

DDGEN

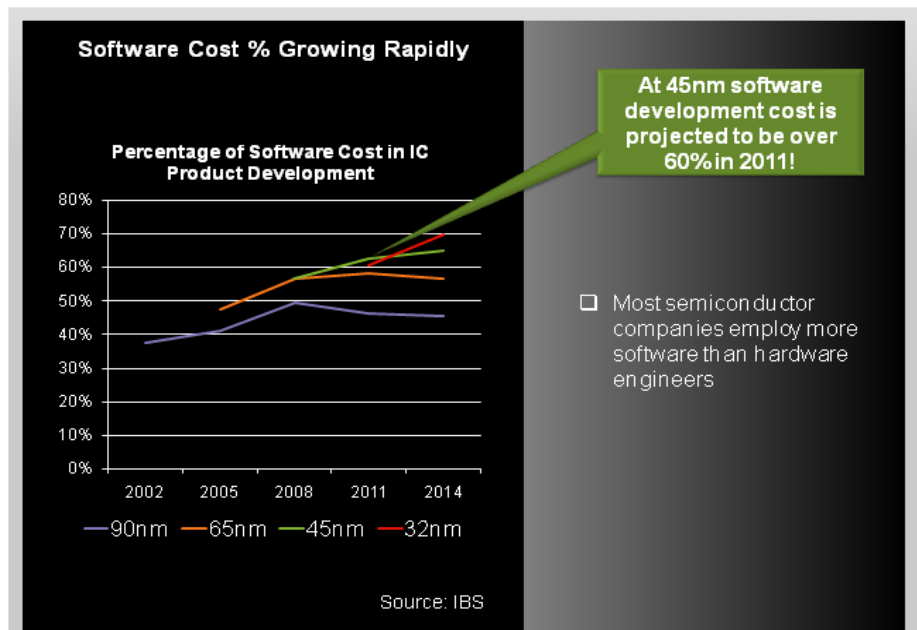
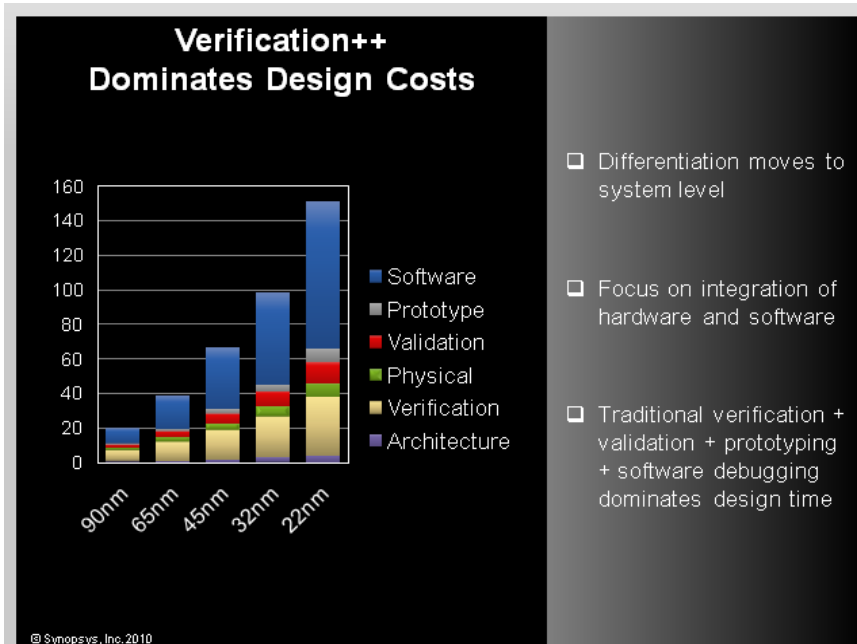
A Device Driver Generator

Sandeep Pendharkar, Vayavya Labs,

Workshop on Hardware Dependent Software
Solutions for SOC Design, DATE 2011

- Background
- Introduction to DDGEN
 - Motivation
 - DDGEN Methodology
- DDGEN Code Generation
- Evaluation Results
- Challenges & Future Work
- Q&A

The Landscape

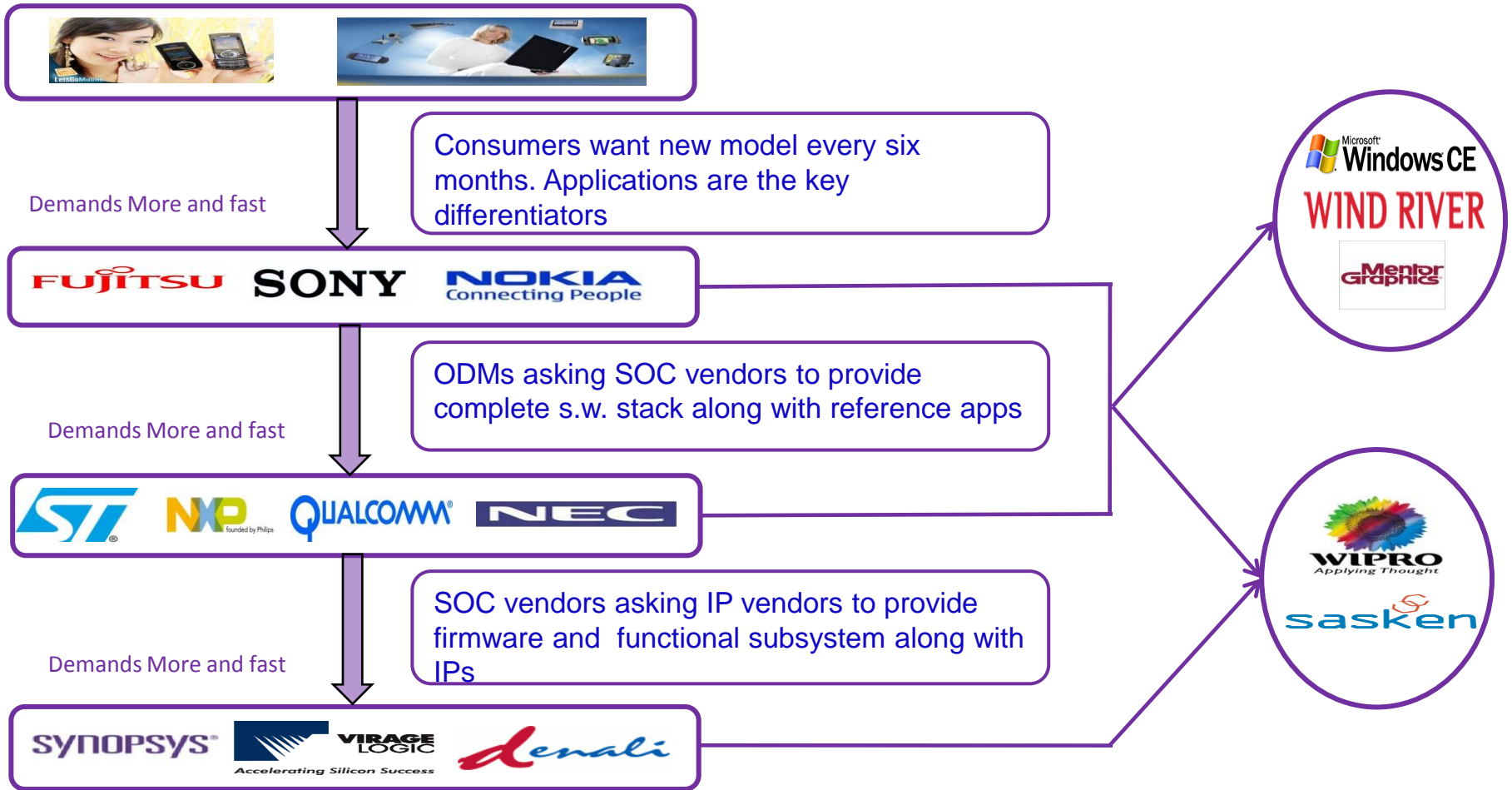


Source: Dr. Aart De Geus, SanJose SNUG 2010 keynote

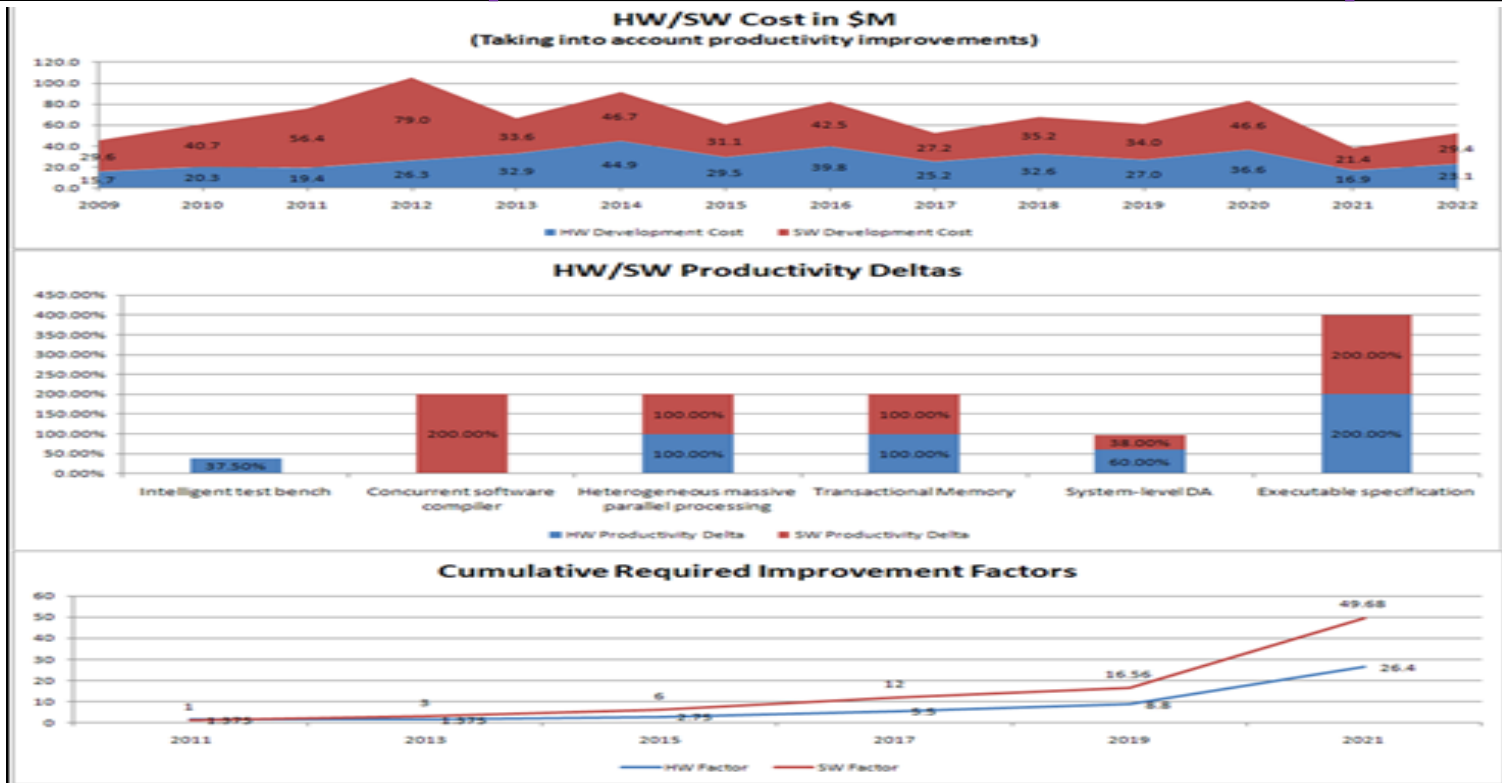
•It's the software!!!

- Software content is increasingly differentiating the system
- Cost of software development becoming higher than the hardware development
- Software development is often the bottleneck

Embedded Market Ecosystem



Holy Grail of Productivity

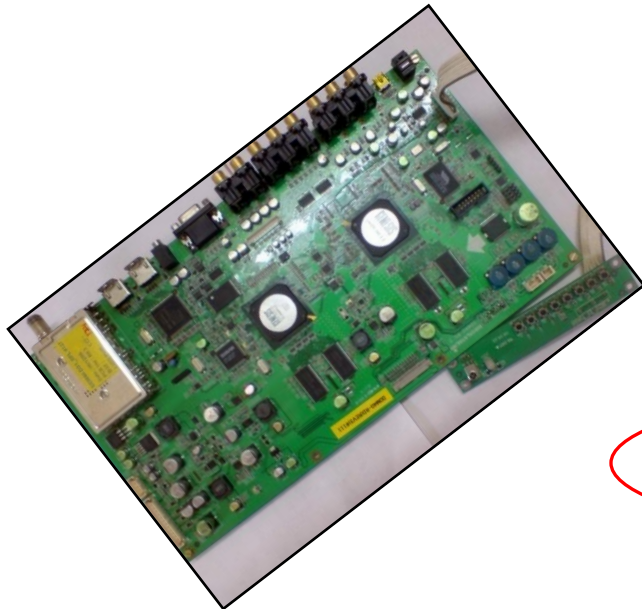


Source: IRTS 2009 Semiconductor Roadmap

- In order to keep up with the increasing complexity of electronic systems over the next 10 years, *automation* needs to provide
 - 26X improvements on the hardware side
 - 50X improvements for the software content

- Background
- **Introduction to DDGEN**
 - Motivation
 - DDGEN Methodology
- DDGEN Code Generation
- Evaluation Results
- Challenges & Future Work
- Q&A

Typical Embedded Platform



Number of SOCs	2(X86 and MIPS based)
App Domain	CE, DTV
Operating System	MontaVista Linux
On chip peripherals(drivers)	23
On board peripherals(drivers)	7
Effort for first release of firmware/drivers	78
Team Size	14
Engineering Support Team	5

Specifications

Partition
& Plan

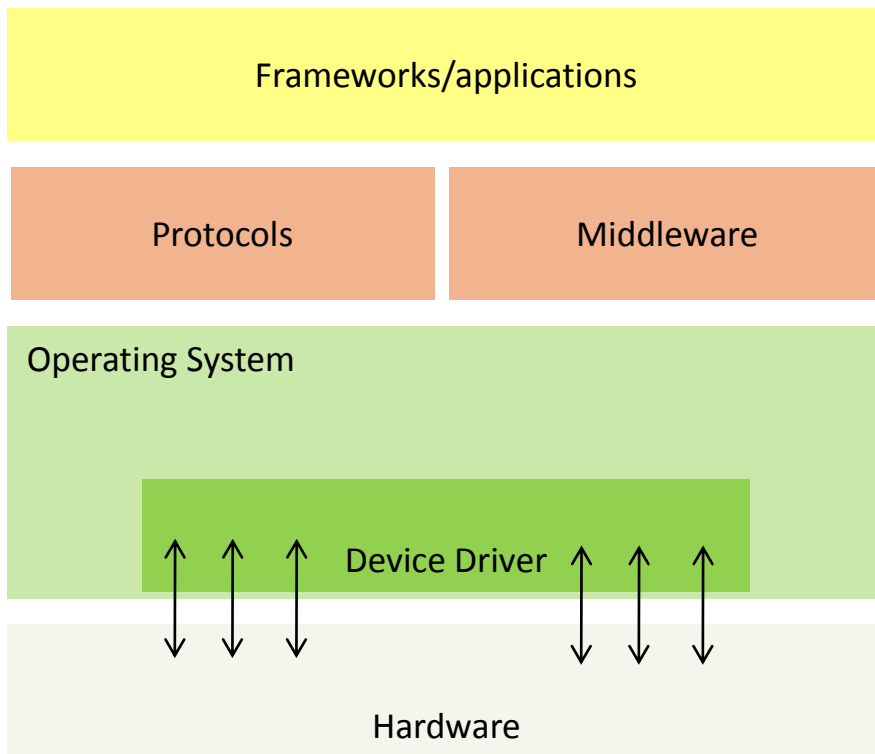
RTL
HW Design

SW Design

Integration
bugs/fix iteration loop

Automatic Generation of device drivers can significantly reduce the overall cost & efforts

Motivation for Device Driver Automation



•A typical embedded system project is characterized by:

- Strict and demanding schedule
- Periodic redesign of h.w. and associated s.w.
- Hardware-Software integration aka device drivers is the most significant pain-point or bottle neck*

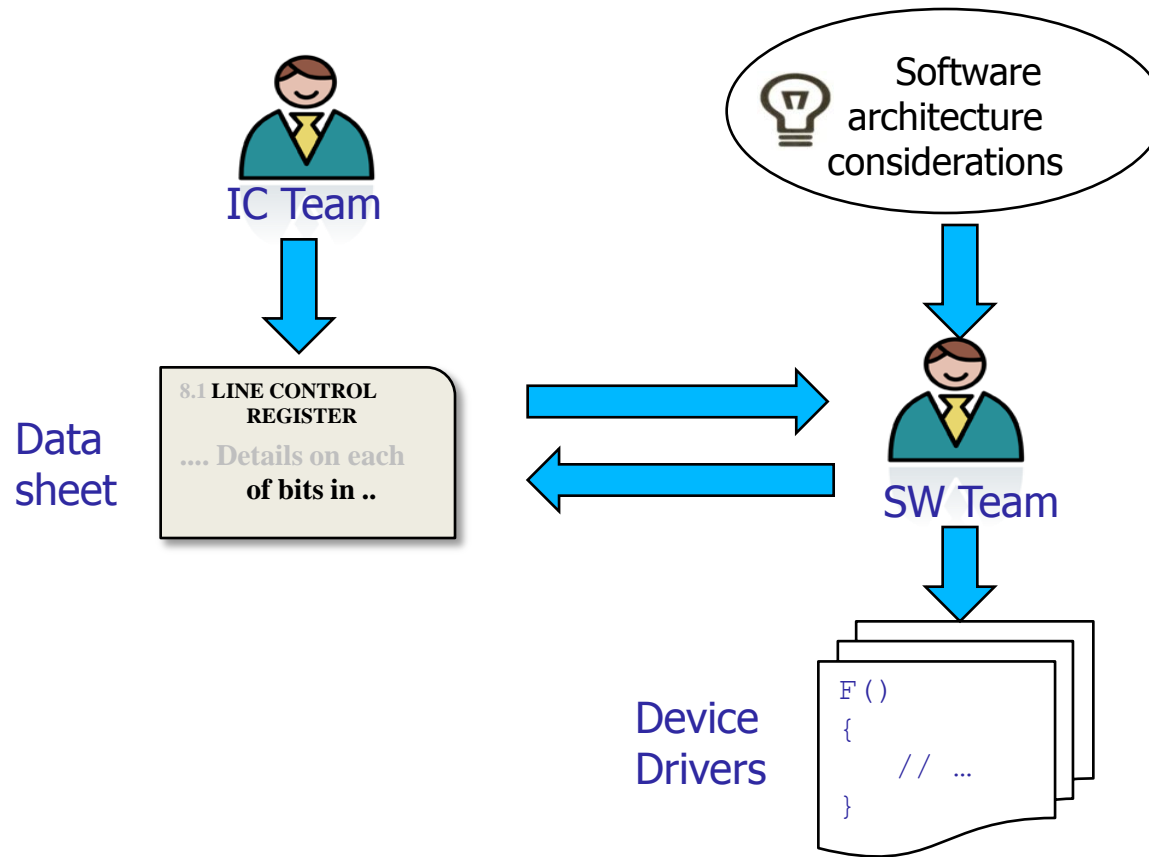
•Device drivers in embedded systems

- Error-prone and time-consuming
- Example: DM355 SOC from TI has around **18** peripherals and the total LOC for the BSP is **~50,000**
- Plethora of operating systems
- Linux, WinCE, Vxworks, etc.
- Variety of driver models on the same operating system..
- Informal communication between the hardware and software teams**

•Market needs productivity improvement tools – tools that can automatically synthesize the device driver software

- Background
- **Introduction to DDGEN**
 - Motivation
 - DDGEN Methodology
- DDGEN Code Generation
- Evaluation Results
- Challenges & Future Work
- Q&A

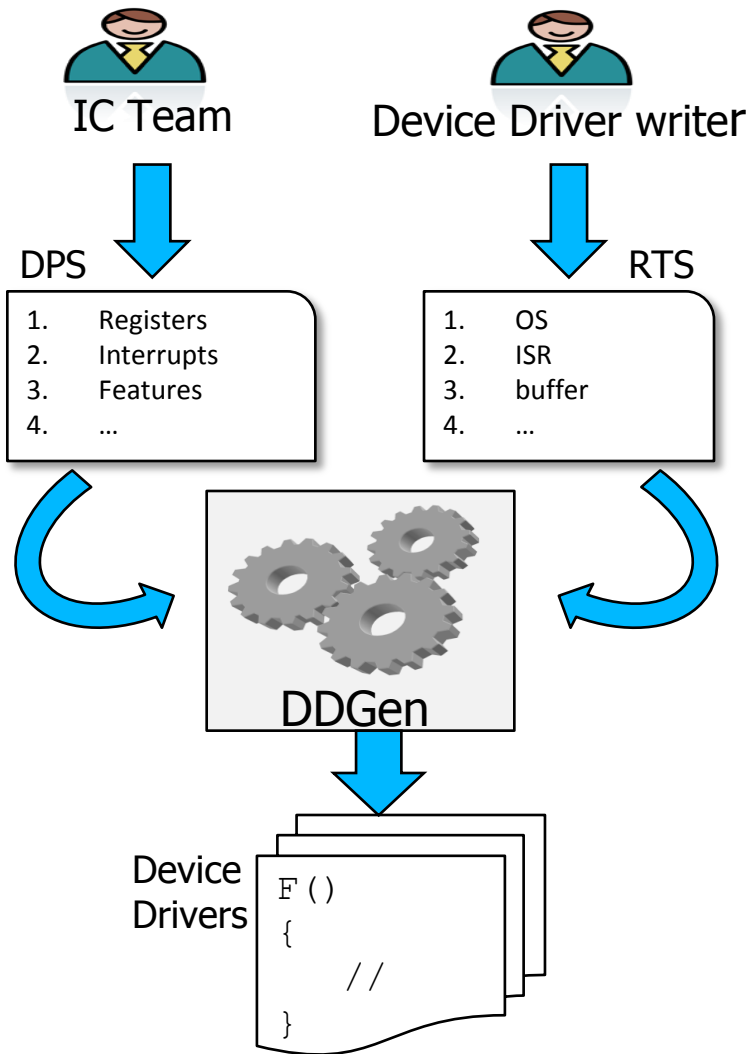
Conventional Way of Driver Development



Informal/ad-hoc communication between IC and software teams

- Word/PDF documents, spreadsheets, emails

DDGEN Methodology



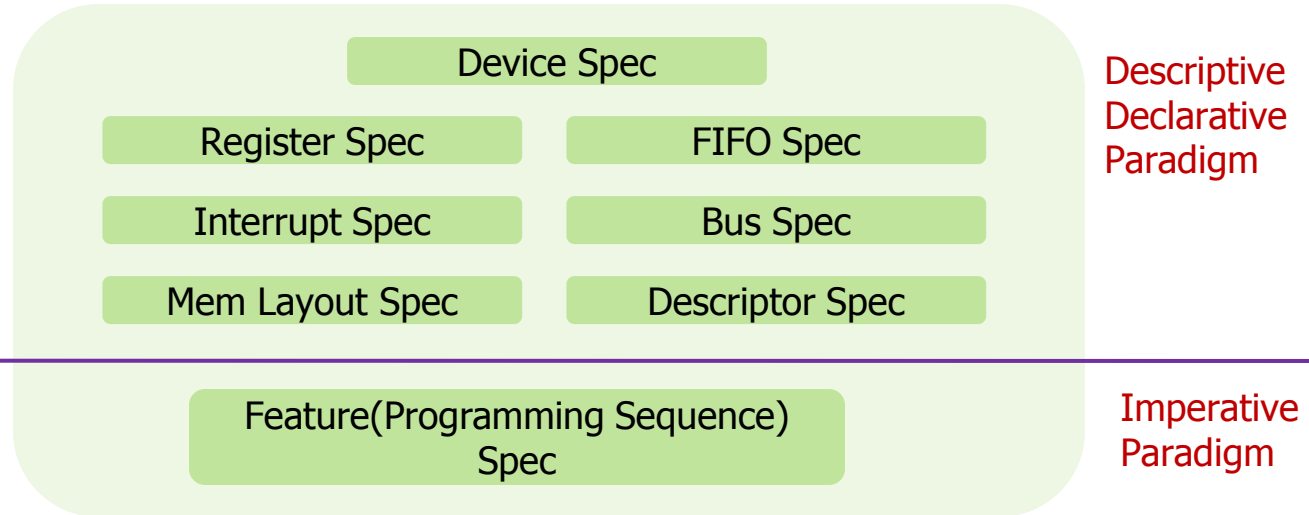
- DDGEN methodology helps formalize communication between hardware and software teams
- Effective separation of the hardware and software concerns
- Think in *problem domain* rather than the *implementation domain*
- DPS is IPXACT compliant
- Generates full-fledged device driver code
 - APIs depending upon driver model
 - ANSI C code including OS calls
 - Support multiple operating systems

DPS: Device Program Sequence

RTS: Run Time Specification

DDGen: Device Driver Generator

DPS Entities



- DPS is a Domain Specific Language
 - External Specification of a Peripheral
 - Focus on Peripheral Usage
- Collection of Several Sections called as Specifications

Register Specification

10.4.52 Transmit Control Register - TCTL (00400h; R/W)

This register controls all transmit functions for the Ethernet controller.

10.4.52.0.1 TCTL Register Bit Description

Field	Bit(s)	Initial Value	Description
Reserved	0	0b	Reserved Write as 0b for future compatibility.
EN	1	0b	Transmit Enable The transmitter is enabled when this bit is set to 1b. Writing 0b to this bit stops transmission after any in progress packets are sent. Data remains in the transmit FIFO until the MAC is re-enabled. Software should combine this operation with reset if the packets in the FIFO should be flushed.
Reserved	2	0b	Reserved Reads as 0b. Should be written to 0b for future compatibility.
PSP	3	1b	Pad Short Packets 0b = Do not pad. 1b = Pad short packets. Padding makes the packet 64 bytes long. This is not the same as the minimum collision distance. If padding of short packet is enabled, the value in TX descriptor length field should be not less than 17 bytes.
CT	11:4	0fh	Collision Threshold This determines the number of attempts at retransmission prior to giving up on the packet (not including the first transmission attempt). While this can be varied, it should be set to a value of 15 in order to comply with the IEEE specification requiring a total of 16 attempts. The Ethernet back-off algorithm is implemented and clamps to the maximum number of slot-times after 10 retries. This field only has meaning while in half-duplex operation.
COLD	21:12	3fh	Collision Distance Specifies the minimum number of byte times that must elapse for proper CSMA/CD operation. Packets are padded with special symbols, not valid data bytes. Hardware checks and pads to this value plus one byte even in full-duplex operation. The default value is 64 bytes = 512-bit time.
SWXOFF	22	0b	Software XOFF Transmission When set to a 1b, the MAC schedules the transmission of an XOFF (PAUSE) frame using the current value of the PAUSE timer. This bit self clears upon transmission of the XOFF frame.
Reserved	23	0b	Reserved
RTLIC	24	0b	Re-transmit on Late Collision When set, enables the MAC to re-transmit on a late collision event.
Reserved	27:25	00h	Reserved

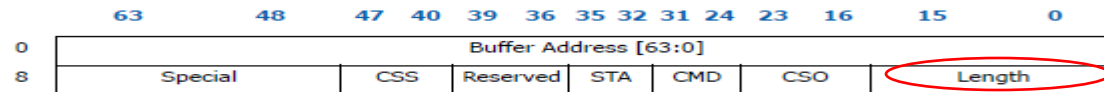
```
TCTL[32] @ 0x400 { //Transmit Control
    SW_AXS      = RW;
    VALUE_ON_RESET = 0x3003F0F8;
    RESERVED <0:0> = 0x0;
    FIELD EN <1:1> { //Transmit Enable
        CLEARING_MODE = DC;
    }
    RESERVED <2:2> = 0x0;
    FIELD PSP <3:3> { //Pad Short Packets
        CLEARING_MODE = DC;
    }
    FIELD CT <4:11> { //Collision Threshold
        CLEARING_MODE = DC;
    }
    FIELD COLD <12:21> { //Collision Distance
        CLEARING_MODE = DC;
    }
    FIELD SWXOFF <22:22> { //Software XOFF Transmission
        CLEARING_MODE = DC;
    }
    RESERVED <23:23> = 0x0;
    FIELD RTLIC <24:24> { //Re-transmit on Late Collision
        CLEARING_MODE = DC;
    }
    RESERVED <25:27> = 0x0;
    RESERVED <28:28> = 0x1;
    FIELD RRTHRESH <29:30> { //Read Request Threshold
        CLEARING_MODE = DC;
    }
    RESERVED <31:31> = 0x0;
}
```

Descriptor/Memory Layout Specification

3.3.3 Legacy Transmit Descriptor Format

To select legacy mode operation, bit 29 (TDESC.DEXT) should be set to 0b. In this case, the descriptor format is defined as shown in Table 15. The address and length must be supplied by software. Bits in the command byte are optional, as are the Checksum Offset (CSO), and Checksum Start (CSS) fields.

Table 14. Transmit Descriptor (TDESC) Layout – Legacy Mode



```

MEMORY_LAYOUT {
  XMIT {
    pBuffer[64] {
      FIELD buff_addr <0:63> {
        TYPE = RW;
        VALUE_ON_RESET = NA;
      }
    }
    tdes0[64] {
      FIELD pkt_length <0:15> { //length of pkt to be transmitted
        TYPE = RW;
        VALUE_ON_RESET = 0x0;
      }
      FIELD CSO <16:23> { //Checksum Offset
        TYPE = RW;
        VALUE_ON_RESET = NA;
      }
      /*Command Fields(8-bits)*/
      FIELD EOP <24:24> { //End Of Packet
        TYPE = RW;
        VALUE_ON_RESET = 0x0;
      }
    }
  }
}

```

```

DESCRIPTOR_SPEC {
  XMIT {
    ITERATOR_TYPE = CIRCULAR_ARRAY;
    DESCRIPTOR_ALLOCATION = EXTERNAL_MEM;
    START_ADDRESS = TDBALO;
    //DEVICE owns the desc
    OWNERSHIP_FIELD = XMIT.tdes0.DD(0);
    BUFFER_PTR = XMIT.pBuffer;
    DESCRIPTOR_DATA_LENGTH = XMIT.tdes0.pkt_length;
  }
}

```

Feature (Programming Sequence) Specification

```
feature pre_transmit {  
  input XMIT txptr;  
  #configure EOP, IFSC, RS, IDE, DD  
    txdesc.tdes0.EOP = 0x1;  
    txdesc.tdes0.IFCS = 0x1;  
    txdesc.tdes0.RS = 0x1;  
    txdesc.tdes0.IDE = 0x1;  
    txdesc.tdes0.DD = 0x0;  
}
```

```
feature post_receive {  
  input RECEIVE rxptr;  
  
  #reset the STATUS byte  
  rxptr.rdes0 = 0x0;  
}
```

- All the device programming sequences are captured as DPS features
- Typically written by studying the programming guide part of the data sheet
- For an ethernet controller, the pre_transmit feature sets all the relevant fields of the transmit descriptor before packet transmission
- post_receive resets the receive descriptor so that it can be reused
- Both pre_transmit and post_receive are keyword features and hence DDGEN automatically maps them to the right place in the generated C Code

RTS Example

- RTS captures all the software considerations – in effect the low-level driver design
 - Operating systems and the driver model
 - Blocking/non-blocking APIs
 - Buffering schemes
 - Synchronization mechanisms
 - Coding conventions

Interconnect	Code Generation	ISR Specification
<pre>BUS_SPEC { REG_ACCESS_TYPE = MEMORY_MAPPED; TRANSFER_MODE = PIO; BASE_ADDRESS = 0xFF8C000 }</pre>	<pre>CODEGEN_SPEC { REGISTER_ACCESS_IMPLEMENTATION = OS_CALLS; REGISTER_ACCESS_GENERATION = ALL; }</pre>	<pre>ISR_SPEC { ISR_NO = 7; ISR_DRIVER_SYNC = SEMAPHORE; ISR_TYPE = SPLIT; }</pre>

Hardware Interface Layer

DPS Features

```
//DPS features in the DPS file
feature tx_complete {input XMIT txdesc...}
feature rx_okay {input unsigned long rx_status...}
feature rx_length_error {
    input unsigned long rx_status...}
feature set_promiscuous_mode {...}
feature set_all_multicast_mode {...}
```

Corresponding C functions

```
//C functions generated by DDGEN
static INT tx_complete(t_xmit *txdesc)
static INT rx_okay(ULONG rx_status)
static INT rx_length_error(ULONG rx_status)
static INT set_promiscuous_mode(void)
static INT set_all_multicast_mode(void)
```

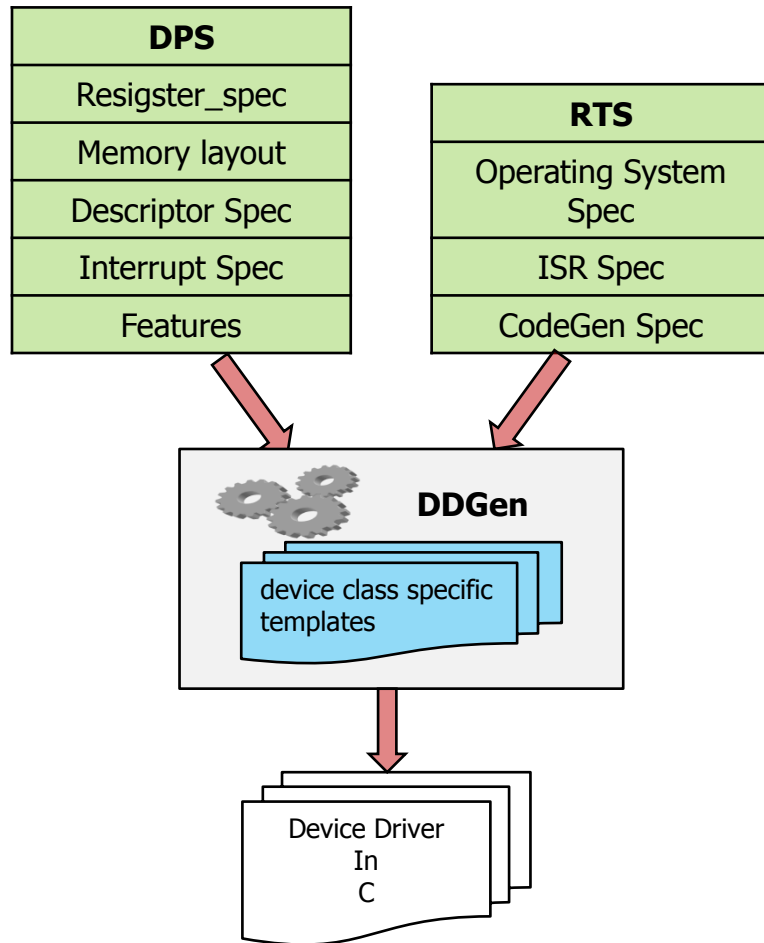
hw_if struct and its initialization

```
struct hw_if_struct {
>INT (* tx_complete)(struct s_xmit *);
    INT (* rx_okay)(ULONG);
    INT (* rx_length_error)(ULONG);
    INT (* set_promiscuous_mode)(void);
    INT (* set_all_multicast_mode)(void);
};
//mapping of C functions/DPS features to hw_if struct
void DWC_ETH_init_function_ptrs(struct hw_if_struct *hw_if)
{
    DBGPR("-->DWC_ETH_init_function_ptrs\n");
>hw_if->tx_complete = tx_complete;
    hw_if->rx_okay = rx_okay;
    hw_if->rx_length_error = rx_length_error;
    hw_if->set_promiscuous_mode = set_promiscuous_mode;
    hw_if->set_all_multicast_mode = set_all_multicast_mode;
    ...
}
```

- Keyword features like device_read/device_write/pre_transmit are automatically mapped by DDGEN in the generated C Code.
- A hardware interface layer – C structure of function pointers is generated for all other *features*.
- Function pointers in this layer need to be manually initialized to the C functions generated from the corresponding DPS feature
- The *hw_if* layer and hence the set of DPS features to be written are very specific to a device-class
 - Requires detailed and thorough understanding of the corresponding application note for that device class

- Background
- Introduction to DDGEN
 - Motivation
 - DDGEN Methodology
- **DDGEN Code Generation**
- Evaluation Results
- Challenges & Future Work
- Q&A

DDGEN Code Generation



- DDGEN code generation involves:
 - Selection of the appropriate code template based on the device-class, Operating System and the driver model.
 - Synthesis of all other parts of the driver C code based on the DPS and RTS
- Internal template is largely imposed by the Operating System chosen in RTS and the corresponding driver model for that OS.
 - 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 device drivers for a specific device-class
- Typically consists of :
 - definition for structures required by the s.w. stack that uses the driver
 - Skeletal/fixed part of code for all the device-class specific APIs that the driver needs to provide
- Remaining portion of these APIs will get generated based on DPS and RTS

Linux Ethernet Controller Driver Template **

DPS

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

RTS

```
OS_spec{
    OS = Linux
    KERNEL_VERSION= 2.6.21
}
BUS_SPEC {
    REG_ACCESS_TYPE =
    PCI_MEMORY_MAPPED;
    TRANSFER_MODE = DMA;
}
```

```
static struct pci_driver I82567_pci_driver =
{
    .name      = "xyz",
    ..probe   = xyz_probe,
    .remove    = xyz_remove
};
static const struct net_device_ops I82567_netdev_ops = {
    .ndo_open      = xyz_open,
    .ndo_stop     = xyz_close,
    .ndo_start_xmit = xyz_start_xmit,
    ....
};
int __devinit xyz_probe (struct pci_dev *pdev,  const struct pci_device_id *id)
{
    struct xyz_prv_data *pdata = NULL;
    struct net_device *dev = NULL;
    INT ret = 0;
    ...
    ret = pci_enable_device (pdev);
    if (pci_request_regions (pdev, DEV_NAME)) {
        printk (KERN_ALERT "%s:Failed to get PCI regions\n",DEV_NAME);
        ret = -ENODEV;
    }
    ...
}
```

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

Code Generation for 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

```
CTRL[32] @ 0x00 {
  SW_AXS          = RW;
  ...
  FIELD FD <0:0> {
    SW_AXS          = RW;
    ...
  }
  ...
}
```

RTS

```
OS_SPEC
  OS = Linux
  .....
}
BUS_SPEC {
  REG_ACCESS =
    PCI_MEMORY_MAPPED;
}
CODEGEN_SPEC {
  REG_ACCESS_IMPLEMENTATION =
    OS_CALLS;
}
```

```
#define CTRL_RgOffAddr ((volatile ULONG *) (BASE_ADDRESS + 0))

#define CTRL_RgWr(data) do {\
    iowrite32(data, (void *) CTRL_RgOffAddr);\
} while(0)

#define CTRL_RgRd(data) do {\
    (data) = ioread32((void *) CTRL_RgOffAddr);\
} while(0)

#define CTRL_FD_Mask (ULONG) (0x1)

#define CTRL_FD_UdfWr(data) do {\
    ULONG v;\
    CTRL_RgRd(v);\
    v = (v & (CTRL_RES_Wr_Mask_29)) | (((0) & (CTRL_Mask_29)) << 29);\
    v = (v & (CTRL_RES_Wr_Mask_21)) | (((0) & (CTRL_Mask_21)) << 21);\
    ...
    v = (v & (CTRL_RES_Wr_Mask_1)) | (((0) & (CTRL_Mask_1)) << 1);\
    v = ((v & CTRL_FD_Wr_Mask) | ((data & CTRL_FD_Mask) << 0));\
    CTRL_RgWr(v);\
} while(0)

#define CTRL_FD_UdfRd(data) do {\
    CTRL_RgRd(data);\
    data = ((data >> 0) & CTRL_FD_Mask);\
} while(0)
```

- As shown in the figure above, read/write macros are generated for register CTRL and its field FD 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
{
  XMIT {
    pBuffer[64] {
      FIELD buff_addr <0:63> {
        TYPE = RW;
        VALUE_ON_RESET = NA;
      }
    }
    tdes0[64] {
      FIELD pkt_length <0:15> { //length of pkt to be
transmitted
        TYPE = RW;
        VALUE_ON_RESET = 0x0;
      }
      FIELD CSO <16:23> { //Checksum Offset
        TYPE = RW;
        VALUE_ON_RESET = NA;
      }
      ...
      /*Status Fields(4-bits)*/
      FIELD DD <32:32> { //Descriptor Done or Ownership
        TYPE = RW;
        VALUE_ON_RESET = NA;
      }
    }
    ...
  }
  DESCRIPTOR_SPEC {
    XMIT {
      ITERATOR_TYPE = CIRCULAR_ARRAY;
      DESCRIPTOR_ALLOCATION = EXTERNAL_MEM;
      START_ADDRESS = TDBAL0;
      OWNERSHIP_FIELD = XMIT.tdes0.DD(0);
      BUFFER_PTR = XMIT.pBuffer;
      DATA_LENGTH =
XMIT.tdes0.pkt_length;
    }
    ...
  }
  BUFFER_SPEC {
    XMIT {
      BUFFER_ALLOCATION = EXTERNAL_MEM;
      BUFFER_SIZE = 1536;
    }
    ...
  }
}

```

```

/* Memory Layout Mask Macro's */
#define XMIT_PBUFFER_BUFF_ADDR 0ull
#define XMIT_TDES0_PKT_LENGTH 0xffff0000

/* Memory Layout Register-Field Read-Write Macros */
#define XMIT_PBUFFER_BUFF_ADDR_Mif_Rd(ptr, data) do { \
data = (ptr & ~XMIT_PBUFFER_BUFF_ADDR) >> XMIT_PBUFFER_BUFF_ADDR_LBIT_POS; \
} while(0)

#define XMIT_PBUFFER_BUFF_ADDR_Mif_Wr(ptr, data) do { \
ptr &= XMIT_PBUFFER_BUFF_ADDR; \
ptr |= (data << XMIT_PBUFFER_BUFF_ADDR_LBIT_POS); \
} while(0)

//C structure corresponding to the xmitmemory layout specified in the DPS
struct s_xmit {
  ULONG_LONG pBuffer;
  ULONG_LONG tdes0;
};

//Prototypes of Descriptor management APIs
INT allocate_buffer_and_desc (struct I82567_prv_data *);
static void tx_descriptor_init (struct I82567_prv_data *);
static void rx_descriptor_init (struct I82567_prv_data *);

```

- 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 SEC
- The Driver model APIs that are generated by DDGEN use the relevant functions for descriptor handling

ISR Generation - Example

```

INTERRUPT_SPEC {
    SUPPORTED_INT_TYPE = LEGACY, MSI;
    INTEL {
        //Tx Descriptor Write Back interrupt
        ICR.TXDW(0x1) {
            INT_TYPE = DEVICE_WRITE;
            ENABLE_FIELD = IMS.TXDW(0x1);
            DISABLE_FIELD = IMC.TXDW(0x1);
            CLEAR_FIELD = AUTO_CLEAR;
        }
        //Tx Queue Empty interrupt
        ICR.TXQE(0x1) {
            INT_TYPE = STATUS;
            ENABLE_FIELD = IMS.TXQE(0x1);
            DISABLE_FIELD = IMC.TXQE(0x1);
            CLEAR_FIELD = AUTO_CLEAR;
        }
        //Rx Descriptor Min Threshold Hit interrupt
        ICR.RXDMTO(0x1) {
            INT_TYPE = STATUS;
            ENABLE_FIELD = IMS.RXDMTO(0x1);
            DISABLE_FIELD = IMC.RXDMTO(0x1);
            CLEAR_FIELD = AUTO_CLEAR;
        }
        //Rx Timer interrupt
        ICR.RXTO(0x1) {
            INT_TYPE = DEVICE_READ;
            ENABLE_FIELD = IMS.RXTO(0x1);
            DISABLE_FIELD = IMC.RXTO(0x1);
            CLEAR_FIELD = AUTO_CLEAR;
        }
    }
}
    
```

```

irqreturn_t xyz_ISR_SW_INTEL(int irq, void * device_id)
{
    ULONG varICR ;
    ULONG status ;
    struct I82567_prv_data *db = (struct xyz_prv_data *)device_id;
    struct net_device *dev = db->dev;

    ICR_RgRd(varICR);

    status = ((ULONG)varICR << 0);

    if((status & MASK) == 0x0) {
        return IRQ_NONE;
    }

    if (GET_VALUE(varICR, ICR_TXDW_LPOS, ICR_TXDW_HPOS) & 1) {
        xyz_tx_interrupt(dev, db);
    }
    if (GET_VALUE(varICR, ICR_TXQE_LPOS, ICR_TXQE_HPOS) & 1) {
        GStatus = S_ICR_TXQE ;
    }
    if (GET_VALUE(varICR, ICR_RXDMTO_LPOS, ICR_RXDMTO_HPOS) & 1) {
        GStatus = S_ICR_RXDMTO ;
    }
    if (GET_VALUE(varICR, ICR_RXTO_LPOS, ICR_RXTO_HPOS) & 1) {
        if(napi_schedule_prep(&db->napi)) {
            xyz_disable_interrupt(exyz_ICR_RXTO);
            napi_schedule (&db->napi);
        }
    }
    return IRQ_HANDLED;
}
    
```

- **ICR.TXDW** is an interrupt identifier for an interrupt of type *device_write* as specified in the INTERRUPT SPEC. The corresponding transmit interrupt handler **xyz_tx_interrupt** is invoked in the ISR
- Similarly **ICR.RXTO** corresponds to the *device_read* interrupt and hence receive is scheduled in the generated ISR.

Mapping of DPS features to function callbacks

- Keyword features like `device_read`, `device_write`, `init`, `finit`, `pre_transmit` are directly mapped by DDGEN to the appropriate place in the template based on the device class.
- C functions generated for other DPS features need to be manually mapped in the generated C code.
- The templates consists of "placeholder" function pointers. These pointers need to be initialized to the appropriate functions

```
FEATURE tx_complete { INPUT XMIT txdesc;
#check the DD(ownership) bit
(txdesc.tdes0.DD == 0x1)?return = 1: return = 0;
}
```

```
struct hw_if_struct {
    INT (* tx_complete)(struct s_xmit *);
    ...
}
static INT tx_complete(t_xmit *txdesc)
{
/*check the DD(ownership) bit*/
XMIT_TDES0_DD_Mlf_Rd( txdesc->tdes0, varDD );
if( varDD == 0x1 ) return 1 else return 0;
}
void xyz_init_function_ptrs(struct hw_if_struct *hw_if) {
    hw-if->tx_complete = tx_complete
    ...
}
static void xyz_tx_interrupt(struct net_device *dev,
                            struct xyz_prv_data *pdata)
{
    struct s_xmit *txptr = NULL;
    ....
    txptr = pdata->tx_desc_ptr[index];
    ...
    while (tx_left > 0) {
        index = dirty_tx % TX_DESC_CNT;
        txptr = pdata->tx_desc_ptr[index];
        if (!hw_if->tx_complete (txptr)) {
            break;
        }
    }
    ...
}
```

- The status of a transmitted descriptor needs to be obtained in the transmit interrupt handler – `xyz_tx_interrupt`
- The ownership bit `tdes0.DD` contains the status and DPS feature `tx_complete` returns this status
- The driver template for `xyz_tx_interrupt` is designed with a callback `tx_complete` to obtain the descriptor status
- All such function pointers are organized in a structure – `hw_if_struct`
- The DPS feature `tx_complete` 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 `tx_complete` is initialized in `xyz_init_function_ptrs`. All function pointers need to be similarly initialized over here

- Background
- Introduction to DDGEN
 - Motivation
 - DDGEN Methodology
- DDGEN Code Generation
- **Evaluation Results**
- Challenges & Future Work
- Q&A

DDGEN Evaluation Results

DMA Controller, Interrupt Controller, Event handler,
clock distribution unit

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

SD Controller with Internal DMA

Activity	Effort in person days
Writing DPS/RTS	6
Integration and testing generated driver in the target environment	22
Total Effort using DDGEN	29
Total Effort for manual driver generation	88

Ethernet Controller

Activity	Effort in person days
Writing DPS/RTS	8
Integration and First Version driver code	1.5
Total Effort using DDGEN for complete test plan	19.5
Total Effort for manual driver generation	Not available

- Almost 300% productivity improvement
- Porting to a different Operating system involves modifying the appropriate RTS field and re-generating the driver

- Background
- Introduction to DDGEN
 - Motivation
 - DDGEN Methodology
- DDGEN Code Generation
- Evaluation Results
- **Challenges & Future Work**
- Q&A



- Perceptions

- I don't think its possible
- I don't think device driver can be automatically generated!
- Who is using it?
- You got to be kidding

- Customer Adoption Challenges

- Paradigm Shift
- How will the tool fit in my existing flow/methodology?
- Lots of existing legacy and robustized code. Can't throw it away
- How quickly can a new OS be supported?
- DPS is proprietary

DPS Challenges

DPS canonization

- Current form of DPS is non-canonical
- Overlap between Register Specification and Memory Layout
- Overlap between FIFO_SPEC and ENDPOINT SPEC for USB devices

DPS Enhancements

- Standardization via IPXACT
- Support for invariants/constraints
- Making Feature (Programming Sequence) Specification non-procedural

Hardware Interface Layer

- hw_if for some device classes is very big and can become potentially unmanagable
- Often Operating System view gets imposed on DPS

Debug

- Debugging in C and mapping back to DPS can be non-trivial
- Need to define debugging at specification (DPS) Level

Support for IP Features and Configurability

Data Transfer Type

Slave Mode

Buffered DMA

Internal DMA

Descriptor Iterator

Circular

Linked

Dual Buffer

MAC-PHY Interface Type

UTMI

ULPI

IP Features and Configurability

Internal DMA

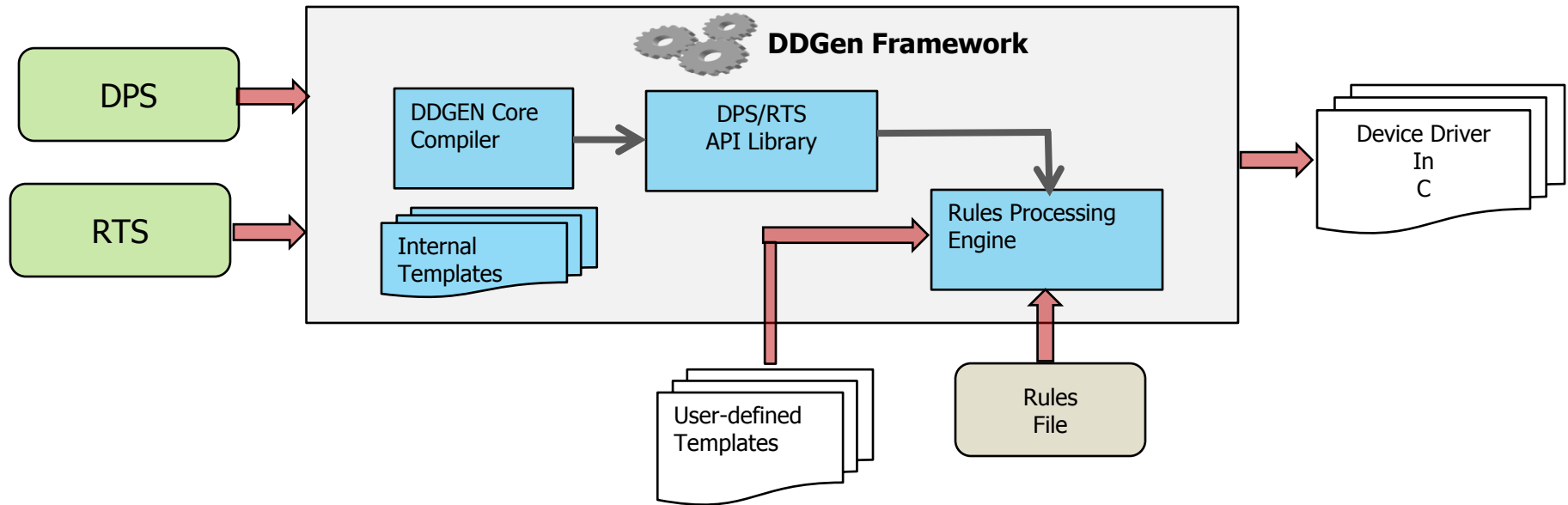
Linked

UTMI

Specific Use Case of the IP

- Typical IP from an IP vendor consists of several features and is highly configurable
- Usage of the IP in an SOC involves only few of the features and configuration
- Reference device driver from the IP Vendor is a superset
 - Supports all the features and configurations
- Current use model of DDGEN cannot generate a super-set driver
 - Typical DPS is written only for a specific use-case and DDGEN generates the device driver accordingly
- DDGEN needs to support generation of super-set device drivers

From Tool to Synthesis framework



- DDGen Core Compiler:

- Generates Register Access Routines
- ISR and translates Features(Programming Sequences) to C functions
- Exposes the DPS and RTS data model through an API library

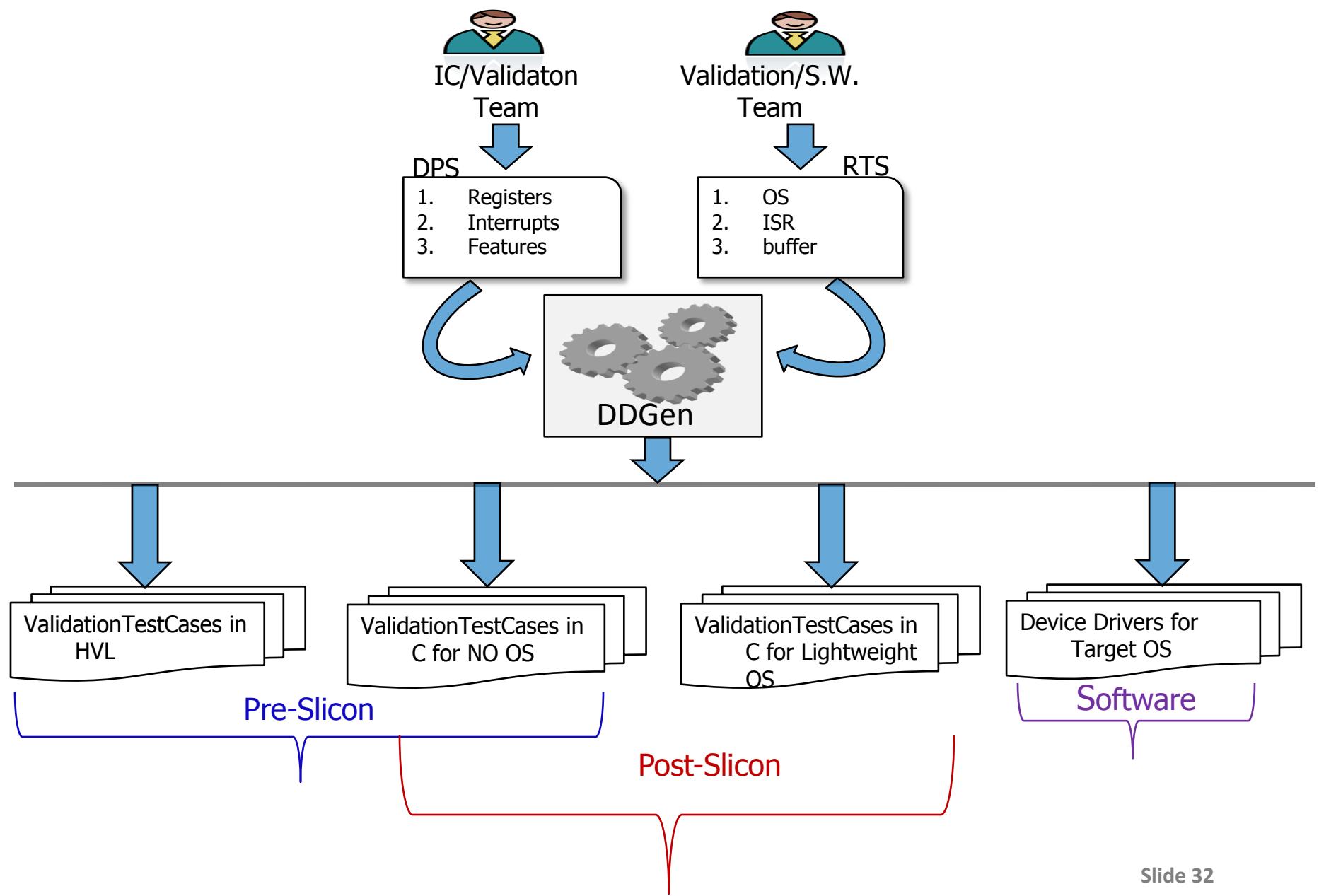
- Device driver generation through DDGen internal templates or user supplied templates

- Infrastructure to import user-defined templates

- For User-defined Templates

- User specifies the rules for orchestration of template blocks via a rules file which is processed by the Rules Processing Engine
- A Typical rule specified in terms of DPS and RTS fields
- Actions for each rules can be specified using the DPS/RTS API library

Unified Pre/Post-Silicon Validation using DDGEN



- Background
- Introduction to DDGEN
 - Motivation
 - DDGEN Methodology
- DDGEN Code Generation
- Evaluation Results
- Challenges & Future Work
- **Q&A**

Merci..