УДК 621.3.049.771.14

Прототипирование кода драйверов OS Linux в пространстве пользователя с использованием языка высокого уровня lua

А.В. Андрианов

ЗАО НТЦ «Модуль», andrew@ncrmnt.org

***Аннотация* — В статье рассмотрен метод быстрого прототипирования драйверов для OS Linux с использованием языка высокого уровня lua полностью в пространстве пользователя, применимый на этапе ПЛИС-прототипирования; делаются выводы относительно спектра задач, допускающих этот способ отладки.**

***Ключевые слова* — linux, kernel, lua, FPGA, prototype.**

I. Введение

При разработке СБИС одной из наиболее трудозатратных задач является разработка программного обеспечения, которое впоследствии войдет в состав пакета ПО, поставляемого вместе с микросхемой разработчикам решений. Для сокращения времени выхода продукта на рынок и для упрощения процесса верификации отдельных блоков СБИС разработка системного ПО начинается на ранних этапах с ПЛИС-прототипирования отдельных блоков.

ПЛИС-прототипирование обычно проводится на оборудовании, главными потребителями которого являются разработчики. В отличие от коммерческих потребительских устройств тут нет жестких требований по времени загрузки и производительности, которые бы дали конкурентные преимущества на рынке. Более того, многие отладочные функции ядра OS Linux, которые включают разработчики на этапе компиляции для удобства отладки, существенно замедляют процесс загрузки и работы.

В виде примера можно привести время загрузки (от подачи питания до командной строки OS Linux) отладочного набора ARM Realview EB, составляющее порядка одной минуты. При включении отладочного вывода различных подсистем ядра это время существенно увеличивается и может достигать пяти минут.

Перезагрузки на этом этапе неизбежны, как по причине выявленных проблем с аппаратным обеспечением, так и по причине ошибок программистов, которые часто наблюдаются на этом этапе. Таким образом, даже при максимальной автоматизации процесса программист тратит существенное количество рабочего времени, ожидая окончания процесса загрузки системы. Сборка драйвера в виде динамически загружаемого модуля позволяет лишь незначительно сократить количество перезапусков, так как большая часть типичных ошибок переводит ядро OS Linux в нерабочее состояние и влечет за собой необходимость перезагрузки.

На реальном примере, драйвере для криптографического IP блока, на начальных этапах требовалось до 30 перезагрузок ПЛИС-макета в день.

Потому главной целью ставилась разработка метода, который может значительно сократить количество необходимых перезапусков системы в процессе разработки и отладки.

Разрабатываемый на этом этапе драйвер можно считать только прототипом, так как он содержит избыточное количество отладочных функций, которые никак не используются при работе на реальной СБИС, а только ухудшают читаемость кода. Во многих случаях такой драйвер переписывается с нуля после выхода микросхемы для устранения недочетов или в силу слишком большого числа отличий между прототипом и реальной СБИС.

Для сокращения времени разработки и отладки на данном этапе необходимо создавать прототип драйвера. Для его создания предлагается использовать интерпретируемый язык программирования высокого уровня lua, запущенный в пространстве пользователя, библиотеку для языка lua, содержащую набор функций, облегчающих отладку и минимальный, общий для всех случаев, драйвер (lprobe).

Выбор языка lua обусловлен его удобством интегрирования в приложения, малым количеством зависимостей, а также очень низким порогом вхождения. Это позволяет использовать его разработчикам аппаратуры для верификации разрабатываемых ими блоков.

Lua уже используется для прототипирования драйверов в ядре операционной системы NetBSD. В случае NetBSD, интерпретатор lua запускается в пространстве ядра и имеет привязки к основным API вызовам ядра. К сожалению, для OS Linux подобная система прототипирования драйверов отсутствует.

Для написания полноценного драйвера устройства на lua необходимо:

- Иметь доступ к диапазону физической памяти, где располагаются регистры устройства.
- Наличие возможности обработки прерываний от устройства.
- Выделять один или несколько буферов физически непрерывной памяти для выполнения DMA операций.

На сегодняшний день существуют два штатных механизма в ядре OS Linux, которые позволяют пользовательскому процессу получить доступ к физической памяти: devmem и UIO.

Драйвер devmem обеспечивает доступ ко всему физическому адресному пространству через специальное устройство /dev/mem, и не представляет каких-либо функций для обработки прерываний.

UIO (Userspace I/O) является более сложным механизмом ядра и позволяет разработку полноценных драйверов полностью в пространстве пользователя. В первую очередь, это необходимо для приложений, где требуется высокая производительность и нужно снизить накладные расходы на копировании данных из памяти приложения в пространстве пользователя в пространство ядра (zero-copy). Отличительной чертой UIO является возможность работы с прерываниями. Текущая реализация драйвера уровня ядра lprobe использует подсистему UIO, добавляя набор дополнительных проверок.

Драйвер уровня ядра обеспечивает приложению доступ к физической памяти отлаживаемого блока, транслирует прерывания в пространство пользователя, а также позволяет выделять буфера физически непрерывной памяти из lua окружения.

Драйвер старается максимально отслеживать некорректное поведение IP блока и, по возможности, сообщать разработчику о типичных ошибках в работе аппаратуры:

1) В случае, когда dma операция затронула данные за границами рабочего буфера, например, при записи данных, не кратных размеру слов на данной архитектуре.
2) В случае «interrupt storm», когда флаг прерывания от устройства выставляется непрерывно.

Приложение пространства пользователя обменивается данными с драйвером через специальное символьное устройство в /dev. Текущая реализация компилируется в виде расширения языка lua и вместе со вспомогательной библиотекой на языке lua представляет окружению простой высокоуровневый интерфейс для работы с устройством.

II. Модификация ядра OS Linux

Для использования данного метода, необходимо внести набор изменений в ядро OS Linux. Набор изменений включает себя patch файл, содержащий сам драйвер lprobe. В случаях, когда на системе не поддерживается технология Flatterned Device Tree, необходимо в коде архитектурно-специфичной инициализации объявить структуру platform_device. В ней необходимо задать диапазон используемой устройством физической памяти, а также номера прерываний.

При использовании технологии Flatterned Device Tree для подключения lprobe достаточно объявить эти данные в .dts файле и задействовать его при запуске системы.

При корректной регистрации устройства одним из вышеуказанных методов в журнале загрузки ядра можно будет увидеть сообщения вида:

[20.444057] lprobe: attaching probe 'mcrypto'

Драйвер поддерживает одновременную регистрацию и использование любого количества устройств.

III. Использование в пространстве пользователя

После успешной регистрации устройства ядром операционной системы доступен набор расширений языка lua для работы с устройствами /dev/lprobeX

Для этого их необходимо кросс-компилировать под целевую платформу. Для удобства работы пакет библиотек пространства пользователя допускает компиляцию в статически скомпонованный бинарный файл, что избавляет от необходимости интеграции пакета ПО lprobe в существующие системы сборки (buildroot, OpenEmbedded, и т.п.). Достаточно просто скопировать один исполняемый файл в корневую файловую систему.

Порядок работы с устройством представлен ниже.

A. *Задание карты памяти*

Первое, что рекомендуется сделать - задать адреса отдельных регистров IP блока и функциональное назначение отдельных битов в специальном формате.

Создание такого формата конфигурации регистровой карты преследует цель уменьшить количество ошибок, которые могут появиться на стадии отладки IP блока в случаях, когда будут добавляться или удаляться регистры.

Эти данные будут учтены при выводе отладочных сообщений. На основе этого описания можно потом будет генерировать как заголовочный файл языка C, так и заголовочный файл для языка Verilog HDL утилитами reg2c и reg2verilog, соответственно. Этот шаг необязателен, но существенно повышает читаемость отладочных сообщений.

Пример задания карты памяти IP блока в виде lua таблицы приведен ниже:

```
local map = {
  [0x0] = {
    name = "control",
    desc = "Control Register",
```

```
    bits = {
          ["0:0"] = {
            name = "power",
            desc = "Power control",
            values = {
              [0] = "off",
              [1] = "on"
            },
            ["2:1"] = {
              name = "mode",
              desc = "Mode of operation",
              values = {
                    [0] = "read",
                    [1] = "write",
                    [2] = "reserved",
                    [3] = "reserved",
              },
            },
          },
    },
  },
  [0x4] = {
    name = "data_sz",
    desc = "Data xfer size",
    bits = {
          ["16:0"] = {
            name = "datalen",
            desc = function(io)
              if ("read" == io.read("control.mode")) then
                    return "Bytes read"
              else
                    return "Bytes written"
              end
            end
          }
    }
  },
}
return map;
```

Подобное описание допускает применение так называемого «семантического сахара» языка lua: любое из полей описания 'desc' можно объявить как функцию, которая должна вернуть строку с текущим описанием. Это применимо в случаях, когда назначение и, соответственно, описание значения конкретных битов зависит от значений, установленных в других регистрах блока.

Так как на данной стадии проектирования в регистровую карту блока могут вноситься изменения, то в операциях чтения и записи можно использовать, помимо адресов, *путu,* в том числе и до отдельных битов. Для наглядности в примере выше можно прочитать данные из регистра datalen по пути data_sz.datalen.

B. Создать lua сценарий, который будет содержать код драйвера

Для начала работы достаточно создать lua сценарий и подключить библиотеку lprobe. После этого можно открыть одно или несколько устройств lprobe вызовом lprobe_open. Открывать устройства можно как по абсолютному пути, например /dev/lprobe0, так и по имени указанному в platform_device/DeviceTree.

Пример:

```
crypto,err = lprobe_open(«/dev/lprobe0»);
crypto,err= lprobe_open(«iowriter»);
```

Если устройство было открыто успешно, то функция вернет объект, через который можно будет работать с этим устройством, а переменная err будет иметь значение nil. Если устройство не существует или произошла какая-то другая ошибка, то переменная crypto в примере выше будет иметь значение nil, а переменная err будет содержать строку с описанием ошибки.

Дальнейшая работа с устройством подразумевает последовательность записи и чтения регистров устройства и обработку прерываний.

Обработка прерываний может задаваться полностью в пространстве пользователя, и выглядит как *сопрограмма* на языке lua.

Технологию сопрограмм часто называют общей многопоточностью. Сопрограмма Lua представляет собой независимый поток выполнения. Несмотря на это, в отличие от потоков в традиционных многопоточных системах, сопрограмма может приостановить свое выполнение только в результате явного вызова функции yield.

В приведенном ниже примере демонстрируется задание обработчика прерываний.

```
p = lprobe_open("mcrypto");
irq = p:get_irq("interrupt");

function irq_handler(irq)
  print("IRQ Arrived\n!");
end

irq.setHandler(irq);
lprobe_wait();
```

Библиотека языка lua собирает статистику прерываний, таких как количество пришедших прерываний и время между прерываниями. При достижении предельно допустимой частоты прерываний (irq_threshold) драйвер уровня ядра маскирует прерывание и уведомляет об этом разработчика.

IV. Дополнительные инструменты

В комплект библиотеки входит большой набор функций, созданных для удобства отладки. В этой части приведены три наиболее интересных, на взгляд автора, инструмента.

A. *Мониторы.*

Монитором здесь называется отдельный поток, который будет следить за изменениями состояния отдельных битов в регистре и сообщать о них

Пример:

```
mon = regs:monitor({ 0x40, 0x44, 0x80 });
mon = regs:monitor(«status», «control»);
mon:start();
...
mon:stop();
```

Мониторинг состояния выполняется отдельным потоком вне окружения lua. Это обеспечивает максимально возможную скорость опроса. При необходимости можно ограничить частоту, задав промежуток времени между опросами.

По умолчанию вывод информации производится в стандартный поток ввода/вывода, но есть возможность и перенаправить его в файл.

```
mon.toFile(«path/to/log»);
```

B. *Работа с DMA*

Отдельный набор функций нацелен на работу с dma буферами. Эти функции позволяют выделять физически-непрерывную память для DMA на стороне ядра и представляют собой небольшую прослойку, которая позволяет отслеживать наиболее частые аппаратные ошибки при работе с DMA, такие как переход границ буфера при работе с неполными словами.

C. *Интерактивный режим*

Интерактивный режим включается вызовом из сценария функции lprobe_interactive() и позволяет при возникновении внештатной ситуации предоставить разработчику интерактивную командную строку с возможностью быстрого доступа ко всем регистрам устройства.

Заключение

Описанный в статье метод позволяет создавать полноценные драйвера в пространстве пользователя на языке высокого уровня и ограничивает количество необходимых перезапусков системы исключительно случаями возникновения аппаратных проблем, что может существенно сократить временные затраты на этой стадии проектирования СБИС.

Область применимости данного метода ограничивается случаями отладки драйверов устройств вне основных подсистем ядра OS Linux и подготовки тестовых сценариев, воспроизводящих найденные ошибки. Прототипы драйверов, написанные на lua, не следует использовать в конечном продукте, а также для тестов производительности, так как из-за избыточности многих проверок и накладных расходов, связанных с работой в пространстве пользователя, может серьезно уменьшиться производительность.

Литература

[1] Ierusalimschy, Roberto; De Figueiredo, Luiz Henrique; Celes, Waldemar: Lua 5.1 Reference Manual. Lua.org, Rio de Janeiro. 2006.

[2] Ierusalimschy, Roberto; De Figueiredo, Luiz Henrique; Celes, Waldemar: The evolution of an extension language: a history ofLua. SBLP 2001 invited paper. URL: http://www.lua.org/history.html. (дата обращения: 05.05.2014).

[3] Marc Balmer. Lua as a Configuration And Data Exchange Language // FOSDEM 2013, URL: http://www.netbsd.org/~mbalmer/lua/ (дата обращения: 05.05.2014).

[4] URL: http://github.com/nekromant/lprobe (дата обращения: 06.02.2014).