A Mathematically Verified Device I^2C Driver Using ASD

Herman Roebbers

Nov-2-2009

TASS software professionals

Dedicated to make it work.

How to develop defect-free device drivers with ASD:Suite

Contents

Introduction

NXP wants to evaluate ASD

Case study:

- Linux I^{2}C driver for ARM processor

Introduction

In fact two case studies were done:

1. ASD with C++ generator and stripped BOOST library (reported on at previous ASD UGM)
2. ASD with C generator (beta version) and OS Abstraction Layer implementation for Linux kernel mode (reported on here)

Objectives (1)

Verify benefits of ASD

Most important benefits of interest to NXP:

- Delivering higher quality product with same or reduced amount of effort
- Reducing cost of future maintenance
- Achieving at least equivalent performance to existing product

TASS software professionals

Objectives (2)

Determine if ASD modeling helps in verifying correctness of device driver software

Assess whether ASD:Suite is a useful and practical tool for developing device drivers

Development Process

- Obtain HW doc to create device Interface Model
- Create driver model
- Verify combination works (model check)
- Create device foreign component
- Create interrupt -> ASD event interface
- Create Linux kernel OSAL implementation (once)
- Create Linux kernel driver / loadable module
- Execute test bench (existing driver test bench)
- Fix problems in foreign components

TASS
software professionals

Who did what?

NXP provided

- Domain knowledge for the project

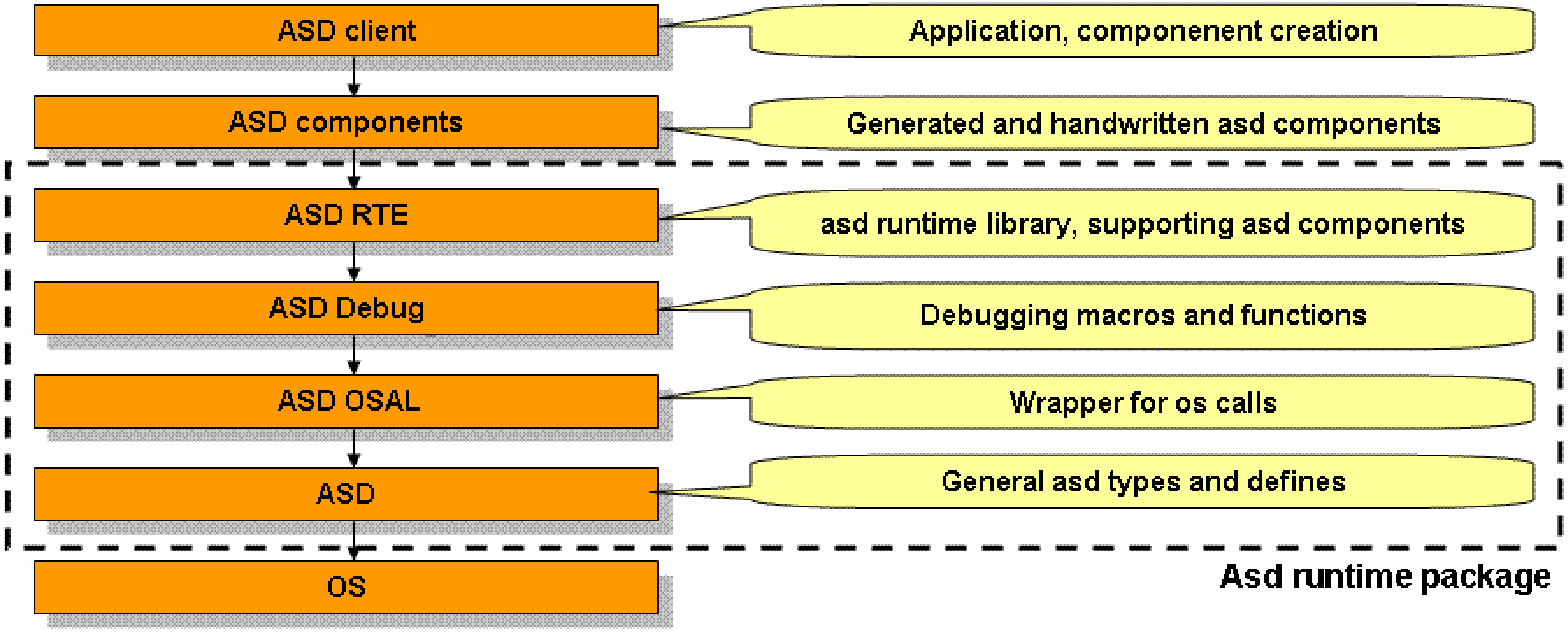
Verum

- Modeling of the driver specification and design

TASS provided expertise

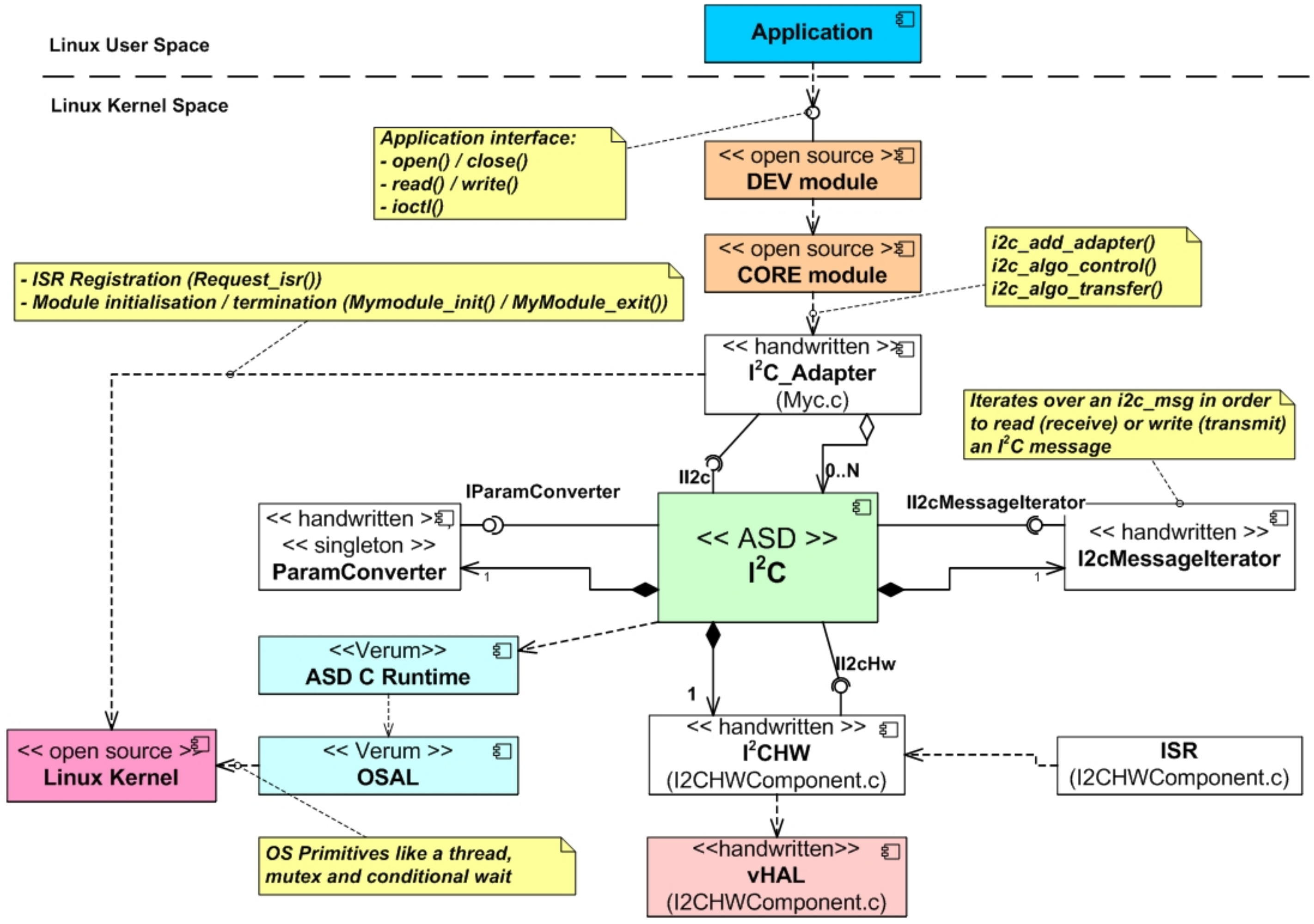
- Input for the C code generation process
- Linux kernel space OSAL implementation
- Interrupt management

General Structure Of ASD Component


ASD client
Application, componenent creation
ASD components
Generated and handwritten asd components
ASD RTE
asd runtime library, supporting asd components
ASD Debug
Debugging macros and functions
ASD OSAL
Wrapper for os calls
ASD
General asd types and defines
OS
Asd runtime package

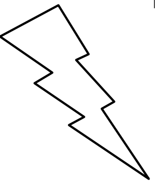
TASS
software professionals

General Structure Of Linux Device Driver With ASD


Linux User Space
Linux Kernel Space
Application
Application interface:
- open() / close()
- read() / write()
- ioctl()
<< open source >> DEV module
<< open source >> CORE module
i2c_add_adapter()
i2c_algo_control()
i2c_algo_transfer()
- ISR Registration (Request_isr())
- Module initialisation / termination (Mymodule_init() / MyModule_exit())
<< handwritten >> I²C_Adapter (Myc.c)
Iterates over an i2c_msg in order to read (receive) or write (transmit) an I²C message
II2c
0..N
IParamConverter
II2cMessageIterator
<< handwritten >> << singleton >> ParamConverter
<< ASD >> I²C
<< handwritten >> I2cMessageIterator
1
1
<<Verum>> ASD C Runtime
II2cHw
1
<< open source >> Linux Kernel
<< Verum >> OSAL
<< handwritten >> I²CHW (I2CHWComponent.c)
ISR (I2CHWComponent.c)
OS Primitives like a thread, mutex and conditional wait
<<handwritten>> vHAL (I2CHWComponent.c)

TASS software professionals

Connecting Interrupts to An ASD Component



interrupt

```
i2c_isr() {
   event = deal_with_IRQ();
   kfifo_put(event);
   queue_work(
          interrupt_wq,
          &i2c_event_work)
}
```

workqueue function

```
i2c_event_handler(){
  while (msg_in_kfifo()){
     kfifo_read(…,&intdata,…);
     /* put msg in DPC queue
          and schedule DPC thread */
     schedule_DPC_callback(&int_data);
  }
}
```

The Linux Kernel OSAL

Mapping ASD thread onto Linux kernel thread

- ASD thread function is:

  `void *(*asdThread_func)(void*arg)`
- Linux kernel thread function is:

  `int (*kthread_func)(void *arg)`

In order to provide thread control some extra management support has to be added

The Linux Kernel OSAL

Mapping ASD thread onto Linux kernel thread

```
typedef struct
{
  asdThread_func func;          Thread function
  void* arg;                    Thread function arg
  struct completion finish;     For thread termination
  struct task_struct *thread;   OS thread reference
} asdThread;
```

The Linux Kernel OSAL

```
void asdThread_init(asdThread* self,
   asdThread_func func, void* arg)
{
  self->func = func;    Thread function to thread
  self->arg = arg;      Thread func arg to thread
  init_completion(&self->finish); Termination
  self->thread = kthread_run(thread_wrapper,
   (void*) self, "work_kthread");  Start thread
  ASD_ASSERT(!IS_ERR(self->thread));
}
```

TASS
software professionals

The Linux Kernel OSAL

```
static int thread_wrapper(void* arg)
{
    asdThread* self = (asdThread*)arg;
    self->func(self->arg);  Call thread_func
    complete(&self->finish);   End thread
    return 0;
}
```

The Linux Kernel OSAL

```
void asdThread_join(asdThread* self)
{
  ASD_ASSERT(!wait_for_completion_timeout(
   &self->finish, 250));
}
```

Results : Code size

Code size of original NXP driver code	
built-in.o	12468 bytes

Code sizes for ASD generated C code + driver + ASD OSAL.	
Handwritten ASD runtime lib incl. OSAL	4824
Handwritten code	4876
ASD generated code	12048
Total code in mymodule.o	21748

Results : Code size

Code size of original NXP driver code	
built-in.o	12468 bytes

Code sizes for ASD generated C code + driver + ASD OSAL.	
Total code in mymodule.o	21748 bytes

Code sizes for ASD generated C++ code + driver + BOOST	
built-in.o	155 Kbytes

Results : Performance

Execution time	Old driver	ASD + OSAL
Send of 2 bytes	380 µs	386 µs
Time in interrupt	60 µs	20 µs

Less time in interrupt means more responsive system

Results : Discussion

Functionality:

- Original handwritten code implements I^{2}C master as well as slave functionality
- ASD code only implements I^{2}C master functionality
- Multi-client support works (didn’t work for C++ version)

Results : Discussion

Code size:

- ASD driver is bigger, but includes RTE incl. OSAL, handwritten HW interface and interrupt connection. OSAL is small and reusable for other drivers.
- ASD code can be bigger because more situations are covered than in non-ASD code.
- HW interface can be written more optimally to require smaller code size.

Issues

HwI2C component interface was not reviewed with HW guys to check equivalence with actual device behavior prior to implementation. This caused problems and required several updates to the model to get right.

Issues

Stress test runs for several hours and then stops -> EEPROM I2C device write fails. New I2C EEPROM cures issue. Is the problem in the model or in the implementation?

What happens when EEPROM write fails and times out? And generates completion or NACK after timeout? Current timeout hardcoded, based on EEPROM timeout spec, should be set higher.

What happens with old driver on EEPROM write fail?

TASS software professionals

Conclusions

- HW I/F model must be validated together with HW experts before anything else!!!!!!
- Behavioral issues in ASD part cannot be present because of model checking (assumes correct HW model!)
- ASD approach is feasible for Linux device driver development
- Footprint as well as performance are comparable with handwritten code
- Less time in interrupt = more responsive system

Conclusions

- Small overhead due to OSAL and RTE
- Current implementation of HW component not written with performance in mind. First get it right!
- Several driver/OSAL parameters are hardcoded, of which some are dependent on the attached I^2C device. So they should be `#defined` or parameterized, maybe via separate `ioctl`'s. FIFO size should be read from device.

Conclusions

- Driver development time: I don’t know original driver effort and don’t know ASD effort. OSAL development effort should not be counted as this can be reused.
- Driver quality: passes stress test for several hours. Cause of hangup still unknown

TASS software professionals

Conclusions

This project has clearly shown

- ASD modeling helps in developing and verifying deeply embedded software
- Using the C code generator is beneficial and practical for device drivers.

Conclusions

Biggest advantage seen

- Rigorous specification process enforced with ASD.
  - Software designers are forced to think before they implement, and ASD helps them ensure a complete and correct specification.
- Race conditions and deadlocks due to unexpected interleaving of activities are prevented by the model checker
- Developers can perform manual timing checks on guaranteed defect free code.

TASS
software professionals

Conclusions

Model checker revealed > 700,000 unique execution scenarios

- Without ASD: > 700,000 test cases required.
- With ASD: No need to test

=>Major reduction in testing effort achieved.

Future work (@Verum)

- Optimize code generation
- Reduce footprint of foreign components
- Providing of OSAL compliance test code for validating OSAL implementations
- Providing guidelines for development of OSAL implementation
- Providing guideline on development of HW Interface code

TASS
software professionals

Questions???

TASS software professionals

Dedicated to make it work.