I^2C Device on Linux

Roy Ratcliffe[1,*]

Abstract

The integration of the I^2C protocol within embedded systems has become a standard practice for facilitating communication between central processing units and external peripherals. This article explores the implementation of an I^2C wrapper library within SWI-Prolog, leveraging Linux's capabilities to provide robust access to I^2C devices through a user-friendly interface. By mapping the I^2C communication model to Unix file descriptors, the library simplifies operations such as opening devices, querying functionalities, and executing read/write commands in a Prolog environment. Notably, the absence of a separate close operation is justified; the library adopts a lazy approach that relies on garbage collection for descriptor management. This methodology reflects typical usage scenarios in embedded applications where a persistent I^2C connection is maintained throughout the service's lifecycle. The article outlines the design, functionality, and potential applications of the Prolog pack, providing developers with an efficient tool for I^2C interaction in embedded Linux systems.

Keywords: Embedded, C, Prolog

1. Introduction

I^2C is a standard ubiquitous protocol for communication within embedded systems between a central processing unit and its external peripherals. Most embedded chip-sets incorporate I^2C communication hardware and embedded vendors include driver-level software within their Hardware Abstraction Layer for efficiently handling low-level transfers over I^2C. The protocol clocks bits over two wires *SDA* and *SCL*, serial data and serial clock respectively. Some embedded systems refer to the protocol as a "two-wire" interface for this reason.

But what about Linux? Though arguably not an embedded operating system, it can be found in embedded scenarios with sufficiently powerful cores equipped with sufficient amounts of memory. Embeddable distributions of Linux typically provide kernel drivers that map the two-wire I^2C write-read protocol to standard Unix file descriptor accessors. The developer opens a "character device" and uses standard writes and reads to perform transfers. I^2C devices appear at `/dev/i2c-` followed by a number to identify the controller channel.

In object modelling terms, the "descriptor" model of input-output for Unix appears below, Figure 1. A "descriptor" acts as a connector between a user-land program and the kernel. Linux maps an I^2C connection to a Unix descriptor. The *input-output control* method allows for arbitrary command requests. The Linux kernel driver uses this interface to configure the channel or to perform advanced transfers; user programs configure the I^2C slave address using this generalised interface.

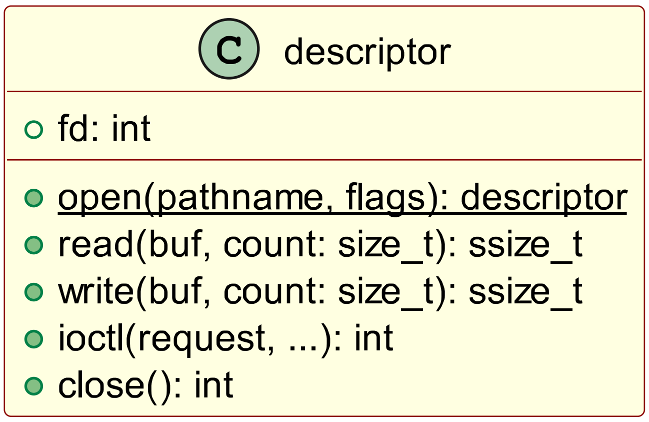
C
descriptor
fd: int
open(pathname, flags): descriptor
read(buf, count: size_t): ssize_t
write(buf, count: size_t): ssize_t
ioctl(request, ...): int
close(): int

Figure 1: Unix descriptor "class" model. This is a conceptual depiction. The *open* method appears in the model as a static answering a *descriptor* instance. The C library answers the underlying *file descriptor* handle, in practice, and the instance methods receive the handle as the first argument.

[*] Corresponding author
Email address: `roy@ratcliffe.me` (Roy Ratcliffe)
[1] See more hackery at GitHub.

It sounds simple. Is it? The following sections develop a simple wrapper library for SWI-Prolog [1]. Find the completed sources for the Prolog pack on GitHub and at the SWI-Prolog add-ons.

2. Prolog Pack

To test out the interface, a "foreign" library for Prolog will wrap the descriptor model. The pack will provide the following I^{2}C predicates.

- `i2c_open(+Dev, -I2C)` to open a device.
- `i2c_funcs(+I2C, ?Funcs)` to query functionality.
- `i2c_slave(+I2C, +Addr)` to configure the slave address.
- `i2c_write(+I2C, ++Bytes, ?Actual)` to write bytes.
- `i2c_read(+I2C, +Expected, ?Bytes)` to read bytes.

The predicate descriptions show the argument modes. Note the double plus when writing *Bytes*, a list of integer terms. Also note, the read operation has partial ground *Bytes*. Reading performs a complete read using a stack-based buffer and then unifies the resulting octets with the argument. The read succeeds when the buffer contents successfully unify with the argument. In other words, reading can check against some fully complete or partially complete expectations about the response.

Notice that the predicates exclude a close operation.

2.1. No Close

The pack does not provide an `i2c_close` predicate. That may seem strange. The developer can open a device but not close it. It takes a lazy approach instead. The open descriptor only closes when the garbage collector releases the I^{2}C blob. This carries with it one disadvantage: holding the descriptor open longer than necessary.

Typical usage does not require eager closing, however. The typical embedded scenario keeps its I^{2}C connection open indefinitely. The connection and its open file descriptor belong to a service that operates continuously. The open descriptor only needs to close when the service ends. The garbage collector releases the device blob on program termination. Linux itself will close the descriptor when its owner process dies.

2.2. Device Blob

In essence, the Prolog pack provides a "blob" for an I^{2}C descriptor. The code extract below lists the C code that defines the blob and illustrates how it unifies a *Term* variable with a file descriptor fd integer. In Prolog's world, the blob exists as an atom with some invisible, opaque state.

```c
PL_blob_t i2c_dev_blob_type =
{ .magic = PL_BLOB_MAGIC,
  .name = "i2c_dev",
  .release = release_i2c_dev,
  .write = write_i2c_dev,
};

int unify_i2c_dev(term_t Term, int fd)
{ struct linux_i2c_dev *blob = PL_malloc(sizeof(*blob));
  (void)memset(blob, 0, sizeof(*blob));
  blob->fd = fd;
  return PL_unify_blob(Term, blob, sizeof(*blob), &i2c_dev_blob_type);
}
```

Opening an I^{2}C device becomes a simple process: opening the device, turning a device number into a path and asking the kernel to open it for read-write access.

```c
foreign_t i2c_open_2(term_t Dev, term_t I2C)
{ int dev;
  char pathname[PATH_MAX];
  int fd;
  if (!PL_get_integer(Dev, &dev)) PL_fail;
  Ssnprintf(pathname, sizeof(pathname), "/dev/i2c-%d", dev);
  if (0 > (fd = open(pathname, O_RDWR))) return i2c_errno("open");
  return unify_i2c_dev(I2C, fd);
}
```

The final `return` statement calls the blob unification function listed previously.

3. Usage

It works well. The following section takes the pack for a 'drive' by talking to a PCA9685.

3.1. NXP Semiconductor's PCA9685

What is the PCA9685? To paraphrase NXP Semiconductor,

> "The PCA9685 is a 16-channel LED controller that operates via I^{2}C-bus, specifically designed for Red/Green/Blue/Amber (RGBA) colour backlighting applications. Each LED output features its own 12-bit resolution, equating to 4096 brightness levels, managed by a dedicated PWM (Pulse Width Modulation) controller. This controller can be programmed to operate at frequencies ranging from a typical 24 Hz to 1526 Hz, with the duty cycle adjustable between 0% and 100%, allowing for precise control over brightness levels. Notably, all outputs maintain the same PWM frequency, ensuring consistency in lighting performance."

```prolog
%!  led_adr(?OnOff, ?LH, ?Adr0) is nondet.
%
%   Adr0 is the PWM control's on-off low-high relative register address
%   offset, between 0 and 3 inclusive.

led_adr(on,  l, 0x00).
led_adr(on,  h, 0x01).
led_adr(off, l, 0x02).
led_adr(off, h, 0x03).

%!  reg_adr(?Reg, ?Adr) is nondet.
%
%   Maps the entire PCA9685 register file.

reg_adr(mode(Mode), Adr) :-
    between(1, 2, Mode),
    Adr is 0x00 + Mode - 1.
reg_adr(subadr(SubAdr), Adr) :-
    between(1, 3, SubAdr),
    Adr is 0x02 + SubAdr - 1.
reg_adr(allcalladr, 0x05).
reg_adr(led(LED, OnOff, LH), Adr) :-
    between(0, 15, LED),
    led_adr(OnOff, LH, Adr0),
    Adr is 0x06 + Adr0 + (0x04 * LED).
reg_adr(led(all, OnOff, LH), Adr) :-
    led_adr(OnOff, LH, Adr0),
    Adr is 0xfa + Adr0.
```

```
reg_adr(pre(scale), 0xfe).
reg_adr(test(mode), 0xff).
```

3.1.1. *Reading the first mode register*

This becomes a straightforward clause sequence: open the device $Dev = 1$ by number, configure the slave address $Addr = 40_{16}$.

```
    i2c_dev(Dev),
    i2c_open(Dev, I2C),
    pca9685_addr(Addr),
    i2c_slave(I2C, Addr),
    ai(I2C),
    % Write 00 to the Control Register.
    % It determines access to the other registers.
    i2c_write(I2C, [0x00]),
    i2c_read(I2C, [Byte]).
```

3.2. *Enabling the control register's auto-increment (AI) feature*

The PCA9685 disables the control register's auto-increment (AI) function on restart. It needs enabling. Enable AI idempotently by reading the $Mode1$ register. If the register has zero in the fifth bit, perform a write-back with the fifth bit set. Retain all other bits.

```
ai(I2C) :-
    reg_adr(mode(1), Adr),
    rd(I2C, Adr, [Mode1]),
    (   Mode1 /\ 2'0010_0000 =\= 0x00
    ->  true
    ;   Mode1_ is Mode1 \/ 2'0010_0000,
        wr(I2C, Adr, [Mode1_])
    ).
```

Non-AI mode is less useful. Arguably, AI-enabled *should* be the default. If the Control Register retains its content, the device accesses the same register at every read operation.

4. Conclusions

The Linux kernel driver makes I^2C easy. The half-duplex protocol fits neatly beneath the standard Unix file descriptor access interface: `open`, `ioctl`, `read`, `write`. It also supports a more sophisticated interface using message buffers—not covered here.

The pack presents the simplest connection to an I^2C controller. It does not use `i2c_msg` but instead leaves that step to future iterations. That would eliminate task switching between transfer segments, however. User-land would only resume at the end of all transfer segments. The kernel handles the entire transfer in such a case. The model based on Unix descriptors gives a simple level of access but not one without limitations. It adds some latency between reads and writes. Descriptor read-write is not an optimal implementation by any means but proves adequate for most requirements.

References

[1] J.-C. Rohner, H. Kjellerstrand, Prolog for scientific explanation, in: D. S. Warren, V. Dahl, T. Eiter, M. V. Hermenegildo, R. Kowalski, F. Rossi (Eds.), Prolog: The next 50 Years, Springer Nature Switzerland, pp. 372–385. doi:10.1007/978-3-031-35254-6_30.
URL https://doi.org/10.1007/978-3-031-35254-6_30