Разработка переносимого драйвера устройств для встраиваемых систем

Лобода А.А., Орел ГТУ

Множество устройств, предназначенных для расширения функций компьютера, как правило, оснащается драйверами под операционную систему (ОС) Windows, доминирующей на рынке домашних ПК. Нередко производитель оборудования уделяет внимание и ОС Linux.

Отдельным типом ОС являются встраиваемые системы, позволяющие решать задачи сбора сигналов с датчиков, управления ресурсами критических по времени объектов, обеспечения функционирования сложных систем цифровой обработки сигналов.

Многообразие встраиваемых ОС заставляет разработчика ПО соблюдать правила написания кода, чтобы при портировании драйвера под очередную ОС не приходилось проделывать всю работу с нуля и при этом сам код оставался «читабельным». Для встраиваемых ОС и операционных систем реального времени (ОСРВ) традиционным подходом является модель кросс-разработки, когда инструментальная ЭВМ использует одну ОС (обычно это Windows), а целевая - другую (рис. 1).

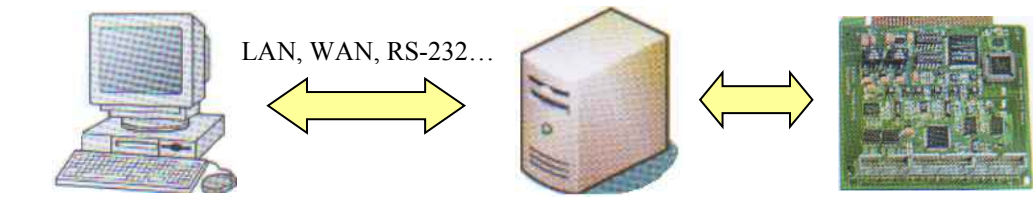

Рис. 1. Традиционная модель разработки

Для написания драйвера необходимо знать о следующих понятиях:

- процессах и нитях, их различиях в Windows- и Unix-системах;
- механизмах синхронизации;

- управлении памятью — режиме пользователя и режиме ядра, понятии о выгружаемой памяти;
- отличиях между пространством ввода-вывода и памятью ввода-вывода;
- приоритетах и механизмах обработки прерываний;
- режимах прямого доступа к памяти.

Под разработкой переносимого драйвера понимается создание такого проекта на языке Си, который бы мог компилироваться не только разными компиляторами (GCC, Microsoft C++), но и для разных архитектур процессоров, например, ОС QNX работает на MIPS, StrongArm, PowerPC, SH-4, ARM, SH-4, Intel' XScale™ Microarchitecture и x86.

Кроме возможности кросс-платформенного программного обеспечения работать на разных типах компьютеров, портируемость хорошо влияет на сам код. Ошибки, которые не распознает один компилятор, могут быть обнаружены другим. Кроме того, общий исходный код для различных операционных систем увеличивает устойчивость системы, надёжность кода и возможность повторного его использования. Следует учесть, что тестирование приложения на разных платформах помогает в отладке — трудновоспроизводимая ошибка может быть обнаружена на другой платформе.

Отметим некоторые «общие» черты ОС Windows, Linux, QNX:

- компиляторы Си существуют для всех платформ и архитектур;
- обращение к драйверу происходит так же, как и работа с обычными файлами, следовательно, применимы стандартные функции open(), read(), write() и другие;
- драйверы имеют схожую модель взаимодействия с системой (рис. 2).

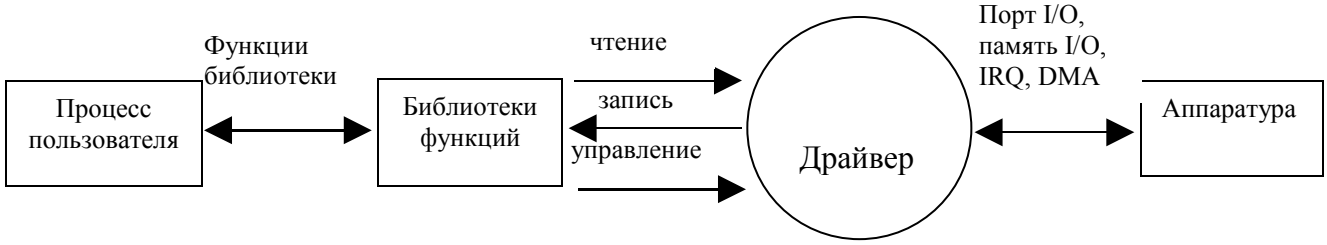

Рис. 2. Общая модель взаимодействия драйвера с системой

Основная проблема при разработке драйверов - различие в структуре и основных принципах создания драйверов для различных видов ОС (Windows, Linux, QNX).

Внешне образ драйвера представляется трёхсторонней моделью. Одна сторона общается с программой пользователя, которой необходимо получить доступ к управляемой драйвером аппаратуре. Механизм взаимодействия основан на методах open(), read(), write(), iocontrol() и т.д. Другая сторона взаимодействует с ядром через системные вызовы для собственных нужд драйвера. Это выделение памяти, установка callback-обработчиков событий ядра и др. Третья сторона ведёт диалог с аппаратной частью. Для ОС Windows основным пакетом и первоисточником информации для разработки драйверов является Microsoft DDK (Driver Development Kit), который содержит в своём составе различные утилиты, примеры драйверов и компилятор.

Механизм общения драйвера с ядром ОС в Linux и Windows схожий. Следует отметить, что от версии к версии, в отличие от Windows, в ядре Linux могут сильно меняться структуры данных и системные вызовы.

При компиляции драйвера необходимы заголовочные файлы ядра Linux. Имена файлов, входящих в исходный код ядра Linux, могут различаться регистром символов. Если на инструментальной ЭВМ установлена ОС Windows, может получиться конфликт имён, так как Windows не различает регистр символов в названиях файлов. Поэтому модель кросс-разработки в данном случае менее удобна, и компиляцию проекта лучше делать на целевой ЭВМ.

По-другому происходит разработка драйвера для операционной системы QNX. QNX — это операционная система реального времени, обеспечивающая предсказуемость и гарантированное время отклика при любом внешнем воздействии. В этой системе, основанной на микроядре, другая идеология построения драйверов (обычно употребляют синоним — ресурс-менеджер). Модель взаимодействия ресурс-менеджера с системой представлена на рис. 3.

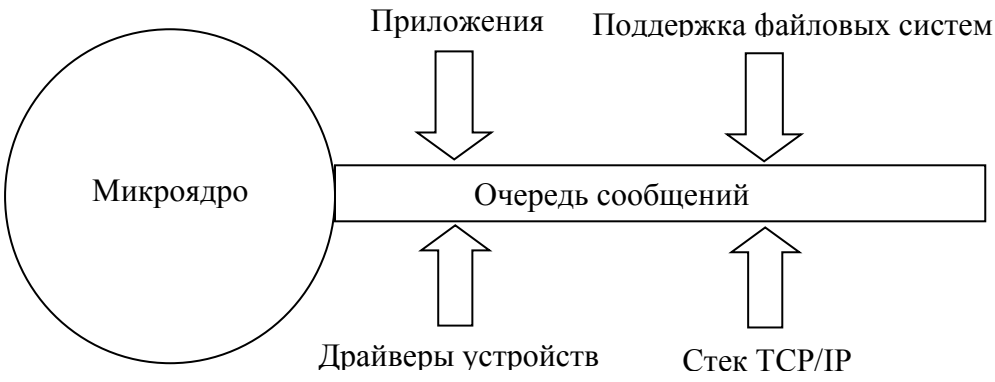

Рис. 3. Модель взаимодействия драйвера с системой

Отличие внутреннего устройства драйвера QNX от других систем, построенных на монолитном ядре, это то, что ресурс-менеджер является пользовательским процессом системы. Ресурс-менеджер не является ни частью ядра, ни модулем, это сервер, принимающий сообщения от других программ и переводящий их на язык обслуживаемого им устройства.

QNX Momentics содержит в себе мощный набор программного инструментария для разработки целевого программного обеспечения реального времени. Среда разработки включает в себя Application Builder — средство для разработки графических приложений под Photon, представляющее из себя RAD-систему (Rapid Application Development — быстрая разработка приложений), схожую с Borland Delphi или Microsoft Visual Basic. Среда Momentics создана на базе технология Eclipse и поддерживает кросс-платформенную разработку под Windows, Linux и Solaris.

Очевидный плюс QNX в том, что это система «из коробки», то есть весь инструментарий, все программы и утилиты, требующиеся для полноценной и комфортной работы, уже включены в дистрибутив системы.

Кроме различия общих принципов построения ОС (что ведет за собой взаимную несовместимость драйверов под разные системы), существуют также и другие проблемы разработки драйверов. Например, проблема размера предопределенных типов данных и проблема порядка байтов.

Для различных типов процессоров машинное слово имеет свой предопределённый размер. Изначально предполагалось, что Си-программисты будут считать размер типа int соответствующим размеру машинного слова, то есть оптимальному для процессора. Но на сегодняшний момент большинство компиляторов приняло за аксиому равенство sizeof(int) = 4. К сожалению, стандарт ANSI не предусматривает чётких правил для размеров фундаментальных типов данных языка Си, поэтому можно предположить, что

- sizeof(int) = sizeof(long) = sizeof(void *)
- sizeof(int) = размер машинного слова
- sizeof(int) = 4

Насчитывается несколько разновидностей моделей, определяющих размеры основных типов данных языка Си (табл. 1). Большинство Unix подобных систем (Linux, QNX, ...) поддерживает модель LP64, тогда как Microsoft использует LLP64 для 64-разрядных версий Windows.

Практически все 32-битовые системы поддерживают ILP32. Модели LP32 и ILP64 мало распространены и встречаются редко.

В таблице 2 представлены типы данных компилятора GCC в ОС Linux. Отметим, что поддерживается тип long long, чего стандарт LP64 не требует. Такие расширения компилятора для переносимого кода использовать не рекомендуется.

Таблица 1

Модели, определяющие размеры фундаментальных типов данных языка Си

Datatype	LP64	ILP64	LLP64	ILP32	LP32
char	8	8	8	8	8
short	16	16	16	16	16
_int	32	—	32	—	—
int	32	64	32	32	16
long	64	64	32	32	32
long long	—	—	64	—	—
pointer	64	64	64	32	32

Таблица 2

Размеры типов данных ОС Linux в зависимости от архитектуры ЭВМ

arch	char	short	int	long	ptr	long-long	u8	Ul6	u32	u64
1386	1	2	4	4	4	8	1	2	4	8
alpha	1	2	4	8	8	8	1	2	4	8
armv4l	1	2	4	4	4	8	1	2	4	8
ia64	1	2	4	8	8	8	1	2	4	8
m68k	1	2	4	4	4	8	1	2	4	8
mips	1	2	4	4	4	8	1	2	4	8
ppc	1	2	4	4	4	8	1	2	4	8
spare	1	2	4	4	4	8	1	2	4	8
sparc64	1	2	4	4	4	8	1	2	4	8
x86_64	1	2	-	8	8	8	1	2	4	8

Нет чёткого ответа на вопрос, в каком порядке должна передаваться цифровая информация: от младшего бита к старшему или от старшего к младшему. Процессоры, передающие первым младший разряд, называются little-endian, старший, соответственно, big-endian. Проблема порядка байтов проявляется при передаче многобайтового числа на компьютер с другой архитектурой, без соблюдения соглашений о том, какой из байтов является старшим, а какой младшим.

Удобная конфигурация стенда разработки представлена на рис. 1. По возможности Windows, Linux и QNX необходимо поставить на один жесткий диск и сделать образ системы, чтобы в случае отказа быстро восстановить рабочую среду. При написании кода часто возникают «мистические» ошибки, когда всё проверено, но программа не работает. Первый плюс такой конфигурации стенда в том, что можно быстро перегружаться между системами и выявлять место ошибки. Второй плюс заключается в том, что написание драйвера обычно идёт для нового «железа». И два одинаковых на вид устройства могут иметь совершенно разные аппаратные особенности. Поэтому для сужения возможного круга ошибок разработку и эксперименты лучше вести на одном устройстве.

Для улучшения переносимости кода рекомендуется исключить из драйвера и вынести в интерфейсную библиотеку операции с плавающей точкой и файлами. Если предполагается работать с другими устройствами, доступ к которым

предоставляет операционная система (COM-порты и т.п.), архитектура драйвера должна проектироваться с учётом этого.

В большинстве драйверов используют общие механизмы:

- задержки;
- выделение памяти;
- синхронизация;
- печать отладочных сообщений;
- доступ к портам ввода-вывода.

Важно определить, какие ресурсы будут использоваться драйвером, и создать свой интерфейс с функциями, не зависящими от операционной системы.

Разберем решение проблемы порядка байтов.

Можно предположить, что большинство программистов знакомо с x86 процессорами, которые построены на архитектуре little-endian. Покажем ошибки, которые возникают при работе приложений на процессорах big-endian. Определим массив:

```
short data [ 2 ] = { 0x0001, 0x0203 };
```

В архитектуре x86 информация в памяти разместится слева направо:

01 00 03 02

Рассмотрим код преобразования типов через указатели:

```
unsigned long x = 0x03020100;
unsigned long *ptr_x = &x;
unsigned char x_char;
x_char = *(unsigned char*)ptr_x;
```

После выполнения на little-endian процессоре xchar будет равно 0, на big-endian — трём. Выход из ситуации — использовать дополнительную переменную и возложить преобразования на компилятор:

```
unsigned long temp = *ptr_x;
x_char = (unsigned char)temp;
```

Теперь независимо от типа системы xchar будет содержать младший байт переменной temp.

Основным правилом при разработке кросс-платформенных драйверов является соответствие стандартам. Также следует избегать расширений компиляторов Borland C, GCC и др. Необходимо использовать приемы программирования, позволяющие избавиться от проблемы порядка байт. Использование стандарта ANSI языка C интерфейса POSIX и других вещей которые были стандартизированы дает неплохой шанс написанным программам запуститься на любой платформе.

Список литературы:

1. Овод М. Разработка переносимого драйвера устройств для встраиваемых систем [Текст] / Максим Овод // СТА. – 2007. – №2. – С. 94-98.
2. Финогенов К.Г. Основы разработки прикладных виртуальных драйверов [Электронный ресурс] / К.Г. Финогенов // Электронный журнал «Компьютер Пресс», 2001. www.compress.ru/article.aspx?id=10015&iid=418
3. Электронный справочник [Электронный ресурс] www.ru.wikipedia.org/wiki/Драйвер