

## Case Study on Debugging of DDGEN Generated Device Driver Code

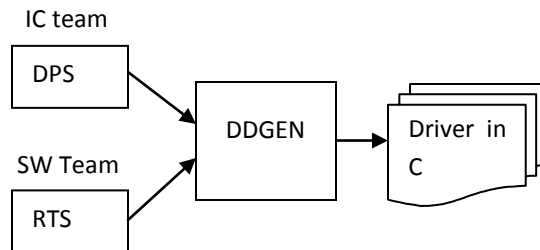
Someshwar D, Pavitra M, Sandeep P.  
{someshward, pavitra, sandeep}@vayavyalabs.com  
Vayavya Labs, Belgaum India

### Abstract

This paper describes our experience in debugging of DDGEN generated device driver code. DDGEN generates device driver code in C from a high-level device specification (DPS) and the corresponding software architecture specification (RTS) as input. Unlike the usual development methodologies where debugging is typically performed at the same level as the input specification<sup>1</sup>, in DDGEN based methodology, we debug the generated C code and not the DPS or RTS. Fixing of the bug mostly involves modifying the input specification (DPS or RTS). Our case study consisting of specific examples of debugging as well as the debugging guidelines presented in this paper will ensure that a DDGEN user can efficiently as well as effectively debug generated C code and further map the fix back to DPS or RTS.

### 1. Introduction

Device driver development is error-prone, complex, time-consuming and often a bottleneck in embedded system projects due to various reasons - strict and aggressive schedules, plethora of operating systems and driver models to name a few. DDGEN relieves the device driver writer of this tedium by automatically generating the device driver.



As shown in the above figure, DDGEN requires two input specifications viz. DPS and RTS. DPS (Device Programming Specification [2]) captures all relevant aspects (registers, interrupts, fifos, programming sequence etc.) of a given device. Thus DPS is a formal specification language. RTS (Run Time Specification [2]) captures software considerations like the target operating system, driver model, interrupt handling mechanism etc. DDGEN compiles both the input specifications and automatically synthesizes the device drivers. Refer [1] for a complete discussion on DDGEN.

While the higher level of abstraction enables a user to think in "problem domain" and thus achieve very high productivity, it also presents interesting challenges insofar as debugging of the generated C code is concerned. Note that defining an effective debug architecture and methodology at DPS level is non-trivial and is currently not in our scope.

Our experience of using DDGEN at various customer evaluations indicate that debugging can be performed at the level of the generated C code and the corresponding fix can be effectively mapped to the input specifications - DPS or RTS.

Section 2 describes our debug experience in details for one such customer evaluation. Based on our experience, we present some high-level guidelines in Section 3. We conclude in section 4.

---

<sup>1</sup> In several cases, people do debug assembly output though the input specification is a language like C.

## 2. Debugging of DDGEN generated driver Code in C

As discussed earlier, we present our experience in debugging of DDGEN generated C code during the tool evaluation at one particular customer site. We first describe the test environment and the high level debug methodology. Our debugging experience is described as a collection of *issues*. For each of the issues encountered during testing of the generated code in the target environment, we first present the symptom that was initially observed. We then present a detailed analysis, the root cause as well as the actual fix for that particular issue.

### Test Environment:

The customer used DDGEN to generate a device driver for a MMC controller. The target Operating System was Linux kernel version 2.6.11. Note that the MMC controller was on an FPGA board which in turn was connected to the host PC (execution environment) over a PCI interface. The generated driver was tested for PIO (Programmed IO) as well as DMA mode of operation.

### High Level Debug Methodology:

Given below is the high-level debug methodology that was followed for debugging the generated C code:

- Instrument the driver code with "printk" statements at the entry and exit of each driver API and assert statements at the start of a function to check for consistency of input arguments. Note that the user does not have to perform this instrumentation manually. DDGEN automatically instruments driver APIs with a macro called "dbgpr". The assert statements are also generated automatically on specifying the "defensive programming" option in the RTS. Note that the user can use dbgpr in any part of the generated code as required.
- Analyze the log file thus generated to trace the driver execution flow.
- Further fine-grained instrumentation in leaf level functions by using dbgpr to isolate the problem. DDGEN does not perform this instrumentation automatically but the user is expected to do it manually.
- Dumping of register values at appropriate places to debug the programming sequences. DDGen generates a macro called "dump\_regs" to dump all the register values and also instruments the generated driver code with this macro at a few places like Interrupt handler, read, write APIs etc. The user can further instrument the code manually by using dump\_regs in any part of the generated code as required.
- Usage of additional tools like lspci, setpci to debug symptoms that were first observed in the PCI subsystem.

### Issue 1:

#### Original Symptom:

After loading the driver module, the host System became unresponsive with the display terminal scrolling uncontrollably.

#### Detailed Analysis:

As specified earlier, the driver code was instrumented with printk statements - including the entry point probe function as well as the ISR. Analysis of the generated log file showed that the ISR was being invoked as soon as the driver module was loaded, even before the driver got registered with the mmc core of the Linux kernel. i.e. the ISR was getting executed before the probe function. In a functionally correct driver, the ISR should be invoked only after the driver initialization and registration has happened in the probe<sup>2</sup> function and the interrupts have been enabled. We concluded that a spurious interrupt logged in the device was causing out of step execution of the ISR.

#### Root Cause:

---

<sup>2</sup> probe is the initialization function for all platform drivers on Linux.

Spurious interrupts logged in the device even before the driver becomes functional can result in a totally unpredictable behaviour of the driver code. For example, the MMC controller has an interrupt that gets generated on successful execution of an MMC command by the controller. Servicing of this interrupt involves invoking an mmc core callback named `mmc_request_done`. Such an interrupt, when previously logged results in invocation of this callback even without a transfer request in the first place.

**Fix:**

The fix involved clearing of all the previously logged interrupts as soon as the driver module was loaded even before the driver gets registered with the mmc core. This was achieved by adding the programming sequence to clear the interrupts in the `INIT` feature of `DPS`. `INIT` is a keyword feature in `DPS`. All the programming sequences corresponding to device initialization are specified in `INIT`. `DDGEN` synthesizes `INIT` to a corresponding C function. `DDGEN` also ensures that this C function is called in the `probe` API of the generated driver code. Note that within the `probe` API, this `INIT` function is executed before the driver is registered with the mmc core.

Given below is a snippet of the `DPS` `INIT` sequence

```
INIT {
// device initializations
# Code added: Clearing any logged interrupts
RINTSTS = 0xffffffff;

# enable interrupts
CTRL.enable_int = 0x1;

// some more device initializations
}
```

**Issue 2:**

**Original Symptom:**

After loading the driver module, the host System became unresponsive with the display terminal scrolling uncontrollably.

**Detailed Analysis:**

During debugging, we observed that the card detect interrupt was logged in the MMC controller when the MMC card was inserted in the card slot for testing before the driver module was loaded on the host system. The corresponding action for this interrupt involves a callback function from the mmc core - `mmc_detect_change`. This callback failed because it occurred even before the driver had been registered with the mmc core. Further analysis revealed that the device interrupts had been enabled even before the driver was registered with the mmc core.

**Root Cause:**

Initially, the interrupt enabling was being done in the `INIT` feature of `DPS`. As discussed earlier, `DDGEN` synthesizes this feature to a C function and inserts a call to this function in the `probe` API of the driver as shown below:

```
INT __devinit dwc_probe(struct device *pdev_ptr)
{
//some code

irq_rval = request_irq(IRQ_NO, dwc_ISR_SW_DWC_INTERRUPTS,
SA_INTERRUPT, DEV_NAME, prvdata_ptr);
if(irq_rval != 0) {
```

```
    printk(KERN_ALERT "Unable to register IRQ %d\n", IRQ_NO);
    ret = -1;
    goto isr_fail;
}

DBGPR("dwc_probe: Base_addr: %#x : irq:%d\n", (UINT)base_addr
      , IRQ_NO);

INIT_WORK(&(prvdata_ptr -> dwc_rw_work), dwc_transfer_data,
          (void *)prvdata_ptr);
INIT_WORK(&(prvdata_ptr->dwc_cd_work), dwc_completed_command,
          (void *)prvdata_ptr);
prvdata_ptr -> dwc_wq = create_singlethread_workqueue(DEV_NAME);
prvdata_ptr -> mmc_host = host_ptr;
prvdata_ptr -> irq = IRQ_NO;
dev_set_drvdata(pdev_ptr, prvdata_ptr);

// device specific init routine
dwc_yinit(); //DPS INIT

DBGPR_REGS();

/* Initialize DPS feature calls using function pointers */
dwc_init_function_ptrs (&(prvdata_ptr->hw_if));

ret = mmc_add_host(host_ptr);
if(ret) {

//some code

return ret;
}
```

The probe function contains a call to the `mmc_add_host` API that registers the driver with the mmc core in the Linux kernel thus indicating that the device is ready to function. Since `INIT` feature in DPS is meant for device initialization, DDGEN inserts a call to the corresponding C function before the `mmc_add_host` invocation because all device initialization needs to happen before the driver gets registered with the mmc core.

As a result, the interrupts were being enabled before registration of the driver (since `INIT` in DPS had programming sequence to enable the interrupts) causing the corresponding interrupt handler to be executed as well.

**Fix:**

The fix goes in DPS. Interrupts should be enabled only after the driver is registered with the `mmc_core`. The programming sequence for enabling the interrupts was moved from the `INIT` sequence to another DPS feature called `POST_INIT`. DDGEN inserts a call to the corresponding C function after `mmc_add_host`. Specifying the interrupt enabling programming sequence in `POST_INIT` ensured that the interrupts were enabled only after the device driver was registered with the mmc core in the Linux kernel. The programming sequence for `POST_INIT` is given below:

```
description = "Enables the interrupts of the device globally";
feature POST_INIT
{
    CTRL.int_enable = 1;
}
```

Given below is a snippet of the generated C code, specifically the relevant portion of the probe function generated by DDGEN:

```
int __devinit dwc_probe(...)
{
    // some code
    devicename_yinit(); // corresponds to the INIT feature in DPS

    ret = mmc_add_host(host_ptr);
    if(ret) {
        printk(KERN_ALERT "failed to add 'devicename' driver");
        goto err_out;
    }
    POST_INIT();
    // some code
}
```

### Issue 3:

#### Original Symptom:

When the MMC driver module was unloaded, the host system froze and became unresponsive.

#### Detailed Analysis:

Note that the driver execution log (generated due to "printk" statements instrumented in the driver source code) did not provide any insight. On analyzing the kernel log, we observed that the PCI IRQ line was getting disabled after unloading of the driver module. The Ethernet controller interfaced to the PCI subsystem of the host stopped functioning as well because the PCI IRQ line was disabled. Note that this PCI IRQ line was shared by all the devices interfaced over the PCI subsystem including the MMC controller as well as the Ethernet controller. Since the symptom was observed on unloading of the driver module, we decided to closely inspect the `remove` API of the driver (`remove` gets invoked when the driver is unloaded). From the kernel log, we concluded that a PCI device interrupt was orphaned. Since the PCI is a shared bus with a shared IRQ line, the devices have to cooperate to service the interrupts. Orphaned interrupts in the PCI subsystem typically occur due to an ISR being freed without the corresponding interrupts of that device being disabled. The `remove` API of the generated driver code was freeing the ISR but not disabling the MMC controller interrupts. As a consequence, PCI ISR chain was broken eventually resulting in disabling of the PCI IRQ line.

#### Root Cause:

The Interrupt handler (ISR) was being freed in the `remove` API of the driver. However, the MMC controller interrupts were not being disabled. As a result, the PCI subsystem attempted to service an orphaned interrupt from the MMC controller. This in turn resulted in the PCI subsystem freeing all the IRQ handlers on that IRQ line and disabling the IRQ line itself. This resulted in disabling of ISRs of all the devices on the PCI subsystem and a faulty behaviour in the system.

#### Fix:

The fix goes in DPS. We incorporated the `FNIT` keyword feature in the DPS. The programming sequence to disable all the MMC controller interrupts was specified in this feature. Similar to `INIT`, `FNIT` is a keyword feature in DPS and typically contains programming sequences related to unloading of the device. DDGEN automatically inserts a call to the corresponding C function in the `remove` API of the driver.

The `FNIT` feature is shown below:

```
FNIT {
    CTRL.int_enable = 0;
```

```
}
```

With this fix, the device did not generate any interrupts after the driver was unloaded. Any number of iterations involving loading and unloading of the driver worked fine with this fix.

Given below is a snippet of the generated C code, specifically the fix with the interrupts being disabled in the FINIT API which was called in the remove function.

```
UINT dwc_yexit(void)
{
    CTRL_int_enable_UdfWr( 0 );
    return Y_SUCCESS;
}

INT __devexit dwc_remove(struct device *pdev_ptr)
{
    // some code
    dwc_yexit(); //FINIT

    //remove any pending function calls
    //and destroy the created work structure
    flush_workqueue(prvdata_ptr -> dwc_wq);
    destroy_workqueue(prvdata_ptr -> dwc_wq);
    mmc_remove_host(host_ptr);
    dev_set_drvdata(pdev_ptr, NULL);

    // some code
}
```

#### Issue 4:

##### Original Symptom:

Failure in card enumeration by the MMC controller. Further, the card insertion event (indicated by a corresponding interrupt for card detection) was not occurring even after removing and re-inserting the MMC card.

##### Detailed Analysis:

The initialization sequence of the MMC controller device involves Bus mode (open drain or push pull), power and clock settings. As per the mmc core and the corresponding driver architecture in Linux, any device settings or configuration needs to be done in the driver API called "set\_ios". Note that these configuration sequences were coded as explicit features in DPS and calls to the corresponding C functions were inserted in the set\_ios API.

The Linux mmc core performs card enumeration (as per the MMC protocol) by sending the relevant enumeration commands to the device. The mmc core achieves this by invoking the mmc\_request driver API which in turn writes the command sent by the mmc core in the command register of the device. As per the device data sheet, we expected the MMC controller to update its response registers with the response received for the command sent by the mmc core and raise the corresponding *command done* interrupt as well. We observed that the device driver was not receiving this *command done* interrupt for the first command (command 0) in the enumeration sequence itself.

Since card enumeration failed in the initial steps itself, our debug focus was the steps prior to this enumeration sequence that is the device configuration sequence executed in set\_ios. Instead of debugging the generated C code (as in previous cases), we reviewed the corresponding DPS features directly since the device configuration sequence was actually specified in the DPS and its easier as well as faster to verify it at the DPS level than the

generated C code. Our review of the related DPS features with respect to the programming sequences specified in the device data sheet<sup>3</sup> revealed an error in the register programming sequence for the clock settings.

---

<sup>3</sup> We have used the words device data sheet, device programming guide, device program book interchangeably. They all mean the same document that contains device details and the programming sequence details.

**Root Cause:**

Clock programming sequence in DPS for setting MMC controller clock was incorrect. Until the clocks are programmed properly, the MMC controller does not function. The DPS sequence did not match the datasheet sequence.

**Fix:**

Fix goes in DPS. Fixing the DPS clock programming sequence rectified the issue.

Following sequence highlights the fixes added to clock programming sequence.

```
DESCRIPTION ="This function is used to set the clock";
FEATURE clock_settings
{
    input unsigned int ios_clk;
    input unsigned char en_dis;
    local unsigned int clkdiv ;

    CLKENA = 0x0;
    CMD.update_clock_reg_only = 0x1;
    CMD.wait_prvdata_complete = 0x1;
    CMD.start_cmd = 0x1;
    POLL CMD.start_cmd UNTIL (CMD.start_cmd == 0x0);

    if(en_dis == 0x1) {
        clkdiv = DEVICE_INPUT_CLOCK / ios_clk;
        CLKDIV.clk_divider0 = clkdiv;
        CLKSRC = 0x0;
        CMD.update_clock_reg_only = 0x1;
        CMD.wait_prvdata_complete = 0x1;
        CMD.start_cmd = 0x1;
        POLL CMD.start_cmd UNTIL (CMD.start_cmd == 0x0);

        CLKENA = 0x1;
        CMD.update_clock_reg_only = 0x1;
        CMD.wait_prvdata_complete = 0x1;
        CMD.start_cmd = 0x1;
        POLL CMD.start_cmd UNTIL (CMD.start_cmd == 0x0);
    }
}
```

The card started responding to the commands being sent by the MMC controller after this fix, thus resulting in successful enumeration of the inserted card.

**Issue 5:**

**Original Symptom:**

Card mounting failed after successful card enumeration as a result of which no data transfer (read/write) operation could be executed.

**Detailed Analysis:**

The Linux kernel is expected to mount the card after its successful enumeration. This operation ultimately involves accessing the Master Boot Record (MBR) on the MMC card. The mmc core initiates a read cycle (by calling the mmc\_request API in the driver) to read this MBR. This read operation involves the MMC controller typically reading one page size (normally 4096 bytes) of data from the card. During debugging of the mmc\_request API and the read interrupt handler, we observed that the driver was reading only 40 bytes of data. As a result, the MBR

was not being read correctly thus eventually resulting in the card not getting mounted. On further debugging of the `device_read` API, we found that incorrect values were being set in the "`blocksize`" and "`bytecount`" registers of the device. The MMC controller computes the size of data transfer operation from these two registers. Note that these values to be written are themselves computed from the following parameters passed by the `mmc_core` to the `mmc_request` API:

```
data->blocks           // Number of Blocks
data->blkksz_bits      //Block size, which is given as shift bit size.
```

Note that the size of the block to be read has to be computed as given below:

```
(data->blocks << data->blkksz_bits)
```

Further this computation is required for Linux kernel version 2.6.11 only. The generated code was erroneously computing the block size as given below:

```
(data->blocks * data->blkksz_bits)
```

The above resulted in an incorrect value being written in the "`blocksize`" register of the mmc controller which in turn resulted in the device reading only 40 bytes<sup>4</sup> from the MMC card.

#### Root Cause:

Typically a read request is submitted to driver with details of transfer in a structure from the mmc core. The total transfer size calculation changes from kernel to kernel due to changes in the data structures. For kernel version 2.6.11 this computation has to be performed as specified previously. Note that this computation is part of the device class specific template in DDGEN and computed value is passed as an argument to the `device_read` feature in DPS.

#### Fix:

Fix goes in DDGEN. The MMC controller specific template for kernel version 2.6.11 was corrected to perform the block size computation as specified earlier.

#### Issue 6:

##### Original Symptom:

The test system froze and became unresponsive after insertion of the MMC controller driver module.

##### Detailed Analysis:

The test system froze after the insertion of the MMC controller driver module. We reviewed driver log to trace the function calls and check the register dumps. The register dumps were correct as per the expected values but the function call sequence had issues. The main issue with the function call trace was the callbacks to the mmc core from the interrupt context. Note that ideally, no callbacks should be done from interrupt context. This is because callbacks from the interrupt context could potentially (if the callback is doing a lot of processing) degrade the test system's responsiveness. Any heavy work, be it a callbacks or data transfer etc. should not be done inside the ISR and has to be deferred or scheduled for execution later. As a result of deferring, the heavy work happens only after the completion of ISR and hence the system performance and responsiveness is unaffected.

The above issue was due to a callback "`mmc_request_done`" to mmc core on completion of a request sent by the mmc core. Any data or non-data transfer operation is in the form of a *request* sent by the mmc core to the MMC controller driver through the `request` API. After completion of the *request*, the MMC controller driver does a callback to the mmc core to inform about the completion of the *request*. In case of an error or a failure of the

---

<sup>4</sup> "blocksize" was 4 and "bytecount" was 10

*request*, the mmc core retries the *request* using the MMC controller driver's *request* API. With the retry happening due to data transfer errors, all the heavy work was happening in the interrupt context.

Also note that these *retry-requests* sent by the mmc core are completed only after interrupts such as the *completed command* interrupt are received. Note that the callback was initially invoked directly from the interrupt context and as a result the corresponding *retry requests* were also occurring in the interrupt context eventually resulting in nesting of the driver ISR. Note that, if the code is executing in the interrupt context, Linux will not re-enter an ISR.

Due to the above mentioned reasons the system's responsiveness was degraded and resulted in system freeze.

**Root Cause:**

Callbacks to the mmc core from the interrupt context, as explained above, with heavy work were the main reason for the above symptom.

**Fix:**

Fix for the above issue was to split the ISR into a top half and bottom half and execute the callback using the *mmc\_request\_done* in the bottom half. This would relieve the ISR from the retry loop on request failure. Splitting of the ISR into top and bottom half was specified in the RTS specification. It is recommended that any callbacks to the Linux core and other heavy duty work (data transfers) be done inside the bottom half.

Change in the RTS input field ensured that the callback *mmc\_request\_done* gets deferred for execution rather than getting called directly in the ISR. Thus, even in case of failures in servicing transfer requests from the mmc core, the system will not freeze.

The value of the RTS field *ISR\_TYPE* was modified from "SINGLE" to "SPLIT". Following RTS snippet shows the RTS changes:

```
//Before Fix

ISR_SPEC {
    ISR_TYPE    = SINGLE;
    SHARED_IRQ = YES;
}
```

```
//After Fix

ISR_SPEC {
    ISR_TYPE    = SPLIT;
    SHARED_IRQ = YES;
}
```

**Issue 7:**

**Original Symptom:**

For DMA version of the driver module, unloading and loading the module for the second time resulted in a failure. The system eventually became unresponsive and had to be re-booted.

**Detailed Analysis:**

This issue was caused after the driver module was unloaded and reloaded back. Hence the first suspect was the *remove* routine. We used PCI utilities such as *lspci* and *setpci* to access the PCI configuration space of the device and ascertain if it was getting changed after unloading. We found that the call to *pci\_disable\_device* inside the *remove* routine had disabled the BUS MASTERING capability of the

MMC controller. This capability is required for the MMC controller to perform data transfers using the internal DMA.

We manually enabled the BUS MASTERING capability and found that the mentioned issue was resolved. Note that the PIO mode operation of the driver did not reveal this problem because BUS MASTERING is required only for DMA transfers.

**Root Cause:**

The call to `pci_disable_device` in the `remove` API of the generated driver was disabling the BUS MASTERING capability thus leading to the above mentioned symptom.

**Fix:**

As in the previously described issue, the MMC controller specific template in DDGEN was fixed by removing this call to `pci_disable_device` from the driver's `remove` API.

### 3. Guidelines for Debugging of DDGEN Generated device drivers

In addition to the high-level debug methodology and our specific experiences in debugging of DDGEN generated driver code for an MMC controller as explained in the previous section, we present a comprehensive set of guidelines below:

1. It is important that a DDGEN user has some basic familiarity on how DDGEN synthesizes the driver code from the input DPS and RTS specification. At a very high-level, the DDGEN generated driver code can be viewed as a composition of the following components:
  - Device class specific template (this template is part of the DDGEN internal template library). The template is based on the device class, operating system and the driver model and hence imposes certain architecture on the generated device driver.
  - Translation and mapping of programming sequences specified as DPS features (explicit features as well as keyword features). The explicit features get mapped manually via function pointers while keyword features are automatically mapped by DDGEN
  - Parts of the driver code get synthesized directly from other specifications in the DPS. For example macros to enable/disable an interrupt are generated from the `Interrupt Spec` in DPS.
  - Apart from the `Operating System and Driver Model` attribute in the RTS, some other attributes also impact DDGEN code generation. (for example the `ISR_SPEC`).

Refer [1] for a complete discussion on DDGEN. Depending upon the device for which the driver is being generated, please ensure that you read the corresponding Application Note for that particular device class as well. The application note gives more detailed information specific to the device class and its vagaries. Note that for each device class, you would need to write explicit features in DPS to specify programming sequences that cannot be encoded in any of the keyword features. Further each of these explicit features gets mapped to a certain place in the generated code. This mapping is done by a set of function pointers which serve as placeholders in the generated code. These function pointers are manually assigned the address of the generated C functions corresponding to the DPS features. It is important that the user is aware of this mapping process for effective debugging of the generated driver code.

2. It is imperative that the programming sequences specified in the device programming book (or data sheet) are correctly captured in DPS. Often a single programming sequence may span across multiple sections in the programming book and the user needs to ensure that the programming book is thoroughly understood to effectively write the programming sequence. Similarly, please ensure that other specifications in the DPS viz. `register specification`, `bus_specification`, `fifo_specification`, `interrupt specification` etc. are correctly specified as well.

- While testing the generated driver code in the target test environment, we recommend that the user views the driver functionality as a collection of functional states through which a single successful execution has to sequentially pass. The user can then focus on ensuring that the driver successfully passes through each of these states. For example, the MMC controller device driver can be viewed as a collection of following sequential states:

Function State Name	Description
Driver registration	Essentially means successful execution of the <code>probe</code> API
Device Configuration	Successful execution of the device configuration and settings – DPS features for bus mode/clock settings invoked in the <code>set_ios</code> API of the driver.
Card Enumeration	Successful card enumeration done by series of commands (as per the protocol) sent by the Linux <code>mmc_core</code> via the <code>request</code> API in the driver. This should result in an interrupt to indicate end of a command and ISR action is to callback the <code>mmc_core</code> through a callback function: <code>mmc_request_done</code>
Card mounting	This actually means success in the first read operation which eventually reads the MBR from the card. The <code>blocksize</code> and <code>bytecount</code> are passed by the <code>mmc_core</code> to the <code>mmc_request</code> API and the appropriate number of bytes should be read. This also involves correct handling of the read interrupt by the ISR in the driver.
Successful read operation	Multiple read operations by the <code>mmc</code> core are successful. For example a data file should successfully get copied from the MMC card to the host system.
Successful write operation	Multiple write operations by the <code>mmc</code> core are successful. For example a data file should get copied from the host system onto the MMC card.

- We recommend that the actual testing and debugging activity be focused on one functional state at a time (at least initially till the driver successfully executes a few test cases). The actually debugging can be effectively performed due to the instrumentation mechanisms specified in the previous section. In case, a particular functional state is not being executed correctly, in majority of the cases, the analysis of driver log generated during execution will lead to an incorrect or missing programming sequence. Note that most of the programming sequences are captured in DPS features. To further debug a potentially incorrect programming sequence, we suggest that the user directly review the DPS feature and verify it with the corresponding device programming guide. Since DPS allows you to specify programming sequences at a higher level of abstraction, it is much easier to spot errors in a programming sequence at the level of a DPS feature than the generated C code.
- In case of a missing programming sequence, please add this programming sequence to the appropriate DPS features. In some cases, you might have to write a new DPS feature and manually map the corresponding generated C function at an appropriate place in the generated C code.
- Note that in some cases, you might identify an error in a particular DPS feature. It is possible that the programming sequence itself does not have any problems (and you can also verify this by cross-checking with the device programming guide). Often the register details pertaining to the programming sequence might have been captured incorrectly in the Register Specification of DPS. For example – the length of a register field might has been specified incorrectly.
- DDGEN also generates some programming sequences automatically based on several specifications in the DPS like the FIFO Specification, Interrupt Specification, USB Specification etc. Please review these sections for any suspected issues in the corresponding areas.

For example the programming sequence to enable/disable/clear an interrupt could have a problem due to an error in the Interrupt Specification of DPS. The error could be as simple as incorrectly capturing the `Interrupt Enable` register field for a particular Interrupt. DPS writers often create parts of a specification by copying some previous parts and then modifying them. This is often error-prone. For example the Interrupt details for `read` Interrupt are first specified. Subsequently, to capture the details for the `write` interrupt, the user copies all the data for the `read` interrupt and appropriately modifies all the fields except the interrupt enable

field for the *write* interrupt.

8. Some issues in the generated driver could be a result of an incorrect specification for some of the attributes in RTS. Note that these issues would typically never occur in the initial function states of the driver and are most likely to occur only after the device initialization and driver registration states. Also note that platform driver model in Linux (as well as the layered MDD/PDD model in WinCE) enforces a strict architecture on the driver and very few attributes are configurable in the RTS. Typically only the buffering mechanism, ISR architecture (single ISR or split ISR, tasklets or work queues) and the synchronization primitives can be varied. For any errors in the data transfer operations (for example system performance degrades during a read or write operation), we recommend that you review the RTS specification and make the appropriate modifications.
9. A production quality driver often requires that you introduce some delays to synchronize driver execution to the correct device state. This synchronization is most often achieved by a *trial and error* mechanism during the debugging phase to identify the optimal delay required. Several times, the placement of this delay statement is crucial as well. We recommend that the user try and capture this delay as a part of the relevant programming sequence in the corresponding DPS feature itself. If this is not possible, please write a dummy DPS feature containing this delay and manually insert a call to this DPS feature in the generated code.
10. In few cases, there could be a bug in the DDGEN tool itself, as described in Issue 5 and hence the fix for this bug would be in the DDGEN tool as well.

## 4. Conclusion

DDGEN synthesizes device driver code from two input specifications: DPS and RTS. A typical development methodology requires that we provide debugging tools and aids at the level of DPS and RTS. Note that defining debug architecture and methodology at the level of DPS and RTS is non-trivial. We recommend debugging the generated C code itself. Our debug experiences in various customer evaluations indicate that debugging the generated C code is highly efficient and the fix of any bug can be effectively mapped to DPS or RTS as well. The code instrumentation mechanisms provided by DDGEN further help in this process. Our case study discussed in Section 2 and the comprehensive guidelines specified in Section 3 should enable a DDGEN user to jumpstart debugging of the generated C code. We conclude that a fix for a typically bug involves relevant corrections in the DPS or the RTS. Finally, some bugs in the generated could be a result of a genuine issue in the DDGEN tool itself.

## 5. References

- [1] Sandeep P, Venugopal K: DDGEN, An Automated Device Driver Generation Tool for Embedded Systems (<http://www.design-reuse.com/articles/23316/automated-device-driver-generation-tool.html>)
- [2] DDGen FAQs (<http://www.vayavyalabs.com/products/ddgen-faq>)