Philosophy of Device Driver Development with I2C Example

B. Vamsi Krishna

Structuring the Device Interface:

The foundation of device driver development lies in accurately mirroring the device's architecture within software. This is achieved by defining a data structure that corresponds to the memory-mapped control and status registers of the device. Such a structure is typically a C struct that directly maps to the registers of the hardware device.

For instance, in an I2C device driver, the struct may look like this:

```
struct I2CDevice {
uint16_t control; // Control register
uint16_t status; // Status register
uint16_t data; // Data register
uint16_t address; // Address register
};
```

State Management Variables:

The second stage involves determining the necessary variables to effectively monitor and manage the state of both the hardware device and the driver itself. These variables serve as a medium to reflect the current status of the device, track its usage, and manage various operational aspects.

In the context of our I2C driver, we might have:

```
typedef struct {
struct I2CDevice* i2c;
int is_initialized;
} I2CDriverState;
```

Initialization Routine:

The third phase is the development of an initialization routine. This function is responsible for setting up the hardware device in a known, stable state. It involves configuring the device's registers to default values and preparing it for operation.

For our I2C device, the initialization might look like this:

```
void initialize_i2c(I2CDriverState* state) {
// Initialization code
state->i2c->control = I2C_ENABLE;
state->is_initialized = 1;
}
```

Developing an API for the Driver:

After successfully initializing the device, the next step is to build a set of routines that collectively form the API for the device driver. These functions allow for interaction with the device, such as reading from or writing to it.

The API for our I2C driver might include functions for starting and stopping communication, writing data, and reading data:

```
void i2c_start(I2CDriverState* state);
void i2c_stop(I2CDriverState* state);
void i2c_write(I2CDriverState* state, uint8_t data);
uint8_t i2c_read(I2CDriverState* state, int ack);
```

Interrupt Service Routines (ISRs):

Finally, the development of Interrupt Service Routines (ISRs) is critical, especially in real-time systems where immediate response to hardware events is required. ISRs handle interrupts generated by the device, allowing for efficient and timely responses to events such as data availability or transmission completion.

An example ISR for our I2C driver might be:

```
void i2c_isr(void) {
// ISR code to handle I2C interrupts
}
```


Contact me @
www.linkedin.com/in/balemarthyvamsi
www.balemarthyvamsi.com