DDGEN: An Automated Device Driver Generation Tool for Embedded Systems

Sandeep Pendharkar, Venugopal Kolathur
Vayavya Labs, Belgaum, India

Abstract

This paper describes a methodology for automatically generating device drivers for embedded systems.
We formally specify the device behavior and attributes in an input specification called DPS (Device programming sequence). Software architecture considerations are similarly captured in another specification called RTS (Run time specification). Our tool, DDGEN takes both these specifications as input and generates a full-fledged device driver code for the target operating system. In this paper, we discuss DPS, RTS and the DDGEN synthesis engine. We also argue about the productivity benefits resulting due to the increased level of abstraction and effective separation of the hardware and the software concerns.

1. Introduction

A typical embedded system project is characterized by very strict and demanding schedule requirements. System designers have to regularly re-design both the hardware as well as the software for newer versions of the product. It is a well-acknowledged fact that in modern embedded systems, software development takes more time than the hardware/IC design.

On the software side, device driver development is often the bottleneck. It is inherently error-prone and complex due to the need for thorough knowledge about the innumerable peripherals that exists in a typical embedded system. Writing of high quality, optimized device drivers is difficult and time consuming. For example, the DM355 Digital Multimedia SOC from TI has around 18 peripherals and the Board Support Package[8] is approximately 50,000 lines of code.
.
This problem is further exacerbated by the fact that the communication between the hardware and the software teams is mostly informal in the form of data sheets, spreadsheets or emails.

The embedded system space is also characterized by a plethora of operating systems like Linux, VxWorks, Windows CE, Windows Mobile, Symbian, Micrium, QNX to name a few. Thus device drivers often need to be written for multiple operating systems. The set of interfaces or APIs (typically known as the driver model) that the driver exposes to the upper levels of the embedded software also varies depending upon the device. Often, multiple driver models exist for a given device class. For example Linux has four different kinds of driver models for video devices viz. *frame buffer*, *direct frame buffer*, *Video-for-Linux* and *OpenGL*.

To address the above mentioned complexities, we propose a domain specific language (DSL) called DPS(Device Programming Sequence) to capture the device attributes and behavior. Similarly, the various software architecture and run-time considerations are captured in another specification called RTS(Runtime Specification). A corresponding tool, DDGEN compiles this DPS specification and generates the device driver after analyzing the RTS input.

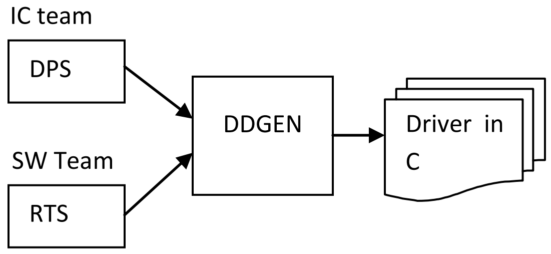

Fig (1)

DPS formalizes the communication between the hardware and software team. It essentially captures all the relevant aspects (registers, interrupts, fifos, programming sequence etc.) of a given device. As shown in fig (1), we propose that DPS be written by the hardware(IC) team. The RTS should be designed by the actual device driver developer. The RTS captures the operating system, the driver model and other software aspects like synchronization, software buffering, interrupt handling mechanism etc. Our methodology effectively separates the hardware and the software concerns. By raising the abstraction level, it enables thinking in the "problem domain" than the "implementation domain".

We discuss related work and approaches in the next section. Section 3 elaborates on the DDGEN framework. Our results are captured in Section 4. We conclude and discuss future work in Section 5.

2. Related Work

A variety of approaches have been proposed in the past to address some of the complexities mentioned. Devil[5] primarily describes an abstraction for capturing register access including bit-level operations and generates a library of register access functions. HAIL[1] is similar to Devil but introduces an invariant specification and thus enables generation of debug code to catch various bugs. HAIL also addresses the problem of handling various bus structures. Thibault et al[6] introduce GAL – a domain specific language for video drivers. Note that the underlying concept and approach of designing a domain specific language is valuable. But GAL is essentially restricted to the video domain only.

Wang et al [4] capture device specification in terms of register programming, state machines for transition between device states and core functions that get synthesized in C for a virtual environment. This virtual environment is manually mapped to a particular target platform. Note that in our approach DPS allows capturing of the device specification and transition between device states as well. Our approach directly synthesizes C code for the target platform and environment. While virtual environment enables reuse, we believe it is not be scalable for the innumerable number of operating systems in the embedded systems domain.

Several commercial tools like Bitwise[3] exists as well to read register specification in some form and generate the low level register access routines in C but they do not generate a complete device driver for a given operating system.

Note that RTS in our approach allows capturing of the software architecture and decisions to a very fine level of granularity and this is noticeably absent in all of the previous approaches. Except for [5] in some manner, none of the previous attempts effectively separate the hardware (device) concerns from the software concerns.

3. DDGEN Framework

In this section, we describe the input specifications viz. DPS and RTS using a UART device as an illustrative example. Note that some specific aspects of DPS are explained using a USB device controller as a reference. We then describe the driver generation process by the synthesis engine of DDGEN.

3.1 DPS

DPS is composed of several sections known as specifications. It is a formal mechanism to capture a typical device data sheet. As specified earlier, we propose that the IC team creates the DPS for a given device. Some of the specifications common across all the device classes are:

- `device_spec`
- `register_spec`
- `interrupt_spec`
- `fifo_spec`
- `interrupt_spec`
- `features`

Note that in some cases, DPS also consists of specifications which are very specific to a particular device class.

`device_spec:`

```
DEVICE_SPEC
{
  device_name = AT91SAM9263_USART;
  device_class = SERIAL;
}
```

Fig (2)

The `device_spec` captures the name of the device and the class of the device. DPS has certain predefined classes like `SERIAL`, `BUS_CONTROLLER`, `NETWORK`, `AUDIO` etc.

DDGEN infers the target driver model from `device_class` and the corresponding attributes like `operating system` and `driver_model` in the RTS.

`Register_spec`:

```
CONTROL_REGISTERS {
  LCR [8] @ 1 {
   type = RW;
   field wls <0:1> {
     type = RW;
     clearing_mode = DC;
     value_on_reset = 0;
   }
  . . .
}
```

Fig (3)

The `register_spec` captures all the registers of the device . For each register, it also captures the size of the register, the offset at which it is placed from the base address and its read/write type as well. DPS also allows logical grouping of bits within a register into sub-fields. All the above mentioned attributes can potentially be specified for each such field as well. The value of most of the attributes in other specifications is typically specified in terms of these field names. DDGEN automatically generates read/write access routines (macros in C) for all these registers and their fields. The appropriate bit masks required for these read/write routines are generated as well.

`Features`:

```
FEATURE device_read {
  OUTPUT char data;
  poll LSR.dr until
 (LSR.dr == 1) data = RBR
  ERROR (LSR.oe == 0x1 ||
         LSR.pe == 0x1 ||
         LSR.fe == 0x1 ||
         LSR.bi == 0x1 );
}
```

Fig (4)

`Feature` Specification is the only non-structural specification in DPS and is meant to capture the programming sequences for the device. It allows declaring of local variables, arguments to the feature and return value as well.
Arithmetic operations and conditional execution is also allowed. These operations can be directly specified on register fields. `Features` also contain certain constructs like `poll` and `wait` which enable effective capture of a sequence. Fig (4) shows a feature named `device_read` which reads data from the `RBR` register only after the corresponding status field `dr` indicates its availability. `device_read` and `device_write` are keywords in DPS thus conveying specific information to DDGEN. These features correspond to reading and writing data from a device. Similarly `init` is a keyword feature as well and corresponds to configurations and sequences required for initialization of the device. Apart from these features having a special meaning, DPS also enables writing of any *explicit* features.

DDGEN directly translates these features to C functions. The synthesis process is explained in details in section 3.3.

`Interrupt_spec`:

```
INTERRUPT_SPEC {
  IIR.IntID(1){
   int_type =  device_read;
   enable_field = IER.erbfi(0x1);
   disable_field= IER.erbfi(0x0);
   clear_field =  AUTO_CLEAR ;
   }
   …
 }
```

Fig (5)

The `interrupt_spec` captures the interrupts supported by the device. Fig(5) above shows the read/receive interrupt for a UART. Each interrupt is identified by a register field and the corresponding value in that field indicating the occurrence of the interrupt. Thus ‘1’ in the register field `IIR.IntID` indicates that a read interrupt has occurred. Since `device_read` is a pre-defined keyword, DDGEN automatically infers that this read interrupt has to be handled by the corresponding `device_read` feature. This particular interrupt is enabled by writing '1' to the register field `IER.erbfi`. The register fields for disabling and clearing this interrupt are specified as well. DDGEN automatically generates routines for enabling/disabling and clearing each of the interrupts specified in `interrupt_ spec`. Note that DPS also allows an *explicit feature* to be specified for each interrupt as well. DDGEN automatically inserts a call to the corresponding C function in the Interrupt Service Routine(ISR) that is generated.

Device-class specific specification

As mentioned earlier, apart from the generic specifications mentioned above, DPS also consists of a few specifications which are very specific to a particular device. Given below is one such example for a USB device controller.

```
USB_DEVICE_SPEC {
   …
  end_point 0 @ UDP_FDR0 {
  supported_transfer_type =
   control(UDP_CSR0.EPTYPE('b00)),
   bulk(UDP_CSR0.EPTYPE('b10),
   interrupt(UDP_CSR0.EPTYPE('b11));
  direction =
   IN(UDP_CSR0.DIR(0)),
   OUT(UDP_CSR0.DIR(1));
  enable =
   UDP_CSR0.EPEDS('b1);
  disable =
   UDP_CSR0.EPEDS('b0);
  … } }
```

Fig(6)

The `usb_device_spec` essentially captures the relevant attributes for all the endpoints of a USB device. Fig (6) above shows some of the configuration details for the endpoint 0. Note that UDP_FDR0 is the data register for this endpoint. Each endpoint often supports multiple types of data transfers (control, bulk, isochronous, interrupt). This is captured by the `supported_transfer_type` attribute. The register fields for configuring the endpoint direction as well as enabling/disabling of the endpoint are specified as well. Thus setting the register field `UDP_CSR0.DIR` to 0 indicates that the host is ready to receive data from the device controller. Similarly, setting `CSR0.EPEDS` to `1` automatically enables the endpoint `0`.

Note that capturing the endpoint details in the above manner enables DDGEN to automatically synthesize the relevant USB device driver model across various operating systems primarily because each keyword/attribute has a specific intent associated with it. In case of device-classes which do not have such a custom specification, the relevant details are specified as explicit features in DPS. The corresponding C functions that get generated need to be manually integrated with the driver model synthesized by DDGEN.

3.2 RTS

As specified earlier, RTS (Runtime Specification) enables the software team to focus purely on the software considerations. At a broad level, RTS captures the following aspects:

- Operating system and corresponding driver model to be generated. For example Windows CE allows "monolithic" or "layered" driver architecture[9]. Similarly all controller devices require a platform driver model[10] on Linux 2.6.11 onwards.
- Generation of asynchronous or blocking/non-blocking APIs.
- Buffering model required in the software
- Coding styles and conventions being followed
- Support for re-entrancy and synchronization mechanism between the main driver thread and the interrupt handler.

We describe some of the RTS specifications in further details:

Interconnect:

```
BUS_SPEC {
 REG_ACCESS_TYPE = MEMORY_MAPPED;
 TRANSFER_MODE = PIO;
 BASE_ADDRESS = 0xFF8C000
}
```

Fig(7)

The bus_spec defines the interconnect for the device. Fig (7) indicates that the device registers are memory mapped. REG_ACCESS_TYPE indicates whether the registers are memory mapped or accessed over an IO port. If the registers are accessed over a bus like I2C or SPI, RTS allows capturing of that information as well. The bus_spec also captures the mode of data transfer – either PIO or DMA. The base address of the device is captured as well.

Interrupt Specification :

```
ISR_SPEC {
 ISR_NO = 7;
 ISR_DRIVER_SYNC = SEMAPHORE;
 ISR_TYPE = SPLIT;
}
```

Fig(8)

This specification defines the manner in which the Interrupt Service Routine needs to be implemented. ISR_NO specifies the interrupt number used in registering the ISR with the kernel. ISR_DRIVER_SYNC specifies the synchronization mechanism between the main driver thread and the ISR. This is typically achieved using semaphores. Note that global variables are typically used for synchronization only if the underlying host processor(on which the device driver is executed) does not have any operating system on it.

3.3 Driver Synthesis

We briefly outline the DDGEN driver generation process below:

- DDGEN automatically identifies the driver registration mechanism and the synchronization APIs to be used based on the target operating system.
- DDGEN internally maintains a library of templates. These templates are designed based on various aspects like the operating system, the device class, the driver model, and the API behavior (asynchronous, blocking/non-blocking etc.).
- Register access routines are generated based on the attributes specified in the register_spec in DPS and the reg_access_type specified in the RTS
.
- DDGEN translates each feature specified in DPS to a corresponding function in C. The pre-defined features are like device_read, device_write, init, finit are directly mapped to the appropriate API in the chosen driver model. The corresponding template in the internal library is appropriately tailored during code generation. For the DPS and RTS mentioned earlier, the device_read feature is directly mapped to the *read* API of the *character driver* model[7] in Linux. All *explicit features* are translated to C functions
- DDGEN generates the ISR routine based on the interrupt_spec in DPS and ISR_spec in the RTS. For the interrupt_spec given in figure(5), the tool automatically infers that the interrupt has to be serviced by the corresponding read API. The ISR is then generated accordingly. The synchronization code between the ISR and main driver thread is generated based on the ISR_DRIVER_SYNC field in the ISR spec of RTS.

3.4 Software Space Exploration using RTS

Conventional method of device driver writing is often iterative. The programmer takes certain design decisions and implements the driver – typically in *C*. Once the working code is obtained, the driver undergoes several refinements to meet various criteria. For example, an audio driver initially implemented using a *ping-pong* buffer might have to be modified to use a *circular linked-list*. Similarly, a *read* API initially implemented as a blocking call might need to be modified to a non-blocking call. Each such change involves direct modification to the *C* code itself.

We term the above process as *Software Space Exploration.* This exploration can be very effectively and efficiently done using RTS. The RTS is analogous to a low-level design document. A change in a design decision essentially means changing the value of a relevant attribute in the RTS. The driver can then be re-generated using DDGEN.

4. Results

The table below shows the results of DDGEN evaluation at one of our customer sites. DDGEN was used for generating drivers for various peripherals like DMA controller, Interrupt controller, Event handler, Clock distribution unit:

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

Thus DDGEN methodology for generating drivers resulted in almost 300% productivity improvement. Note that porting the same drivers to a different operating system involves making the relevant changes in RTS and using DDGEN to generate the drivers for that particular target operating system.

5. Conclusion and Future Work

Traditionally, Device driver writing is highly error-prone, cumbersome and time-consuming due to a variety of reasons like very strict schedule, informal communication between the hardware and the software teams and a plethora of operating systems. DDGEN methodology formally captures the device specifications in DPS and the software architecture in RTS. The tool automatically generates complete device driver code in C thus improving productivity and alleviating the problems discussed earlier.

Our future work involves designing of suitable optimization knobs in RTS such that the DDGEN synthesis engine can be influenced to generate code which is optimized for performance and/or size. DPS is a proprietary DSL thus inhibiting its wide-spread adoption. We're working on adopting IPXACT[2] as the device specification mechanism. IPXACT currently allows capture of the register information only. Our work involves designing of suitable vendor extensions for mapping all other DPS specifications in IPXACT. The current library of code generation templates has been designed based on our vast domain expertise and experience. In future we would like to evolve a methodology for any customer specific template to be imported in the tool as well.

8. Acknowledgements

We would like to acknowledge the contribution of all the DDGEN product team members in various activities, right from conceptualization to implementation, testing, benchmarking and several enhancements that they continue to incorporate.

9. References

[1]J. Sun, W. Yuan et al. HAIL: A Language for Easy and Correct Device Access

[2]Spirit Consortium IPXACT 1.4 specification. Website: http://www.spiritconsortium.org

[3]Bitwise Register Management: http://www.duolog.com

[4]S. Wang, S. Malik, R. Bergamaschi: Modeling and Integration of Peripheral Devices in Embedded Systems

[5]F. Merillon, Laurent Reveillere: Devil: An IDL for Hardware Programming

[6]S. Thibault, R Marlet, C. Consel: A Domain Specific Language for Video Device Drivers

[7]Jonathan Corbet et al: Linux Device Drivers.

[8] The BSP was created as a part of an internal porting project.

[9] http://msdn.microsoft.com/en-us/library/aa923899.aspx

[10] Linux Platform Drivers: http://lxr.linux.no/linux+v2.6.29/Documentation/driver-model/platform.txt