Максим Овод

Разработка переносимого драйвера устройства для встраиваемых систем

О драйверах и операционных системах

В отличие от множества печатных и электронных изданий, освещающих вопросы разработки драйверов для различных операционных систем, материал этой статьи обозначит общие черты систем, и особое внимание будет уделено принципам и организационной структуре драйверов, которая позволит успешно «собирать» их под ОС Windows 2000/XP/XP Embedded, Linux и QNX6. Множество устройств, предназначенных для расширения функций компьютера, как правило, оснащается драйверами под ОС Windows, доминирующей на рынке домашних ПК. Нередко производитель оборудования уделяет внимание и операционной системе Linux. Мир встраиваемых систем отличает интересная особенность, какая бы ни была поставлена задача: сбор сигналов с датчиков, или управление ресурсами критических по времени объектов, или обеспечение функционирования сложных систем цифровой обработки сигналов – в каждом случае мы можем выбрать отдельную операционную систему, которая будет успешно работать. Многообразие встраиваемых операционных систем (одних операционных систем реального времени насчитывается более ста) заставляет разработчика ПО соблюдать некоторые 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 (ОС РВ) традиционным подходом является модель кросс-разработки, когда инструментальная ЭВМ использует одну ОС (обычно это «классическая» платформенная ОС типа Windows), а целевая – другую (рис. 1).

Невероятно сложно, скорее всего невозможно, написать драйвер без понимания внутреннего устройства операционной системы. Поэтому, как минимум, необходимо иметь общие представления о следующих понятиях:

- процессы и нити, их различия в Windows- и Unix-системах,
- механизмы синхронизации,
- управление памятью — режим пользователя и режим ядра, понятия о выгружаемой памяти,
- отличие между пространством ввода-вывода и памятью ввода-вывода,
- приоритеты и механизмы обработки прерываний,
- режим прямого доступа к памяти.

Стоит ли писать портируемый драйвер?

Термины «портируемый» и «переносимый» можно рассматривать по-разному. Для кого-то код, написанный для разных версий компилятора, уже считается переносимым. Под разработкой переносимого драйвера мы будем понимать создание такого проекта на языке Си, который бы мог компилироваться не только разными компиляторами (GCC, Microsoft C++), но и под разными архитектурами процессоров, например, ОС QNX работает на MIPS, PowerPC, SH-4, ARM, StrongArm, Intel® XScale™ Microarchitecture и x86.

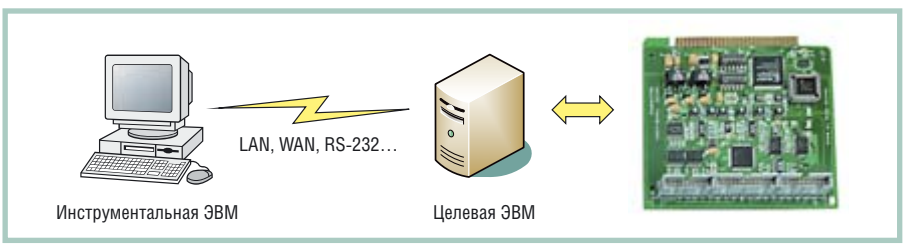

Рис. 1. Традиционная модель разработки ПО для встраиваемых систем

Кроме очевидного преимущества того, что кросс-платформенное программное обеспечение может работать на разных типах компьютеров, есть ещё один немаловажный плюс — портируемость хорошо влияет на сам код. Во-первых, ошибки, которые не распознает один компилятор, с лёгкостью могут быть обнаружены другим. Во-вторых, общий исходный код для проектов под различными операционными системами увеличивает робастность, надёжность вашего кода и возможность повторного его использования. И в-третьих, тестирование приложения на разных платформах помогает в отладке — скрытая, трудновоспроизводимая ошибка может быть легко обнаружена на другой платформе.

Казалось бы, между Windows, Linux и QNX нет ничего общего – это абсолютно разные системы с разной идеологией и организацией. Однако, взглянув поглубже, можно отметить некоторые «общие» черты, интересующие нас как разработчиков драйверов:

- ориентация на язык Си – компиляторы 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).

Заметим, что на схеме, представленной на рис. 2, в частном случае библиотека функций может отсутствовать. Стандартный приём создания переносимого приложения — спрятать непортируемый код в отдельные модули. На рис. 3 показан один из возможных вариантов переносимого драйвера, представляющий собой двухуровневую структуру. На первом уровне находится общий портируемый код, который не должен зависеть от операционной системы. На второй уровень будут добав-

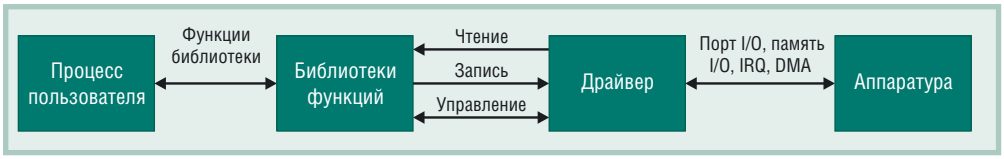

Рис. 2. Общая модель взаимодействия драйвера с системой

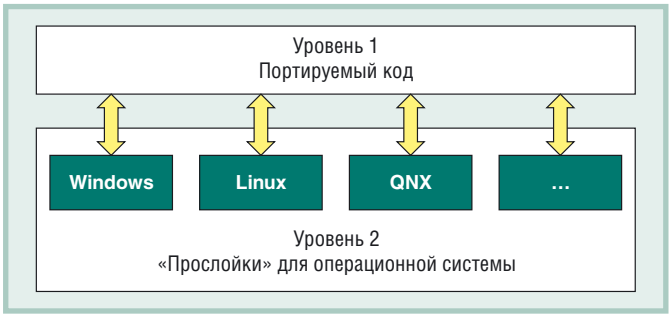

Рис. 3. Один из вариантов структуры переносимого драйвера

ляться так называемые «драйверные прослойки», выполняющие специфичную для ОС работу:

- процедуры инициализации и захвата ресурсов,
- установку обработчиков прерываний,
- регистрацию callback-функций для реакции драйвера на события ядра ОС,
- освобождение ресурсов при выгрузке драйвера и т.п.

WDM-драйверы Windows

Важно отметить, что на смену WDM-драйверам (Windows Driver Model) Microsoft предлагает WDF (Windows Driver Foundation). Одна из главных особенностей новой драйверной модели — возможность создания драйверов, работающих в пространстве пользователя, в результате чего даже серьёзная ошибка, допущенная программистом, не должна влиять на стабильность системы.

WDM-модель проектировалась как совместимая снизу вверх, то есть драйвер, написанный для Windows 2000, будет успешно работать под Windows XP/XP Embedded/2003.

Внешне образ драйвера для нас представляется трёхсторонней моделью (рис. 4). Одна из сторон общается с программой пользователя, которой необходимо получить доступ к управляемой драйвером аппаратуре. Механизм взаимодействия основан на вышеупомянутых методах open(), read(), write(), iocontrol() и т.д. Другая сторона взаимодействует с ядром через системные вызовы для собственных нужд драйвера. Это может быть, например, выделение памяти, установка callback-обработчиков событий ядра и др. И последняя, третья сторона ведёт диалог с аппаратной частью. Обо всех инструментах разработки, которые могут понадобиться, упомянуть просто не представляется возможным, и, конечно, есть необходимый минимум программ и утилит, без которых не обойтись. Для ОС Windows основным пакетом и первоисточником разработки драйверов является Microsoft DDK (Driver Development Kit), который содержит в своём составе множество полезных утилит, примеров драйверов и собственно компилятор. Создание и редактирование исходного кода удобно делать в IDE Visual Studio. Из сторонних утилит интересны программы просмотра отладочных сообщений DebugView (http://www.microsoft.com/technet/sysinternals/Miscellaneous/DebugView.mspx). Порой не обойтись без отладчика, один из лучших — это SoftICE. Огромный портал, посвящённый разработке драйверов под Windows, расположен по адресу http://www.osronline.com.

Драйверы Linux

Механизм общения драйвера с ядром операционной системы в Linux и Windows довольно схожий. Если абстрагироваться от внутреннего устройства ядра драйвера в этих системах, то можно сказать, что модель взаимодействия драйвера с системой выглядит одинаково (рис. 4). Очень важным моментом является ветка ядра Linux, так как от версии к версии, в отличие, скажем, от Windows, в ядре Linux могут сильно меняться структуры данных и сами системные вызовы. Развиваются две основные ветки — 2.4 и 2.6. Существует ряд документов (http://lwn.net/Articles/driver-porting), помогающих переходить на новую ветку 2.6. Разработку желательно вести на оригинальном стабильном ядре (vanilla source), взятом с сервера http://www.kernel.org. В этом случае оно не имеет специальных «патчей», считается наиболее стабильным, так как протестировано множеством специалистов. Также стоит обратить внимание на версии компилятора GCC и основных библиотек. При компиляции проекта драйвера практически всегда необходимы заголовочные файлы ядра Linux. Интересно, что имена файлов, входящих в исходный код ядра Linux, могут различаться лишь регистром символов, что для Unix-систем вполне естественно. 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. Во многие дистрибутивы входит интегрированная среда разработки Eclipse IDE. Из утилит интересен отладчик Linice (http://www.linice.com), напоминающий своим интерфейсом Windows SoftICE. Большое количество информации об операционной системе Linux доступно на сайте www.linuxjournal.com. А по адресу http://lxr.linux.no можно найти путеводитель on-line по исходному коду ядра, начиная с его первой версии.

Драйверы QNX

QNX — это операционная система реального времени, обеспечивающая предсказуемость и гарантированное время отклика при любом внешнем воздействии. В этой системе, основанной на микроядре, совершенно другая идеология построения драйверов (обычно употребляют синоним — ресурс-менеджер). Поэтому остановимся на этом моменте подробнее. Модель взаимодействия ресурс-менеджера с системой представлена на рис. 5.

Главное и, пожалуй, самое ценное для нас отличие внутреннего устройства драйвера от других систем, постро-

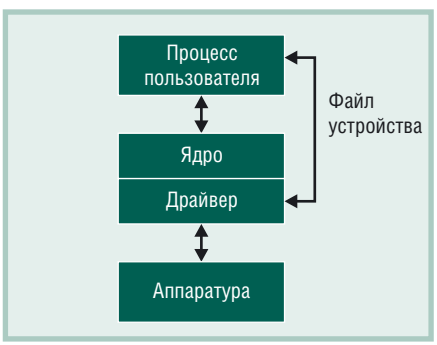

Рис. 4. Модель взаимодействия драйвера с компонентами системы в ОС Windows и Linux

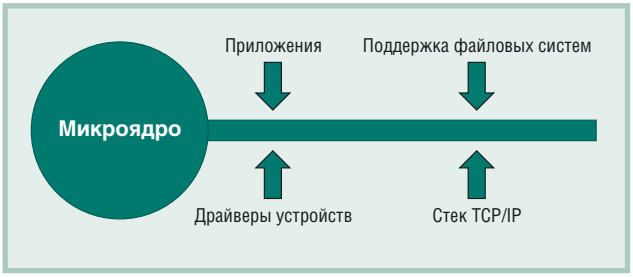

Рис. 5. Модель взаимодействия драйвера с компонентами системы в ОС QNX

енных на монолитном ядре, это то, что ресурс-менеджер является самым обыкновенным пользовательским процессом системы. То есть, в принципе, написание драйвера для QNX ничем не отличается от написания обыкновенной программы. Ресурс-менеджер не является частью ядра, не встраивается в него в виде модуля, это просто сервер, принимающий сообщения от других программ и переводящий их на язык обслуживаемого им устройства.

QNX Momentics 6.3 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), очень схожую с такими инструментами, как Borland Delphi или Microsoft Visual Basic. Среда Momentics создана на базе технологии 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. Как яркий пример этого отметим великолепную утилиту System Profiler, предназначенную для отладки системных приложений. System Profiler (рис. 6) работает в паре со специальным инструментальным ядром QNX и позволяет заглянуть внутрь системы, чтобы понять, что же происходит «за кулисами». С помощью инструментального ядра можно отследить очень много событий системы, включая

- системные вызовы ядра,
- работу менеджера процессов,
- отладку прерываний,
- работу планировщика,
- переключения контекстов.

В связи со спецификой ОС и соответственно её меньшей популярностью существует немного примеров программирования драйверов. Хорошим подспорьем для начинающего программиста будут готовые шаблоны ресурс-менеджера, написанные Игорем Желтиковым, которые можно свободно загрузить со страницы http://resmgr.narod.ru. Большие порталы сообщества программистов и пользователей QNX доступны по адресам

http://qnx.org.ru и

http://www.openqnx.com.

С чего начать

Удобная конфигурация стенда разработки представлена на рис. 1. По возможности Windows, Linux и QNX необходимо поставить на один жёсткий диск и сделать образ системы, чтобы в случае отказа быстро восстановить рабочую среду. При написании кода часто возникают «мистические» ошибки, когда, казалось бы, всё что можно проверено, но программа всё равно 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, только что вышедшего из опытной партии в свет «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 настоятельно 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.

Все драйверы по-своему уникальны, так как создаются для нового типа оборудования, но в большинстве из них используются 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. Например, доступ к портам ввода-вывода можно сделать на основе макроопределений:

```
#ifdef OS_WINNT

#define WRITE_PORT8(port, value)\
WRITE_PORT_UCHAR( (PUCHAR)(port),\
(value))
#define READ_PORT8(port)\
```

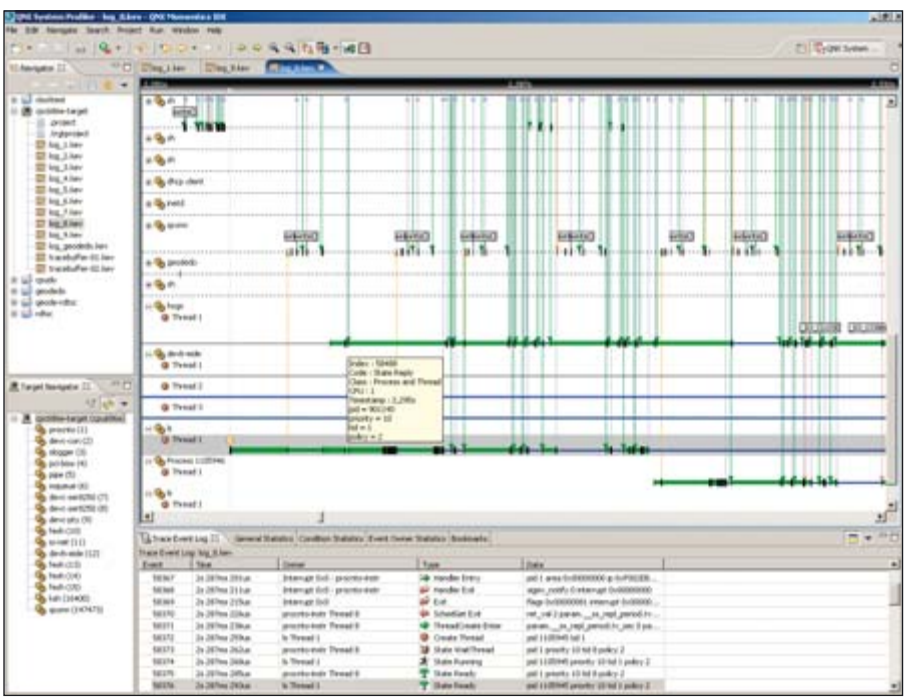

Рис. 6. Отладка приложения с помощью System Profiler

```
(READ_PORT_UCHAR( (PUCHAR) (port) ))

#endif // _NT_

#ifdef OS_QNX

#define WRITE_PORT8(port, value)\
out8((port), (value))
#define READ_PORT8(port) in8(port)

#endif // _QNX_

#ifdef OS_LINUX

#define WRITE_PORT8(port, value)\
outb((value), (port))
#define READ_PORT8(port) inb(port)

#endif // _LINUX_
```

Размер предопределённых типов данных

Для различных типов процессоров машинное слово имеет свой родной, предопределённый размер. Изначально предполагалось, что Си-программисты будут считать размер типа int соответствующим размеру машинного слова, то есть оптимальному для процессора, и так было многие годы. Но наступил момент, когда большинство разработчиков 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 довольно редко.

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

Исходя из сказанного, желательно всегда определять типы данных с помощью typedef:

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

```
typedef signed char         my_s8;
typedef unsigned char       my_u8;

typedef signed short        my_s16;
typedef unsigned short      my_u16;

typedef signed int          my_s32;
typedef unsigned int        my_u32;


typedef float               my_f32;
typedef double              my_f64;

#ifdef OS_WINNT
typedef signed __int64              my_s64;
typedef unsigned __int64            my_u64;
#endif

#ifdef OS_LINUX
typedef signed long long int        my_s64;
typedef unsigned long long int      my_u64;
#endif
```

Проблема порядка байтов

Сыздавна 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. Изначально термины little-endian и big-endian совсем не имели отношения к информатике, любознательный читатель может ознакомиться с историей их появления в статье по адресу http://www.rdrop.com/~cary/html/endian_faq.html#danny_cohen.

Можно предположить, что большинство программистов знакомо с x86 процессорами, которые построены на архитектуре little-endian. Постараемся показать подводные камни, которые ожидают 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
```

Процессоры PowerPC, SPARC, а также заголовки пакетов протокола TCP/IP хранят данные в формате big-endian, что очень удобно для отладки и просмотра hex-образов дампов памяти:

```
00 01 02 03
```

Взгляните на вполне обычный код преобразования типов через указатели

Таблица 2

Размеры типов данных ОС Linux в зависимости от архитектуры ЭВМ

arch	char	short	int	long	ptr	long-long	u8	u16	u32	u64
i386	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
sparc	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	4	8	8	8	1	2	4	8

```
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 x_char 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 x_char будет содержать младший байт переменной temp.

Очень важно соблюдать порядок байтов при записи файлов или передаче информации по сети.

```
long x = 0x1;
int result = write( handle, &x, sizeof( x ) );
```

В результате чтения записанной информации по сети или с дискового носителя можно получить неверный результат:

```
long x;
int result = read( handle, &x, sizeof( x ) );
```

Если ЭВМ имеет тот же порядок байтов, содержимое x будет равно 1, в противном случае x примет значение 0x01000000 (при условии, что оба компьютера 32-разрядные).

Порядок байтов в конкретной машине можно определить с помощью программы на языке Си (testendian.c):

```
#include <stdio.h>
unsigned short x = 1; /* 0x0001 */

int main(void)
{
 printf(«%s\n», *((unsigned char *) &x)
 == 0 ? "big-endian" : "little-endian");
 return 0;
}
```

Результаты запуска на big-endian машине (SPARC):

```
$ cat /proc/cpuinfo | grep ^cpu
cpu : TI UltraSparc IIi
$ gcc -o testendian testendian.c
$ ./testendian
big-endian
```

Результаты запуска на little-endian машине (x86):

```
$ cat /proc/cpuinfo | grep '^model name'
model name : Intel(R) Pentium(R) 4 CPU
2.66GHz
$ gcc -o testendian testendian.c
$ ./testendian
little-endian
```

Выравнивание данных

Некоторые процессоры не могут читать или писать многобайтовые числа по точно указанному адресу памяти. Зачастую сложности в доступе возникают, если адрес не кратен размеру числа, поэтому при размещении данных в памяти рекомендуется использовать выравнивание, то есть слова следует размещать по чётным адресам, двойные слова — по адресам, кратным четырём, и т. д. Архитектура x86 поддерживает произвольный доступ к памяти, но обращение к невыравненным данным вынуждает процессор делать дополнительные циклы чтения или записи. Доступ к невыравненным данным на некоторых RISC-процессорах может вызывать исключение с последующей остановкой системы. Поэтому следует очень осторожно отнестись к манипуляциям с указателями и директивам компилятора. Для максимальной переносимости следует выбрать наибольшее выравнивание из требований к архитектурам процессоров, поддерживаемых вашим программным обеспечением.

Рассмотрим пример:

```
struct _foo{
char a;
long b;
} foo;

foo.a = 1;
foo.b = 2;
```

Как структура будет расположена в памяти? Всё зависит от платформы и настроек директив компилятора. По умолчанию для оптимизации доступа часто включено выравнивание данных, поэтому sizeof(foo) заранее не известно. В памяти может быть

```
01 00 00 00 02
```

или

```
01 e8 00 00 00 02
```

или даже

```
01 f6 a5 20 00 00 00 02
```

При включённом выравнивании данных компилятор сам вставляет дополнительные «пустые» байты для оптимизации доступа к структуре. В приведённых примерах, байты e8, f6, a5 — это всего лишь старые значения, хранящиеся в памяти, поэтому рекомендуется делать её предварительное обнуление.

Заключение

Что же должны помнить разработчики при создании переносимого кода, при создании программ, которые будут перенесены на системы с другой архитектурой, например, базирующиеся на микроядре? Основная мысль, содержащаяся в ответах на эти вопросы, это соответствие стандартам. Придерживайтесь стандартов! Избегайте расширений компиляторов Borland C, GCC и др. Использование стандарта ANSI языка Си, интерфейса POSIX и других вещей, которые были стандартизированы, даст неплохой шанс вашим программам запуститься на любой платформе.

В статье не упомянута операционная система Windows CE, созданная Microsoft специально для встраиваемых систем, которая, кстати сказать, доросла уже до версии 6. Поэтому в следующих выпусках мы детально рассмотрим структуру и процесс создания драйвера Windows CE. ●

Рекомендуемая литература

1. QNX® Momentics® Integrated Development Environment User's Guide. — Ottawa : QNX Software Systems Ltd., 2004.
2. B. Hook.Write Portable Code: An Introduction to Developing Software for Multiple Platforms. — San Francisco : No Starch Press, 2005.
3. Walter Oney. Programming the Windows Driver Model. — Redmond : Microsoft Press, 1999.
4. Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman. Linux Device Drivers, Third Edition. — Sebastopol : O'Reilly Media, Inc., 2005.
5. http://www.unix.org
6. http://www.ansi.org
7. http://www.wikipedia.org

Автор — сотрудник фирмы Fastwel
119313, Москва, а/я 242
Тел.: (495) 234-0639
Факс: (495) 232-1654
E-mail: info@fastwel.ru