Свен Даммер

О реальной стоимости «доморощенной» Linux

В статье детально описывается процесс построения специализированного дистрибутива встраиваемой Linux в кросс-платформенной среде. Раскрываются неочевидные источники опасностей и скрытых затрат при использовании «ручного» подхода к построению собственной Linux-платформы, приводится сравнение «ручного» и коммерческого подходов на примере платформы Wind River Linux.

Вступление

Исходя из статистики клиентских запросов и обзоров аналитических компаний типа Gartner, число новых проектов ПО для встраиваемых систем, для которых в качестве ОС была выбрана Linux, растёт. Среди самых очевидных преимуществ Linux – доступность исходного текста из тысяч онлайн-источников, отсутствие лицензионных отчислений, большое количество системного, связующего и прикладного ПО, а также вся мощь и гибкость, которые может дать встраиваемому ПО многозадачная ОС. Ядро Linux и окружающее его открытое ПО являются центром новой экосистемы разработки, оптимизации и развёртывания встраиваемых приложений.

Однако, к сожалению, мощь, гибкость и лёгкая доступность ещё не означают, что создание и поддержка Linux-платформы для встраиваемого устройства вручную является тривиальной задачей. Всё большее количество разработчиков в последнее время стали осознавать, что использование «доморощенной» Linux несёт в себе риск невольно свернуть их с пути к реальной цели – обеспечению конкурентных преимуществ разрабатываемому устройству. В данной статье описываются базовые компоненты Linux-платформы, инструменты и последовательность шагов по созданию, развёртыванию и тестированию этих компонентов, а также основные ключевые моменты поддержки любых «доморощенных» программных решений для встраиваемых систем.

Компоненты Linux-платформы

Термином «Linux-платформа» (или «дистрибутив Linux») обычно называют базовый комплект ПО, необходимый для построения ядра Linux и системных и прикладных программ для заданного устройства. Эти базовые компоненты всегда одинаковы, вне зависимости от того, является ли целевое устройство настольным или встраиваемым компьютером. Любая Linux-платформа всегда состоит из трёх базовых элементов:

- ядра Linux, то есть версии ядра Linux, скомпилированной для данного процессора (оно предоставляет поддержку интегрированных аппаратных модулей, шин, необходимых протоколов связи и обмена данными, а также наиболее часто встречающихся периферийных устройств);
- корневой файловой системы, то есть набора системных и прикладных программных пакетов, используемых для создания корневой файловой системы (корневая файловая система поддерживает основные системные сервисы, и с неё загружаются прикладные программы);
- инструментария разработки, то есть компилятора и сопутствующих утилит, позволяющих разработчику создавать ПО, исполняемое на целевой аппаратуре.

Поддержка своей собственной Linux-платформы означает, что все эти три компонента вам нужно будет заполучить, скомпилировать и поддерживать самостоятельно. Далее в статье мы увидим, что построение этих ключевых компонентов на одном настольном компьютере – процесс хоть и непростой, но управляемый. Однако когда в процесс разработки встраиваемого ПО вовлечено более одного разработчика и более одной рабочей станции, а также когда к разработке добавляются развёртывание, поддержка и регулярное обновление платформы на всём предприятии (или даже просто в пределах одного подразделения), возникает необходимость в дополнительной инфраструктуре.

Получение, модификация и конфигурирование ядра Linux

Попытка начать проект разработки ПО с получения исходных текстов ядра, которое вы намерены использовать на целевом устройстве, может показаться странной – в конце концов, у вас ещё даже нет инструментария, чтобы этот код скомпилировать. Однако если вы хотите собрать свой собственный инструментарий, то для сборки всех его компонентов необходимо,

чтобы заголовочные файлы ядра были модифицированы в соответствии с выбранной архитектурой процессора (более детально об этом – в следующем разделе).

Выбор конкретной версии ядра Linux для проекта обычно зависит от того, какой уровень поддержки вашей целевой аппаратуры обеспечивается основной веткой кода ядра, а также от того, для каких версий ядра существуют необходимые вам «заплатки» (патчи). «Заплатки» обычно добавляют поддержку новых процессоров и устройств, а также содержат исправления ошибок, пока не включённые в основную ветку кода ядра. Предоставляются такие «заплатки» обычно производителями «кремния», которые используют Linux для внутренних целей, но не предлагают коммерческих Linux-решений. Другие «заплатки» зачастую можно взять с сайтов, посвящённых конкретной архитектуре.

В дополнение к этому для некоторых классов процессоров (например, процессоров без аппаратного диспетчера памяти – MMU) часто требуются специфические «заплатки», обычно доступные только для конкретных версий ядра. Несмотря на то что поддержка процессоров без MMU уже интегрирована в основную ветку ядра 2.6, всё ещё существует множество дополнительных «заплаток», которые могут быть необходимы для поддержки вашей целевой аппаратуры. И, наконец, «заплатки», добавляющие поддержку отраслевых спецификаций типа CELF (Consumer Electronics Linux Forum) или CGL (Carrier Grade Linux), могут требовать предварительного применения ряда других «заплаток».

Когда вы определитесь с номером версии ядра Linux и всех необходимых «заплаток» и получите исходный текст, интеграция «заплаток» в код ядра может не пройти из-за конфликтующих изменений в коде. Чтобы устранить эти конфликты, вам придётся вручную редактировать исходный текст ядра. Процесс этот обычно носит итеративный характер, и вы вынуждены будете пройти через множество циклов модификации и проверки, пока у вас получится предположительно корректное дерево исходных текстов ядра Linux для вашего устройства. С этого момента вы можете начать конфигурировать ваше ядро – скомпилировать его вы пока не можете, но теперь у вас есть хотя бы заголовочные файлы, которые потребуются для построения инструментария кросс-компиляции.

Поиск/построение инструментального пакета и сопутствующего ПО

Самая серьёзная проблема любого проекта, основанного на «доморощенной» Linux, – это инструментальный пакет для построения системного и прикладного ПО, а также ядра ОС. Большинство Linux-платформ используют инструментарий на базе GNU Compiler Collection, свободно распространяемого пакета компиляторов, включающего в себя самый распространённый в мире компилятор языка Си – GCC. Использование GCC предполагает наличие ещё двух программных пакетов: библиотеки языка Си и набора утилит, включающих в себя ассемблер, компоновщик, библиотекарь и ряд других инструментов, обеспечивающих создание исполняемых программ и сопутствующих библиотек для целевого устройства. Совокупность всех этих компонентов и называется инструментальным пакетом.

GCC портирован на большое число различных процессорных архитектур, так что физическая возможность генерировать бинарные модули для большинства доступных аппаратных платформ обычно уже есть. Аналогично стандартная библиотека Си и утилиты тоже портированы на множество различных систем. Однако поддержка последних моделей процессоров обычно требует соответствующих «заплаток» для утилит, GCC и библиотеки языка Си, которые нужно заполучить, интегрировать, разрешить все конфликты и т.п.

С генерацией бинарных модулей для встраиваемых систем связана ещё одна интересная проблема. Поскольку Linux на вашем устройстве ещё не работает, то резидентную (так называемую родную. – *Прим. пер.*) версию GCC вам выполнять физически негде. К тому же у большинства встраиваемых аппаратных платформ просто недостаточно ресурсов, чтобы хранить и выполнять компилятор и связанные с ним компоненты. По этим причинам разработка для Linux-платформы обычно производится на обычном настольном компьютере с использованием специального инструмента, называемого кросс-компилятором. Кросс-компилятор выполняется на инструментальной системе, но бинарные файлы, которые он генерирует, предназначаются для целевой системы с другой архитектурой. Инструментальная система отличает кросс-компиляторы от резидентных компиляторов по префиксам в именах – эти префиксы содержат названия целевых платформ.

Если вам повезёт, то вы сможете найти и скачать готовый пакет кросс-инструментария для процессора, используемого в вашей целевой системе. Однако, сделав это, вы сразу становитесь заложником умений того человека, который этот пакет собрал, и его выбора включённых в пакет библиотек. Даже если вы не прочь попытать счастья, имейте в виду – вы рискуете поставить свой проект в зависимость от неподдерживаемого программного компонента.

Удобство скачивания готового кросс-инструментария может быстро улетучиться, если вы впоследствии столкнётесь с проблемами в процессе его использования (особенно в плане производительности результирующих бинарных модулей) и вам будет некуда обратиться за исправлениями и поддержкой. Многие проекты встраиваемого ПО также требуют архивации исходных текстов, из которых был построен инструментарий, а они запросто могут быть недоступны, потому что инструментарий строили не вы. Когда возникает одна или обе из перечисленных проблем, проектная команда часто принимает решение собрать свой собственный кросс-инструментарий, чтобы не зависеть от доступного, но неподдерживаемого стороннего решения.

Построение среды кросс-компиляции в составе кросс-компилятора GCC, стандартной библиотеки языка Си и набора утилит может оказаться чрезвычайно сложной задачей. «Заплатки» и обновления для конкретных архитектур, процессоров и устройств рассеяны по множеству Web-сайтов, и все их придётся перед сборкой инструментария интегрировать вручную. Как и в случае с конфигурированием кода ядра, наложение «заплаток» из разных источников зачастую не срабатывает из-за конфликтов в версиях кода, которые также придётся разрешать самостоятельно. Процесс этот обычно итерационный, и циклов модификации и проверки потребуется больше одного.

Как только исходный код будет готов, вам нужно будет построить все компоненты среды кросс-компиляции в правильном порядке. Сначала с по-

мощью резидентной версии GCC строятся утилиты для вашей целевой системы. Затем GCC доводится до состояния, в котором его можно использовать для построения библиотеки Си. Следом вы строите библиотеку, а потом достраиваете GCC.

Если ваша целевая платформа стандартная, то в процессе построения инструментального пакета вам могут помочь свободно распространяемые инструменты типа crosstool Дэна Кегела и buildroot Эрика Андерсена. Однако их требуется предварительно сконфигурировать, выбрав правильные версии компилируемых компонентов, и то оптимизации кода для вашего процессора они могут и не предоставить. Если вы строите инструментальный пакет для более экзотической встраиваемой системы или вам требуется конкретная версия GCC, утилит и/или библиотеки Си, то даже с использованием этих инструментов вы будете вынуждены создавать свои собственные конфигурации.

И последнее, что следует учесть при построении инструментального пакета, – это то, что для встраиваемой платформы может быть необходимо использование нескольких различных версий библиотеки Си. Стандартная GNU-библиотека (glibc) может оказаться слишком велика, и результирующие бинарные модули получатся слишком объёмными, чтобы развернуть их на выбранной целевой системе. Для использования во встраиваемых приложениях был разработан ряд других Си-библиотек, таких как diet libc, klibc, newlib и uClibc, – их применение позволяет сократить размер бинарных модулей. Интегрирование дополнительной Си-библиотеки в инструментальный пакет может оказаться сложной задачей и требовать дополнительных навыков по сравнению с обычным процессом построения инструментального пакета. К тому же различные библиотеки могут подпадать под различные модели лицензирования – в частности, diet libc и klibc подпадают под GPL, а не LGPL, что может вылиться в проблемы с защитой вашей интеллектуальной собственности.

Несмотря на то что путь, описанный в данном разделе, сложен, запутан и требует фундаментального понимания устройства всех компонентов инструментального пакета, он был не раз пройден. Впрочем, выбирая в качестве платформы «доморощенную» Linux, помните, что способность поддерживать в актуальном состоянии инструментальный пакет и все его компоненты – очень непростая задача. Начиная такой проект, в первую очередь убедитесь, что у вас есть люди, способные её выполнить. В дальнейшем вам нужно также быть уверенными, что вы либо сохраните этих людей в проекте, либо будете иметь доступ к экспертам с равноценными знаниями и навыками, способным разрешить проблемы, возникающие в процессе разработки, а также выпускать и поддерживать необходимые обновления.

Построение ядра

Построение ядра Linux для встраиваемой платформы с использованием кросс-компилятора во многом аналогично построению ядра Linux для настольного компьютера, но с двумя основными отличиями.

Во-первых, при построении ядра нужно установить две переменные окружения: ARCH, задающую целевую архитектуру, и CROSS_COMPILE, определяющую нужный префикс имени кросс-компилятора. Во-вторых, поскольку основная целевая платформа для Linux – это системы с архитектурой x86/IA-32, вы должны быть готовы к решению проблем с порядком следования байтов и прочими архитектурно-зависимыми нюансами, которые не были протестированы и исправлены в коде ядра после применения всех необходимых «заплаток». Вам также нужно быть уверенными в том, что вы включили в ядро драйверы для всех необходимых устройств, протоколов и файловых систем, которые вам понадобятся в процессе собственно загрузки ОС.

После множества циклов компиляции ядра, идентификации ошибок, исправления этих ошибок и повторной компиляции вы в результате получите код ядра, который компилируется «чисто» и может выполняться на вашей целевой аппаратуре. Следующим шагом будет установка ядра на целевую систему и его тестирование.

Если ваша целевая платформа – готовое коммерческое решение, то она, скорее всего, поставляется в комплекте с предустановленным монитором загрузки или начальным загрузчиком, способным найти и развернуть ядро и передать ему управление. Если вы разрабатываете собственную аппаратуру, не содержащую BIOS, для установки ядра или нестандартного загрузчика вам понадобятся средства прямого доступа к оборудованию (например, JTAG). Существуют прекрасные свободно распространяемые начальные загрузчики типа U-Boot или RedBoot, но их использование требует специальных навыков, потому что их нужно дорабатывать для конкретной платформы, компилировать, инсталлировать и поддерживать. Аналогично JTAG-адаптеры тоже требуют наличия у разработчика специальных навыков, особенно когда это касается инсталляции, выполнения и отладки нестандартных программных решений.

К этому моменту ядро может стартовать на вашей встраиваемой системе, инициализировать устройства и останавливаться на попытке найти корневую файловую систему. Если это ещё не так, значит, вам потребуется пройти через несколько циклов построения ядра и устранить проблемы типа некорректных базовых адресов и отображения памяти, чтобы ваше ядро могло правильно инициализировать оборудование и отображать диагностические сообщения. Если вы видите сообщение о том, что ядро не смогло найти корневую файловую систему, то это значит, что вы достигли важной вехи в вашем проекте. Несмотря на то что тестирование вашей Linux-платформы ещё даже не начиналось, вы уже близки к моменту, когда её можно будет протестировать и начать писать для неё приложения – то, ради чего и затевался проект.

Создание базовой корневой файловой системы

Корневая файловая система – это файловая система, содержащая системные и пользовательские приложения, которые ядро Linux может выполнять, а также необходимую системную информацию типа файлов устройств и конфигурационных файлов. Во встраиваемых системах используются различные типы корневых файловых систем в зависимости от того, используется ли в них энергонезависимый носитель (флэш-память, жёсткий диск и т.п.) или нет.

Системы на основе встраиваемого Linux, не использующие долговременного хранилища данных, обычно загружаются с RAM-диска, созданного из сжатого образа файловой системы (его часто называют начальным RAM-диском), или с файловой системы в ОЗУ, разворачиваемой непосредственно в памяти ядра (известной также как initramfs) – этот способ доступен толь-

ко в Linux 2.5 и старше. Начальные RAM-диски могут быть самого различного формата, включая ext2, romfs, cramfs или squashfs; у каждого формата свои достоинства и недостатки, но хранить данные между перезагрузками системы RAM-диск не может.

Системы на основе встраиваемого Linux, снабжённые долговременным хранилищем, поддерживают ряд файловых систем, у каждой из которых – свои различные характеристики. Некоторые файловые системы, как, например, поддерживающие выравнивание износа флэш-памяти (JFFS2) или транзакционные с поддержкой быстрого перезапуска, используемые на жёстких дисках (ext3, JFS, XFS, ReiserFS, Resiser4 и т.д.), привязаны к определённому типу носителя. Выбор файловой системы, которую вы будете разворачивать на целевой системе с жёстким диском, – фундаментальное решение, требующее как поддержки со стороны ядра, так и навыков работы с административными утилитами выбранной файловой системы.

Процесс создания корневой файловой системы, по сути, одинаков вне зависимости от того, развёртываете вы файловую систему в ОЗУ или на диске. Первым шагом будет определение необходимых программных пакетов, которые вам нужно будет расположить в файловой системе, чтобы обеспечить корректную инициализацию ОС, старт ключевых сервисов и поддержку приложений, которые будут выполняться на вашей целевой системе.

Большинство проектов, основанных на «доморощенной» Linux, начинают с использования корневой файловой системы, предоставляемой пакетом BusyBox – многофункциональным модулем (в оригинале "multi-call binary" – единый бинарный модуль, совмещающий в себе функции более чем одной утилиты. – *Прим. пер.*), способным выполнять функцию практически любой утилиты работы с файловой системой в Linux.

Вы можете сконфигурировать и скомпилировать этот пакет так, что он будет, по сути, единственным компонентом вашей корневой файловой системы (за исключением нескольких файлов устройств и множества символьных связей, по которым бинарный модуль BusyBox вызывается под разными именами).

Если на этот момент вам будет необходимо поставить дополнительные программные пакеты, то каждый из них можно скачать с домашней страницы соответствующего проекта в Интернете. Чтобы выбрать правильные пакеты, предоставляющие необходимые сервисы и поддерживающие нужные вашему устройству протоколы, вашей команде программистов надо хорошо ориентироваться в доступных программных пакетах для встраиваемой Linux и возможных альтернативах для них. Зачастую пакетов, предоставляющих схожую функциональность, доступно несколько; принимая решение о том, какому пакету доверить ту или иную функциональность, вам нужно будет учитывать характеристики имеющихся пакетов, их признанность и популярность в сообществе разработчиков, стабильность, уровень активности в проекте, а также имена конкретных личностей, вовлечённых в разработку.

Когда вы скачаете пакет, вам надо будет его сконфигурировать и скомпилировать кросс-компилятором, соблюдая правильную организацию дерева каталогов, которая будет «видна» пакету на целевом устройстве. В процессе кросс-компиляции может потребоваться пройти через несколько циклов устранения проблем, связанных с выбран-

ной архитектурой целевого процессора, перед тем как пакет будет корректно собран. Как только это произойдёт, вы сможете установить пакет в каталог, представляющий собой корневую файловую систему, удалить ненужные файлы (например, локальную копию документации, поддержку интернационализации и т.п.) и перейти к следующему пакету. Отладка будет потом.

Если ваша встраиваемая система снабжена жёстким диском, вы можете отформатировать его под файловую систему того же типа, что и на вашем инструментальном настольном компьютере, смонтировать её и установить пакеты непосредственно в точку монтирования. Если вы используете файловую систему во флэш-памяти (например JFFS2), то, чтобы сгенерировать образ корневой файловой системы из представляющего её каталога, вам потребуется дополнительная утилита.

Традиционный подход к созданию начального RAM-диска из стандартной файловой системы формата ext2 заключается в том, что нужно создать пустой образ файловой системы, смонтировать его, скопировать в этот смонтированный образ каталог, представляющий вашу корневую файловую систему, затем отмонтировать образ и сжать его. Начальные RAM-диски других форматов (например, romfs, cramfs или squashfs) предоставляют утилиты построения файлов образа в соответствующем виде, аналогично утилите, используемой с JFFS2. Аналогичный функционал предоставляют более новые утилиты для файловых систем формата ext2, такие как genext2fs. Файловые системы initramfs можно строить вручную и интегрировать их в ядро в процессе его построения или (что используется чаще) позволить ядру построить их для вас в процессе компиляции.

Если вы используете систему без начального загрузчика или без способности распознать отдельный образ корневой файловой системы, вам придётся каждый раз перекомпилировать ядро, чтобы встроить в него начальный RAM-диск или построить/интегрировать более свежую архивную initramfs. Если вы используете корневую файловую систему, расположенную на флэш-носителе или жёстком диске, и загружаете Linux при помощи монитора начальной загрузки, то вы сможете указать местоположение файловой системы в командной строке ядра. В противном случае вам придётся ещё и перекомпилировать ядро, чтобы оно правильно распознало корень файловой системы (что, впрочем, вы захотите сделать в любом случае перед началом тестирования и развёртывания).

Теперь вы можете использовать ваш начальный загрузчик или JTAG-эмулятор для копирования нового ядра и образа корневой файловой системы на ваше целевое устройство. Затем вам потребуется пройти через несколько циклов загрузки, отладки, повторного построения и копирования ядра, пока вы не добьётесь корректного старта и корректного монтирования и чтения/записи данных в корневую файловую систему.

Это ещё одна важная веха в построении «доморощенной» Linux-платформы, поскольку теперь вы можете начать тестировать ваше ядро и файловую систему и интегрировать драйверы устройств и прикладные программы, необходимые вашему устройству.

Интеграция дополнительных драйверов и приложений

Как только ваша «доморощенная» Linux-платформа начнёт загружаться и корректно выполнять приложения с выбранного вами типа корневой файловой системы, можно начинать писать драйверы устройств, специфичных для вашего оборудования. Если вы используете готовое коммерческое оборудование, драйверы могут прилагаться вместе с «заплатками», доступными от производителя микросхем или самой платы. В противном случае вам придётся писать драйверы самостоятельно и либо встраивать их непосредственно в ядро, либо кросс-компилировать как подключаемые модули, которые ядро сможет загрузить из корневой файловой системы.

Встраивать драйверы непосредственно в код ядра или компилировать отдельно и загружать как внешние модули – решение корпоративное. Драйверы, встраиваемые непосредственно в ядро, должны выпускаться под лицензией GNU General Public License (GPL), а значит, вам придётся сделать их исходный текст доступным по запросу для всех пользователей вашей платформы.

Если вы решите встроить ваш драйвер непосредственно в дерево исходных текстов ядра, то чтобы его протестировать, вам нужно будет заново построить и развернуть ядро. Если вы компилируете ваши драйверы как отдельные модули, которые будут загружаться из файловой системы, вам нужно будет встроить их в файловую систему. Если она при этом выполнена в формате образа флэш-памяти, то этот образ тоже нужно будет перестраивать. Если же образ файловой системы встраивается непосредственно в ядро (как, например, начальный RAM-диск или архив initramfs), то придётся перестраивать само ядро.

Разработка приложений, делающих ваше устройство уникальным и предоставляющих необходимый пользователям функционал, очевидно, во многом зависит от целевого рынка вашего устройства. Однако вашему приложению могут потребоваться дополнительные сервисы, которые пока что в созданной вами корневой файловой системе недоступны. Например, сетевым устройствам часто необходим программный пакет, обеспечивающий им возможность динамически получать IP-адрес, поддерживать удалённый доступ через SSH, передавать файлы по FTP, иметь свой Web-сервер для удалённого конфигурирования и т.п.

Каждый пакет, обеспечивающий подобные возможности, обычно доступен в нескольких версиях, выбор наиболее соответствующей требованиям, наиболее свежей и стабильной и укладывающейся во вносимые вашим устройством ограничения по ресурсам, лежит на вас.

Когда вы определитесь с тем, какие пакеты добавить в корневую файловую систему, зайдите на домашнюю страницу каждого пакета в Интернете. Скачав пакет, вы должны будете сконфигурировать и попытаться кросс-компилировать его. Это может потребовать множества итераций, пока компиляция не завершится успешно, а также разрешения взаимных зависимостей между данным пакетом и другими. Модификация кода также часто бывает необходима при кросс-компиляции пакетов для процессоров без аппаратного диспетчера памяти (MMU) из-за использования ряда недоступных Си-функций.

Как только кросс-компиляция пакета пройдёт успешно, его можно будет установить в корневую файловую систему, перестроить её (при необходимости), пересобрать ядро (если нужно) и затем развернуть корневую файловую систему и/или ядро заново.

Тестирование «доморощенных» программных платформ

Если предположить, что у вас есть все необходимые навыки, самостоятельная поддержка встраиваемой Linux-платформы может быть эффективной с точки зрения затрат, поскольку всё, включая нужные драйверы и связующее ПО, является доступным и бесплатным. Однако перспектива основывать свои приложения на бесплатном ПО и вручную доработанном ядре может оказаться пугающей. И несмотря на то что в любом уважающем себя проекте разработки ПО отводится место для тестового кода, системные тесты для кода ОС и утилит, находящихся за пределами вашего контроля, – это совершенно другая история.

После написания сценариев локального и удалённого тестирования для приложений и драйверов, разработанных вами для вашего устройства, популярным решением для тестирования самой Linux-платформы обычно является использование свободно распространяемого ПО, входящего в состав проекта LTP (Linux Test Project). LTP представляет собой набор из нескольких тысяч тестов, разработанных для проверки ядра Linux и связанного с ним функционала. Вам, скорее всего, понадобятся не все тесты из состава LTP, поскольку часть из них будет относиться к функциональности, которая в вашем устройстве не используется; однако данный пакет позволяет выбрать используемый набор тестов вручную.

После скачивания исходных текстов пакета тестирования LTP с его кросс-компиляцией могут возникнуть те же проблемы, что и со всеми остальными пакетами, изначально предназначенными для систем на базе x86 (и протестированными на них), что снова потребует нескольких итераций исправления с последующей перекомпиляцией. После успешной компиляции всех тестов (или требуемого подмножества) нужно будет определить, как развернуть их на целевой системе, выполнить там, а также собрать и отобразить полученные результаты.

В дополнение к тестам, доступным в составе LTP, существуют и другие свободно распространяемые пакеты тестирования, например LSB (Linux Standard Base). В зависимости от того, в какой отрасли промышленности вы работаете, к вашей программной платформе могут предъявляться дополнительные требования совместимости, связанные с необходимостью соответствия отраслевым стандартам; например, для производителей сетевого оборудования это может быть соответствие спецификации CGL (Carrier Grade Linux). В этих случаях от вашей платформы может потребоваться прохождение тестов наподобие включённых в LSB.

У разных встраиваемых аппаратных платформ объём доступной оперативной и постоянной памяти сильно отличается. Ограничения по ресурсам могут помешать вам развернуть на целевой аппаратуре все необходимые тесты одновременно; в этом случае вам придётся придумывать, как итеративно выполнять тесты по очереди, разворачивая каждый тест отдельно, выполняя его, собирая результаты и затем освобождая ресурсы для проведения следующего теста.

Масштабирование и поддержка «доморощенной» Linux

Как уже было показано, многие компании «вырастили» свою собственную

Linux-платформу и выпустили продукты на её основе. Иными словами, разработка и развёртывание встраиваемой программной платформы на одной конкретной настольной рабочей станции – процесс хоть и сложный, но управляемый.

Однако когда в процесс оказываются вовлечены более одного разработчика и более одной рабочей станции, способность работать с Linux-платформой в масштабе всего предприятия определяется тем, насколько просто её установить и поддерживать на множестве рабочих мест. В связи с этим большинство Linux-платформ обычно снабжаются дополнительным ПО, которое облегчает выполнение данных задач и включает в себя:

- инсталлятор, упрощающий процесс установки Linux на настольную рабочую станцию, с которой она уже может быть развёрнута на встраиваемой аппаратуре;
- администратор программных пакетов, создающий образ корневой файловой системы (или каталог, её содержащий) для целевого устройства;
- какой-либо механизм отслеживания версий установленного инструментального пакета и дополнительных компонентов, включённых в его состав.

Несмотря на то что ключевые компоненты Linux-платформы концептуально одинаковы для всех дистрибутивов, инструменты установки и администрирования, предназначенные для развёртывания и поддержки среды разработки ПО для встраиваемых устройств, таковыми не являются. Например, в большинстве Linux-систем стандартным администратором пакетов является RPM. Однако чтобы использовать его для управления пакетами, установленными в корневой файловой системе встраиваемого устройства, его нужно перекомпилировать с указанием использовать свою собственную базу данных установленных пакетов, а не базу данных RPM, отслеживающую пакеты, установленные на инструментальной рабочей станции. Аналогично, примитивные решения типа монолитных tar-архивов, несмотря на свою простоту, неэффективны, требуют большой аккуратности, а их недостаточная модульность не позволяет поддерживать обновления.

Кроме краткосрочных затрат ресурсов, потраченных на разработку, формирование дистрибутивов и развёртывание «доморощенной» Linux-платформы, важно также учитывать долгосрочные вложения ресурсов в постоянное поддержание ядра и программных пакетов в актуальном состоянии. В общем случае поддержка и регулярное обновление среды кросс-компиляции, ядра Linux, корневой файловой системы и всех ваших приложений требует очень широкого спектра навыков. Аналогично отслеживание многочисленных программных пакетов и версий ядра в онлайн-сообществе, поиск необходимых «заплаток» для реализации требований по безопасности, производительности и т.п., интеграция этих «заплаток» и повторное развёртывание платформы по мере её обновления – всё это сложные задачи, требующие больших затрат времени, особенно если дело касается обновления подсистем ядра и драйверов устройств. Соответственно, для выполнения этих задач вам может потребоваться дополнительный персонал, не имеющий отношения к непосредственным задачам вашей компании, но который необходимо иметь в распоряжении всякий раз, когда подобные задачи возникают.

Резюме

Несмотря на то что краткосрочный выигрыш от использования «доморощенной» Linux-платформы может быть очень существенным, зачастую это всего лишь надводная часть айсберга. Подвох в том, что скрытые затраты не так просто разглядеть. Опыт разработки устройств и прикладного ПО для них сильно отличается от опыта, необходимого для построения, развёртывания и поддержки встраиваемой Linux-платформы. Разработка и поддержка инструментария кросс-компиляции, интеграция «заплаток», построение и поддержка ядра ОС и корневой файловой системы, предоставляющей необходимые вашей платформе сервисы, часто требуют наличия дополнительного персонала и долгосрочного вложения ресурсов. При этом к основной (как деловой, так и технической) деятельности компании эти вложения непосредственного отношения не имеют.

Пройдя этот путь, многие производители устройств обнаружили, что перспектива иметь собственную Linux-платформу может оказаться обескураживающей. По мере увязания проектов в непредвиденных проблемах на самых разных стадиях разработки привлекательность «бесплатного» дистрибутива резко падает. Как следствие, постоянно растущая сложность устройств и сжатые временные рамки проектов вынуждают производителей устройств переходить на коммерческие дистрибутивы Linux.

Компания Wind River – наиболее динамично развивающийся производитель коммерческих Linux-платформ в индустрии встраиваемых приложений. В число платформ, предоставляемых компанией Wind River, входят масштабируемые решения, оптимизированные для самых различных вертикалей: промышленной автоматизации, сетей и телекоммуникаций, автомобилестроения, потребительской электроники, медицинского приборостроения. Все эти платформы доступны для многих популярных процессорных архитектур и отладочных плат, применяемых сегодня в разработке встраиваемых приложений.

Linux-платформа Wind River для потребительской электроники, компактная и с малым временем загрузки, идеальна для мобильных устройств и приставок. Платформа для сетевых устройств соответствует спецификации Carrier Grade Linux и оптимизирована для коммерческих ATCA-решений. Платформа общего назначения поддерживает широкий спектр оборудования и подходит для реализации самых разнообразных устройств.

Каждая из коммерческих Linux-платформ Wind River базируется на «первоисточнике» (Linux 2.6) и включает в себя интегрированный комплект разработчика на базе Eclipse. Поддержка платформ обеспечивается глобальной командой экспертов и подкреплена более чем 20-летним опытом компании Wind River в области ПО для встраиваемых систем. В дополнение к этому Wind River предлагает пользователям Linux-платформ консалтинговые услуги по разработке устройств, пакетов поддержки оборудования (Board Support Packages – BSP) и драйверов, а также оптимизации производительности. ●

Автор – Свен Даммер (Sven Dummer), руководитель направления Linux компании Wind River
Перевод Николая Горбунова, сотрудника фирмы ПРОСОФТ
Телефон: (495) 234-0636
E-mail: info@prosoft.ru