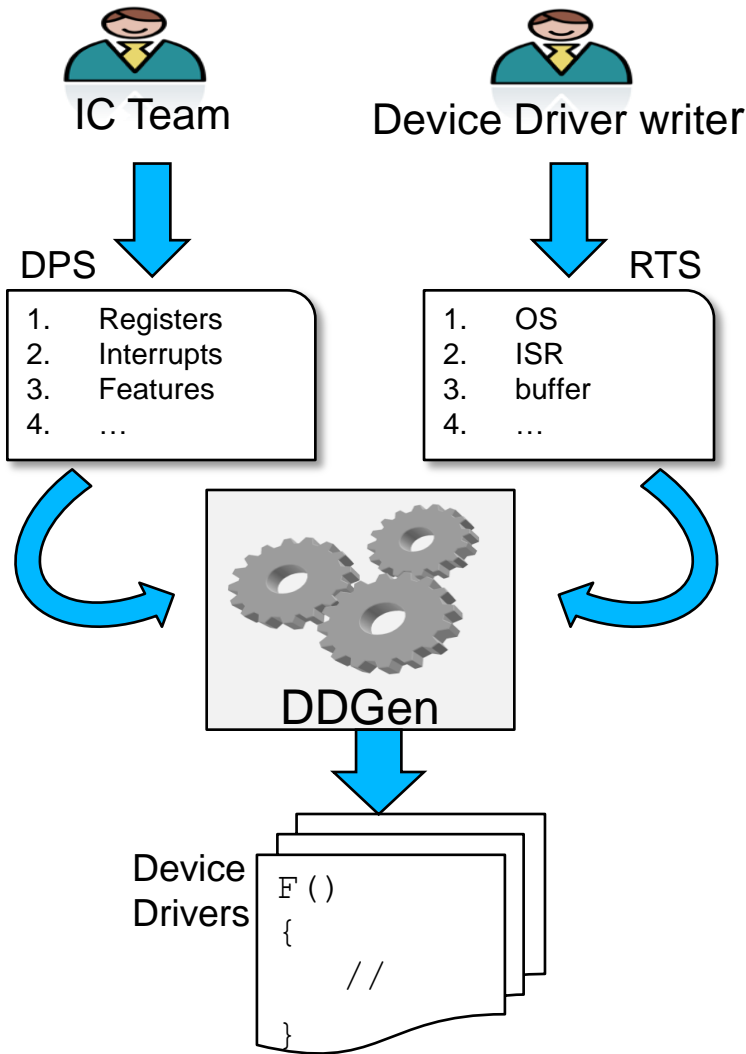
Key aspects in manually writing the USB device driver

Brief Introduction to DDGEN

DDGEN Demystified

Summary

DDGen Methodology



- DDGEN methodology helps formalize communication between hardware and software teams
- Effective separation of the hardware and software concerns
- Think in *problem domain* rather than the *implementation domain*

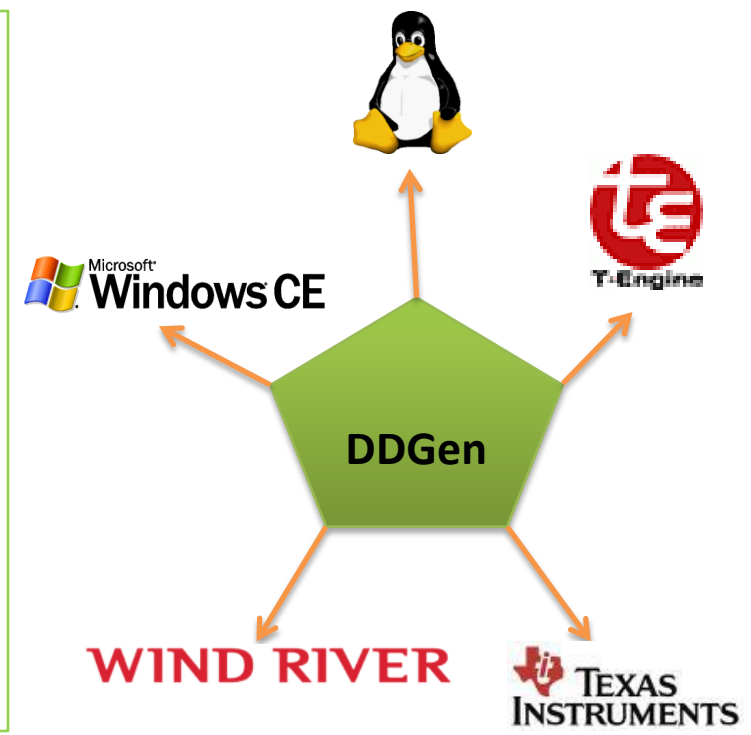
DPS: Device Programming Specification

RTS: Run Time Specification

DDGen: Device Driver Generator

DDGen – Details

- ~300% productivity gain in device driver development
- Generates
 - ANSI C code including OS calls. Fully functional device driver
 - Unit test code of device driver
- Supports
 - A range of device complexities: UART, I2C, USB, Ethernet, PCI, PCI Express, SD/MMC controllers...
 - Many popular Operating Systems, null OS



Activity	Effort in person days
Writing DPS/RTS	19
Integration and testing generated driver in the target environment	12
Total Effort using DDGEN	31
Total Effort for manual driver generation	90

USB device driver model on Linux

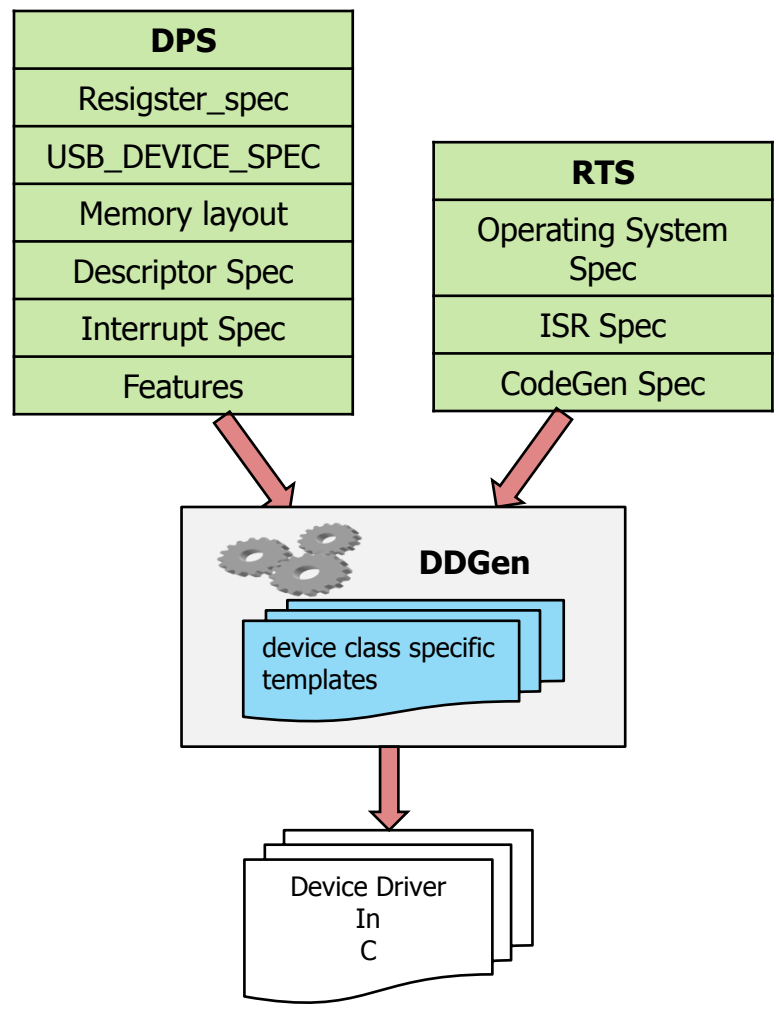
Key aspects in manually writing the USB device driver

Brief Introduction to DDGEN

DDGEN Demystified

Summary

DDGEN Code Generation at High-Level



- DDGEN code generation involves:
 - ✓ Selection of the appropriate code template based on the device-class, Operating System and the driver model. For USB, the corresponding template gets selected
 - ✓ Synthesis of all other parts of the driver C code based on the DPS and RTS
- Internal template is largely imposed by the USB driver model for linux
 - ✓ Consists of the stack specific structures that get used
 - ✓ Some of the APIs that eventually get constructed from DPS features
- Synthesis of various driver aspects as given below:
 - ✓ Register read/write routines are directly generated from Register spec in DPS and CodeGen Spec in RTS
 - ✓ C structure for descriptors from memory layout spec in DPS
 - ✓ Descriptor iteration and management from descriptor spec in DPS
 - ✓ ISR from Interrupt Spec in DPS and ISR spec in RTS
 - ✓ Features in DPS get translated to C functions. Non-keyword features need to be manually mapped to corresponding function pointers in templates

What is a template?

- Template is the static part of the driver code. It will essentially be same across all the drivers that for a specific device-class
- Typically consists of :
 - ✓ definition for stack structures required by the USB driver model
 - ✓ Skeletal/fixed part of code for all the endpoint control related APIs and platform driver APIs
 - Remaining portion of these APIs will get generated based on DPS and RTS

Linux USB Device Controller Driver Template **

DPS

```
Device_spec {
    device_class = USB;
    manufacturer_name=xyz;
}
```

RTS

```
OS_spec{
    OS = Linux
    DRIVER_MODEL = PLATFORM
}
```

```
struct usb_ep_ops xyz_ep_ops = {
    .enable = xyz_ep_enable,
    .disable = xyz_ep_disable,
    .alloc_request = xyz_ep_alloc_request,
    .free_request = xyz_ep_free_request,
    .queue = xyz_queue,
    .dequeue = xyz_dequeue,
    ...
};
...
static INT __init xyz_probe(struct platform_device *pdev)
{
    INT retval;
    struct xyz_controller *usbDevController;
    logMsg("-->xyz_probe(pdev %p)\n", pdev);
    if(NULL == pdev) {
        printk(KERN_WARNING "Platform device not found\n");
        return -ENODEV;
    }
    usbDevController = &usbDevice;
    ...
    platform_set_drvdata(pdev, usbDevController);
    ...
}
```

**
Text in Black is template. While Text in blue color is generated by the tool

Code Generation of Register Access

- Register access routines are directly generated from the register specification in the DPS and the bus_spec and Codegen spec in RTS

DPS

```

ENDPTNAK[32] @0x0178 {
  SW_AXS = RW;
}
FIELD EPTN <16:31> {
  SW_AXS = RW;
}
...
}
...
}

```

RTS

```

BUS_SPEC {
  REG_ACCESS = MEMORY_MAPPED;
}
CODEGEN_SPEC {
  REG_ACCESS_IMPLEMENTATION =
  DIRECT;
}

```

```

#define ENDPTNAK_RgOffAddr (
  *(( volatile ULONG *) )(BASE_ADDRESS + 0x178))

#define ENDPTNAK_RgWr(data) do {\
  ((ENDPTNAK_RgOffAddr) = data);\
} while(0)

#define ENDPTNAK_RgRd(data) do {\
  ((data) = ENDPTNAK_RgOffAddr);\
} while(0)

#define ENDPTNAK_EPTN_Wr_Mask
  (ULONG)(0xffff)

#define ENDPTNAK_EPTN_UdfWr(data) do {\
  ULONG v;\
  v = ENDPTNAK_RgOffAddr;\
  v = ((v & ENDPTNAK_EPTN_Wr_Mask) |
  ((data & ENDPTNAK_EPTN_Mask)<<16));\
  ENDPTNAK_RgOffAddr = v;\
} while(0)

#define ENDPTNAK_EPTN_UdfRd(data) do {\
  data = ((ENDPTNAK_RgOffAddr >> 16 ) & ENDPTNAK_EPTN_Mask);\
} while(0)

```

- As shown in the figure above, read/write macros are generated for register ENDPTNAK and its field EPTN as well
- If REG_ACCESS is specified as I2C/SPI/PCI, then appropriate wrapper calls are generated to access registers
- If REG_ACCESS_IMPLEMENTATION is specified as OS_CALLS, then register read/writes are generated with ioread/iowrite calls instead of direct pointer access

Code Generation for Descriptors

```

MEMORY_LAYOUT
{
  ENDPOINT_TRANS_DESC {
    ELEMENT_ALIGN = 32;
    ALLOC_ALIGN = 4096;

    DWORD0[32] {
      ...
    }

    DWORD1[32] {
      field stat<0:7> ...
      field total_bytes_to_transfer<16:30> {
        type = rw;
        value_on_reset = 0;
      }
    }
    DWORD2[32] {
      field curr_offset<0:11> {
        type = rw;
        value_on_reset = 0;
      }

      field buffer_ptr_page0<12:31> {
        type = rw;
        value_on_reset = 0;
      }
    }
  }
  ...
}
DESCRIPTOR_SPEC
{
  NUM_OF_DESCRIPTORs = 8;
  ITERATOR_TYPE = ARRAY;
  DESCRIPTOR_ALLOCATION = EXTERNAL_MEM;
  DESCRIPTOR_TERMINATE =
    ENDPOINT_TRANS_DESC.DWORD0.T('b1');
  DESCRIPTOR_DATA_LENGTH=
    ENDPOINT_TRANS_DESC.DWORD1.total_bytes_to_transfer;
  BUFFER_PTR =
    ENDPOINT_TRANS_DESC.DWORD2.buffer_ptr_page0,
  ....
}
BUFFER_SPEC {
  BUFFER_SIZE = 4096;
}

```

```

//macros to access fields (corresponding to the memory layout) in the generated C structure
#define ENDPOINT_TRANS_DESC_DWORD1_TOTAL_BYTES_TO_TRANSFER 0x8000ffff
#define ENDPOINT_TRANS_DESC_DWORD2_BUFFER_PTR_PAGE0 0xffff

//C structure corresponding to the memory layout specified in the DPS
struct s_ENDPOINT_TRANS_DESC {
  UINT DWORD0;
  UINT DWORD1;
  UINT DWORD2;
  UINT DWORD3;
  UINT DWORD4;
  UINT DWORD5;
  UINT DWORD6;
  UINT padding;
} __attribute__((aligned(32)));
typedef struct s_ENDPOINT_TRANS_DESC t_ENDPOINT_TRANS_DESC;

//Prototypes of Descriptor management APIs
init_descriptor (struct usb_dev_prvdata *host);
alloc_descriptor (struct usb_dev_prvdata *host);
create_descriptor (struct usb_dev_prvdata *host);
alloc_buffer(struct usb_dev_prvdata *host);
init_buffer(struct usb_dev_prvdata *host);

```

- Descriptor Layout is specified using MEMORY_LAYOUT spec in DPS
- The type of descriptor, number of descriptors and other details are specified in the DESCRIPTOR_SPEC
- MEMORY_LAYOUT in DPS gets translated to a structure in C
- Macros to access the corresponding fields are also generated
- Descriptor management code gets synthesized based on the DESCRIPTOR SPEC
- The Driver model APIs that are generated by DDGEN use the relevant functions for descriptor handling

- ISR is generated based on the INTERRUPT_SPEC and USB_DEVICE_SPEC in DPS and the ISR_SPEC in RTS
- INTERRUPT_SPEC
 - ✓ Routines to enable/disable each specific interrupt based on the corresponding attributes
 - ✓ Code to clear each interrupt that has occurred as well
- USB_DEVICE_SPEC
 - ✓ Routines to handle reset, resume, suspend interrupts for the device controller are generated from the INTERRUPT_STATUS section in the USB_DEVICE_SPEC
 - ✓ Code to handle transmit and receive interrupt for each endpoint is also generated from endpoint specific details in the USB_DEVICE_SPEC and the interrupt details in the INTERRUPT_SPEC
 - ✓ Code to handle SETUP_DATA_STATUS pertaining to device enumeration also generated based on the corresponding attributes specified for endpoint0
- ISR_SPEC
 - ✓ IRQ number with which the ISR should be registered is obtained from the RTS
 - ✓ DDGEN automatically generates the call to register the ISR
 - ✓ Decision on whether processor context should be saved in the ISR is also obtained from RTS

ISR Generation - Example

```

INTERRUPT_SPEC
{
  USB_ISR {
    USBSTS.URI('b1) {
      int_type = status ;
      clear_field = COW USBSTS.URI('h1);
      enable_field = USBINTR.URE('h1);
      disable_field = USBINTR.URE('h0);
    }

    ENDPTSETUPSTAT.ENDPTSETUPSTAT0('b1) {
      int_type = status ;
      clear_field = COW
      ENDPTSETUPSTAT.ENDPTSETUPSTAT0('h1);
      enable_field = ENDPTCTRL0.RXE('h1);
      disable_field = ENDPTCTRL0.RXE('h0);
    }
    ...
  }
}

USB_DEVICE_SPEC {
  NUMBER_OF_ENDPOINTS = 16;
  INTERRUPT_STATUS {
    RESET = USBSTS.URI('b1);

    SUSPEND = USBSTS.SLI('b1);
    RESUME = USBSTS.SRI('b1);
  }
}

END_POINT 0 {
  SIZE = 64;
  SETUP_DATA_STATUS =
  ENDPTSETUPSTAT.ENDPTSETUPSTAT0('h1);
  SETUP_DATA_BUFFER =
  ENDPOINT_QUEUE_HEAD.DWORD10;
  ...
}

```

```

irqreturn_t xyz_ISR_SW_USB_ISR(int irq, void * device_id)
{
  ULONG varUSBSTS ;
  ULONG varENDPTSETUPSTAT ;
  ULONG varENDPTCOMPLETE ;
  USBSTS_RgRd(varUSBSTS);
  ENDPTSETUPSTAT_RgRd(varENDPTSETUPSTAT);
  ENDPTCOMPLETE_RgRd(varENDPTCOMPLETE);

  if (GET_VALUE(varUSBSTS, USBSTS_URI_LPOS, USBSTS_URI_HPOS) ==
      USBSTS_URI_RESET) {
    xyz_bus_reset_int();
    GStatus = S_RESET ;
    USBSTS_URI_UdfWr(1);
  }

  if (GET_VALUE(varENDPTSETUPSTAT, ENDPTSETUPSTAT_ENDPTSETUPSTAT0_LPOS,
      ENDPTSETUPSTAT_ENDPTSETUPSTAT0_HPOS) ==
      ENDPTSETUPSTAT_ENDPTSETUPSTAT_clear) {
    ENDPTSETUPSTAT_ENDPTSETUPSTAT0_UdfWr(1);
    usbd_setup_isr_handler();
    GStatus = S_CLEAR ;
  }
  ...
}

```

- **USBSTS.URI** is an interrupt identifier for an interrupt of type status as specified in the INTERRUPT SPEC. INTERRUPT_STATUS section in USB_DEVICE_SPEC indicates that this interrupt corresponds to a reset interrupt for the device endpoints and hence in the generated code the reset handler – **xyz_bus_reset_int** gets invoked
- Similarly **ENDPTSETUPSTAT.ENDPTSETUPSTAT0** corresponds to the SETUP_DATA_STATUS interrupt for endpoint0 in the USB_DEVICE_SPEC and hence the setup handler – **usbd_setup_isr_handler** gets synthesized in the generated code

- Endpoint configuration APIs are automatically generated from the USB_DEVICE_SPEC in DPS

```

USB_DEVICE_SPEC {
  END_POINT 0 {
    SIZE = 64;
    SUPPORTED_TRANSFER_TYPE =
      CONTROL(ENDPTCTRL0.RXT('b00'));
    IN(ENDPTCTRL0.TXE('b1')) {
      ENABLE = ENDPTCTRL0.TXE('b1');
      DISABLE = ENDPTCTRL0.TXE('b0');
      ...
    }
  }
}

```

```

xyz_enable(struct usb_ep *ep, const struct usb_endpoint_descriptor *desc)
{
  init_ep_queue_head(desc->bEpAddress, desc->bmAttributes,
                    desc->wMaxPacketSize);
  ...
  configure_endp(desc->bEpAddress, desc->bmAttributes);
  ...
}

configure_endp(UCHAR epadr, UCHAR bmAttributes)
{
  UINT type = (bmAttributes & USB_ENDPOINT_XFERTYPE_MASK);
  ..
  switch(epadr) {
    case USB_EP0_IN :
      ENDPTCTRL0_RXT_UdfWr(type);
      → ENDPTCTRL0_TXE_UdfWr(1);
      break ;
    case USB_EP0_OUT :
      ...
  }
}

```

- The enable API (xyz_enable) is generated automatically from the USB_DEVICE_SPEC
- In configure_endp:
 - ✓ ENDPOINT0 is configured based on the direction specified by the class driver. This particular code snippet shows it being configured for IN direction by writing 1 to the field TXE in the endpoint 0 control register ENDPTCTRL0
- Note that code to configure all the end points gets generated in a similar fashion

Synthesis of read functionality

```

USB_DEVICE_SPEC
{
  end_point 1 @ DR0
  {
    received_interrupt_status =
    ENDPTCOMPLETE.ERCE1('h1);
  }
  ....
}

INTERRUPT_SPEC
{
  ENDPTCOMPLETE.ERCE1('b1) {
    int_type = device_read;
    source_field = USBSTS.UI('b1);
    clear_field = COW ENDPTCOMPLETE.ERCE1('b1);
  }
  ....
}

FEATURE device_read
{
  input unsigned short epno;
  output unsigned char data[];
  input unsigned short count;
  local unsigned short idx;
  ....
  repeat(idx < count) {
    data[idx] = DR[epno];
    idx = idx + 1;
  }
}

```

```

irqreturn_t xyz_ISR_SW_USB_ISR(int irq,, void * device_id)
{
  if (GET_VALUE(varENDPTCOMPLETE, ENDPTCOMPLETE_ERCE1_LPOS, ENDPTCOMPLETE_ERCE1_HPOS) ==
  ENDPTCOMPLETE_ERCE_clear) {
    handle_ep_request(1)
    ENDPTCOMPLETE_ERCE1_UdfWr(1);
  }
  ....
}

static INT handle_ep_request(USHORT epNo)
{
  struct xyz_ep *ep = &usbDevice.epInfo[epNo];
  struct xyz_request *req = NULL;

  if(!list_empty(&ep->queue)) {
    req = list_entry(ep->queue.next, struct ATMEL_request, queue);
  }

  if(req) {
    if(ep->is_in) {
      return transmit_data(ep, req);
    }
    else {
      return receive_data(ep, req);
    }
  }
  return 0;
}

static INT receive_data(struct xyz_ep *ep, struct xyz_request *req)
{
  int count = usbd_get_reieve_data_count(ep->ep_no);
  ....
  device_read(ep->ep_no, buf, count);
  ....
}

```

- The read interrupt details for endpoint 1 are captured in the USB_DEVICE_SPEC and INTERRUPT_SPEC in the DPS. The DPS also shows the keyword feature `device_read` – consisting of the programming sequence to read data in slave mode. Since it's a keyword feature, DDGEN knows that the corresponding generated C function has to be invoked for handling the receive interrupt of endpoint 1.
- In the generated C code, the function `handle_ep_request` is called for both receive/transmit interrupt with the corresponding endpoint number as the argument. This function retrieves the request packet from the queue and correspondingly calls `receive_data` for the receive_interrupt. `receive_data` in turn has a call to `device_read`. Note that while the C function `device_read` is generated from the corresponding DPS feature `device_read`, other section of the code like `handle_ep_request/receive_data` are part of the USB device specific template. The `data write` functionality is also generated similarly
- Incase of an internal DMA, the `device_read` feature in DPS will have programming sequence to setup the memory descriptors and initiate DMA xfer

- Explicit features in DPS get translated to C code.
- The function thus generated needs to be manually mapped at an appropriate location in the generated C Code
- The template code consists of “placeholder” function pointers. These pointers need to be initialized to the appropriate functions

FEATURE `post_transmit`

```
{
  INPUT_ENDPOINT_TRANS_DESC desc;
  return = desc.DWORD1.stat;
}
```

```
typedef INT (*get_transmit_status_t)(void *);
...
struct hw_if_struct {
  get_transmit_status_t get_transmit_status;
  ...
}
INT post_transmit(void *td) {
  t_ENDPOINT_TRANS_DESC *desc = (t_ENDPOINT_TRANS_DESC*)td;
  if ( ((desc->DWORD1 & ENDPOINT_TRANS_DESC_DWORD1_STAT) >>
    ENDPOINT_TRANS_DESC_DWORD1_STAT_LBIT_POS) != 0 ) {
    return -1;
  } else {
    return 0;
  }
}
void xyz_init_function_ptrs(struct hw_if_struct *hw_if) {
  get_transmit_status = post_transmit
  ...
}
INT xyz_transfer_complete(UCHAR epadr) {
  ....
  get_ep_status(epadr, DESC_FREE, &pep, &preq);
}
INT get_ep_status(UCHAR ep_adrs, UINT use_flag,
  struct xyz_ep **pep_ret,
  struct xyz_request **preq_ret) {
  ...
  get_transmit_status(desc);
}
```

- The status of a transmitted descriptor for a specific endpoint needs to be obtained in the data transfer complete interrupt handler – `xyz_transfer_complete`
- The field `DWORD1.stat` in the descriptor actually contains this status
 - `Memory_layout` in DPS captures the descriptor layout
 - `Explicit feature post_transmit` returns the status in DPS
- The USB driver template is designed with a callback function pointer `get_transmit_status` to obtain the descriptor status
- All such function pointers are organized in a structure – `hw_if_struct`
- The DPS feature `post_transmit` gets translated to a corresponding C function by the same name as well
- Mapping of DPS features to function pointers involves initializing all the function pointers in the `hw_if_struct` to the correct C functions (generated from DPS features)
- The function pointer `get_transmit_status` is initialized to the function `post_transmit` in `xyz_init_function_ptrs`
 - All function pointers need are similarly initialized over here

USB device driver model on Linux

Key aspects in manually writing the USB device driver

Brief Introduction to DDGEN

DDGEN Demystified

Summary

- The USB device controller driver model essentially consists of the platform driver APIs and a set of APIs pertaining to various configurations of the endpoints
- DDGEN methodology effectively generates a full-fledged Linux driver for the USB device controller . This includes:
 - ✓ Generation of platform driver as well as endpoint configuration APIs
 - ✓ Code for Buffer and descriptor management
 - ✓ Interrupt Service Routine to handle all the Interrupts
- DDGEN identifies the USB driver template based on the device_class specification in DPS and the Operating System specification in RTS
- Buffer and Descriptor management code is generated based on the memory_layout, descriptor_spec and buffer_spec in DPS
- ISR code is generated from the interrupt_spec in DPS and isr_spec in RTS
 - ✓ Code to handle endpoint specific interrupts is generated from the usb_device_spec in DPS
- Endpoint configuration APIs are generated from the endpoint details captured in the usb_device_spec
- The `device_read` and `device_write` features get synthesized to the corresponding read/write functionality in the generated C code.
- Some of the APIs make use of callbacks. DDGEN provides a methodology to map explicit features in DPS to these function pointers.

Thank You!