Министерство образования и науки Российской Федерации
Федеральное агентство по образованию
Федеральное государственное образовательное учреждение высшего профессионального образования «Санкт-Петербургский государственный университет»
Математико-механический факультет

Кафедра Системного Программирования

Калмук Александр Игоревич

Технология автоматизации тестирования встраиваемых систем

Дипломная работа

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Терехов А. Н.

Научный руководитель:
д. ф.-м. н., профессор Терехов А. Н.

Рецензент:
асп. Козлов А. П.

Санкт-Петербург
2014

SAINT-PETERSBURG STATE UNIVERSITY
Mathematics & Mechanics Faculty

Software Engineering Department

Alexander Kalmuk

Test automation technology for embedded systems

Graduation Thesis

Admitted for defence.
Head of the chair:
professor A.N. Terekhov

Scientific supervisor:
professor A.N. Terekhov

Reviewer:
postgraduate A.P. Kozlov

Saint-Petersburg
2014

Оглавление

Введение **4**

1. Постановка задачи **7**

2. Обзор **8**

2.1. Интеграционное тестирование 8

2.1.1. TETware RT 8

2.1.2. OpenTest 8

2.1.3. DejaGnu 9

2.1.4. Autotestnet 10

2.1.5. tcltest 10

2.2. Модульное тестирование 12

2.3. Тестирование аппаратуры 12

2.3.1. eCos 13

2.3.2. Embox 14

3. Реализация **15**

3.1. Интеграционное тестирование 15

3.2. Модульное тестирование 19

3.3. Тестирование аппаратуры 20

4. Апробация **23**

Заключение **25**

Введение

Встраиваемые системы получили применение во многих сферах человеческой деятельности - начиная от массового производства mp3-плееров и заканчивая созданием дорогостоящих и высоконадежных марсоходов. Зачастую отличительной особенностью таких систем является ограниченность аппаратных ресурсов, таких как размер ОЗУ, флеш-памяти, и отсутствие привычных устройств ввода-вывода - клавиатуры и монитора; а также минимальное энергопотребление. Это обусловлено тем, что в отличие от систем общего назначения, встраиваемые системы должны выполнять некоторую специфичную задачу, то есть им не нужна вся мощь универсальных систем.

Специфичность задачи и ограниченность аппаратных ресурсов встраиваемой системы влечет за собой усложнение процесса тестирования таких систем. Если на обыкновенном настольном компьютере мы можем установить любое необходимое окружение для тестирования, писать и тут же выполнять тесты, то в случае встраиваемых систем это не так. Чаще всего из-за отсутствия таких возможностей для встраиваемых систем их приходится либо тестировать вручную, либо не тестировать вовсе, что существенно увеличивает время разработки и уменьшает качество итоговой системы. Решением этой проблемы могло бы послужить заимствование сложившихся подходов к тестированию из области настольных компьютеров.

При разработке ПО для обычных систем, как правило, всегда присутствуют два типа тестирования - модульное и интеграционное. Они представляют базовый набор, без которого невозможно перейти к системному тестированию. Кроме того, так как под разработкой встраиваемых систем часто подразумевается и разработка аппаратной составляющей, то тестирование аппаратуры также является необходимым.

Модульное тестирование ПО позволяет проверять на корректность отдельные модули исходного кода программы. Представим себе, что мы разрабатываем POSIX-совместимое приложение для встраиваемой системы и модульные тесты для него. Так как разработка тестов ведется на локальной машине, хочется иметь возможность запускать написанные тесты там же, но при этом иметь возможность выполнять их и на целевой платформе без внесения изменений в исходный код тестов. Существует множество средств для модульного тестирования на c/c++ [13], предназначенных в основном для тестирования на настольных компьютерах. Но, опять же, из-за ограниченности ресурсов нужен фреймворк, который будет легковесным и сможет работать

как на настольном компьютере, так и на встраиваемой системе. При этом легковесность не должна быть в ущерб удобству разработки тестов.

Интеграционное тестирование обычно проводится после модульного, предшествует системному, и служит для тестирования взаимодействия групп модулей. Ограниченность аппаратных и программных ресурсов затрудняет выполнение интеграционных тестов непосредственно на целевой платформе, так как для выполнения тестов требуется настройка соответствующего окружения. Для решения этой проблемы обычно создают удаленное подключение по сети к встраиваемой системе, настраивают удаленный ввод-вывод и запускают тестируемое ПО на различных входных данных, анализируя получаемые результаты на локальной машине [15].

Помимо тестирования программной части встраиваемой системы, важную роль играет тестирование периферийных устройств, а именно тот период, когда аппаратура уже разработана, и можно проводить тестирование на полной интегрированной аппаратной платформе. Для таких целей удобно было бы использовать какой-нибудь простой скриптовый язык, позволяющий быструю разработку, но вместе с тем этот язык должен иметь прямой доступ ко всем ресурсам системы, включая периферию и физическую память. Часто одной из составных частей такого тестирования является проверка состояния регистров периферийных устройств. В этом случае удобно использовать маленькую операционную систему, которая будет работать на целевой платформе и позволит легко собирать всю необходимую информацию и отправлять ее каким-либо образом на компьютер тестировщика.

На сегодняшний день существует множество средств, помогающих в решении проблемы с тестированием встраиваемых систем. Как будет показано далее, эти средства либо слишком универсальны, и, как следствие, требует немало ресурсов от встраиваемой системы, либо наоборот слишком просты и требуют реализации дополнительных надстроек. То есть если на встраиваемой системе аппаратные ресурсы ограничены, то разработчикам такой системы не остается иного выбора как самим создавать средства автоматизации тестирования, либо отказываться от них вовсе. Компании-производители встраиваемых систем, создают свои собственные средства, но в целом тестирование остается ручным.

Для автоматизации тестирования нужно, чтобы исполнение тестов и проверка результатов осуществлялись программно, а не человеком. В случае модульного тестирования фреймворк должен быть установлен на встраиваемой системе, чтобы иметь

доступ к глобальным функциям всех модулей, и, по сути, должен являться легковесным аналогом таких средств как CUnit, которые используются как средства автоматизации на обычных системах. В случае интеграционного тестирования модули системы представляются как “черный ящик”, и поэтому необходимости в установке фреймворка на целевую платформу нет. Поэтому можно не думать о легковесности, а автоматизировать загрузку тестов, получение и анализ результатов. Для тестирования аппаратуры часто удобна не только автоматизация, а совмещение автоматизированного тестирования и ручного. Например, после того как пройдены все автоматические тесты, тестировщику может потребоваться тут же вручную проверить некоторые регистры, которые не были проанализированы автоматическими тестами. То есть для тестирования аппаратуры фреймворк было бы удобно установить на стороне встраиваемой системы и поддерживать как ручное, так и автоматизированное тестирование.

1. Постановка задачи

Целью данной дипломной работы являлось разработать технологию автоматизации тестирования встраиваемых систем с ограниченными ресурсами. Для достижения цели были поставлены следующие задачи:

- Автоматизация интеграционного тестирования
- Автоматизация модульного тестирования
- Автоматизация тестирования периферийных устройств

2. Обзор

2.1. Интеграционное тестирование

2.1.1. TETware RT

TETware [8] - это универсальное средство для управления тестированием. Оно поддерживает не распределенное тестирование (как на локальной машине, так и на удаленной), и распределенное тестирование, когда части одного теста исполняются на разных машинах. TETware можно собрать с поддержкой TETware RT для тестирования встраиваемых систем и систем реального времени. TETware RT состоит из двух частей - полноценный фреймворк с управлением и журналированием результатов тестирования на стороне хоста и его легковесный аналог на стороне встраиваемой системы. Для запуска теста на целевой платформе тестовый сценарий, написанный на языке Си, линкуется с библиотекой TCM (Test Case Manager) и библиотекой API (поддержка потоков, процессов, синхронизации тестов, и т.д.).

Для удаленного выполнения тестов нужно реализовать всего 5 функций на целевой платформе: tet3rt_msgrt_open(), tet3rt_msgrt_close(), tet3rt_ msgrt_send(), tet3rt_msgrt_recv(), tet3rt_rt_exit(). Кроме того, в составе TETware RT имеются две готовые реализации этих функций - через последовательный порт и через сокеты. Таким образом, TETware RT решает проблему с тяжеловесностью, предоставляя пользователю как готовые решения, так и возможность самому реализовать TCM.

Однако у этой системы имеется один существенный недостаток. Для TETware неудобно писать тестовые сценарии на языке Си, который получил широкое применение в разработке встраиваемых систем, в виду чего была добавлена поддержка C++, Java, Perl, sh и Python. Таким образом, от встраиваемой системы требуется поддержка одного из этих языков, что не всегда уместно.

2.1.2. OpenTest

OpenTest [10] - это уже проект с открытым исходным кодом, основанный на другом проекте STAF [9] (Software Testing Automation Framework), и являющийся частью Arago Project. Основу для исполнения тестов на целевой платформе составляет TEE (Test Execution Engine), представляющий собой сервис STAF. Сервис STAF - это обыкновенный процесс ОС, который находится в режиме ожидания и ждет команд от планировщика STAF. Планировщик посылает TEE сервису, который запущен на

встраиваемой системе, команду выполнить тест, после чего сервис выполняет тест и отсылает обратно сообщение о результатах в виде XML-файла.

Сами тесты создаются через web интерфейс системы TestLink, после чего TestLink экспортирует тесты в XML файлы и делает запрос на исполнение теста планировщику STAF. Такой подход является универсальным, так как пользователь может реализовать что угодно в виде сервиса, однако это и создает определенные сложности. Во-первых, TEE должен быть запущен на целевой платформе и, следовательно, иметь все необходимое окружение STAF. Во-вторых, каждый отдельный тест представляет собой набор входов и выходов, являющихся строками, которые задаются через web интерфейс. Но часто выходной результат может описываться более сложным образом, чем просто строка, например, как результат выполнения некоторой команды bash. И проверка этого результата возложена на TEE, который работает на стороне целевой платформы, что подразумевает, как минимум, наличие некоторых основных утилит (sh, grep, awk, и т.д.), то есть предполагается наличие Linux или какой-то другой ОС с достаточно большими требованиями.

2.1.3. DejaGnu

DejaGnu [5] преподносится как “фреймворк для тестирования других программ”, и его целью является предоставить единый фронтенд для всех тестов. Основные свойства этого фреймворка:

- написан на Expect [7],- средстве для автоматизации интерактивных приложений, таких как telnet, ftp и т.д.;
- уровень абстракции, позволяющий легко переносить тесты на любую платформу, где они должны исполняться;
- тестирование на соответствие стандарту POSIX.

Тесты пишутся на Expect, с использованием процедур DejaGnu. DejaGnu предоставляет минимальный набор процедур (pass, fail, xfail, unsupported), чтобы сообщить о результате выполнения теста. Помимо этого имеется возможность легко устанавливать соединение по протоколам telnet и ssh. Для запуска тестов используется утилита runtest в составе DejaGnu, которая запускает тесты и журналирует результат их исполнения. Порядок исполнения тестов никак не специфицирован.

DejaGnu удобен тем, что от встраиваемой системы почти ничего не требуется. Единственное, что нужно - это какое-нибудь соединение по сети и минимальный командный интерпретатор для запуска тестируемого приложения.

Однако этот фреймворк почти не предоставляет примитивов для тестирования, например, нельзя легко проверить, что код возврата некоторой команды равен 0.

2.1.4. Autotestnet

Autotestnet [1] - еще одно средство для тестирования, основанное на Expect. Характеризуется своей простотой. Чтобы писать тесты не требуется знание скриптовых языков. Для добавления нового теста требуется создать папку для набора тестов и задать для всех будущих тестов из этой папки тип подключения к тестируемой платформе - telnet, ssh или minicom. После этого в созданный набор можно добавлять отдельные тесты в формате - <входные данные> ||| <выходные данные> ||| <таймаут>.

Autotestnet, так же как и DejaDnu, примечателен тем, что почти ничего не требует от тестируемой платформы. Существенными недостатками являются:

- отсутствие возможности запуска тестов из командной строки (имеется только GUI);
- <выходные данные> можно сравнивать только с регулярным выражением, нет возможности сравнения с результатом выполнения какой-нибудь команды. Например, “ping -c 1 localhost ||| echo “1 received” ||| 10” будет ожидать строку “echo “1 received”” (не позже чем через 10 секунд), а не результат выполнения команды echo.

2.1.5. tcltest

Tcltest [7] - это один из стандартных пакетов в составе дистрибутивов Tcl. Он характеризуется простотой и гибкостью. Тест может включать в себя до девяти частей, которые исполняются последовательно - проверка ограничений, действия, выполняемые перед тестом и после теста, режимы проверки результатов, режимы вывода результата и другие. Например,

```
test example-1.1 {test file existence} -setup {
    set file [makeFile {} test]
```

```
} -body {
    file exists $file
} -cleanup {
    removeFile test
} -result 1
```

Части setup и cleanup нельзя сделать общими для всех тестов, их нужно прописывать в каждом отдельном тесте. Кроме этого, каждый тест проверяет ровно одно условие - соответствие шаблону, записанному после слова “result”. Несколько условий проверить нельзя.

Существует несколько улучшений tcltest - PTL, TTXN, New Test Package. Улучшения сводятся либо к возможности проверять несколько условий внутри одного теста, либо к понятности оформления. Например, вышеприведенный тест, переписанный на TTXN, будет выглядеть так:

```
Setup:    { ... }
Test:     { ... }
Cleanup:  { ... }
Expected:{ 1 }
```

Подводя итог обзора, можно сказать, что существует множество средств тестирования, но все они не предназначены для тестирования встраиваемых систем по одной из следующих причин:

- большие требования от встраиваемой системы - поддержка языков c++, java, python и их библиотек, предустановленный набор основных утилит Unix-подобных систем - grep, awk и т.д. (TETware, Opentest, Autotest);
- отсутствие встроенных средств для проверки вывода и результата исполнения тестируемой программы (DejaGnu);
- неудобный синтаксис для написания большого числа однотипных тестов (TclTest);
- отсутствие встроенных средств для взаимодействия со встраиваемой системой. Иными словами нет возможности легко проверить, что результат выполнения программы на удаленной системе совпадает с ожидаемым (TclTest и его улучшения - PTL, TTXN, New Test Package).

2.2. Модульное тестирование

Самые популярные среди средств модульного тестирования на языке Си - это CUnit [3], Unity [14] и Embedded Unit [6]. Одной из особенностей Unity и Embedded Unit является их легковесность - они не используют никаких библиотечных функций, за исключением setjmp() и longjmp() и, может быть, printf(). Это позволяет устанавливать их на практически любую систему. CUnit использует malloc() и соответственно проигрывает в гонке за ресурсы.

У всех приведенных фреймворков отдельный тест представляет собой функцию типа void (*test_case)(void), внутри которой располагаются макросы вида test_assert(condition), позволяющие проверять на истинность выражение с аргументами различных типов языка Си. Тесты можно объединять в наборы. В Embedded Unit и CUnit добавление теста в набор происходит при помощи функции, которой в качестве аргументов передается строка (имя теста) и указатель на функцию теста. Этот подход неудобен тем, что после написания теста его нужно не забыть зарегистрировать внутри функции main(), иначе тест не будет исполняться. В Unity название функций тестов должны начинаться с “test_”. Специальный скрипт генерирует код на Си, сходный с кодом для Embedded Unit и CUnit, именуя тест по части имени функции, следующим за “test_”. Это неудобно тем, что имена функций становятся очень длинными, а сообщения об исполнении тестов превращается в набор слитных слов и подчеркиваний.

Таким образом, для средств модульного тестирования на языке Си необходима ручная регистрация тестов, которой нет в некоторых фреймфорках для других языков, например, в googletest и JUnit. В googletest каждый отдельный тест регистрируется при помощи макроса TEST, в который передается имя теста, и имя набора тестов, к которому будет относиться данный тест. В JUnit регистрация осуществляется посредством добавления аннотации @Test над методом класса, представляющим тест. Однако эти фреймворки предназначены для C++ и Java соответственно, но не для Си.

2.3. Тестирование аппаратуры

При тестировании периферийных устройств необходимо иметь доступ к физической памяти и регистрам. Для этих нужд в контексте встраиваемых систем хоро-

шо подходит Tcl [11] - скриптовый язык высокого уровня, разработанный Джоном Оустерхаутом. С помощью Tcl можно легко подключаться к тестируемому устройству или внутреннему API приложения, вызывать тестируемые функции, проверять результат и сообщать об ошибках.

Когда нужно установить Tcl внутрь встраиваемой системы, сначала устанавливается POSIX-совместимая операционная система, на которую затем устанавливается Tcl. Для этих целей можно использовать два вида ОС:

- ОС общего назначения. В ходе ее установки на встраиваемую систему с сильно ограниченными ресурсами возникнут проблемы из-за размера образа. Помимо этого придется решать проблему с виртуальной памятью. То есть взять ОС и легко перенести ее на целевую платформу не получится;

- ОС для встраиваемых систем. Тут мы сталкиваемся с другой проблемой. Лишь небольшое число из них поддерживают POSIX, который требуется для Tcl. Кроме того, не все ОС поддерживают высокую конфигурируемость, чтобы уменьшить размер образа и не включать ничего лишнего.

Рассмотрим несколько модульных POSIX совместимых ОС для встраиваемых систем, на которые можно портировать интерпретатор Tcl.

2.3.1. eCos

eCos [16] - это популярная ОСРВ для встраиваемых систем с ограниченными ресурсами, которая характеризуется высокой портируемостью и малым потреблением оперативной памяти. Еще одной ее особенностью является модульность. Она очень легко собирается в одной из стандартных конфигураций под целевую платформу. Для этого сначала из исходных файлов eCos собирается утилита ecosconfig под платформу, на которой будет собираться eCos. С помощью этой утилиты можно выбрать целевую платформу и одну из предоставляемых конфигураций. Например, команда "ecosconfig new sparc_leon net" создаст стандартную конфигурацию под архитектуру Sparc с поддержкой сети.

Далее в конфигурацию можно добавлять дополнительные пакеты. Например, для того чтобы добавить поддержку файловой системы FAT в стандартную конфигурацию нужно выполнить команду "ecosconfig add CYGPKG_FS_FAT". И тут возникает проблема. Потому что эта команда сообщит о конфликтах - о зависимостях от других

пакетов, которые не включены в сборку. То же самое касается и добавления новых тестов в конечную конфигурацию. Если имеется множество пакетов с тестами, то при включении в сборку некоторого числа из них, придется включать и все пакеты по зависимостям. Такие конфликты не получается разрешить автоматически посредством вызова “ecosconfig resolve” из-за того, что в конфликт вовлечены пользовательские настойки [4]. Поэтому приходится добавлять все недостающие пакеты вручную.

Кроме того, модульность eCos довольно ограничена. Например, не получится включить только поддержку UDP в сетевом стеке, исключив TCP. Еще одним неудобством является отсутствие удобно читаемого списка модулей в итоговой конфигурации. ecosconfig генерирует файл ecos.ecc, размером порядка 15000-20000 строк, в котором содержатся описания модулей, интерфейсов, опций. Прочитать такой файл и понять, что на самом деле будет зашито во встраиваемую систему, довольно сложно.

2.3.2. Embox

Embox - это ОСРВ, одной из особенностей которой, также как и у eCos, является модульность и конфигурируемость. Цель, которую предполагают достигнуть разработчики - это применение кода проекта на всех стадиях разработки встроенной системы, то есть на этапе отладки оборудования в качестве системы тестирования, на этапе запуска основной ОС в качестве системного загрузчика, и наконец, в качестве полноценной ОС с различными характеристиками [17]. Система конфигурирования в данном проекте гораздо лучше развита и в частности в ней решены проблемы, которые обнаружились в eCos. На момент написания диплома в Embox не было поддержки интерпретатора Tcl. И его перенос стал одной из решаемых мной задач в рамках данной работы.

3. Реализация

В своей реализации я выделил три независимые части, необходимые для полноценного процесса разработки встроенных систем:

- Средства для интеграционного тестирования
- Средства для модульного тестирования
- Средства для тестирования и отладки периферийных устройств

Ниже будут рассмотрены детали и особенности реализации каждой из этих частей.

3.1. Интеграционное тестирование

Как было показано выше, существующие средства тестирования требуют серьезной поддержки со стороны операционной системы: поддержка языков c++, python и их библиотек, стандартные утилиты для Unix систем - grep, awk и др. То есть, от встраиваемой системы требуется наличие ОС типа Linux. Но зачастую встраиваемые системы не имеют достаточно ресурсов для этого. Однако, даже самая маленькая ОС, установленная на такую систему, имеет командный интерпретатор для запуска приложений. А так как для интеграционного тестирования система представлена как "черный ящик", то ничего кроме запуска приложения на различных входных данных и не требуется. Это означает, что можно "разгрузить" встраиваемые системы. При этом фреймворк не должен быть таким же общим как, например, DejaGnu. Часто тесты выглядят так - запустить программу с такими-то входными данными и с такими-то опциями, проверить вывод и результат возврата. Поэтому следует иметь определенный набор функций, который позволит делать это просто. Кроме того, пользователь не должен задумываться о том, что он посылает команды некоторой удаленной системе - код должен выглядеть так, будто все происходит локально [2].

В качестве языка для реализации был выбран Expect - расширение языка Tcl, предназначенное для автоматизации и тестирования в ОС Unix. Набор однотипных тестов, располагается в одном файле, и также реализуется на Tcl. Рассмотрим пример теста:

```
package require autotest
namespace import autotest::*
```

```
TEST_SETUP_TARGET     {test_setup_target}
TEST_TEARDOWN_TARGET {test_teardown_target}

TEST_SETUP_HOST     {test_setup_host}
TEST_TEARDOWN_HOST {test_teardown_host}

TEST_CASE {ping test} {
        test_assert_return_zero “ping -c 1 ”10.0.2.10 b
}
```

Первые две строчки означают, что мы подключаем реализованную мной библиотеку autotest, которая представляет собой package в Tcl. Ниже вызываются процедуры TEST_SETUP и TEST_TEARDOWN, которые регистрируют процедуры, исполняемые до и после каждого теста соответственно. Также имеются аналогичные процедуры TEST_SUITE_SETUP и TEST_SUITE_TEARDOWN, которые регистрируют обработчики, вызываемые единожды перед запуском всех тестов или после завершения последнего теста соотвественно. Постфиксы TARGET и HOST означают, что эти процедуры будут исполняться на удаленной встраиваемой системе или на локальной машине.

Каждый отдельный тест представляет собой обыкновенную процедуру на Tcl - TEST_CASE, которой передается два аргумента - название теста и сам тест. Процедура TEST_CASE вызывает процедуры setup и teardown, они зарегистрированы для выполнения на локальной машине, и далее передает управление процедуре test_exec. Эта процедура устанавливает соединение по протоколу TELNET между локальной машиной, где находятся тесты, и встраиваемой системой. Далее она вызывает процедуры setup и teardown, если они были зарегистрированы для встраиваемой системы, и затем запускает тесты.

Тесты запускаются при помощи реализованного мной скрипта run_tests (см. рисунок 1). При запуске он проверяет, были ли переданы в качестве аргументов имена файлов содержащих тесты, если нет, то ищет файл с расширением .config, в котором содержится список имен файлов с тестами. После этого запускаются все наборы тестов (один файл - один набор) в порядке перечисления. Каждый тест запускается в отдельном процессе. Это сделано для того, чтобы при ошибке в ходе выполнения теста можно было сразу же завершить процесс с ненулевым кодом возврата, а затем обработать этот код возврата уже внутри скрипта run_tests. После завершения на-

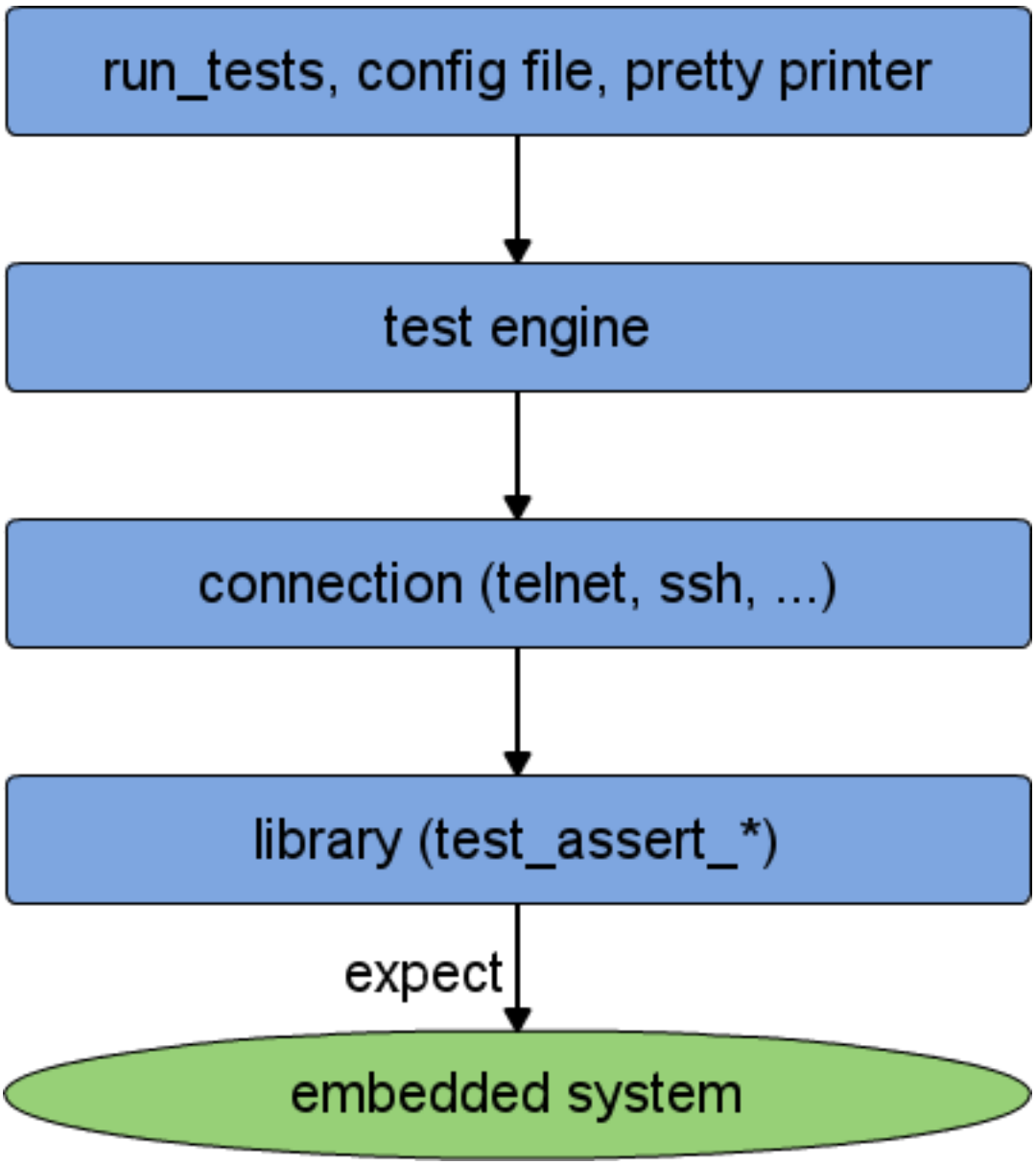

Рис. 1: Архитектура

бора тестов вызываются процедуры TEST_SUITE_TEARDOWN_TARGET/HOST, а также выводится информация о завершении набора тестов, включая информацию о завершении каждого теста. Кроме этого подробная информация о ходе выполнения тестов сохраняется в файл testrun.log в текущей директории. Скрипт run_tests можно запускать с параметром stop_on_error, который равен 0, если после ошибки в одном из тестов в исполняемом наборе нужно перейти к исполнению следующего набора тестов, и равен 1 в противном случае. Также можно передавать в качестве аргументов настройки для сетевого соединения - ip адреса встраиваемой системы и локальной машины, пароль и имя пользователя. Если эти параметры не были пере-

даны в качестве аргументов, то берутся значения по умолчанию. Увидеть подробные настройки можно запустив run_tests с опцией –verbose.

Каждый отдельный тест состоит из набора команд, которые будут последовательно исполняться командным интерпретатором на встраиваемой системе. После завершения исполнения каждой команды, ее вывод будет сравниваться с ожидаемым. Это реализовано с помощью набора процедур:

- test_assert_return_zero - проверяет, что команда завершилась с кодом возврата 0
- test_assert_string_equal - проверяет, что вывод команды точно совпадает с ожидаемым
- test_assert_regexp_equal - проверяет, что вывод команды удовлетворяет некоторому регулярному выражению

Каждая из процедур работает следующим образом. Отправляется команда по telnet с помощью внутренней процедуры Expect - send(). Далее обрабатываются случаи, когда команда зависла, когда результат ожидаемый и когда результат не совпадает с ожидаемым. В первом и третьем случаях будет выведено сообщение об ошибке и тест аварийно завершится, при этом будет напечатано имя файла, номер строки и сама строка, на которой произошла ошибка.

Имеется возможность ставить точки останова в вышеперечисленных библиотечных процедурах. Это сделано для того, чтобы можно было подключаться отладчиком к тестируемой системе ровно в тот момент, когда ей должна быть послана команда, на которой не прошел тест. Стандартные средства отладки не позволяют ставить точки останова на произвольную строку внутри процедуры, и предлагают обходиться отладочной печатью [12].

Для того чтобы поставить точку останова на вызове библиотечной процедуры, последним параметром нужно передавать строку “b”. Например, в вышеприведенном примере кода точка останова поставлена на строке ”test_assert_return_zero ”ping -c 1 10.0.2.10” b”. Если был указан параметр “b “, то сразу после входа в процедуру она сообщает об этом, печатая номер строки, название файла с точкой останова и саму строку, а затем входит в режим ожидания нажатия клавиши (команда “gets stdin”). В этот момент можно подключаться отладчиком к тестируемой системе, и после нажатия клавиши выполнение скрипта продолжается.

3.2. Модульное тестирование

Как уже было сказано, язык Си очень часто применяется в разработке встраиваемых систем [18]. Но существующие фреймворки для модульного тестирования на языке Си довольно тяжеловесны и требуют от пользователя дополнительных действий для регистрации тестов. Этот недостаток я устранил в данной работе.

В составе ОС Embox имеется внутреннее средство для модульного тестирования. Оно отличается своей легковесностью - от ОС требуется только наличие функций setjmp(), longjmp() и возможность каким-либо способом выводить информацию о результатах исполнения тестов (например, используя printf()). Тесты регистрируются при помощи макроса TEST_CASE, который добавляет структуру теста в специальную линкер секцию, упорядочивая тем самым структуры в массив. При старте ОС специальный модуль в системе проходит по этому массиву структур и запускает зарегистрированные тесты.

Отдельные тесты объединяются в наборы и регистрируются при помощи макроса EMBOX_TEST_SUITE. Ниже приведен пример теста:

```
EMBOX_TEST_SUITE("suite "name);

TEST_SETUP(setup_func);
TEST_TEARDOWN(teardown_func);

TEST_CASE "(test "name) {
        test_assert_not_null(pointer);
}
```

Этот фреймворк решает проблему с регистрацией тестов, то есть тесты объявляются единожды и нигде отдельно не регистрируются. Однако на момент написания диплома этот фреймворк был внутренним в Embox. Это означает, что тесты можно было запускать только из Embox, даже если они могли бы быть выполнены на любой POSIX совместимой ОС. Для решения данной задачи этот фреймворк был портирован мной на ОС Unix.

Для этого я полностью перенес код фреймворка без изменений, добавив при помощи команды линкера “INSERT AFTER <section>” секции, которые необходимы для корректной работы макроса TEST_CASE. Кроме того, ему требуется некоторое уникальное имя при объявлении теста. Это решается путем объявление специально-

го макроса __EMBUILD_MOD__ во время компиляции, которому присваиваются уникальные значения для разных наборов тестов. Для этих целей мной был реализован bash скрипт, которому в качестве аргументов передается директория с тестами, а он генерирует make-файл, в котором содержатся правила для компиляции тестов из указанной директории.

3.3. Тестирование аппаратуры

Для тестирования аппаратуры обычно удобно применять языки, предоставляющие возможность быстрого прототипирования. Одним из таких языков является Tcl. И как было сказано в обзоре, он хорошо подходит для тестирования аппаратуры. Давайте рассмотрим следующий пример. Пусть имеется некоторое периферийное устройство, которое нужно протестировать. Пусть у этого устройства есть регистры R и S. Требуется вычитывать значение регистра R, когда значение статусного регистра S равно нулю. И в случае если значение регистра S равно нулю, то нужно подождать 1 миллисекунду, пока значение в регистре S сбросится. Это реализует следующий код:

```
while {1} {
        set val [exec mem <&S>]
        if {val == 0} {
                puts $sock [exec mem <&R>]
                usleep 1000
        }
}
```

где &S и &R - это адреса регистров S и R соответственно. Команда mem считывает значение из физической памяти по указанному адресу. Команда usleep входит в ожидание на передаваемое ей число микросекунд (в примере - 1 миллисекунда). Этот пример доказывает необходимость двух вещей: во-первых, Tcl - для возможности написания циклов и условий, а во-вторых, возможности вызова внешних команд (mem, usleep) напрямую из интерпретатора Tcl.

У Tcl имеется популярный интерпретатор tclsh, который требует для работы POSIX. Отсюда появилось решение выбрать какую-нибудь маленькую POSIX-совместимую ОС и портировать на нее Tcl. Это сделало бы возможным тестирование аппаратуры с ограниченной памятью. Как было показано в обзоре, для этих целей лучше всего подходят Embox или eCos в силу своих маленьких размеров и конфигурироемости.

Но eCos проигрывает Embox по двум параметрам: в eCos используется ручное разрешение зависимостей, а Embox обладает более высокой конфигурироемостью. Поэтому было решено портировать tclsh на Embox. Процесс портирования можно разделить на три этапа:

1. Запуск tclsh.

2. Поддержка вызовов команд Embox из tclsh.

3. Поддержка перенаправления ввода-вывода. То есть, например, команда "set var [exec mem <addr>]" сохранит 4 байта из физической памяти по адресу <addr> в переменную var.

На первом этапе нужно было интегрировать Tcl в общую инфраструктуру Embox. Для этого я сконфигурировал tclsh так, чтобы исключить из него потоки, разделяемые библиотеки, динамическую загрузку и выгрузку библиотек, после чего размер tclsh уменьшился в два раза. Далее был добавлен соответствующий модуль для системы сборки, и реализованы недостающие функции. После этого нужно было, чтобы tclsh запустился. Для этого я добавил в образ Embox главный файл init.tcl, history.tcl, который предоставляет процедуры для управления историей вызовов, и tclIndex для того, чтобы эти файлы были обнаружены.

Далее нужно было разобраться с тем, как интерпретатор Tcl исполняет команды и расширить этот механизм возможностью вызова внутренних команд Embox. Это потребовалось для того, чтобы сделать возможным прямой доступ к физической памяти и регистрам устройств посредством внутренних команд Embox, так как Tcl не имеет подобных механизмов. Интерпретатор Tcl исполняет любую команду следующим образом: сначала он ищет ее среди своих встроенных команд, затем, если не находит, ищет в подгружаемых библиотеках. Если оба варианта поиска завершились неудачей, то это означает, что команда либо внешняя, либо такой команды не существует вовсе и вызывается процедура ::unknown из скрипта init.tcl, который входит в состав tclsh. Процедура ::unknown имеет две реализации - для win32 и для unix. Реализация для unix наиболее близка к Embox, но она сложна тем, что требует наличие некоторых системных директорий, которые отсутствуют в Embox. Кроме того, она пытается запускать исполняемые файлы из этих директорий, что невозможно сделать в Embox. Поэтому я реализовал свою версию процедуры ::unknown, которая

вызывает уже функцию интерпретатора на языке Си - Tcl_ExecObjCmd, передавая саму команду и ее аргументы. Функция Tcl_ExecObjCmd также была заменена на собственную реализацию, в которой команда, полученная из процедуры ::unknown, запускается напрямую из Embox.

И наконец, нужно было научиться перенаправлять ввод-вывод. Для этого пришлось создавать отдельную задачу (упрощенная версия процесса, не имеющая виртуального адресного пространства), в котором исполняется команда, перенаправляя свой вывод в pipe. Внутри Tcl_ExecObjCmd результат итеративно вычитывается из pipe, после чего оборачивается в специальный объект - TclNewStringObj и далее передается интерпретатору. Ниже приведен однострочный пример того, как удобно можно вычитывать и тут же отправлять по сети значения регистров устройств:

```
puts $socket [mem 0x40000000],
```

где mem - внутренняя команда Embox.

4. Апробация

Мною был разработан набор интеграционных тестов для сетевых утилит в ОС Embox: ntpdate, telnetd, ping, rlogin, ftp, sshd. Эти тесты выполняются на CI сервере, который используется в проекте. На сервере установлен Buildbot - средство для автоматизации сборки и тестирования. Buildbot собирает все стандартные конфигурации для каждой из поддерживаемых в проекте аппаратных платформ и запускает модульные и интеграционные тесты. К CI серверу можно подключаться и узнавать через веб интерфейс результаты сборки и результаты выполнения тестов для различных конфигураций Embox.

Если сборка проекта прошла успешно, Buildbot переходит к исполнению тестов. Для этого собранный образ Embox запускается на симуляторе QEMU. В момент старта системы исполняются модульные тесты, включенные в конфигурацию. В случае ошибки на каком-либо из тестов система не загружается, и выводится сообщение о том, что тест не пройден.

Если все модульные тесты были успешно пройдены, и система загрузилась, то запускается telnetd - сервис, позволяющий подключаться к системе по протоколу TELNET. После того как сервис запущен, начинается выполнение интеграционных тестов. Для этого на сервере вызывается скрипт run_tests.exp, которому передается в качестве аргумента имя файла, содержащего тесты - tests.config. В этом файле перечислены интеграционные тесты, которые должны быть выполнены. Кроме этого в нем присутствуют необходимые настройки для соединения по сети - IP адрес виртуальной машины с Embox и IP адрес виртуального tap интерфейса на сервере, при помощи которого происходит общение с виртуальной машиной. Первым исполняется тест на ping, для проверки того, что виртуальная машина видна в сети. Вторым запускается тест на telnetd для проверки того, что данный сервис корректно работает. Если этот тест завершился успешно, то это означает, что и настройка соединения прошла успешно, и можно приступать к выполнению остальных тестов. Результаты выполнения тестов журналируются в файл testrun.log, содержимое которого после завершения всех тестов выводится в stdout.

Технология тестирования аппаратуры на основе Embox и Tcl была апробирована на реальной аппаратной платформе, использующей ПЛИС с реализованным в нем процессором LEON3 и нестандартной периферией. Для этого понадобилось собрать

интерпретатор Tcl под архитектуру SPARC.

На вышеописанной аппаратной платформе имеется сетевая карта GRETH 10/100 Ethernet MAC. У нее есть контрольный регистр, который отображен на кусочек физической памяти размером 32 бита. В документации GRETH сказано, что если в шестой бит этого регистра записать 1, то сетевая карта перезагрузится, после чего бит автоматически сбрасывается. С помощью разработанной технологии я реализовал тест, который несколько раз перезагружает сетевую карту и проверяет, что бит действительно сбрасывается.

Заключение

В рамках данной дипломной работы мною был разработан набор средств для автоматизации тестирования, который может быть использован при разработке встраиваемых систем. Данный набор включает в себя средство автоматизации интеграционного тестирования, средство автоматизации тестирования периферийных устройств, а также средство модульного тестирования для языка Си. Данные средства были применены в рамках проекта по созданию программно-аппаратного комплекса на базе ПЛИС и встроенного ПО на базе ОСРВ Embox. В ходе эксплуатации данных средств было продемонстрировано улучшение процесса разработки системы в целом, в частности, ошибки выявлялись на гораздо более ранней стадии жизненного цикла продукта.

Список литературы

[1] Autotestnet. — URL: http://autotestnet.sourceforge.net.

[2] Broekman Bart, Notenboom Edwin. Testing Embedded Software. — Addison-Wesley, 2003.

[3] CUnit. — URL: http://cunit.sourceforge.net.

[4] Conflicts and constraints in eCos. — URL: http://ecos.sourceware.org/docs-3.0/user-guide/conflicts-and-constraints.html.

[5] DejaGnu. — URL: http://www.gnu.org/software/dejagnu/.

[6] Embedded Unit. — URL: http://embunit.sourceforge.net.

[7] Expect. — URL: http://expect.sourceforge.net.

[8] Group The Open. TETware. — URL: http://tetworks.opengroup.org/Products/tetware.htm.

[9] STAF. — URL: http://staf.sourceforge.net.

[10] TETware. — URL: http://arago-project.org/wiki/index.php/Opentest.

[11] Tcl. — URL: http://www.tcl.tk.

[12] Tcl debugging. — URL: http://wiki.tcl.tk/473.

[13] Unit testing tools. — 2014. — URL: http://www.opensourcetesting.org/unit_c.php.

[14] Unity. — URL: http://throwtheswitch.org/white-papers/unity-intro.html.

[15] Vandecappelle Arnout. Testing Embedded Software. — URL: http://mind.be/?page=embedded-software-testing.

[16] eCos. — URL: http://ecos.sourceware.org/docs.html.

[17] tcltest. — URL: http://wiki.tcl.tk/1502.

[18] Баррет С.Ф. Встраиваемые системы. Проектирование приложений на микроконтроллерах семейства 68HC12 / HCS12 с применением языка C. — Книга по Требованию, 2007.