Linux Device Drivers

a brief introduction

Alexandre Moreno

Seenergy Corp.
alexandre@seenergy.com.tw

February 2012

Introducton

Device drivers are the abstraction layer between software concepts and hardware circuitry:

- They provide a standard interface to perihperals, hiding the details of how devices work
- Almost every system operation eventually maps to a physical device
- The OS kernel embeds device drivers for every peripheral device present in the system
- Unix motto also applies here, i.e. provide mechanism, not policy
- have seven times the bug rate of other kernel code[1]

1: *pdos.csail.mit.edu/6.097/readings/osbugs.pdf*

Classes of devices

Typically device drivers are classified as follows:

- Character devices: simplest type. They are accessed as a stream of bytes, sequentially
- Block devices: I/O operations transfer whole blocks (e.g. 512 bytes), can be random-accessed and host a filesystem.
- Network interfaces: packet delivery, uses network subsystem API; do not use device nodes in order to be accessed

But this taxonomy is not universal, e.g. an USB device might appear in the kernel as char device (serial port), block device (USB flash drive) or network device (USB Ethernet interface)

Linux kernel—user-space API

Linux provides different mechanisms for communicating between user and kernel space:

- System calls: functions[1] which are useful for many application (about 380)
- ioctl syscall: catch-all for non-standard file operations
- Virtual file systems: procfs, sysfs, configfs, debugfs
- sysctl: configure kernel parameters at runtime
- Netlink: sockets-based interface e.g. iptables/netfilter
- Upcall: allows kernel modules start a program in userspace

1: syscalls aren't usually invoked directly from userspace, but rather thru C library wrappers. This is because a system call requires, among other things trapping to kernel mode, set errno variable etc. which is arch specific. From userspace it's just a C library function.

Common practise

From the previous list of interfaces, device drivers usually:

- implement file operations, like open, read, write, poll, mmap etc. which implements the system calls with the same name
- might use ioctls for other device-specific operations, e.g. query capabilities, tuning, etc.
- use the pseudo file system /proc to expose internal data structures

Everything is a file

The UNIX philosophy is quoted as "everything is a file". That what really means is that everything is a stream of bytes—in the file system namespace.
Sockets and pipes are the exception, lacking a file name. A more precise definition would be "everything is a file descriptor"
The advantage of this approach is that you can use the same (file-based) interfaces on different things.

http://yarchive.net/comp/linux/everything_is_file.html

File operations

file operations (simplified)

```
struct file_operations {
    struct module  *owner;
    loff_t         (*llseek)         (struct file *, loff_t, int);
    ssize_t        (*read)           (struct file *, char __user *, size_t, loff_t *);
    ssize_t        (*write)          (struct file *, const char __user *, size_t, loff_t *);
    unsigned int   (*poll)           (struct file *, struct poll_table_struct *);
    long           (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long           (*compat_ioctl)   (struct file *, unsigned int, unsigned long);
    int            (*mmap)           (struct file *, struct vm_area_struct *);
    int            (*open)           (struct inode *, struct file *);
    int            (*flush)          (struct file *, fl_owner_t id);
    int            (*release)        (struct inode *, struct file *);
    int            (*fsync)          (struct file *, loff_t, loff_t, int datasync);
    int            (*aio_fsync)      (struct kiocb *, int datasync);
    int            (*fasync)         (int, struct file *, int);
    int            (*lock)           (struct file *, int, struct file_lock *);
};
```

Character devices are accessed as files. Drivers should provide implementations of some of these functions. Thus one opens a device, reads/writes from/to it, and closes it, using the same file I/O semantics used for regular files.

include/linux/fs.h

Loadable Modules

- Linux has the ability to extend the kernel functionality at runtime using modules
- Device drivers can also be added to the kernel in this fashion (benefits are a smaller kernel, on demand loading gives you a better footprint and no kernel recompilation to add new modules)
- You can use the module-init-tools package, which contains a set of programs for loading, inserting and removing kernel modules.

```
Example:
    obj-$(CONFIG_FOO) += foo.o

  $(CONFIG_FOO) evaluates to either y (for built-in) or m (for module).
  If CONFIG_FOO is neither y nor m, then the file will not be compiled
  nor linked.
```

hello, world (1)

helloworld.c

```c
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void)
{
    printk(KERN_ALERT "Hello, world\n");
    return 0;
}

static void hello_exit(void)
{
    printk(KERN_ALERT "Goodbye, world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Makefile

```makefile
ifneq (£(KERNELRELEASE),)
    obj-m := helloworld.o
else
    KERNELDIR ?= /lib/modules/$(shell uname -r)/build
    PWD := $(shell pwd)
default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
```

hello, world (2)

And this is a simple session showing how to bring it to life

```
$ ls
helloworld.c  helloworld.ko  helloworld.mod.c  helloworld.mod.o
helloworld.o  Makefile  modules.order  Module.symvers

$ sudo insmod ./helloworld.ko
$ dmesg | tail -1
[151235.953044] Hello, world

$ cat /proc/modules | grep helloworld
helloworld 680 0 - Live 0xf8252000

$ modinfo helloworld.ko
filename:       helloworld.ko
license:        Dual BSD/GPL
srcversion:     736D100661E927970863868
depends:
vermagic:       2.6.32-39-generic SMP mod_unload modversions 586

$ sudo rmmod helloworld
$ dmesg | tail -1
[151277.360643] Goodbye, world
```

Driver entry and exit points

model_init

```
module_init(hello_init);
```

The driver's init() routine specified in module_init() will be called during boot (when statically linked in) or when the module is dynamically loaded. The driver passes its structure of operations to the device's subsystem when it registers itself during its init().

model_exit

```
module_exit(hello_exit);
```

module_exit will wrap the driver clean-up code with cleanup_module when used with rmmod when the driver is a module. If the driver is built-in, module_exit has no effect.

Access the Linux kernel using the /proc filesystem

prochello.c

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/proc_fs.h>  /* Necessary because we use the proc fs */

static struct proc_dir_entry *proc_ent;
static int proc_read(char *buffer, char **start, off_t offset, int size, int *eof, void *data)
{
  char *hello_str = "Hello, world!\n";
  int len = strlen(hello_str); /* Don't include the null byte. */
  strcpy(buffer, hello_str);
  *eof = 1;
  return len;
}
static int simpl_init(void)
{
  proc_ent = create_proc_entry("simpl", 0, NULL);
  if(!proc_ent){
    remove_proc_entry("simpl", NULL/*&proc_root*/);
    return -ENOMEM;
  }
  proc_ent->read_proc = proc_read;
  return 0;
}
static void simpl_exit(void)
{
  remove_proc_entry("simpl", NULL/*&proc_root*/);
}
module_init(simpl_init);
module_exit(simpl_exit);
```

Interrupts

At any given time the system is either in user or kernel (aka supervisor) mode. When the kernel runs, it is either in:

- process context: on behalf of the user, e.g. thru a system call, or
- atomic context: or interrupt context, servicing a device, e.g. the system timer.

Thus, a device driver is either in process or interrupt context. These contexts are very different regarding concurrency and latency of operations, and that's the reason we need deferred work APIs.

Interrupt handlers

Run asynchronously, and are registered using request_irq(), which asks the kernel to call them when that IRQ line raises and interrupt. If that request fails, you would need a fallback routine. Once you finish processing, you can either return:

- not handled: you check your registers and happens that your device neither reads or writes anything, so means some other device was using that line
- handled: you're done or
- deferred: Deferred means that after the handler has performed a possible time-critical part of the handling, some other work, is done later, by so-called bottom-halves

Bottom halves

Deferring work idea: split time-sensitive work from non-time-sensitive one. For that purpose, the kernel provides:

- timers
- workqueues
- kernel threads

One particular useful analogy from OO design, is the Reactor/Proactor pattern.

Memory-mapped peripherals

Embedded systems, particularly those based on systems-on-a-chip, usually lack of a standard peripheral bus—e.g. USB, with discovery capabilities, for most integrated devices. Thus these devices need to be explicitly instantiated by kernel code. The Platform device API is provided for that purpose. At boot time, the bootloader passes the machine type it is booting in a register.

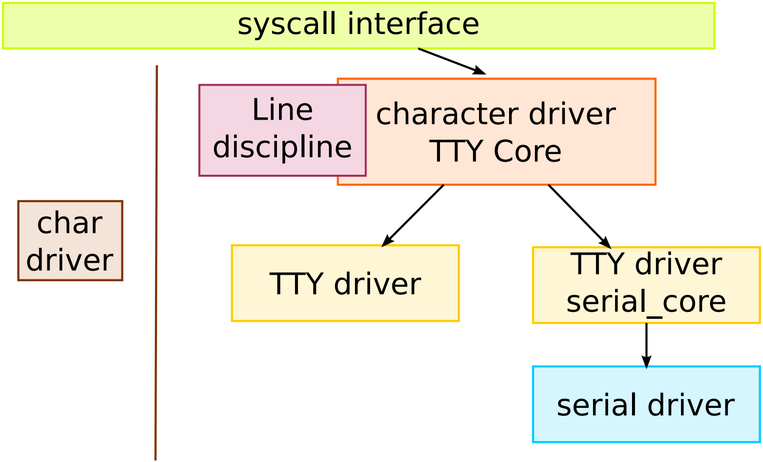
https://www.kernel.org/doc/Documentation/driver-model/platform.txt

Device trees

The previous approach, means the kernel needs the **entire description of the hardware**. A new approach is to use a device tree, a textual description of the hardware configuration. The bootloader loads too images, the kernel (e.g. uImage) and the device tree, and then passes the address of the table in a register to the kernel, instead of the machine type.

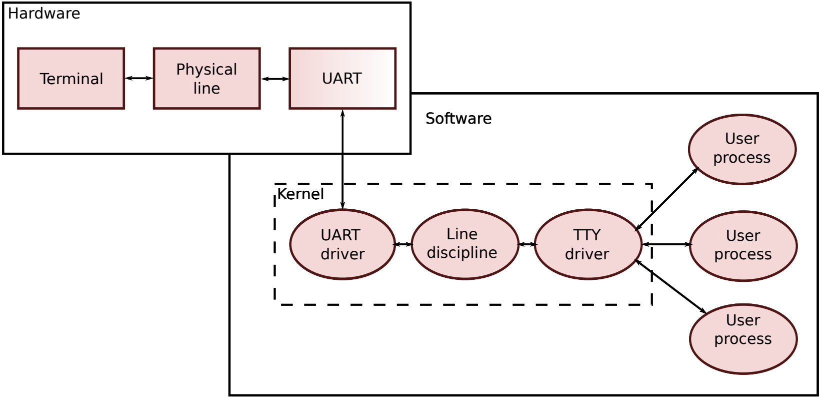
http://lwn.net/Articles/448502/

The TTY layer



The Linux console (1)

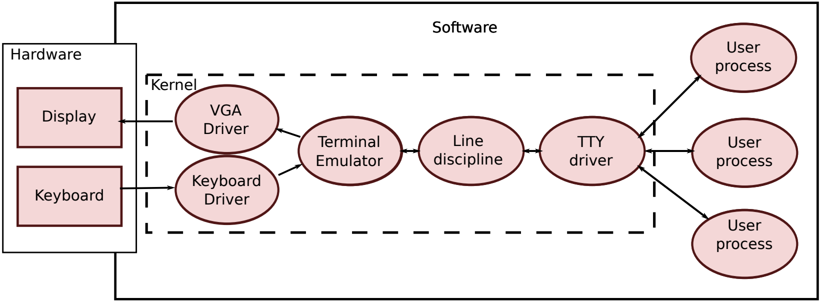
In embedded Linux, it's common to use a serial port as console



The serial ports are named ttyS0, ttyS1, etc. (COM1, COM2, etc. in DOS/Windows), and accessed thru /dev/ttyS0, etc.

The Linux console (2)

In your desktop Linux, you have virtual consoles (/dev/tty1, etc.).
Normally in the 7th virtual console start the X Window System



The other 6 VT are text consoles.
From your desktop, when using a terminal emulator such as xterm or a network login service such as ssh a pseudo-terminal device (dev/pts0 etc.) is used.

Check it out in your Linux desktop machine

```
$ ps aux | grep tty
root       834  0.0  0.0   1788   468 tty4     Ss+  Mar27   0:00 /sbin/getty -8 38400 tty4
root       837  0.0  0.0   1788   472 tty5     Ss+  Mar27   0:00 /sbin/getty -8 38400 tty5
root       845  0.0  0.0   1788   472 tty2     Ss+  Mar27   0:00 /sbin/getty -8 38400 tty2
root       846  0.0  0.0   1788   472 tty3     Ss+  Mar27   0:00 /sbin/getty -8 38400 tty3
root       849  0.0  0.0   1788   472 tty6     Ss+  Mar27   0:00 /sbin/getty -8 38400 tty6
root       970  1.8  6.8  90380 69812 tty7     Ss+  Mar27  42:51 /usr/bin/X :0
-nr -verbose -auth /var/run/gdm/auth-for-gdm-e3KUoh/database -nolisten tcp vt7
root      1145  0.0  0.1   3160  1796 tty1     Ss   Mar27   0:02 /bin/login --
alex     28851  1.0  0.3   6224  3632 tty1     S    17:56   0:05 -bash
alex     28880  2.0  0.1   2712  1080 tty1     R+   18:05   0:00 ps aux
alex     28881  1.0  0.0   3324   836 tty1     S+   18:05   0:00 grep --color=auto tty
```

The init process reads the file /etc/ttys and, for every terminal device that allows a login, does a fork followed by an exec of the program getty

Check it out in your Linux desktop machine

```
$ ps aux | grep tty
root       834  0.0  0.0   1788   468 tty4     Ss+  Mar27   0:00 /sbin/getty -8 38400 tty4
root       837  0.0  0.0   1788   472 tty5     Ss+  Mar27   0:00 /sbin/getty -8 38400 tty5
root       845  0.0  0.0   1788   472 tty2     Ss+  Mar27   0:00 /sbin/getty -8 38400 tty2
root       846  0.0  0.0   1788   472 tty3     Ss+  Mar27   0:00 /sbin/getty -8 38400 tty3
root       849  0.0  0.0   1788   472 tty6     Ss+  Mar27   0:00 /sbin/getty -8 38400 tty6
root       970  1.8  6.8  90380 69812 tty7     Ss+  Mar27  42:51 /usr/bin/X :0
-nr -verbose -auth /var/run/gdm/auth-for-gdm-e3KUoh/database -nolisten tcp vt7
root      1145  0.0  0.1   3160  1796 tty1     Ss   Mar27   0:02 /bin/login --
alex     28851  1.0  0.3   6224  3632 tty1     S    17:56   0:05 -bash
alex     28880  2.0  0.1   2712  1080 tty1     R+   18:05   0:00 ps aux
alex     28881  1.0  0.0   3324   836 tty1     S+   18:05   0:00 grep --color=auto tty
```

```
$ stty -a < /dev/tty1
speed 38400 baud; rows 30; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
eol2 = <undef>; swtch = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R;
werase = ^W; lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 hupcl -cstopb cread -clocal -crtscts
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon ixoff
-iuclc -ixany -imaxbel -iutf8
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig icanon -iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt
echoctl echoke
```

The init process reads the file /etc/ttys and, for every terminal device that allows a login, does a fork followed by an exec of the program getty

Data structures

The generic UART driver is defined by the following structures:

- A data structure representing a driver : uart_driver
  - Single instance for each driver
  - uart_register_driver() and uart_unregister_driver()
- A data structure representing a port : uart_port
  - One instance for each port (several per driver are possible)
  - uart_add_one_port() and uart_remove_one_port()
- interface between serial_core and the hardware specific driver: uart_ops
  - Linked from uart_port through the ops field

https://www.kernel.org/doc/Documentation/serial/driver uart_ops

```
struct uart_ops {
  unsigned int  (*tx_empty)(struct uart_port *);
  void          (*set_mctrl)(struct uart_port *, unsigned int mctrl);
  unsigned int  (*get_mctrl)(struct uart_port *);
  void          (*stop_tx)(struct uart_port *);
  /* ... */
};
```

uart_driver

```
struct uart_driver {
  struct module   *owner;
  const char      *driver_name;
  const char      *dev_name;
  int              major;
  int              minor;
  int              nr;
  struct console  *cons;
};
```

uart_port

```
struct uart_port {
  spinlock_t              lock;       /* port lock */
  unsigned long           iobase;     /* in/out[bwl] */
  unsigned char __iomem   *membase;   /* read/write[bwl] */
  int                     (*handle_irq)(struct uart_port *);
  unsigned int            irq;        /* irq number */
  unsigned long           irqflags;   /* irq flags  */
  unsigned int            uartclk;    /* base uart clock */
  unsigned int            fifosize;   /* tx fifo size */
  unsigned char           x_char;     /* xon/xoff char */
  unsigned char           regshift;   /* reg offset shift */
  unsigned char           iotype;     /* io access style */
  /* ... */
  const struct uart_ops    *ops;
  resource_size_t         mapbase;    /* for ioremap */
  struct device           *dev;       /* parent device */
  /* ... */
};
```

linux/drivers/char/serial_core.h

altera_uart_ops

```
static struct uart_ops altera_uart_ops = {
  .tx_empty       = altera_uart_tx_empty,
  .get_mctrl      = altera_uart_get_mctrl,
  .set_mctrl      = altera_uart_set_mctrl,
  .start_tx       = altera_uart_start_tx,
  .stop_tx        = altera_uart_stop_tx,
  /* ... */
};
```

extended uart_port

```
struct altera_uart {
  struct uart_port port;
  struct timer_list tmr;
  unsigned int sigs;
  unsigned short imr;
};
```

altera_uart_driver

```
static struct uart_driver altera_uart_driver = {
.owner         = THIS_MODULE,
.driver_name   = DRV_NAME,
.dev_name      = "ttyAL",
.major         = SERIAL_ALTERA_MAJOR,
.minor         = SERIAL_ALTERA_MINOR,
.nr            = CONFIG_SERIAL_ALTERA_UART_MAXPORTS,
.cons          = ALTERA_UART_CONSOLE,
};
```

```
static struct altera_uart altera_uart_ports[MAXPORTS];
```

linux/drivers/tty/serial/altera_uart.c

init routines

register the driver

```
static int __init altera_uart_init(void)
{
  int rc;

  rc = uart_register_driver(&altera_uart_driver);
  if (rc)
    return rc;
  rc = platform_driver_register(&altera_uart_platform_driver);
  if (rc) {
    uart_unregister_driver(&altera_uart_driver);
    return rc;
  }
  return 0;
}
```

unregister the driver

```
static void __exit altera_uart_exit(void)
{
  platform_driver_unregister(&altera_uart_platform_driver);
  uart_unregister_driver(&altera_uart_driver);
}

module_init(altera_uart_init);
module_exit(altera_uart_exit);
```

the __init macro tells the compiler to put that function in the .init section (also __init_data and __exit)

I/O Memory Allocation and Mapping

allocating and mapping the UART registers

```
static int __devinit altera_uart_probe(struct platform_device *pdev)
{
  struct uart_port *port;
  struct resource *res_mem;
  int i = pdev->id;
  /* ... */
  port = &altera_uart_ports[i].port;
  res_mem = platform_get_resource(pdev, IORESOURCE_MEM, 0);
  if (res_mem)
    port->mapbase = res_mem->start;
  else if (platp->mapbase)
    port->mapbase = platp->mapbase;
  /* ... */
  port->membase = ioremap(port->mapbase, ALTERA_UART_SIZE);
  /* ... */
  port->ops = &altera_uart_ops;
  /* ... */
  uart_add_one_port(&altera_uart_driver, port);
  /* ... */
  return 0;
}
emap is needed because we need the kernel provides page tables to access it.
```

There is also a pairing remove() method, that undoes what probe () did (remove the port, etc.). This methods are part of the struct platform_driver (include/linux/platform_device.h)
The __dev_init macro says that this is not a hot-pluggable device

read/write operations in the device registers

read a memory-mapped register

```
static u32 altera_uart_readl(struct uart_port *port, int reg)
{
  return readl(port->membase + (reg << port->regshift));
}
```

write a memory-mapped register

```
static void altera_uart_writel(struct uart_port *port, u32 dat, int reg)
{
  writel(dat, port->membase + (reg << port->regshift));
}
```

read/write operations in the device registers

read a 32-bit value

```
#define readl(c)          ({ u32 __v = readl_relaxed(c); __iormb(); __v; })
#define readl_relaxed(c)  ({ u32 __r = le32_to_cpu((__force __le32) __raw_readl((c))); __r; })
#define __raw_readl(a)   ( __chk_io_ptr(a), *(volatile unsigned int __force   *)(a))
```

Check out how the read macros

- introduce a memory barrier (avoid reordering of instructions),
- takes care of endianness
- use the *volatile* keyword to prevent the compiler from register caching.

I/O regions should already configured as noncached and nonbuffered memory by the kernel in init code, to avoid accessing stale data from cache.

Interrupt handling

rx/tx interrupt service routine

```
static irqreturn_t altera_uart_interrupt(int irq, void *data)
{
  struct uart_port *port = data;
  unsigned int isr;

  isr = altera_uart_readl(port, ALTERA_UART_STATUS_REG) & pp->imr;
  spin_lock(&port->lock);
  if (isr & ALTERA_UART_STATUS_RRDY_MSK)
    altera_uart_rx_chars(pp);
  if (isr & ALTERA_UART_STATUS_TRDY_MSK)
    altera_uart_tx_chars(pp);
  spin_unlock(&port->lock);

  return IRQ_RETVAL(isr);
}
```

attach UART with interrupt vector

```
static int altera_uart_startup(struct uart_port *port)
{
  int ret;

  ret = request_irq(port->irq, altera_uart_interrupt, 0, DRV_NAME, port);
  spin_lock_irqsave(&port->lock, flags);
  /* Enable RX interrupts now */
  pp->imr = ALTERA_UART_CONTROL_RRDY_MSK;
  writel(pp->imr, port->membase + ALTERA_UART_CONTROL_REG);
  spin_unlock_irqrestore(&port->lock, flags);

  return 0;
}
```

GCC hacks in the Linux kernel

- inline, deprecated, used and const ... functions
- Statement expressions (i.e. the ({ and }) constructs).
- Declaring attributes of a function / variable / type (__attribute__)
- typeof
- Zero length arrays
- Macro varargs
- Arithmetic on void pointers
- Non-Constant initializers (automatic variables)
- Assembler Instructions (not outside arch/ and include/asm/) asm ()
- Function names as strings (__func__).
- __builtin_constant_p()

Object-oriented design tecniques used in the kernel (1)

Inheritance from a base class by embedding it as a first member.

extended uart_port

```
struct altera_uart {
  struct uart_port port;
  struct timer_list tmr;
  unsigned int sigs;
  unsigned short imr;
};
```

The C standard requires that no padding is introduced by the compiler at the start of a struct.

Object-oriented design tecniques used in the kernel (2)

Example showing the use of the container_of().

container_of macro

```
static unsigned int altera_uart_get_mctrl(struct uart_port *port)
{
  struct altera_uart *pp = container_of(port, struct altera_uart, port);
  unsigned int sigs;

  sigs = (altera_uart_readl(port, ALTERA_UART_STATUS_REG) &
    ALTERA_UART_STATUS_CTS_MSK) ? TIOCM_CTS : 0;
  sigs |= (pp->sigs & TIOCM_RTS);

  return sigs;
}
```

Access to implementation-specific data, like a derived class.

Object-oriented design tecniques used in the kernel (3)

Method Dispatch, using virtual function tables

virtual method table

```
struct uart_ops {
  unsigned int  (*tx_empty)(struct uart_port *);
  void          (*stop_tx)(struct uart_port *);
  void          (*start_tx)(struct uart_port *);
  void          (*send_xchar)(struct uart_port *, char ch);
  void          (*stop_rx)(struct uart_port *);
/* ... */
};
```

structure which contains only function pointers where the first argument of each is a pointer to some other structure (the object type) which itself contains a pointer to this vtable