

Разработка IoT для ESP32 и ESP8266 с помощью JavaScript

Практическое руководство
по XS и Moddable SDK

—
Питер Ходди
Лиззи Прадер

Разработка IoT для ESP32 и ESP8266 с помощью JavaScript

**Практическое
руководство по XS
и Moddable SDK**

**Питер Ходди
Лиззи Прадер**

Оглавление

Об авторе.....	xvii
----------------	------

Введение.....	xxvii
---------------	-------

Глава 1: Начало работы.....	1
------------------------------------	----------

Требования к компонентам	1
--------------------------------	---

Требования к программному обеспечению	6
---	---

Загрузка примера кода	6
-----------------------------	---

Настройка среды сборки	7
------------------------------	---

Использование <code>xsbug</code>	7
--	---

Важные функции для примеров в этой книге.....	9
---	---

Примеры запуска	10
-----------------------	----

Установка хоста	10
-----------------------	----

<code>mcconfig</code>	11
-----------------------------	----

Подтверждение установки хоста	11
-------------------------------------	----

Установка <code>helloworld</code>	12
---	----

<code>mcrun</code>	13
--------------------------	----

Завершение.....	13
-----------------	----

TABLE OF CONTENTS

Исправление проблем	14
Устройство не подключено/не распознано.....	14
Несовместимая скорость передачи	16
Устройство не в режиме загрузчика	17
Добавление дисплея	18
Подключение дисплея к ESP32	19
Подключение дисплея к ESP8266	20
Установка helloworld-gui	22
Заключение	24
Глава 2:JavaScript для программистов	25
Основной синтаксис	26
“Hello, world”	26
Точки с запятой.....	27
Объявление переменных и констант	27
Оператор if.....	29
Оператор switch	30
Циклы	32
Типы.....	33
undefined	34
Логические значения	35
Числа	35
Строки	39
Функции	47
Объекты	58
null	68
Сравнения	68
Сравнение объектов	70
Ошибки и исключения	71

Классы.....	75
Конструктор класса и методы.....	75
Статические методы	77
Подклассы	78
Частные поля.....	84
Частные методы.....	86
Использование функций обратного вызова в классах	87
Модули	90
Импорт из модулей	90
Экспорт из модулей	93
Модули ECMAScript и модули CommonJS.....	94
Глобальные переменные	94
Массивы.....	96
Сокращение массива	97
Доступ к элементам массива.....	97
Перебор массивов	99
Добавление и удаление элементов массива	101
Поиск массивов	103
Сортировка массивов.....	104
Двоичные данные	106
Буфер массива.....	106
Типизированные массивы	107
Представления данных.....	114
Управление памятью.....	118
Класс Date	121
Программирование, управляемое событиями.....	125
Заключение	126

Глава 3: Сеть.....	127
О сети	128
Подключение к Wi-Fi.....	129
Подключение из командной строки	130
Подключение к коду.....	131
Подключение к любой открытой точке доступа	134
Установка сетевого хоста.....	137
Примеры установки	137
Получение информации о сети.....	137
Выполнение HTTP-запросов	138
Основы.....	138
GET	140
Потоковая передача.....	141
GET JSON	143
Создание подкласса HTTP-запроса.....	145
Настройка заголовков запроса	147
Получение заголовков ответа	148
POST	149
Обработка ошибок.....	150
Защита соединений с помощью TLS	151
Использование TLS с классом SecureSocket	152
Публичные сертификаты	152
Частные сертификаты	154
Создание HTTP-сервера	155
Основы.....	155
Ответ на запрос.....	157
Ответ на JSON PUT.....	158

Получение запроса на потоковую передачу	160
Отправка потокового ответа	161
mDNS	163
Требование имени	164
Поиск услуги.....	165
Реклама услуги.....	167
Веб-сокет	168
Подключение к серверу WebSocket	169
Создание сервера WebSocket.....	171
MQTT.....	173
Подключение к серверу MQTT.....	173
Подписка на тему	175
Публикация в теме.....	176
SNTP	177
Дополнительные темы	178
Создание точки доступа Wi-Fi.....	178
Промисис и асинхронные функции.....	181
Заключение	183
Глава 4: Bluetooth с низким энергопотреблением (BLE)	185
Основы BLE	186
Центральные и периферийные устройства GAP.....	186
Клиенты и серверы GATT	187
GAP против GATT	187
Профили, услуги и характеристики	188
BLE API модифицируемого SDK	189
Класс BLEClient	190
Класс BLEServer	191

TABLE OF CONTENTS

Установка BLE-хоста	191
Создание сканера BLE	192
Создание двусторонней связи	193
Подключение к периферийному устройству	195
Получение уведомлений	198
Создание монитора сердечного ритма	200
Определение и развертывание сервисов	200
Реклама	203
Установление соединения	204
Отправка уведомлений	206
Ответ на запросы на чтение	211
Установление безопасной связи	215
Безопасный монитор сердечного ритма	215
Заключение	220
Глава 5: Файлы и данные	221
Установка хоста файлов и данных	222
Файлы	222
Классы файлов	223
Пути к файлам	223
Операции с файлами	224
Запись в файл	226
Чтение из файла	229
Каталоги	231
Получение информации о файловой системе	233
Preferences (предпочтения)	234
Класс Preference	234
Имена Preference	235

Данные Preference	235
Чтение и запись Preferences	236
Удаление Preferences	237
Не используйте JSON	237
Безопасность	239
Ресурсы	239
Добавление ресурсов в проект	240
Доступ к ресурсам	241
Использование ресурсов.....	241
Прямой доступ к флэш-памяти.....	243
Основы аппаратного обеспечения флэш-памяти.....	244
Доступ к разделам флэш-памяти	246
Пример: часто обновляемое целое число	251
Закключение	254
Глава 6: Аппаратное обеспечение.....	255
Установка аппаратного хоста.....	256
Примечания по монтажу.....	256
Следование инструкциям по подключению	256
Устранение неполадок с монтажом.....	257
Мигание LED	258
Чтение кнопки	260
Другие режимы цифрового ввода.....	262
Мониторинг изменений	266
Управление трехцветным светодиодом	267
Подключение LED	268
Инструкции по подключению ESP32	268
Инструкции по подключению ESP8266	269

TABLE OF CONTENTS

Использование цифрового сигнала с трехцветным LED.....	270
Использование ШИМ с трехцветным LED	271
Вращение вала сервопривода	274
Инструкции по подключению ESP32	275
Инструкции по подключению ESP8266	275
Разбор кода сервопривода.....	276
Получение температуры	277
TMP36	278
TMP102	282
Заключение	294
Глава 7: Звук	295
Варианты динамиков.....	295
Добавление аналогового динамика.....	297
Инструкции по подключению ESP32	298
Инструкции по подключению ESP8266	299
Добавление платы I2S и цифрового динамика	300
Инструкции по подключению ESP32	300
Инструкции по подключению ESP8266	302
Установка аудиохоста	303
Класс AudioOut.....	304
Конфигурация аудиовыхода.....	304
Аппаратные аудиопротоколы	304
Форматы аудиоданных.....	308
Сжатие звука.....	310
Установка длины аудио очереди	310
Воспроизведение аудио с помощью AudioOut	311
Создание экземпляра AudioOut.....	311

Воспроизведение одного звука	312
Повторение звука	313
Обратные вызовы для синхронизации аудио.....	313
Использование команд для изменения громкости.....	314
Воспроизведение последовательности звуков	314
Воспроизведение звуков одновременно	315
Проигрывание части звука	316
Очистка очереди аудио.....	317
Заключение	317
Глава 8: Основы графики	319
Зачем добавлять дисплей?.....	320
Преодоление аппаратных ограничений.....	322
Влияние скорости пикселей на частоту кадров	323
Рисование рамок.....	324
Рендеринг строки сканирования.....	326
Ограничение области рисования.....	327
Пиксели.....	331
Форматы пикселей.....	332
Настройка хоста для пиксельного формата.....	333
Свобода выбора дисплея	335
Графические активы.....	335
Маски.....	336
Шрифты.....	341
Цветные изображения.....	345
Вращение дисплея.....	349
Ротация в программном обеспечении.....	350
Ротация устройства.....	351

TABLE OF CONTENTS

Росо или Piu?	352
Заключение	355
Глава 9: Рисование графики с помощью Росо.....	357
Установка хоста Росо	357
Подготовка к рисованию	358
Рисование прямоугольников.....	361
Заполнение экрана.....	361
Обновление части экрана	363
Рисование случайных прямоугольников	364
Рисование смешанных прямоугольников	366
Рисование растровых изображений.....	370
Маски рисования.....	370
Рисование цветных изображений	377
Рисование изображений JPEG	378
Заполнение цветными изображениями	381
Рисование замаскированных цветных изображений.....	383
Рисование текста.....	387
Рисование тени текста	388
Измерение текста	389
Усечение текста	390
Обтекание текста.....	391
Дополнительные техники рисования.....	393
Ограничение текста блоком	393
Простое повторное использование кода рисования	396
Эффективная визуализация градиентов	398
Сенсорный ввод	402
Доступ к сенсорному драйверу	402
Чтение сенсорного ввода	402

Использование мультитач	404
Применение вращения	404
Заключение	405
Глава 10. Построение интерфейсов с помощью Piui...	407
Ключевые понятия	408
Все является объектом.....	408
Каждый элемент интерфейса является объектом контент....	410
Не все объекты Piui являются объектами содержимого.....	411
Установка хоста Piui	413
«Привет, мир» с Piui.....	414
Шрифты.....	416
Добавление цвета	423
Реакция на события поведением	428
Добавление изображений	435
Рисование части изображения.....	436
Рисование нескольких значков из одного изображения	438
Использование масок	440
Мозаика изображений	442
Создание составных элементов интерфейса.....	448
Создание заголовка.....	449
Создание адаптивных макетов.....	454
Расположение строк и столбцов	456
Прокрутка содержимого	460
Шаблоны для объектов контента.....	463
Создание класса шаблона кнопки.....	463
Аргументы конструктора контента	466

TABLE OF CONTENTS

Доступ к объектам содержимого в контейнере	471
Использование первого, последнего, следующего и предыдущего	471
Accessing Children by Index and Name	472
Accessing Content with Anchors	472
Defining and Triggering Your Own Events	476
Triggering Events on a Content Object	477
Распространение событий внутри контейнера	480
Всплывающие события вверх по иерархии сдерживания.....	484
Анимация.....	488
Уравнения смягчения	488
Анимация объектов контента	489
Анимация переходов.....	494
Рисование графика в реальном времени.....	497
Добавление экранной клавиатуры.....	500
Код пользовательского интерфейса с помощью модулей.....	506
Модули.....	507
Логика приложения	508
Экран-заставка.....	511
Главный экран	513
Добавление дополнительных экранов	517
Заключение	518
Глава 11: Добавление собственного кода	519
Установка хоста	520
Генерация случайных целых чисел	521
Создание собственной функции	521
Реализация нативной функции.....	522
Использование аппаратного генератора случайных чисел.....	523

ОГЛАВЛЕНИЕ

Ограничение случайных чисел диапазоном	526
Сравнение подходов случайных чисел	528
Класс BitArray	529
Использование памяти, выделенной <code>ArrayBuffer</code>	530
Использование памяти, выделенной <code>calloc</code>	533
Уведомления о сигналах Wi-Fi.....	542
Тестовый код	542
Класс <code>WiFiRSSINotify</code>	544
Собственная структура <code>RSSINotifyRecord</code>	544
Конструктор	545
Деструктор.....	550
Функция закрытия	550
Функция обратного вызова	551
Дополнительные методы	554
Отладка нативного кода	554
Доступ к глобальным переменным.....	555
Получение возвращаемого значения функции	556
Получение значений.....	556
Установка значений.....	557
Определение типа значения.....	559
Работа со строками	560
Обеспечение корректности указателей буфера.....	561
Интеграция с C++	562
Использование потоков.....	562
Заключение	563
Глоссарий.....	565

Об авторах

Питер Ходди — инженер и предприниматель, специализирующийся на клиентском программном обеспечении. Он известен созданием компактного и эффективного кода, расширяющего границы пользовательского опыта на потребительском оборудовании. Программное обеспечение, созданное им и его командой, используется в потребительских продуктах массового рынка таких компаний, как Apple, Palm, Sling, HP, Sony и Whirlpool. Питер признает, что первыми пользователями любого продукта являются разработчики, создающие его, и что эти разработчики не могут создавать привлекательные потребительские продукты на нестабильном, сложном или запутанном фундаменте. Поэтому он выступает за инвестиции в отличные инструменты и простую архитектуру времени выполнения.

Питер основал несколько компаний, в том числе Kinoma, которая объединилась с Marvell Semiconductor. Он руководил разработкой QuickTime в Apple в 1990-х годах в качестве заслуженного инженера. Он внес свой вклад в разработку формата файлов QuickTime и его принятие ISO в стандарт MPEG-4. В настоящее время он является членом комитета по стандартам языка JavaScript (ECMA TC39) и председателем ECMA TC53 по «Умным носимым системам и сенсорным устройствам». Питер особенно гордится своей работой по размещению фреймворка KinomaJS и Darwin Streaming Server с открытым исходным кодом. Он продолжает договариваться с десятью патентами, которые носят его имя.

Лиззи Прадер — инженер-программист в компании Moddable, расположенной в районе залива Сан-Франциско. Она скептически относится к IoT, работая в сфере IoT, надеясь сделать потребительские продукты IoT и другие встроенные системы более открытыми и настраиваемыми для конечного пользователя. Она специализируется на разработке пользовательских интерфейсов с сенсорным экраном для встраиваемых систем и создании ресурсов для разработчиков.

ABOUT THE AUTHORS

До Moddable Лиззи работала инженером по связям с разработчиками в команде Kinoma в Marvell Semiconductor, помогая клиентам получить максимальную отдачу от продуктов прототипирования Kinoma на основе JavaScript. Она получила степень бакалавра компьютерных наук в Калифорнийском университете в Беркли. Когда она не сидит за компьютером, ей нравится быть на свежем воздухе (в частности, бегать, ходить в походы и плавать), читать и играть на пианино.

Предисловие

Те из вас, кто читал Википедию от корки до корки, помнят определение гибридной силы. Для остальных из вас:

Гетерозис, сила гибрида или усиление аутбридинга — это улучшенная или повышенная функция любого биологического качества в гибридном потомстве. Потомство является гетерозисным, если его черты усиливаются в результате смешения генетических вкладов его родителей.

— Википедия, страница 69105.

Практическим результатом этого является то, что если вы возьмете отдельные, инбредные штаммы, скажем, кукурузы и скрестите их вместе, вы получите большое сильное сильное растение, отсюда и название «гибридная сила».

Moddable SDK, описанный в этой книге, представляет собой гибрид разработки для встраиваемых систем и JavaScript. Использование Moddable — это короткий путь к очень большому количеству кукурузы.

Если вы разработчик встраиваемых систем, вы наслаждаетесь возможностью приблизиться к металлу, программировать на крошечных недорогих устройствах без возможностей, предлагаемых разработкой на больших системах. Написание на C/C++ и/или языке ассемблера дает вам большую степень контроля, но вы часто сталкиваетесь с проблемой вписывания функциональности в эти небольшие устройства, боретесь с хрупкими средами разработки и отладки и создаете специальные способы обновления кода и управления для конкретных устройств. Ресурсы. Если встроенное устройство оснащено дисплеем или поддерживает беспроводную связь, вам необходимо найти соответствующие инструменты для создания, моделирования и тестирования широкого спектра функций при отсутствии многофункциональной базовой ОС. Большая часть вашей энергии уходит на управление ограничениями этих небольших систем, а не на сами приложения.

FOREWORD

Если вы разработчик JavaScript, вам понравится его производительность, универсальность и повсеместность. Язык снисходителен к новичку, но чрезвычайно эффективен в руках мастера. Он достаточно универсален, чтобы его можно было использовать как во фронтенд-разработке, так и во внутренней инфраструктуре. Его огромное сообщество постоянно обогащает язык и библиотеки и делает его, пожалуй, самым популярным языком за всю историю. Вся эта универсальность и мощь имеют свою цену; до сих пор надежная версия JavaScript не масштабировалась до такой степени, что она поместилась бы на маленьком встроенном устройстве.

Moddable берет лучшее из этих двух штаммов и смешивает их вместе в развивающемся классе новых мощных микроконтроллеров. Разработчикам встраиваемых систем больше не нужно смотреть на JavaScript с завистью. Теперь у них есть доступ к универсальному языку высокого уровня, который по-прежнему позволяет им держаться подальше от металла и сохранять жесткий контроль. Разработчики JavaScript могут получить доступ к смехотворно недорогим устройствам без каких-либо проблем традиционной разработки встраиваемых систем, но при этом работать с возможностями, присущими гораздо более крупным системам.

Работать продуктивно в среде Moddable с его богатым SDK для управления безопасным подключением к Интернету, Wi-Fi, Bluetooth, звуком, графикой, пользовательскими интерфейсами и многим другим невероятно легко. его элегантная полнофункциональная отладка; и широкий выбор целевого оборудования, от недорогого до почти бесплатного.

Моя основная работа заключается в инвестировании в технологические стартапы, и поэтому мне больно видеть, как компании пытаются разрабатывать встраиваемые системы с помощью грубых инструментов. Moddable может сэкономить время и деньги разработчиков и создавать красиво отшлифованные встраиваемые приложения. Как инвестор, я люблю инструменты, которые снижают риск и делают компании более эффективными с точки зрения капитала. Как заядлый производитель, я люблю инструменты, которые делают работу еще более увлекательной.

Если вы разрабатываете встраиваемые продукты, являетесь любителем или просто любите писать код, гибридная мощь Moddable меняет правила игры. Иди выращивай кукурузу.

— Питер Барретт

Вступление

Эта книга представляет собой практическое руководство по написанию программ для продуктов IoT. Каждая глава заполнена компактными, сфокусированными примерами, на которых вы можете учиться, изучать, запускать и изменять. Когда вы прочтете эту книгу, вы будете знать основы создания сложных продуктов IoT на недорогом оборудовании с использованием современного JavaScript.

Продукты IoT отличаются от традиционных продуктов двумя способами: они могут запускать программы и могут обмениваться данными. Их связь часто осуществляется через Интернет, но может быть и более локальной — например, между продуктами в вашей домашней сети Wi-Fi или с вашим телефоном через соединение Bluetooth.

Продукты IoT часто создаются путем добавления микроконтроллера(МК) с возможностями Wi-Fi или Bluetooth к традиционному продукту. Стоимость добавления МК с коммуникационными возможностями сегодня составляет около одного доллара и продолжает падать. При такой цене почти каждый продукт будет продуктом Интернета вещей — не только телевизоры и термостаты, но и лампочки, выключатели, электрические розетки, дверные замки, оконные шторы, открыватели гаражных ворот, потолочные вентиляторы, рисоварки, холодильники, и больше.

Код в этой книге работает на МК ESP32 и ESP8266 от Espressif, которые предлагают невероятную мощность по беспрецедентной цене. Неудивительно, что они широко используются в продуктах IoT и чрезвычайно популярны среди производителей и любителей. Однако то, что вы узнаете из этой книги, не ограничивается этими МК; их можно применять к растущему числу МК таких производителей, как Nordic, Qualcomm и Silicon Labs.

Добавление компонентов IoT к традиционному продукту — это простая часть; сложная часть - это программное обеспечение. Программное обеспечение определяет функции и поведение продукта.

INTRODUCTION

Он определяет, надежен ли продукт и прост в использовании, защищен ли он от внешних атак и уважает ли он конфиденциальность пользователей. Программное обеспечение решает, с какими другими продуктами продукт может взаимодействовать, его энергопотребление, легкость добавления новых функций с течением времени и многое другое.

Программы имеют основополагающее значение для продуктов IoT, однако большая часть отрасли продолжает писать для них программы, используя те же инструменты и методы, которые разработчики встраиваемых программ использовали на протяжении десятилетий. В то время как аппаратное обеспечение продвинулось на порядки, программное обеспечение — нет. Это проблема, потому что сегодня от программного обеспечения продукта IoT ожидают гораздо большего, чем от программы цифрового термостата 1999 года.

JavaScript: новый инструмент

В этой книге представлен новый способ создания программ для продуктов IoT — способ, при котором не нужно пытаться изобретать велосипед, начиная все сначала. Он добавляет один новый инструмент к многим, которые разработчики встраиваемых программ использовали годами: язык программирования JavaScript. Может показаться преувеличением предположение, что язык программирования может преобразовать программы продукта IoT, но это возможно. Современный язык высокого уровня — идеальное противоядие от низкоуровневых методов разработки, которым уже несколько десятков лет.

JavaScript может показаться маловероятной отправной точкой для будущих поколений продуктов IoT. В конце концов, JavaScript начинался как простой язык программирования, чтобы добавить немного интерактивности веб-страницам на заре Интернета. Но по мере развития сети JavaScript развивался вместе с ней; теперь это официально определенный язык программирования, стандартизированный международным комитетом, в который входят представители крупных компаний, включая Apple, Facebook, Google, Microsoft, Mozilla и PayPal. Язык был защищен двумя десятилетиями атак в веб-браузере. Он стал мощным благодаря требованиям все более сложных веб-страниц. Кроме того, он стал надежным и простым в использовании, чтобы удовлетворить потребности миллионов веб-разработчиков по всему миру.

Разработчики, работающие на JavaScript, невероятно продуктивны и проворны. Через несколько часов после стихийного бедствия появляются новые веб-сайты с впечатляющими функциями, реализованными на JavaScript. Крупные веб-сайты, такие как Facebook и LinkedIn, не просто построены на JavaScript, но и ежедневно внедряют новые функции с помощью JavaScript. Серверный JavaScript с Node.js теперь поддерживает целые компании, и многие мобильные приложения созданы на JavaScript.

JavaScript готов к использованию разработчиками IoT. Эта книга подготавливает разработчиков Интернета вещей к использованию JavaScript.

Последние технические достижения

JavaScript не всегда подходил для IoT. Хотя механизмы JavaScript, встроенные в веб-браузеры, невероятно быстры, за эту скорость приходится платить. Эти механизмы слишком велики, чтобы хранить их в недорогом микроконтроллере, и им требуется на несколько порядков больше памяти, чем встроено в аппаратное обеспечение. Движок XS JavaScript отличается: он создан Kinoma и поддерживается Moddable, он оптимизирован для ограничений микроконтроллеров, на которых работают недорогие продукты IoT. XS очень маленький и все еще достаточно быстрый.

Чтобы сохранить небольшой движок, вы можете ожидать, что XS будет поддерживать только часть языка программирования JavaScript, но это не так. XS реализует более 99% версии спецификации языка JavaScript 2020 года; это больше, чем в любом веб-браузере. В качестве оптимизации XS позволяет вам отказаться от многих функций языка, чтобы сделать движок еще меньше, но это ваш выбор. Все функции есть, если вы хотите их использовать.

Недостаточно иметь небольшой, эффективный и совместимый движок JavaScript. Язык JavaScript выполняет только вычисления; ему нужна среда выполнения для взаимодействия с внешним миром. Для веб-страницы среда выполнения — это веб-браузер; для веб-сервера среда выполнения — Node.js. Для продуктов IoT в этой книге в качестве среды выполнения используется Moddable SDK.

Moddable SDK включает в себя большой набор эффективных модулей для решения общих задач, таких как общение через Интернет, управление оборудованием IoT, управление безопасностью и хранение данных. В Moddable SDK также есть что-то необычное для среды выполнения IoT: глубокая поддержка высококачественных, многофункциональных пользовательских интерфейсов на недорогих сенсорных экранах. Хотя вы, возможно, еще не рассматриваете экран для своего продукта IoT, ваши будущие клиенты, вероятно, хотели бы, чтобы вы это сделали, потому что экран упрощает настройку и использование вашего продукта и позволяет ему предоставлять больше функций.

Для встраиваемых программистов и веб-разработчиков

Нет ветеранов-разработчиков программ для Интернета вещей, потому что эта область слишком новая. Мы все еще учимся. Большинство разработчиков IoT имеют один из двух подходов: есть разработчики, создающие программы для встраиваемых систем, которым теперь необходимо создавать продукты IoT с возможностью подключения, сложным поведением и бесчисленными функциями; и есть разработчики, работающие в Интернете, которых призывают создавать продукты IoT на оборудовании с ограничениями ресурсов и производительности, невообразимыми в Интернете.

JavaScript находится на пересечении разработчиков из мира встроенного и веб-программного обеспечения. Сам язык JavaScript воплощает это пересечение, прибыв из Интернета с синтаксисом, очень похожим на языки C и C++, давно любимые разработчиками встраиваемых систем. Код JavaScript и C/C++ может вызывать друг друга, что делает естественным их объединение в продукте. JavaScript уже давно используется для общения в Интернете, что даже привело к появлению формата обмена данными JSON, обычно используемого продуктами IoT для связи.

Эта книга предназначена для разработчиков, имеющих опыт работы как со встроенным программным обеспечением, так и с веб-разработкой. Для опытных разработчиков встраиваемых систем, незнакомых с JavaScript, первая глава знакомит программистов на C и C++ с JavaScript.

Для опытных веб-разработчиков, начинающих работу со встроенными системами, книга содержит советы по оптимизации памяти и производительности, уникальные для разработки продуктов IoT.

Организация этой книги

Вот как устроена эта книга:

- **Глава 1** проведет вас через описание всего аппаратного и программного обеспечения, необходимого для этой книги, и запуск вашего первого приложения JavaScript на микроконтроллере. Попутно она показывает, как использовать полезные функции `xsbug`, отладчика исходного кода JavaScript.
- **Глава 2** представляет собой краткое практическое введение в JavaScript для разработчиков, уже знакомых с C или C++. Представленный здесь язык JavaScript — это тот же язык, который используется в Интернете, но, поскольку эта книга посвящена встраиваемым системам, в главе рассматриваются некоторые аспекты JavaScript, редко используемые веб-разработчиками.
- **Глава 3** посвящена подключению вашего IoT-устройства к сети, включая различные способы подключения, способы связи с использованием различных сетевых протоколов, способы создания безопасных соединений, а также расширенные темы о том, как превратить ваше устройство в частную станцию Wi-Fi и как использовать обещания JavaScript с сетевыми API.
- **Глава 4** посвящена Bluetooth с низким энергопотреблением (BLE) — беспроводной связи, широко используемой между двумя устройствами, находящимися в непосредственной

близости друг от друга. Продукты выбирают использование BLE вместо Wi-Fi, если особенно важно свести к минимуму потребление энергии и когда прямая связь с другим устройством, например мобильным телефоном, является приемлемой альтернативой доступу в Интернет.

- В [главе 5](#) объясняется, как работать с хранимыми данными во встроенных системах, учитывая их ограничения на размер кода, ограниченный объем ОЗУ и ограничения производительности. В нем объясняются три основных способа хранения данных — файлы, настройки и ресурсы — и вводится прямой доступ к флэш-памяти — передовой метод, обеспечивающий наибольшую гибкость.
- [Глава 6](#) знакомит вас с написанием собственного кода JavaScript для взаимодействия с оборудованием. Он включает в себя примеры, для которых требуется всего несколько широкодоступных недорогих датчиков и исполнительных механизмов, и показывает, как взаимодействовать с ними напрямую, используя различные аппаратные протоколы.
- В [главе 7](#) объясняется, как воспроизводить звуки с помощью недорогого динамика, который легко подключить непосредственно к ESP32 или ESP8266. Вы также узнаете, как добиться более высокого качества воспроизведения звука с помощью внешнего аудиодрайвера I2S и как выбрать оптимальный аудиоформат для своего проекта.
- [Глава 8](#) сначала описывает преимущества и экономическую эффективность добавления дисплея к вашему продукту IoT, а затем основы графики на микроконтроллерах, включая справочную информацию об оптимизации и ограничениях, информацию о том, как добавлять графические активы в проекты, а также введение в различные способы рисования. методы. Следующие две главы содержат более подробную информацию.

- В [главе 9](#) рендерер Росо обсуждается на примерах, показывающих, как предоставлять высококачественную и высокопроизводительную графику на недорогих МК. Росо является частью графической библиотеки Commodetto, которая добавляет функции, включая внеэкранные графические буферы, растровые изображения и создание экземпляров графических активов из ресурсов, а также показано на некоторых примерах.
- [Глава 10](#) посвящена Piu, объектно-ориентированной среде пользовательского интерфейса, использующей средство визуализации Росо для отрисовки и упрощения процесса создания сложных пользовательских интерфейсов. В главе представлен обзор того, как работает Piu, и представлены его ключевые возможности с помощью ряда примеров.
- [Глава 11](#) представляет расширенную тему XS в C, низкоуровневого C API, предоставляемого механизмами XS JavaScript, чтобы вы могли интегрировать код C в свои проекты JavaScript (или код JavaScript в свои проекты C). Использование XS в C позволяет оптимизировать использование памяти, повысить производительность, повторно использовать существующие библиотеки кода C и C++ и получить доступ к уникальным аппаратным возможностям.

Что дальше

Изучая главы этой книги, вы приобретете новые знания и навыки. Впереди много путей:

- Если вы профессиональный разработчик продуктов, вы можете применить полученные знания к своим продуктам IoT.

INTRODUCTION

- Если вы управляете облачной службой, вы можете создавать модули, которые помогут разработчикам продуктов IoT упростить подключение к вашей службе.
- Если вы являетесь производителем датчиков, вы можете создавать модули для сбора данных с ваших датчиков, упрощая процесс использования этих датчиков в продуктах IoT.
- Если вы мастер или любитель, вы можете использовать свои новые знания и навыки в своем следующем проекте.

Когда вы освоите использование JavaScript для создания продуктов IoT, вы можете обнаружить, что тратите меньше времени и энергии на то, чтобы заставить продукт работать вообще, и больше времени на то, чтобы он работал лучше, делал больше и был проще в использовании. В этом заключается реальная сила создания продуктов IoT на JavaScript.

Одна из причин, по которой язык JavaScript так быстро развивался в течение столь долгого времени, заключается в том, что разработчики JavaScript всегда делились своими знаниями и опытом в сетевых сообществах. Это совместное использование привело к тому, что огромное количество исходного кода JavaScript было опубликовано под бесплатными лицензиями на программы с открытым исходным кодом. Этот код доступен для изучения, улучшения и дальнейшего развития. Когда вы создаете свои собственные проекты Интернета вещей с помощью JavaScript, подумайте о том, чтобы поделиться ими с другими, чтобы другие разработчики могли учиться у вас и опираться на вашу работу.

Когда вы захотите узнать больше, в Интернете есть множество ресурсов, которые помогут вам:

- Репозиторий этой книги на GitHub содержит все примеры из книги, обновления и исправления ошибок. Вы можете открыть тему, чтобы сообщить о проблеме или задать вопрос.

<https://github.com/Moddable-OpenSource/iot-product-dev-book>

- Moddable SDK включает справочную документацию и примеры для всех возможностей, представленных в этой книге. Он также включает исходный код модулей, так что вы можете узнать, как они работают, и улучшить их в соответствии со своими потребностями. Если у вас есть вопросы или вы обнаружили проблему, вы можете открыть тему.

[https://github.com/Moddable-OpenSource/
moddable](https://github.com/Moddable-OpenSource/moddable)

- В блоге Moddable есть подробные статьи о создании продуктов IoT с использованием JavaScript. Вы можете узнать о последних языковых возможностях, поддерживаемых движком XS JavaScript, возможностях безопасности и новых функциях Moddable SDK.

<https://blog.moddable.com/blog/>

- Twitter — отличный способ быть в курсе последних событий. Вы можете следить за авторами этой книги в Твиттере по адресу @lizzieprader и @phoddie и подписываться на @moddabletech, чтобы быть в курсе последних новостей о Moddable SDK, мероприятиях, встречах и новых проектах, созданных разработчиками.

ГЛАВА 1

https://t.me/it_boooks

Начало работы

В этой главе вы соберете все аппаратное и программное обеспечение, необходимое для этой книги, и запустите свое первое приложение JavaScript на микроконтроллере. Попутно в этой главе также показано, как использовать полезные функции `xdebug`, отладчика исходного кода JavaScript.

Установка всех программных средств и настройка среды разработки занимает немного времени, но как только вы сможете запустить один пример, вы сможете запустить любой пример из этой книги. У вас также будет все необходимое, чтобы начать писать собственные приложения с помощью Moddable SDK.

Требования к оборудованию

Для большинства примеров в этой книге требуется очень мало аппаратного обеспечения, но вам, по крайней мере, потребуется следующее:

- Компьютер с портом USB (версия macOS Sierra 10.12 или более поздней версии, Windows 7 Pro SP1 или более поздней версии или Linux)
- Кабель Micro USB (высокоскоростной, с возможностью синхронизации данных)
- Модуль ESP32 NodeMCU или модуль ESP8266 NodeMCU.

Примечание Все примеры выполняются на ESP32 или ESP8266, за исключением примеров, использующих Bluetooth с низким энергопотреблением (BLE), как описано в главе 4, работают только на ESP32, поскольку ESP8266 не поддерживает BLE. Если вам интересно поэкспериментировать с примерами BLE из этой книги, вам понадобится ESP32.

Примеры были протестированы с модулями ESP32 и ESP8266, показанными на рисунке 1-1..

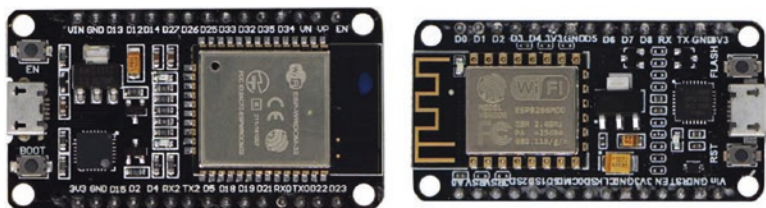


Рис. 1-1. ESP32 (слева) и ESP8266 (справа)

Примеры с датчиками и исполнительными механизмами (главы 6 и 7) требуют нескольких дополнительных компонентов:

- Тактильная кнопка
- Трехцветный светодиод (общий анод)
- Три резистора по 330 Ом
- Микро сервопривод
- Датчик температуры TMP36
- Датчик температуры TMP102
- Миниатюрный динамик (8 Ом, 0,5 Вт)
- Перемычки

Эти аппаратные компоненты показаны на рис. 1-2. Более подробная информация о том, где их можно приобрести, содержится в главах, где они обсуждаются.



Рис. 1-2. Компоненты для глав 6 и 7

Примеры, использующие визуализатор Pico (глава 9) или инфраструктуру пользовательского интерфейса Piu (глава 10), можно запускать на аппаратном симуляторе на вашем компьютере, но настоятельно рекомендуется использовать реальный дисплей и запускать их на ESP32 или ESP8266. Если вам удобно соединять компоненты на макетной плате, вот что вам нужно:

- Сенсорный дисплей ILI9341 QVGA (рис. 1-3), который доступен на eBay и в других местах в Интернете; введите «spi display 2.4 touch» и вы найдете несколько недорогих вариантов. Обратите внимание, что хотя этот дисплей работает хорошо, есть много других вариантов. Moddable SDK включает встроенную поддержку нескольких других дисплеев различной стоимости и качества; см. каталог [document/displays](#) Moddable SDK для получения дополнительной информации.

ГЛАВА 1 НАЧАЛО РАБОТЫ

- Макетная плата
- Перемычки папа-мама

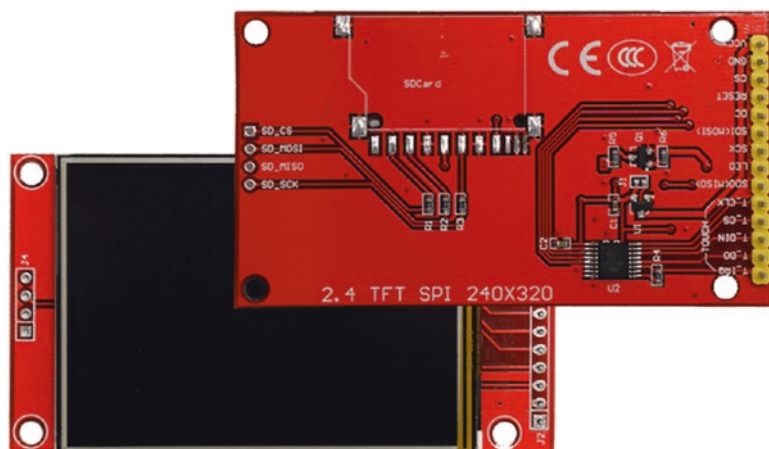


Рис. 1-3. Сенсорный QVGA-дисплей ILI9341.

Если вы не хотите использовать макетную плату, вы можете приобрести Moddable One или Moddable Two на веб-сайте Moddable. Moddable One — это ESP8266, подключенный к емкостному сенсорному экрану; Moddable Two — это ESP32, подключенный к тому же сенсорному экрану. Оба поставляются в виде готовых к использованию комплектов для разработки в компактном форм-факторе. На рис. 1-4 показан модуль с возможностью модификации.

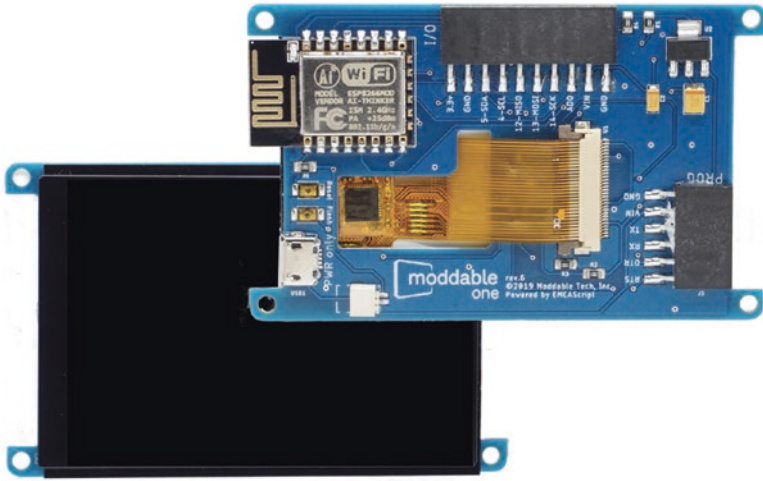


Рис. 1-4. Модифицируемый вариант

Moddable SDK также поддерживает комплекты разработки на основе ESP32 со встроенными дисплеями. Популярным выбором является M5Stack FIRE, показанный на рис. 1-5. Дополнительные сведения о поддерживаемых комплектах разработки см. в репозитории Moddable SDK на GitHub.



Рис. 1-5. M5Stack FIRE

Требования к программному обеспечению

Вам понадобятся следующие программы:

- Редактор кода
- Примеры файлов кода
- Moddable SDK
- Инструменты сборки для ESP32 и/или ESP8266.

Вы можете выбрать любой редактор кода, который вам больше нравится. Существует множество редакторов, совместимых с JavaScript, включая Visual Studio Code, Sublime Text 3 и Atom.

В следующих разделах объясняется, как загрузить примеры файлов кода и настроить Moddable SDK и инструменты сборки для вашего устройства.

Загрузка примера кода

Все примеры доступны по адресу

<https://github.com/Moddable-OpenSource/iot-product-dev-book>. Вы также можете загрузить пример кода из прилагаемой папки с кодами.

Примечание В этой книге перед командами, которые вы вводите в линейной команде, стоит символ `>`. Этот символ не является частью команды; он включен только для того, чтобы пояснить, где начинается каждая отдельная команда.

- В macOS/Linux используйте терминал:

```
> cd ~/Projects
> git clone https://github.com/Moddable-OpenSource/
  iot-product-dev-book
```

- В Windows используйте командную строку (изменив <username> на ваше имя пользователя):

```
> cd C:\Users\<username>\Projects
> git clone https://github.com/Moddable-OpenSource/
  iot-product-dev-book
```

Вам также необходимо установить переменную среды EXAMPLES, чтобы она указывала на вашу локальную копию репозитория примеров, как показано ниже:

- В macOS/Linux:

```
> export EXAMPLES=~/.Projects/iot-product-dev-book
```

- В Windows:

```
> set EXAMPLES=C:\Users\<username>\Projects\
  iot-product-dev-book
```

Настройка среды сборки

Перед сборкой и запуском примеров следуйте инструкциям в документе «Moddable SDK — Начало работы» в каталоге документации Moddable SDK. В этом документе представлены пошаговые инструкции по установке, настройке и сборке Moddable SDK для macOS, Linux и Windows, а также инструкции по установке инструментов, необходимых для работы с ESP32 и ESP8266.

Использование xsbug

Отладчик [xsbug](#) обеспечивает отладку кода JavaScript на уровне исходного кода, выполняемого на движке XS JavaScript. Он подключается к устройствам через USB и имеет графический пользовательский интерфейс (показан на рис. 1-6), облегчающий использование.

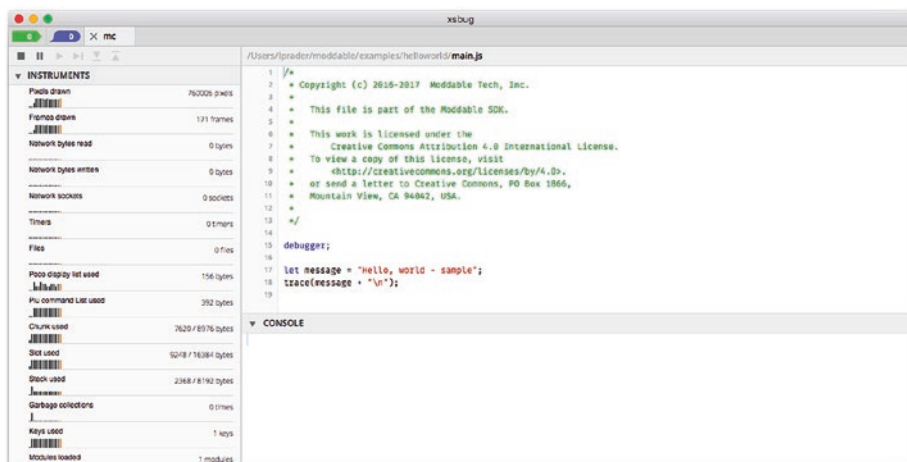


Рис. 1-6. Отладчик xdebug

Подобно другим отладчикам, xdebug поддерживает установку точек останова и просмотр исходного кода, стека вызовов и переменных. Он также предоставляет инструментарий в режиме реального времени для отслеживания использования памяти и профилирования потребления приложений и ресурсов.

Когда вы разрабатываете для микроконтроллера, система сборки автоматически открывает xdebug перед запуском вашего приложения на целевом устройстве.

При разработке для настольного симулятора вам необходимо открыть xdebug самостоятельно, либо дважды щелкнув значок приложения, либо открыв его из командной строки следующим образом:

- В macOS:


```
> open $MODDABLE/build/bin/mac/release/xsbug.app
```
- В Windows/Linux:


```
> xsbug
```


Важные особенности примеров в этой книге

В этой книге `xsbug` упоминается нечасто, потому что примеры уже отлажены. Однако `xsbug` — бесценный инструмент при создании собственных приложений. Наиболее важные функции `xsbug`, использованные в этой книге, следующие:

- Вкладки компьютеров. Каждая виртуальная машина XS, подключенная к `xsbug`, получает собственную вкладку в верхнем левом углу окна (обозначена пунктирной рамкой на рис. 1-7). Щелкнув вкладку, левая панель изменится на представление вкладки машины, где вы можете просматривать приборы, использовать кнопки управления и т. д.
- **Кнопки управления.** Эти графически помеченные кнопки (выделены пунктирной рамкой на рисунке) в верхней части вкладки машины управляют виртуальной машиной. Слева направо: «Kill - Ограничить», «Break - Останов», «Run - Пуск», «Step - Шаг», «In - Войти» и «Out - Выйти».
- **Консоль** — часто полезно иметь возможность просматривать диагностические сообщения во время выполнения приложения. Функция трассировки записывает сообщения в консоль отладки в правом нижнем углу файла `xsbug`.

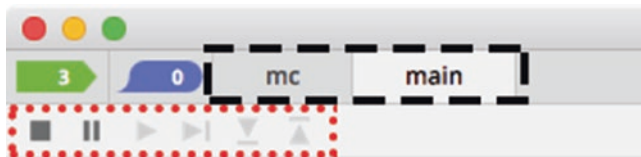


Рис. 1-7. Вкладки и кнопки управления машиной `xsbug`

См. документ `xsbug` в каталоге `document/xs Moddable SDK` для получения полной документации по всем функциям `xsbug`.

Запуск примеров

Установка хоста отдельно, а не установка хоста и примера вместе, значительно ускоряет разработку за счет минимизации количества программ, которые необходимо загрузить. Установка хоста на ваше устройство обычно занимает от 30 до 90 секунд. Как только это будет сделано, вы сможете установить большинство примеров всего за несколько секунд, потому что хост уже содержит прошивку устройства и модули JavaScript, необходимые для примеров.

Вы можете думать о хосте как о базовом приложении. Веб-браузер является хостом, когда вы запускаете JavaScript в веб-браузере; Node.js — это хост, когда вы запускаете JavaScript на веб-сервере.

Примеры в репозитории этой книги организованы по главам, каждая из которых содержит несколько примеров. Чтобы ускорить сборку и запуск примеров, каждая глава имеет свой собственный хост, который содержит программную среду, необходимую для запуска примеров для этой главы; хост — это набор модулей JavaScript, переменных конфигурации и других программ, доступного для использования вашим приложением. Поскольку пространство в микроконтроллерах очень ограничено, невозможно иметь один хост, содержащий все модули, используемые в примерах из этой книги.

Следующие разделы проведут вас через весь процесс установки хоста, а затем приведут пример, начиная с `helloworld`. Обратите внимание, что в контексте этой книги установка приложения приводит к его запуску на устройстве.

Установка хоста

Первым делом нужно прошить устройство для установки хоста. Исходный код хоста каждой главы доступен для чтения в каталоге хоста, если вам интересно. Чтобы использовать хост, все, что вам действительно нужно знать, это то, что он включает в себя все модули, необходимые для соответствующих примеров.

Вы будете использовать инструмент командной строки `mcconfig` для прошивки устройства.

mcconfig

Инструмент командной строки `mcconfig` создает и устанавливает приложения на микроконтроллеры или в симулятор. Здесь приведены команды для установки хоста этой главы на каждой поддерживаемой платформе.

На ESP32 используйте следующие команды:

- В macOS/Linux:

```
> cd $EXAMPLES/ch1-gettingstarted/host  
> mcconfig -d -m -p esp32
```
- В Windows:

```
> cd %EXAMPLES%\ch1-gettingstarted\host  
> mcconfig -d -m -p esp32
```

На ESP8266 используйте следующие команды:

- В macOS/Linux:

```
> cd $EXAMPLES/ch1-gettingstarted/host  
> mcconfig -d -m -p esp
```
- В Windows:

```
> cd %EXAMPLES%\ch1-gettingstarted\host  
> mcconfig -d -m -p esp
```

Подтверждение установки хоста

Как только хост установлен, он записывает сообщение, показанное на рис. 1-8, в консоль отладки.

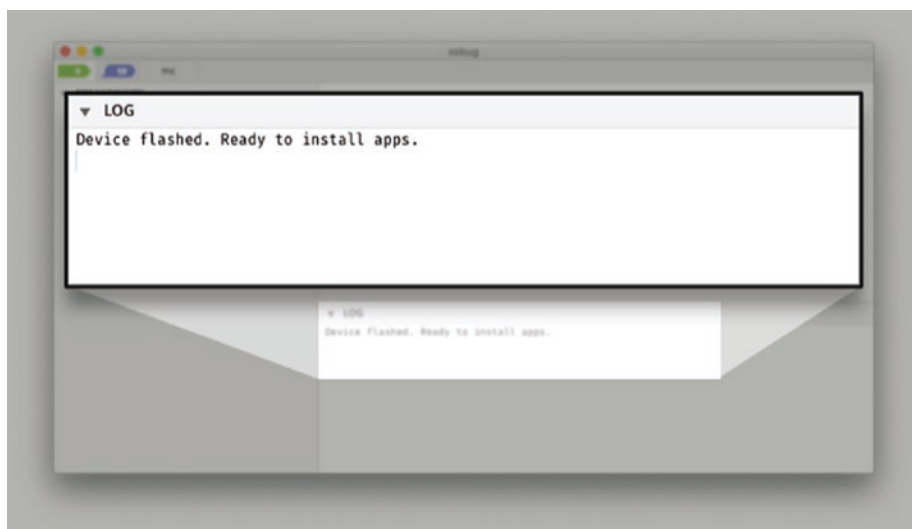


Рис. 1-8. Сообщение от хоста в xsbug

Установка helloworld

Пример helloworld состоит всего из трех строк кода

```
debugger;  
let message = "Hello, World";  
trace(message + "\n");
```

В этом примере используются две важные функции:

- Оператор отладчика, который останавливает выполнение и переходит в [xsbug](#).
- Функция трассировки, которая записывает сообщения в консоль отладки. Обратите внимание, что `trace` не добавляет автоматически символ новой строки (`\n`) в конец сообщения. Это позволяет использовать несколько операторов трассировки для создания выходных данных одной строки. Не забудьте включить символ новой строки в конце строки, чтобы текст

правильно отображался в xdebug. Вы используете `mcrun` для установки примеров.

mcrun

Инструмент командной строки `mcrun` создает и устанавливает дополнительные модули и ресурсы JavaScript, которые изменяют поведение или внешний вид модифицируемых приложений на микроконтроллерах или в симуляторе. И `mcconfig`, и `mcrun` создают сценарии и ресурсы. В отличие от `mcrun`, `mcconfig` также создает собственный код. В терминах JavaScript `mcconfig` создает хост.

После установки примера с помощью `mcrun` устройство перезагружается. Оно перезапустит хост, который, в свою очередь, запустит установленный вами пример.

Используйте следующие команды для установки примера `helloworld`. Убедитесь, что вы изменили `<platform>` на соответствующую платформу для вашего устройства, либо `esp32`, либо `esp8266`.

- В macOS/Linux:

```
> cd $EXAMPLES/ch1-gettingstarted/helloworld
> mcrun -d -m -p <platform>
```

- В Windows:

```
> cd %EXAMPLES%\ch1-gettingstarted\helloworld
> mcrun -d -m -p <platform>
```

Заканчивая

После установки приложения нажмите кнопку `Run`, чтобы увидеть сообщение `Hello, World`, записанное в консоль отладки, как показано на рисунке 1-9.

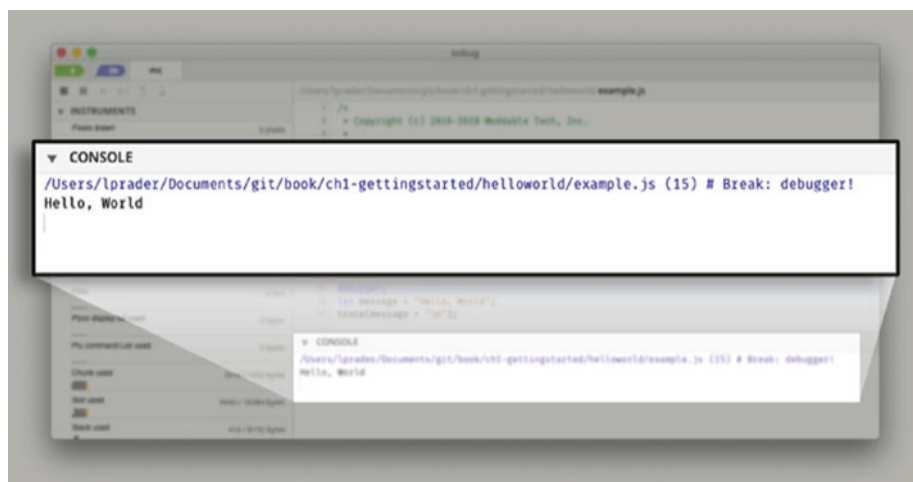


Рис. 1-9. Hello, World написан для консоли в xdebug

Если все пойдет хорошо, вы можете перейти к разделу «Заключение» этой главы, если вы работаете с голой платой NodeMCU. Если вы хотите добавить дисплей — что настоятельно рекомендуется — перейдите к разделу «Добавление дисплея».

Если у вас возникли проблемы, см. следующий раздел.

Исправление проблем

При установке приложения, вы можете столкнуться с препятствиями в виде ошибок или предупреждений; В этом разделе объясняются некоторые распространенные проблемы и способы их решения.

Устройство не подключено/не распознано

Сообщение об ошибке

```
error: cannot access /dev/cu.SLAB_USBtoUART
// ошибка: невозможно получить доступ к
/dev/cu.SLAB_USBtoUART
```

означает, что устройство не подключено к вашему компьютеру или компьютер не распознает устройство. Это может произойти по нескольким причинам:

- Ваше устройство не подключено к компьютеру. Убедитесь, что он подключен, когда вы запускаете команды сборки.
- У вас есть USB-кабель, предназначенный только для питания. Убедитесь, что вы используете USB-кабель с возможностью синхронизации данных.
- Компьютер не распознает ваше устройство. Чтобы устранить эту проблему, см. инструкции для вашей операционной системы.

macOS/Linux

Чтобы проверить, распознает ли ваш компьютер ваше устройство, отключите устройство от сети и введите следующую команду:

```
> ls /dev/cu*
```

Затем подключите устройство и повторите ту же команду. Если ничего нового не появляется, устройство не видно. Убедитесь, что у вас установлен соответствующий драйвер VCP.

Если он виден, теперь у вас есть имя устройства, и вам нужно отредактировать переменную среды UPLOAD_PORT. Введите следующую команду, заменив

```
> export UPLOAD_PORT=/dev/cu.SLAB_USBtoUART
```

Windows

Проверьте список USB-устройств в диспетчере устройств. Если ваше устройство отображается как неизвестное, убедитесь, что у вас установлен соответствующий драйвер VCP.

Если ваше устройство отображается на COM-порту, отличным от COM3, вам необходимо отредактировать переменную среды UPLOAD_PORT. Введите следующую команду, заменив COM3 соответствующим COM-портом устройства для вашей системы:

```
> set UPLOAD_PORT=COM3
```

Несовместимая скорость передачи данных

Следующее предупреждающее сообщение является нормальным и не вызывает беспокойства:

```
warning: serialport_set_baudrate: baud rate 921600 may not work
```

Однако иногда загрузка начинается, но не завершается. Вы можете сказать, что загрузка завершена, когда индикатор выполнения, отслеживаемый на консоли, достигает 100%. Например:

```
..... [ 16% ]
..... [ 33% ]
..... [ 49% ]
..... [ 66% ]
..... [ 82% ]
..... [ 99% ]
..... [ 100% ]
```

Есть несколько причин, по которым загрузка может завершиться на полпути:

- У вас неисправный кабель USB.
- У вас USB-кабель, который не поддерживает более высокие скорости передачи данных
- Вы используете плату, для которой требуется более низкая скорость передачи данных, чем скорость передачи данных по умолчанию, используемая Moddable SDK.

Чтобы решить две последние проблемы, вы можете изменить скорость передачи данных на более медленную следующим образом:

1. Если вы работаете с ESP32, откройте `moddable/tools/mcconfig/make.esp32.mk`; если ESP8266, откройте `moddable/tools/mcconfig/make.esp.mk`.
2. Найдите эту строку, которая устанавливает скорость загрузки 921 600:

```
UPLOAD_SPEED ?= 921600
```

3. установите скорость на меньшее число. Например:
4. `UPLOAD_SPEED ?= 115200`

Устройство не в режиме загрузчика

Эта проблема не редкость, если вы используете определенные платы на основе ESP32. Сообщения о состоянии ненадолго перестают отслеживаться, когда вы пытаетесь прошить устройство, и через несколько секунд вы получаете следующее сообщение об ошибке:

```
A fatal error occurred: Failed to connect to ESP32: Timed out waiting for packet header // Произошла фатальная ошибка: не удалось подключиться к ESP32: время ожидания заголовка пакета истекло.
```

Если устройство не находится в режиме загрузчика, вы не сможете прошить устройство. Если вы используете модуль NodeMCU, выполняйте следующие шаги каждый раз при прошивке: Unplug the device.

1. Удерживайте нажатой кнопку BOOT (обведена кружком на рис. 1-10).
2. Подключите устройство к компьютеру.
3. Введите команду `cconfig`.
4. Подождите несколько секунд и отпустите кнопку BOOT.



Рис. 1-10. Кнопка BOOT на ESP32

Добавление дисплея

Хотя для большинства примеров в этой книге не требуется дисплей, добавление дисплея к ESP32 или ESP8266 значительно улучшает взаимодействие с пользователем. Это позволяет вам делать следующее:

- Показать больше информации, чем несколько мигающих LED
- Создавать современные пользовательские интерфейсы
- Добавить функциональность

Примеры в этой книге предназначены для дисплея с разрешением 240 x 320 или 320 x 240 (например, QVGA). Эти дисплеи обычно имеют размер от 2,2 до 3,5 дюймов и были распространены в первых смартфонах. К этим микроконтроллерам можно подключать дисплеи других размеров, но этот размер хорошо соответствует возможностям этих микроконтроллеров.

В следующих разделах показано, как подключить сенсорный дисплей ILI9341 QVGA к ESP32 или ESP8266. Если вы используете плату разработки, такую как Moddable One или Moddable Two, вы можете перейти к разделу «Установка helloworld-gui».

Подключение дисплея к ESP32

Таб. 1-1 и рис. 1-11 показывают, как подключить дисплей к ESP32.

Таб. 1-1. Соединения для подключения дисплея к ESP32

Дисплей ILI9341	ESP32
SDO/MISO	GPIO12
LED	3.3V
SCK	GPIO14
SDI/MOSI	GPIO13
CS	GPIO15
DC	GPIO2
RESET	3.3V
GND	GND
VCC	3.3V
T_DO	GPIO12
T_DIn	GPIO13
T_CLK	GPIO14
T_IRQ	GPIO23
T_CS	GPIO18

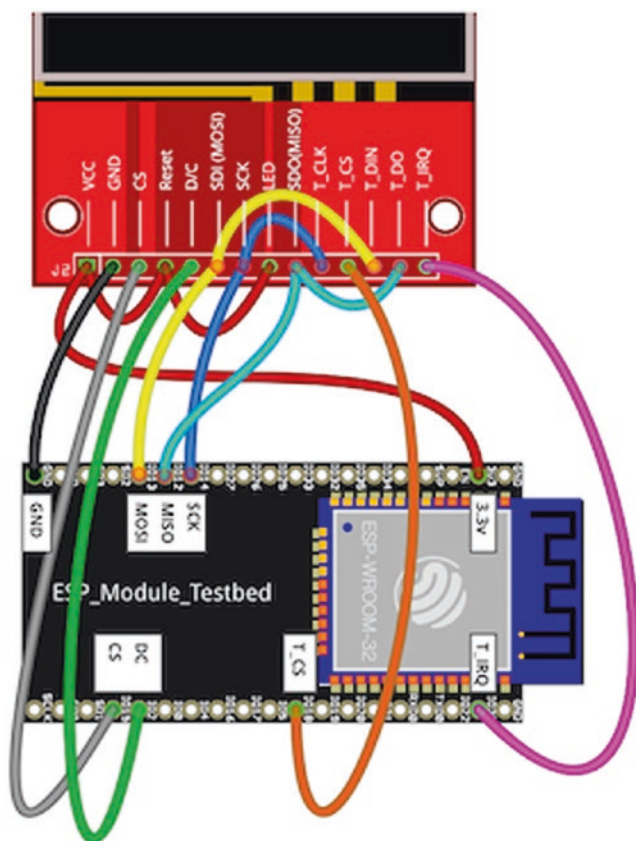


Рис. 1-11. Схема подключения дисплея к ESP32.

Подключение дисплея к ESP8266

Таб. 1-2 и рис. 1-12 показывают, как подключить дисплей к ESP8266.

Table 1-2. Соединения для подключения дисплеяк ESP8266

ILI9341 Display	ESP8266
SDO/MISO	GPIO12
LED	3.3V
SCK	GPIO14
SDI/MOSI	GPIO13
CS	GPIO15
DC	GPIO2
RESET	3.3V
GND	GND
VCC	3.3V
T_DO	GPIO12
T_DIn	GPIO13
T_CLK	GPIO14
T_IRQ	GPIO16
T_CS	GPIO0

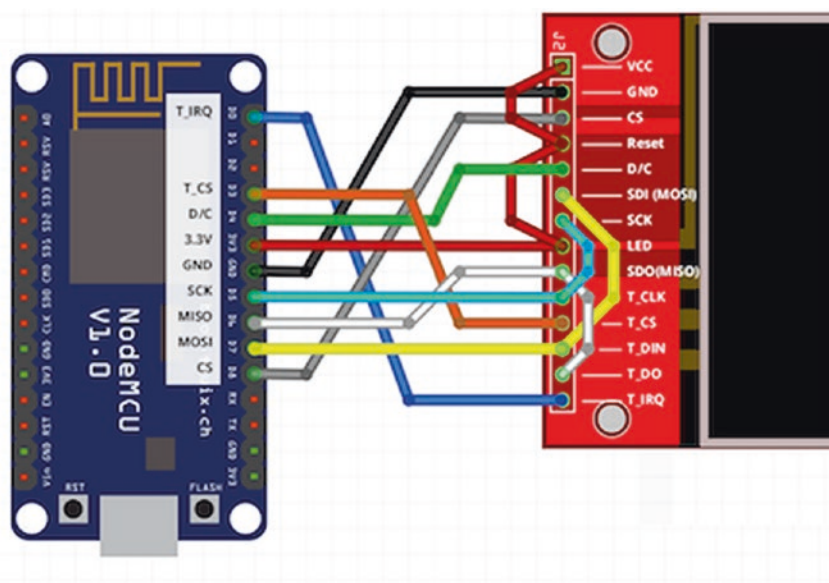


Рис. 1-12. Схема подключения дисплея к ESP8266

Установка helloworld-gui

Пример helloworld-gui — это версия helloworld, которая отображает текст на экране. Если вы самостоятельно подключили дисплей к устройству, перепрошивка устройства с помощью приложения helloworld-gui — хороший способ проверить правильность подключения.

Используемые команды очень похожи на те, которые используются для установки helloworld. Отличие только в идентификаторе платформы. Идентификатор платформы сообщает системе сборки о необходимости включения соответствующих драйверов дисплея и сенсорного экрана. Если вы используете Moddable One, идентификатор платформы — `esp/moddable_one`; для Moddable Two это `esp32/moddable_two`. Если вы добавили дисплей в соответствии с инструкциями в предыдущих разделах, идентификатором платформы будет либо `esp32/moddable_zero`, либо `esp/moddable_zero`.

Используйте следующие команды для установки примера helloworld-gui. Убедитесь, что вы изменили `<platform>` на соответствующую платформу для вашего устройства.

- On macOS/Linux:

```
> cd $EXAMPLES/ch1-gettingstarted/helloworld-gui  
> mcconfig -d -m -p <platform>
```

- On Windows:

```
> cd %EXAMPLES%\ch1-gettingstarted\helloworld-gui  
> mcconfig -d -m -p <platform>
```

Если вы укажете соответствующую платформу и подключите без ошибок, вы увидите экран, показанный на рис. 1-13.

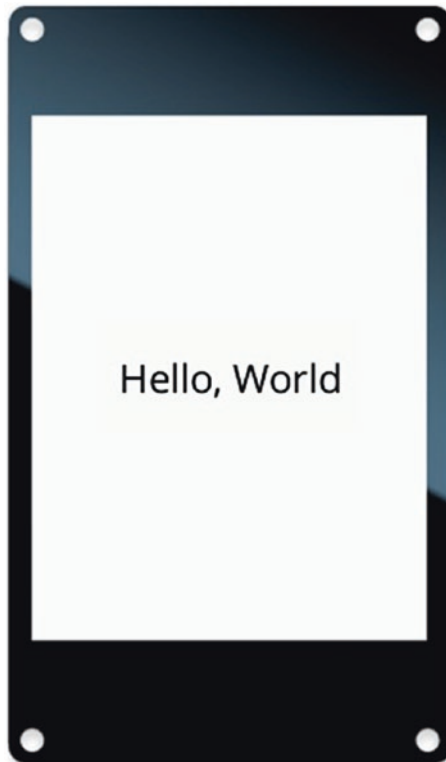


Рис. 1-13. Графическое приложение *helloworld*

Заключение

Теперь, когда ваша среда разработки настроена и вы знакомы с процессом установки примеров из этой главы на свое устройство, вы должны попробовать еще несколько примеров!

К этому моменту у вас есть все материалы и навыки, необходимые для изучения глав со 2 по 10. Эти главы не зависят друг от друга, поэтому вы можете читать их в любом порядке. Когда вы начнете работать с примерами в главе, обязательно установите хост этой главы, иначе вы столкнетесь с ошибками при запуске примеров. Как только вы освоитесь с API-интерфейсами Moddable SDK, вы можете перейти к главе 11, в которой рассматриваются более сложные темы.

ГЛАВА 2

https://t.me/it_boooks

JavaScript для программистов С и С++

Эта глава представляет собой динамичное практическое введение в JavaScript для разработчиков, уже знакомых с С или С++. Предполагается, что вы уже умеете программировать и, возможно, имеете некоторый опыт разработки встроенных систем. Представленный здесь язык JavaScript — это точно такой же язык, который используется в Интернете. Но поскольку здесь основное внимание уделяется встроенным системам, а не веб-браузерам, в этой главе рассматриваются некоторые аспекты JavaScript, которые редко используются разработчиками, работающими в Интернете. Например, учтите, что почти невозможно написать встроенное программное обеспечение без манипулирования двоичными данными (такими как массив байтов); JavaScript поддерживает двоичные данные со встроенными классами `typedarray`, но большинство веб-разработчиков никогда не используют эту функцию, потому что она не нужна при создании веб-страницы. Так что, даже если вы знакомы с JavaScript в Интернете, вы можете прочитать эту главу, чтобы ознакомиться с функциями языка, более распространенными во встроенных системах, чем в Интернете.

Программисты на С и С++ имеют большое преимущество при начале работы с JavaScript, потому что язык очень похож на С. И это не случайно: язык программирования JavaScript был разработан таким образом, чтобы быть похожим на язык программирования Java; Java был создан как

эволюция C++; а C++ был создан, чтобы перенести объектно-ориентированное программирование на C. Многие сходства помогут вам быстро читать и писать код JavaScript. Тем не менее, языки также во многом различны. В этой главе сходства используются в качестве основы, чтобы познакомить вас с некоторыми различиями.

JavaScript уже более 20 лет, и он постоянно развивается. В этой главе представлен современный JavaScript, в том числе функции версии JavaScript 2019 года, а также некоторые (например, закрытые поля), которые планируется включить в будущую версию. Здесь описаны только функции JavaScript, которые являются частью стандартного языка. Из-за долгой истории JavaScript некоторые функции больше не рекомендуются для использования; в этой главе описаны некоторые из них. В частности, JavaScript 5th Edition, стандартизированный в 2012 году, представил строгий режим, который устраняет несколько запутанных и неэффективных функций. Эти исходные варианты поведения остаются доступными в небрежном режиме, который в основном используется для обратной совместимости с веб-сайтами, но в этой книге используется исключительно строгий режим.

Фундаментальный синтаксис

В этом разделе представлены основы, такие как использование JavaScript для вызова функций, объявления переменных и управления потоком выполнения с помощью операторов `if`, `switch`, `for` и `while`. Все они очень похожи в C и JavaScript, но по ходу дела вы узнаете о некоторых важных различиях.

“Hello, world”

Традиционной отправной точкой для изучения C является программа `hello, world` из книги Кернигана и Ритчи «Язык программирования C». В JavaScript это всего одна строка:

```
trace("hello, world\n");
```

Здесь функция `C printf` заменена функцией трассировки из Moddable SDK. (Разработчики, работающие с JavaScript в Интернете, используют `console.log` вместо `trace`.) Как и в C, аргумент функции передается в круглых скобках, а оператор заканчивается точкой с запятой. Строковый литерал, переданный функции, также идентичен — строка, заключенная в двойные кавычки, — и использует знакомую нотацию обратной косой черты (`\`), как в C, для экранирования специальных символов, таких как здесь символ новой строки.

Точки с запятой

Одно существенное различие между C и C++ заключается в том, что точка с запятой в конце инструкции не является обязательной в JavaScript благодаря функции автоматической вставки точки с запятой (ASI). Следующий код разрешен в JavaScript, но не работает в C:

```
trace("hello, ")
trace("world")
trace("\n")
```

Хотя это удобно, так как экономит нажатие клавиши и автоматически исправляет распространенную ошибку пропуска точки с запятой, в некоторых неясных случаях это создает двусмысленность, что может привести к ошибкам. Поэтому операторы всегда следует заканчивать точкой с запятой, а не полагаться на ASI. Линтеры JavaScript, такие как ESLint, включают проверку отсутствия точек с запятой.

Объявление переменных и констант

Переменные в JavaScript объявляются с помощью оператора `let`:

```
let a = 12;
let b = "hello";
let c = false;
```

В отличие от C, объявление переменной не содержит никакой информации о типе (например, `int`, `bool` или `char *`). Это потому, что любая переменная может содержать любой тип. Эта динамическая типизация, которая будет объяснена далее в этой главе, является одной из возможностей JavaScript, к которой программистам на C требуется некоторое время, чтобы привыкнуть.

Имена переменных в JavaScript обычно следуют соглашениям C: они чувствительны к регистру, поэтому `i` и `I` — разные имена, а длина имен переменных не ограничена. Имена переменных JavaScript также могут включать символы Юникода, как в следующих примерах:

```
let garçon = "boy";
let 東京都 = "Tokyo";
let $ = "dollar";
let under_score = "_";
```

Вы объявляете постоянные значения с помощью `const`:

```
const PRODUCT_NAME = "Light Sensor";
const LUMEN_REGISTER = 2;
const USE_OPTIMIZATIONS = true;
```

Любая попытка присвоить значение константе приводит к ошибке, однако, в отличие от C, эта ошибка генерируется во время выполнения, а не во время сборки.

Как показано в листинге 2-1, объявления, сделанные с помощью `let` и `const`, подчиняются тем же правилам области видимости, что и объявления в C.

Листинг 2-1.

```
let x = 1;
const y = 2;
let z = 3;
if (true) {
    const x = 2;
```

```

let y = 1;
trace(x);    // output: 2
trace(y);    // output: 1
trace(z);    // output: 3
y = 4;
z += y;
}
trace(x);    // output: 1
trace(y);    // output: 2
trace(z);    // output: 7

```

JavaScript также позволяет вам использовать `var` для объявления переменных, и это все еще распространено, поскольку `let` является относительно новым дополнением. Однако в этой книге рекомендуется использовать исключительно `let`, потому что `var` не подчиняется тем же правилам области действия, что и объявления в C.

Оператор if

Оператор `if` в JavaScript имеет ту же структуру, что и в C, как показано в листинге 2-2.

Листинг 2-2.

```

if (x) {
    trace("x is true\n");
}
else {
    trace("x is false\n");
}

```

Как и в C, когда блок `if` или `else` представляет собой один оператор, вы можете опустить фигурные скобки, ограничивающие блок:

```
if (!x)
    trace("x is false\n");
else
    trace("x is true\n");
```

Условие в операторе `if` в листинге 2-2 — это просто `x`. В C это означает, что если `x` равен 0, условие ложно; в противном случае это правда. В JavaScript это сложнее, потому что переменная `x` может быть любого типа, а не только числа (или указателя, но указателей в JavaScript не существует). JavaScript определил следующие правила для оценки того, является ли заданное значение истинным или ложным:

- Для логического значения это определение простое: значение либо истинно, либо ложно.
- Для числа JavaScript следует правилу C, рассматривая значение 0 как ложное, а все остальные значения как истинные.
- Пустая строка (строка длиной 0) оценивается как `false`, а все непустые строки оцениваются как `true`.

В JavaScript значение, которое оценивается как истинное в условии, называется «truthy», а значение, которое оценивается как ложное, называется «falsy».

Компактная форма оператора `if`, условный (тернарный) оператор, доступна в JavaScript и имеет ту же структуру, что и в C:

```
let x = y ? 2 : 3;
```

Оператор `switch`

Как показано в листинге 2-3, оператор `switch` в JavaScript выглядит почти так же, как и в C.

Листинг 2-3.

```
switch (x) {  
    case 0:  
        trace("zero\n");  
        break;  
    case 1:  
        trace("one\n");  
        break;  
    default:  
        trace("unexpected!\n");  
        break;  
}
```

Однако есть одно важное отличие: значение, следующее за ключевым словом `case`, не ограничено целочисленными значениями. Например, вы можете использовать число с плавающей запятой (см. листинг 2-4).

Листинг 2-4.

```
switch (x) {  
    case 0.25:  
        trace("one quarter\n");  
        break;  
    case 0.5:  
        trace("one half\n");  
        break;  
}
```

Вы также можете использовать строки (листинг 2-5).

Листинг 2-5.

```
switch (x) {  
    case "zero":  
    case "Zero":  
        trace("0\n");  
        break;  
    case "one":  
    case "One":  
        trace("1\n");  
        break;  
    default:  
        trace("unexpected!\n");  
        break;  
}
```

Кроме того, JavaScript позволяет смешивать типы значений в операторах case, хотя это редко бывает необходимо.

Циклы

В JavaScript есть циклы for и while, которые выглядят аналогично своим аналогам в языке C (см. листинг 2.6).

Листинг 2-6.

```
for (i = 0; i < 10; i++)  
    trace(i);  
  
let j = 12;  
while (j--)  
    trace(j);
```


Циклы JavaScript поддерживают как `continue`, так и `break` (листинг 2-7).

Листинг 2-7.

```
for (i = 0; i < 10; i++) {  
    if (i & 1)  
        continue;    // Пропустить нечетные числа  
    trace(i);  
}  
  
let j = 0;  
do {  
    let jSquared = j * j;  
    if (jSquared > 100)  
        break;  
    trace(jSquared);  
    j++;  
} while (true);
```

Типы

В JavaScript есть всего несколько встроенных типов, из которых создаются все остальные типы. Многие из этих типов знакомы программистам на C и C++, например, `Boolean`, `Number` и `String`, хотя в их версиях JavaScript есть различия, которые вам необходимо понимать. Другие типы, такие как `undefined`, не имеют эквивалента в C или C++.

Обратите внимание, что в этом разделе представлены не все типы. Например, в нем опущены `RegExp`, `BigInt` и `Symbol`, потому что они обычно не используются при разработке на JavaScript для встраиваемых систем; однако они доступны, если они потребуются вашему проекту.

undefined

В С и С++ операция может иметь результат, не определенный языком. Например, если вы забудете инициализировать **x** в следующем коде, значение **y** неизвестно:

```
int x;  
int y = x + 1;      // ??
```

Кроме того, результат функции неизвестен, если вы забудете включить оператор возврата:

```
int add(int a, int b) {  
    int result = a + b;  
}  
int z = add(1, 2);  // ??
```

Ваш компилятор С или С++ обычно обнаруживает такие ошибки и выдает предупреждения, чтобы вы могли исправить проблему. Тем не менее, есть много способов сделать ошибки в С и С++, которые приводят к непредсказуемым результатам кода.

В JavaScript никогда не бывает ситуаций, в которых результат был бы непредсказуем. Одна часть того, как это достигается, заключается в том, что специальное значение не определено, что указывает на то, что значение не было присвоено. В С 0 иногда используется как недопустимое значение для аналогичной цели, но это неоднозначно в ситуациях, когда 0 является допустимым значением.

Когда вы определяете новую локальную переменную, она имеет значение `undefined`, пока вы не присвоите ей другое значение. Если ваша функция завершается без оператора `return`, ее возвращаемое значение не определено. В этой главе вы увидите и другие варианты использования `undefined`.

Строго говоря, JavaScript имеет неопределенный тип, который всегда имеет значение `undefined`.

Логические значения

Булевы значения в JavaScript являются истинными и ложными. Это не то же самое, что 1 и 0 в C; это разные значения, определенные языком.

```
let x = 42;  
let y = x == 42;    // истина  
let z = x == "dog"; // ложь
```

Числа

Каждое числовое значение в JavaScript определяется как значение двойной точности (64-разрядное) с плавающей запятой IEEE 754. Прежде чем вы ахнете в ужасе от влияния этого на производительность микроконтроллера, знайте, что движок XS JavaScript, используемый в Moddable SDK, внутренне сохраняет числа как целые числа и выполняет над ними целочисленные математические операции, когда это возможно. Это гарантирует эффективность реализации на микроконтроллерах без FPU при сохранении полной совместимости со стандартным JavaScript.

```
let x = 1;  
let y = -2.3;  
let z = 5E2;    // 500
```

Есть некоторые преимущества в том, чтобы каждое число было определено как 64-битное число с плавающей запятой. Во-первых, переполнение целых чисел гораздо менее вероятно. Если результат целочисленной операции превышает 32-битное целое число, оно автоматически преобразуется в значение с плавающей запятой. 64-битное значение с плавающей запятой может хранить целые числа до 53 бит, прежде чем потеряет точность. Если вам случится выполнить математическую операцию, которая даст дробный результат, например, деление нечетного целого числа на 2, JavaScript вернет точный дробный результат в виде значения с плавающей запятой.

Бесконечность и NaN

В JavaScript есть специальные значения для чисел:

- Деление на 0 не приводит к ошибке, а вместо этого возвращает бесконечность.
- Попытка выполнить бессмысленную операцию возвращает NaN, что означает «not a number - не число». Например, `5 / "строка"` и `5 + undefined` возвращают NaN, потому что нет смысла делить целое число на строковое значение или прибавлять к целому integer.

Базы

В JavaScript есть специальные обозначения для шестнадцатеричных и двоичных констант:

- Префикс `0x` в числе означает, что оно записано в шестнадцатеричной системе счисления, как и в C.
- Префикс `0b` в числе означает, что оно представлено в двоичной записи, как это поддерживается в C++14.

Эти префиксы полезны при работе с двоичными данными, как в следующих примерах:

```
let hexMask = 0x0F;
let bitMask = 0b00001111;
```

В отличие от C, JavaScript не поддерживает восьмеричные числа с ведущим 0, как в `0557`; если вы попытаетесь использовать его, он выдаст ошибку при построении. Восьмеричные числовые литералы поддерживаются в форме `0o557`.

Числовые разделители

JavaScript позволяет использовать символ подчеркивания (`_`) в качестве числового разделителя для разделения цифр в числе. Разделитель не изменяет значение числа, но может облегчить его чтение. C++14 также имеет числовой разделитель, но вместо него используется одинарная кавычка (`'`).

```
let mask = 0b0101101011110000;
let maskWithSeparators = 0b0101_1010_1111_0000;
```

Побитовые операторы

JavaScript предоставляет побитовые операторы для чисел, в том числе следующие:

- `~` – побитовое NOT
- `&` – побитовое AND
- `|` – побитовое OR
- `^` – побитовое XOR

Он также предоставляет эти побитовые операторы для сдвига битов:

- `>>` – сдвиг вправо
- `>>>` – беззнаковый сдвиг вправо
- `<<` – сдвиг влево

Беззнакового сдвига не остается, потому что сдвиг влево на ненулевое значение всегда отбрасывает бит знака. При выполнении любой побитовой операции JavaScript всегда сначала преобразует значение в 32-битное целое число; любой дробный компонент или дополнительные биты отбрасываются.

Математический объект

Объект `Math` предоставляет многие функции, которые программисты на C используют из заголовочного файла `math.h`. В дополнение к обычным константам, таким как `Math.PI`, `Math.SQRT2` и `Math.E`, он включает в себя общие функции, такие как те, что показаны в листинге 2-8.

Листинг 2-8.

```
let x = Math.min(1, 2, 3); // minimum = 1
let y = Math.max(2, 3);   // maximum = 3
let r = Math.random();    // случайное число от 0 до 1
```

```
let z = Math.abs(-3.2);    // абсолютное значение = 3.2
let a = Math.sqrt(100);   // квадратный корень = 10
let b = Math.round(3.9);  // округленное значение = 4
let c = Math.trunc(3.9);   // усеченное значение = 3
let z = Math.cos(Math.PI); // косинус числа пи = -1
```

Обратитесь к справочнику по JavaScript для получения полного списка значений констант и функций, предоставляемых объектом Math.

Преобразование чисел в строки

В языке C обычным способом преобразования числа в строку является использование `sprintf` для вывода числа в строковый буфер. В JavaScript вы конвертируете число в строку, вызывая метод `toString` числа (да, в JavaScript даже число является объектом!):

```
let a = 1;

let b = a.toString();    // "1"
```

База по умолчанию для `toString` равна 10; для преобразования в недесятичное значение, такое как шестнадцатеричное или двоичное, передайте основание в качестве аргумента `toString`:

```
let a = 240;
let b = a.toString(16);    // "f0"
let c = a.toString(2);     // "11110000"
```

Чтобы преобразовать в нотацию с плавающей запятой, используйте `toFixed` вместо `toString` и укажите количество цифр после запятой:

```
let a = 24.328;
let b = a.toFixed(1);      // "24.3"
let c = a.toFixed(2);      // "24.33"
let d = a.toFixed(4);      // "24.3280"
```

Функции `toExponential` и `toPrecision` предоставляют дополнительные параметры форматирования для преобразования чисел в строки.

Преобразование строк в числа

В C обычным способом преобразования строки в число является использование `sscanf`. В JavaScript используйте либо `parseInt`, либо `parseFloat` в зависимости от того, хотите ли вы получить результат в виде целого числа или значения с плавающей запятой:

```
let a = parseInt("12.3");      // 12
let b = parseFloat("12.30");   // 12.3
```

База по умолчанию для `parseInt` равна 10, за исключением случаев, когда строка начинается с 0x, и в этом случае значение по умолчанию равно 16. Функция `parseInt` принимает необязательный второй аргумент, указывающий базу. В следующем примере анализируется шестнадцатеричное значение:

```
let a = parseInt("F0", 16);    // 240
```

Вы также можете получить доступ к функциям `parseInt` и `parseFloat` как `Number.parseInt` и `Number.parseFloat`; однако это встречается реже.

Строки

В C строки не являются отдельным типом, а просто массивом 8-битных символов. Поскольку строки очень распространены, стандартная библиотека C предоставляет множество функций для работы с ними. Тем не менее, работа со строками в C не проста и может легко привести к ошибкам безопасности, таким как переполнение буфера. C++ решает некоторые проблемы, хотя работа со строками по-прежнему не проста и небезопасна. JavaScript, напротив, имеет встроенный тип `String`, который был разработан, чтобы программистам было легко и безопасно использовать его; это отражает происхождение JavaScript как языка Интернета, где манипуляции со строками распространены при создании веб-страниц.

В JavaScript строки во многом отличаются от обычных строк C. Строка представляет собой последовательность 16-битных символов Юникода (UTF-16), а не массив 8-битных символов. Использование Unicode для представления строк гарантирует, что все приложения могут надежно работать со строковыми значениями на любом языке.

Хотя символы концептуально представляют собой 16-битный Unicode, движок JavaScript может хранить их внутри в любом представлении. Движок XS хранит строки в кодировке UTF-8, поэтому нет дополнительных затрат памяти на символы из обычного 7-битного набора символов ASCII.

Доступ к отдельным символам

Строки JavaScript не являются массивами; однако они поддерживают синтаксис массива `C` для доступа к отдельным символам. Однако, в отличие от `C`, результатом является не Unicode (числовое) значение символа, а односимвольная строка, содержащая символ с этим индексом.

```
let a = "garçon";  
let b = a[3];    // "ç"  
let c = a[4];    // "o"
```

В `C` обращение к недопустимому индексу — например, за концом строки — возвращает неопределенное значение. Для объявленного, как показано в предыдущем коде, `a[100]` будет обращаться ко всему, что окажется в памяти через 100 байтов после начала строки. Доступ может даже вызвать ошибку памяти из-за доступа к неотображенной памяти. В JavaScript попытка прочитать символ за пределами допустимого диапазона строки возвращает значение `undefined`.

Чтобы получить значение Unicode символа по заданному индексу, используйте функцию `charCodeAt`:

```
let a = "garçon";  
let b = a.charCodeAt(3);    // 231  
let c = a.charCodeAt(4);    // 111  
let d = a.charCodeAt(100);   // NaN
```


Изменение строк

C позволяет вам как читать, так и записывать символы в строке. Строки JavaScript доступны только для чтения, их также называют неизменяемыми; вы не можете изменить строку «на месте». Например, присваивание `a[0]` в следующем коде ничего не делает в JavaScript:

```
let a = "a string";  
a[0] = "A";
```

This restriction can be a little difficult to get used to coming from C, but it becomes familiar with some experience using the many methods provided to operate on a string.

Определение длины строк

Чтобы определить длину строки в C, используется функцию `strlen`, которая возвращает количество байтов в строке. Она определяет длину путем сканирования байта со значением 0, потому что строки в C определены так, чтобы заканчиваться 0 байтом. В JavaScript строки представляют собой последовательность символов Unicode без завершающего нулевого символа; количество символов в последовательности известно движку JavaScript и доступно через свойство `length`.

```
let a = "hello";  
let b = a.length; // 5
```

Одна проблема со `strlen` заключается в том, что количество байтов в строке равно длине строки только тогда, когда символы являются 8-битными символами ASCII. Для символов Unicode `strlen` не предоставляет количество символов. Конечно, есть и другие функции, которые это делают, но программисты на C часто ошибаются, когда неправильно используют `strlen` со строками для получения количества символов, что приводит к ошибкам. Свойство длины JavaScript позволяет избежать этой проблемы, поскольку оно всегда возвращает количество символов.

Пример в листинге 2-9 использует свойство `length` для подсчета пробелов в строке.

Листинг 2-9.

```
let a = "zéro un deux";
let spaces = 0;
for (let i = 0; i < a.length; i++) {
    if (a[i] == " ")
        spaces += 1;
}
trace(spaces);
```

Встраивание кавычек и управляющих символов

Строковые литералы в этой главе до этого момента использовали двойные кавычки (") для определения начала и конца строки. Строки, заключенные в двойные кавычки, могут содержать одинарные кавычки (').

```
let a = "Let's eat!";
```

Как и в C, такие строки не могут содержать двойные кавычки. В отличие от C, вы можете разделить строку одинарными кавычками вместо двойных кавычек, что удобно для строк, содержащих двойные кавычки.

```
let a = '"This is a test," she said.';
```

Строки, разделенные одинарными или двойными кавычками, должны полностью содержаться в одной строке. Вы можете включать разрывы строк в строку, используя `\n` для указания символа новой строки; обратная косая черта (`\`) позволяет экранировать символы, как в C.

```
let a = 'line 1\nline 2\nline 3\n';
```

Другой способ обозначить строку в JavaScript — использовать символ обратной кавычки (```). Строки, определенные таким образом, называются литералами шаблонов и имеют несколько полезных свойств, в том числе то, что они могут занимать несколько строк (потенциально делая ваши строки более удобочитаемыми; сравните листинг 2.10 с предыдущим примером).

Листинг 2-10.

```
let a =
`line 1
line 2
line 3
`;
```

Замена строки

Литералы шаблонов предоставляют механизм подстановки, полезный для составления строки из нескольких частей. Функциональность, которую это предоставляет, очень похожа на использование `printf` в C со строкой форматирования. Однако в то время как C отделяет строку форматирования от форматируемых значений, JavaScript объединяет их. Поначалу это может показаться непривычным, но размещение форматируемых значений непосредственно в строке менее подвержено ошибкам.

```
let a = "one";
let b = "two";
let c = `${a}, ${b}, three`;    // "one, two, three"
```

Внутри литерала шаблона символы между `${` и `}` оцениваются как выражение JavaScript, что позволяет выполнять вычисления и применять форматирование к результату:

```
let a = `2 + 2 = ${2 + 2}`;    // "2 + 2 = 4"
let b = `Pi to three decimals is ${Math.PI.toFixed(3)}`;
    // "Pi to three decimals is 3.142"
```

Специальная функция, называемая тегами, позволяет функции изменять поведение литералов шаблонов по умолчанию. Например (как будет показано в главе 4), вы можете использовать эту функцию для преобразования строкового представления UUID в двоичные данные. Детали того, как работают литералы шаблонов с тегами, выходят за рамки этой главы, но использовать их легко: просто поместите тег перед литералом шаблона.

```
let a = uuid`1805`;
let b = uuid`9CF53570-DDD9-47F3-BA63-09ACEFC60415`;
```

Добавление строк

Вы можете комбинировать строки в JavaScript, используя оператор сложения (+):

```
let a = "one";
let b = "two";
let c = a + ", " + b + ", three";    // "one, two, three"
```

JavaScript позволяет добавлять строки к нестроковым значениям, таким как числа. Его правила того, как это работает, обычно дают ожидаемый результат, но не всегда:

```
let a = "2 + 2 = " + 4;              // "2 + 2 = 4"
let b = 2 + 2 + " = 2 + 2";          // "4 = 2 + 2"
let c = "2 + 2 = " + 2 + 2;          // "2 + 2 = 22"
```

Поскольку запомнить все правила преобразования типов при добавлении строк может быть сложно, вместо этого рекомендуется использовать литералы шаблонов, которые более предсказуемы и часто более читабельны.

Преобразование регистра строк

Преобразование строк в верхний или нижний регистр в C является сложной задачей, особенно когда вы работаете с полным набором символов Unicode. JavaScript имеет встроенные функции для выполнения этих преобразований.

```
let a = "Garçon";
let b = a.toUpperCase();    // "GARÇON"
let c = a.toLowerCase();    // "garçon"
```

Обратите внимание, что функции `toUpperCase` и `toLowerCase` не изменяют исходную строку, сохраненную в переменной `a` в предыдущем примере, а возвращают новую строку с измененным значением. Все функции JavaScript, работающие со строками, ведут себя таким образом, потому что все строки неизменяемы.

Извлечение частей строк

Чтобы извлечь часть строки в другую строку, используйте функцию `slice`. Его аргументы — это начальный и конечный индексы, где конечный индекс — это индекс, перед которым завершается извлечение. Если конечный индекс опущен, используется длина строки.

```
let a = "hello, world!";

let b = a.slice(0, 5);        // "hello"
let c = a.slice(7, 12);      // "world"
let d = a.slice(7);          // "world!"
```

В JavaScript также есть функция `substr`, которая предоставляет аналогичные функции для среза, но с немного другими аргументами. Тем не менее, `slice` предпочтительнее, чем `substr`, который поддерживается в основном для устаревшего кода в Интернете.

Повторяющиеся строки

Чтобы создать строку, которая повторяет определенное значение несколько раз, используйте функцию повтора:

```
let a = "-";

let b = a.repeat(3);        // "---"
let c = ".-";
let d = c.repeat(2);        // "-.-"
```

Обрезка строк

При синтаксическом анализе строк часто требуется удалить пробелы (символ пробела, табуляция, возврат каретки, перевод строки и т. д.) в начале или в конце. Функции обрезки удаляют пустое пространство за один шаг:

```
let a = " JS ";
let b = a.trim();           // "JS"
let c = a.trimStart();     // "JS "
let d = a.trimEnd();       // " JS"
```

Функции обрезки можно полностью реализовать на JavaScript (как и большинство строковых функций), но встраивание их в язык означает, что их реализация будет значительно быстрее, а их поведение будет одинаковым во всех приложениях.

Поиск строк

Функция `strstr` в C находит одну строку внутри другой. Функция `indexOf` в JavaScript похожа на функцию `strstr`. Как показано в листинге 2-11, первый аргумент `indexOf` — это искомая подстрока, необязательный второй аргумент — это индекс символа, с которого нужно начать поиск, а результатом функции является индекс, по которому найдена подстрока, или — 1, если не найдено.

Листинг 2-11.

```
let string = "the cat and the dog";
let a = string.indexOf("cat");      // 4
let b = string.indexOf("frog");     // -1
let c = string.indexOf("the");      // 0
let d = string.indexOf("the", 2);    // 12
```

Иногда вам нужно найти последнее вхождение подстроки. В C это требует вызова `strstr` несколько раз, пока не будет найдено больше совпадений. JavaScript предоставляет функцию `lastIndexOf` для этой ситуации.

```
let string = "the cat and the dog";
let a = string.lastIndexOf("the");           // 12
let b = string.lastIndexOf("the", a - 1);    // 0
```

При оценке строк полезно проверять, начинается ли строка с определенной строки или заканчивается. Вы используете `strcmp` и `strncmp` для этого в C. Эта ситуация настолько распространена, что JavaScript предоставляет специальные функции `startsWith` и `endsWith`.

```
if (string.startsWith("And "))
    trace(`Don't start sentence with "and"`);
if (string.endsWith("..."))
    trace(`Don't end sentence with ellipsis`);
```

Функции

Разумеется, в JavaScript есть функции, как и в C. Некоторые функции в обоих языках очень похожи.

```
function add(a, b) {
    return a + b;
}
```

Аргументы функции

Поскольку переменные JavaScript могут содержать значения любого типа, тип аргументов не указывается, а указывается только их имя. Кроме того, в отличие от C и C++, здесь нет объявлений функций; вы просто пишете исходный код функции, а затем любой код, который может получить доступ к функции, может вызвать ее. Этот специальный подход позволяет ускорить кодирование.

В C и C++ вы можете передавать значение аргумента по ссылке, используя указатель, но в JavaScript вы всегда должны передавать аргументы по значению. Следовательно, функция JavaScript никогда не изменяет значение переданной ей переменной. Например, функция добавления в листинге 2-12 не изменяет значение `x`.

Листинг 2-12.

```
function add(a, b) {  
    a += b;  
    return a;  
}  
let x = 1;  
let y = 2;  
let z = add(x, y);
```

Когда вы передаете объект функции, функция может изменять свойства объекта, но не локальную переменную вызова, содержащую объект. Это похоже на передачу указателя на структуру данных в С. В листинге 2-13 функция setName добавляет свойство name к переданному ей объекту. Присвоение нового пустого объекта его параметру a не изменяет значение b.

Листинг 2-13.

```
function setName(a, name) {  
    a.name = name;  
    a = {};  
}  
  
let b = {};  
setName(b, "thermostat");  
// b.name is "thermostat"
```

В С и С++ реализация функции может определять количество переданных ей аргументов и может обращаться к каждому из них с помощью `va_start`, `va_end` и `va_arg`. Это мощные инструменты, но их использование может быть сложным. JavaScript также предоставляет инструменты для работы с аргументами функции. Любые аргументы, не переданные вызывающей программой, устанавливаются в значение `undefined`, поэтому (как это сделано для b в листинге 2.14) вы можете проверить, не был ли передан аргумент.

Листинг 2-14.

```
function add(a, b) {  
    if (b == undefined)  
        return NaN;  
    return a + b;  
}  
  
add(1);
```

Другой способ получить доступ к параметрам, переданным функции, — использовать специальную переменную `arguments`, которая ведет себя как массив, содержащий аргументы. Этот подход похож на использование `va_arg` с дополнительным преимуществом знания количества аргументов. В листинге 2-15 функция `add` принимает любое количество аргументов и возвращает их сумму.

Листинг 2-15.

```
function add() {  
    let result = 0;  
    for (let i = 0; i < arguments.length; i++)  
        result += arguments[i];  
    return result;  
}  
  
let c = add(1, 2);  
let d = add(1, 2, 3);
```

Использование аргументов распространено в JavaScript, но доступно не во всех ситуациях. Он представлен здесь, потому что вы, вероятно, увидите его в коде. Современный JavaScript имеет дополнительную функцию, называемую `arguments.callee`, которая обеспечивает аналогичную функциональность, всегда доступна и является более гибкой (см. листинг 2.16).

Листинг 2-16.

```
function add(...values) {  
    let result = 0;  
    for (let i = 0; i < values.length; i++)  
        result += values[i];  
    return result;  
}
```

Здесь ...values указывает, что все оставшиеся аргументы (в данном примере все аргументы) должны быть помещены в массив с именем values. Код в листинге 2-17 добавляет параметр округления для управления округлением значений перед суммированием.

Листинг 2-17.

```
function addR(round, ...values) {  
    let result = 0;  
    for (let i = 0; i < values.length; i++)  
        result += round ? Math.round(values[i]) : values[i];  
    return result;  
}  
  
let c = addR(false, 1.1, 2.9, 3.5); // c = 7.5  
let d = addR(true, 1.1, 2.9, 3.5);  // d = 8
```

Подобно тому, как остальные параметры объединяют несколько аргументов в массив, синтаксис расширения разделяет содержимое массива на отдельные аргументы. Синтаксис расширения использует тот же синтаксис с тремя точками, что и остальные параметры. Функция в листинге 2-18 суммирует абсолютное значение своих аргументов; сначала он принимает абсолютное значение аргументов, а затем вызывает функцию сложения, используя синтаксис расширения для вычисления суммы.

Листинг 2-18.

```
function addAbs(...values) {
    for (let i = 0; i < values.length; i++)
        values[i] = Math.abs(values[i]);
    return add(...values);
}

let c = addAbs(-1, -2, 3); // c = 6
```

Есть много других применений синтаксиса расширения, например, для клонирования массива:

```
let a = [1, 2, 3, 4];
let b = [...a];           // b = [1, 2, 3, 4]
```

Вы также можете использовать синтаксис расширения для объединения двух массивов:

```
let a = [1, 2];
let b = [3, 4];
let c = [...a, ...b];     // c = [1, 2, 3, 4]
```

В некоторых ситуациях полезно указать значение по умолчанию для аргумента. Это невозможно в C, но это делается в C++ с использованием того же синтаксиса, что и в JavaScript. В JavaScript, поскольку аргументы, не переданные вызывающей стороной, устанавливаются в значение `undefined`, вы можете указать значение по умолчанию для любого параметра, который имеет это значение. Функция в листинге 2-19 принимает значение температуры; если единицы измерения не указаны, по умолчанию используются градусы Цельсия.

Листинг 2-19.

```
function setCelsiusTemperature(temperature) {
    trace(`setCelsiusTemperature ${temperature}\n`);
}
```

```
function setTemperature(temperature, units = "Celsius") {
    switch (units) {
        case "Fahrenheit":
            temperature -= 32;
            temperature /= 1.8;
            break;
        case "Kelvin":
            temperature -= 273.15;
            break;
        case "Celsius":    // преобразование не требуется
            break;
    }
    setCelsiusTemperature(temperature);
}
```

```
setTemperature(14); // Аргумент единиц измерения по умолчанию равен Цельсию
setTemperature(14, "Celsius");
setTemperature(57, "Fahrenheit");
```

В отличие от C, каждая функция в JavaScript имеет возвращаемое значение; функция не может выйти без четко определенного возвращаемого значения. Рассмотрим три функции, показанные в листинге 2-20.

Листинг 2-20.

```
function a() {
    return undefined;
}
function b() {
    return;
}
function c() {
}
```

Функция *a* явно возвращает значение `undefined`. Функция *b* не предоставляет значение оператору `return` и поэтому возвращает неопределенное значение. Функция *c* не имеет оператора возврата, но возвращает неопределенное значение, как и функция *b*, потому что неопределенное значение является значением по умолчанию, которое возвращают все функции. Вы найдете все три формы в коде JavaScript, в зависимости от предпочтений автора кода. Они неразличимы вызывающей функцией.

Напротив, в C разрешен следующий код. Результатом функции *c* является любое значение, оказавшееся в памяти или регистре, зарезервированное для возвращаемого значения.

```
int c(void) {
}
int b = c();    // b неизвестно
```

Передача функций в качестве аргументов

В C принято передавать функции указатель на другую функцию, что позволяет настроить поведение передаваемой функции, например, предоставить функцию сравнения для использования при сортировке. Точно так же функции JavaScript могут быть переданы в качестве аргументов другой функции, как показано в листинге 2-21.

Листинг 2-21.

```
function square(a) {
    return a * a;
}
function circleArea(r) {
    return Math.PI * r * r;
}
function sum(filter, ...values) {
    let result = 0;
```

```

    for (let i = 0; i < values.length; i++)
        result += filter(values[i]);
    return result;
}

let a = sum(square, 1, 2, 3);    // 14
let b = sum(circleArea, 1);     // 3.14...

```

Вы также можете передавать встроенные функции в качестве аргументов. Например, следующий код вычисляет сумму квадратных корней остальных аргументов:

```
let c = sum(Math.sqrt, 1, 4, 9);    // 6
```

Часто при передаче функции эта функция используется только в одном месте. В C реализация функции часто находится не рядом с тем местом, где она вызывается, что ухудшает читабельность. В отличие от C, JavaScript допускает анонимные (безымянные) встроенные функции. В следующем примере вызывается функция `sum`, определенная в листинге 2-21, для вычисления суммы площадей равнобедренных треугольников с использованием анонимной встроенной функции в качестве фильтра:

```

let a = sum(function(a) {
    return a * (a / 2);
}, 1, 2, 3);    // 7

```

Анонимные функции широко используются в коде JavaScript для различных видов обратных вызовов. Видеть исходный код функции в качестве аргумента вызова функции немного необычно, но к этому можно привыкнуть. Если вы предпочитаете отделять реализации функций от вызовов функций, вместо этого вы можете использовать вложенные функции. В листинге 2-22 функция треугольника видна только внутри функции `main`. Использование вложенной функции удерживает реализацию функции фильтра рядом с тем местом, где она используется, что часто повышает удобство сопровождения кода.

Листинг 2-22.

```
function main() {
    function triangleArea(a) {
        return a * (a / 2);
    }

    let a = sum(triangleArea, 1, 2, 3); // 7
}
```

Объявление функций

Как отмечалось ранее, в JavaScript нет объявлений функций, в отличие от C и C++: когда вы объявляете функцию в JavaScript, вы фактически объявляете переменную. Следующая строка кода, используя общий синтаксис объявления функции, создает локальную переменную с именем `example`:

```
function example() {}
```

Следующая строка также создает локальную переменную с именем `example`, присваивая ей анонимную функцию:

```
let example = function() {};
```

Эти две строки кода эквивалентны, и обе функции можно вызывать одинаково. Но поскольку обе формы создают локальную переменную, у вас не может быть функции и локальной переменной с одинаковыми именами. Однако вы можете изменить функцию, на которую ссылается локальная переменная, как показано в листинге 2-23.

Листинг 2-23.

```
function log(a) {
    trace(a);
}

log("one");
```

ведение журнала

```
let originalLog = log; log = function(a) {}
```

```
log("two");
```

```
// Повторно включить ведение журнала
```

```
log = originalLog;
```

```
log("three");
```

Закрывтия

Одной из самых мощных функций функций JavaScript являются замыкания. Они обычно используются для функций обратного вызова. Замыкание связывает функцию с группой переменных вне функции. Ссылки на внешние переменные сохраняются в течение всего времени существования замыкания. Замыкания не существуют в C и были добавлены в C++ только в 2011 году в виде лямбда-выражений; следовательно, многие разработчики, работающие на C и C++, не знакомы с ними. Несмотря на неясное название, замыкания настолько просты в использовании, что легко забыть, что вы их используете.

В листинге 2-24 для реализации счетчика используется замыкание. Функция `makeCounter` возвращает функцию. Вы можете заставить одну функцию возвращать указатель на другую функцию в C, но здесь есть разница: возвращаемая анонимная функция ссылается на переменную с именем `value`, и эта переменная не является локальной для анонимной функции; вместо этого это локальная переменная в функции `makeCounter`, в которой содержится анонимная функция.

Листинг 2-24.

```
function makeCounter() {
    let value = 0;

    return function() {
        value += 1;
```



```

    return value;
  }
}

```

Каждый раз, когда вызывается функция, возвращаемая `makeCounter`, она увеличивает значение и возвращает это значение. Вот как это работает: когда функция ссылается на переменные за пределами своей локальной области видимости, говорят, что она «закрывает» эти переменные, автоматически создавая замыкание. В этом примере использование значения переменной в анонимной функции создает замыкание, которое позволяет получить доступ к значению локальной переменной из `makeCounter`. JavaScript позволяет безопасно использовать эту локальную переменную даже после возврата `makeCounter` и освобождения кадра стека `makeCounter` (см. листинг 2.25).

Листинг 2-25.

```

let counter = makeCounter();

let a = counter(); // 1
let b = counter(); // 2
let c = counter(); // 3

```

Пример в листинге 2-25 делает то, что вы ожидаете: функция `makeCounter` возвращает функцию-счетчик; каждый раз, когда вызывается функция счетчика, она увеличивает счетчик и возвращает новое значение. Но что произойдет, если вы дважды вызовете `makeCounter`? Возвращает ли второй вызов отдельный счетчик или ссылку на первый счетчик? Ответ см. в листинге 2-26.

Листинг 2-26.

```

let counterOne = makeCounter();
let counterTwo = makeCounter();

let a = counterOne(); // 1
let b = counterOne(); // 2

```

```
let c = counterTwo();    // 1
let d = counterTwo();    // 2
let e = counterOne();    // 3
let f = counterTwo();    // 3
```

Как видите, каждый раз, когда вызывается `makeCounter`, возвращаемая им функция имеет новое замыкание с отдельной копией значения.

Если вам сейчас сложно представить, как вы могли бы использовать замыкания в своем собственном коде, не волнуйтесь; многие программисты используют их, даже не осознавая этого. Замыкания распространены в API, которые используют функции обратного вызова; когда функция обратного вызова установлена, она часто закрывает переменные, которые она использует при вызове обратного вызова.

Если у вас есть опыт объектно-ориентированного программирования, вы можете признать, что замыкания, используемые таким образом, похожи на экземпляры объектов, и на самом деле они могут использоваться для этого. Однако у JavaScript есть лучшие альтернативы, использующие классы (представленные позже в этой главе).

Объекты

JavaScript — это объектно-ориентированный язык программирования; C нет. Существует несколько практических способов использования JavaScript без использования объектов. В предыдущих разделах этой главы даже общие операции с числами и строками требовали вызова методов числовых и строковых объектов. C++ — это объектно-ориентированный язык, но C++ и JavaScript используют очень разные подходы к объектам. Например, в C++ есть шаблоны классов, перегрузка операторов и множественное наследование — все это не является частью JavaScript. Если вы переходите с C, вам нужно немного узнать об объектах. Если вы переходите с C++, вам нужно узнать о более компактном подходе JavaScript к объектам. Хорошая новость заключается в том, что миллионы разработчиков успешно использовали объекты в JavaScript для создания веб-страниц, веб-сервисов, мобильных приложений и встроенных прошивок.

Для создания объектов в JavaScript используется ключевое слово `new`, как и в C++. Все объекты в JavaScript происходят от встроенного объекта `Object`. Следующие строки создают экземпляр `Object`:

```
let a = new Object();
let b = new Object;
```

Объект — это особый вид функции, называемый конструктором. Когда конструктор `Object` вызывается с помощью `new`, создается экземпляр `Object`, и функция конструктора выполняется для инициализации экземпляра. Если функции-конструктору не переданы аргументы, круглые скобки для списка аргументов необязательны. Следовательно, предыдущие две строки идентичны; какую форму вы используете, зависит от вашего личного стиля кодирования.

Есть много других объектов, встроенных в JavaScript. В листинге 2-27 показаны примеры вызова конструктора для некоторых из них. Подробности об этих и других встроенных объектах приведены в следующих разделах этой главы.

Листинг 2-27.

```
let a = new Array(10);           //массив длиной 10
let b = new Date("September 6, 2019");

let c = new Date;                // текущая дата и время
let d = new ArrayBuffer(128);    // 128-байтовый буфер
let e = new Error("bad value");
```

Базовый объект `Object` мало что делает сам по себе. Тем не менее, это распространено в коде JavaScript, потому что его можно использовать как специальную запись. В C вы используете структуру (`struct`) для хранения набора значений; в C++ вы используете либо структуру, либо класс (структуру или класс). Объект JavaScript, в отличие от структуры в C or C++, не является фиксированным набором полей. То, что C называет полями, в JavaScript называется свойствами. Как показано в листинге 2-28, вы можете добавлять свойства к объекту в любое время; их не нужно объявлять заранее.

Листинг 2-28.

```
let a = new Object;  
a.one = 1;  
a.two = "two";  
a.object = new Object;  
a.add = function(a, b) {  
    return a + b;  
};
```

Поскольку создание этих специальных объектов является обычным явлением, в JavaScript есть упрощение: вы можете использовать {} вместо нового объекта. Результат идентичен, но код более компактный. Вы можете инициализировать свойства объекта, перечислив свойства в фигурных скобках. Следующий пример эквивалентен предыдущему примеру:

```
let a = {one: 1, two: "two", object: {}},  
    add: function(a, b) {return a + b;}};
```

Разработчики JavaScript, как правило, предпочитают стиль фигурных скобок (и в этой книге он используется почти исключительно), потому что он более компактен и удобен для чтения.

Сокращение объекта

Сокращение объекта. Обычно результат нескольких вычислений сохраняется в локальных переменных, а затем помещается в объект. Когда локальные переменные имеют то же имя, что и свойства объекта, код выглядит избыточным, как в примере в листинге 2-29.

Листинг 2-29.

```
let one = 1;  
let two = "two";  
let object = {};
```

```
let add = function(a, b) {return a + b;};  
let result = {one: one, two: two, object: object, add: add};
```

Поскольку такая ситуация возникает часто, JavaScript предоставляет для нее ярлык. Код в листинге 2-30 эквивалентен предыдущему примеру.

Листинг 2-30.

```
let one = 1;  
let two = "two";  
let object = {};  
let add = function(a, b) {return a + b;};  
let result = {one, two, object, add};
```

Другой ярлык доступен для определения свойств, значением которых является функция. В листинге 2-31 показан простой подход.

Листинг 2-31.

```
let object = {  
  add: function(a, b) {  
    return a + b;  
  },  
  subtract: function(a, b) {  
    return a - b;  
  }  
};
```

В листинге 2-32 показана сокращенная версия, в которой исключено двоеточие (:) и ключевое слово function.

Листинг 2-32.

```
let object = {  
  add(a, b) {  
    return a + b;  
  },  
  subtract(a, b) {  
    return a - b;  
  }  
};
```

Помимо того, что он более компактен и удобочитаем, этот же синтаксис используется для определения классов в JavaScript, как вы скоро увидите.

Удаление свойств

Вы можете не только добавлять свойства к объекту JavaScript в любое время, но и удалять их. Свойства удаляются с помощью ключевого слова `delete`:

```
delete a.one;  
delete a.missing;
```

Как только свойство удалено, получение его из объекта дает значение `undefined`. Вы можете вспомнить, что это то же самое значение, которое возвращается, когда вы пытаетесь получить доступ к символу строки за пределами длины строки. Использование удаления для свойства, которого нет у объекта, не является ошибкой. Например, ошибка не возникает, когда (данный объект `a` со свойствами `один`, `два` и объект) `a.missing` удаляется, как показано ранее.

Программисты на C++ знакомы с удалением как способом уничтожения объекта, поэтому могут ожидать, что удаление свойства приведет к уничтожению объекта, на который ссылается это свойство; однако ключевое слово удаления в JavaScript отличается, как будет показано ниже в разделе «Управление памятью».

Проверка свойств

Поскольку свойства могут появляться и исчезать в любое время, иногда вам нужно проверить, присутствует ли конкретное свойство в объекте. Есть два способа сделать это. Поскольку любое отсутствующее свойство имеет значение `undefined`, вы можете проверить, дает ли получение свойства значение `undefined`.

```
if (a.missing == undefined)
    trace("a does not have property 'missing'");
```

Но не делайте этого! Есть несколько тонких проблем, которые могут возникнуть. Например, рассмотрим этот код:

```
let a = {missing: undefined};
if (a.missing == undefined)
    trace("a does not have property 'missing'");
```

Здесь объект имеет отсутствующее свойство, которое имеет значение `undefined`. Есть и другие причины, по которым эта проверка может не пройти, но пока этого примера достаточно, чтобы продемонстрировать необходимость лучшего решения. Использование ключевого слова `in` — лучший способ проверить существование свойства. Следующий пример работает во всех ситуациях:

```
if (!("missing" in a))
    trace("a does not have property 'missing'");
```

Добавление свойств к функциям

Функции в JavaScript являются объектами, что означает, что вы можете добавлять и удалять свойства функции так же, как и любого другого объекта. В листинге 2-33 определена функция с именем `calculate`, которая поддерживает три операции, каждая из которых соответствует свойству функции, которой присвоена константа: сложение равно 1, вычитание равно 2, а умножение равно 3. Определенные здесь операции аналогичны перечислению в C или C++. Однако вместо того, чтобы определяться отдельно от функции `calculate` в виде перечисления в C или C++, значения операции присоединяются непосредственно к функции, которая их

использует. Этот способ предоставления имен для констант используется в некоторых частях Moddable SDK.

Листинг 2-33.

```
function calculate(operation, a, b) {  
    if (calculate.add == operation)  
        return a + b;  
    if (calculate.subtract == operation)  
        return a - b;  
    if (calculate.multiply == operation)  
        return a * b;  
}  
calculate.add = 1;  
calculate.subtract = 2;  
calculate.multiply = 3;  
  
let a = calculate(calculate.add, 1, 2);           // 3  
let b = calculate(calculate.subtract, 1, 2);      // -1
```

Замораживание объектов

Есть ситуации, в которых вы хотите гарантировать, что свойства объекта не могут быть изменены. У вас может возникнуть соблазн использовать `const` для достижения этого:

```
const a = {  
    b: 1  
};
```

Однако это не работает. Использование `const` не делает объект справа от `=` в объявлении константы доступным только для чтения; в этом примере он делает только доступ только для чтения. Рассмотрим эти последующие задания:


```

a = 3;          // генерирует ошибку
a.b = 2;        // ОК — можно изменить существующее свойство
a.c = 3;        // ОК — можно добавить новое свойство

```

Чтобы предотвратить изменения объекта, который является значением константы, вы можете использовать `Object.freeze`, встроенную функцию, которая делает все существующие свойства объекта доступными только для чтения и предотвращает добавление новых свойств. Как вы можете видеть в листинге 2-34, попытки изменить значение свойства в замороженном объекте или добавить к объекту новое свойство приводят к ошибкам.

Листинг 2-34.

```

const a = Object.freeze({
    b: 1
});

a = 3;          // выдает ошибку
a.b = 2;        // ошибка - невозможно изменить существующее свойство
a.c = 3;        // ошибка - нельзя добавить новое свойство

```

Обратите внимание, что `Object.freeze` возвращает переданный ему объект, что удобно в этом примере, поскольку позволяет избежать добавления строки кода. `Object.freeze` редко используется в JavaScript для Интернета сегодня, но Moddable SDK использует его широко, потому что он позволяет эффективно хранить объекты в ПЗУ или флэш-памяти встроенных устройств, экономя ограниченный объем оперативной памяти.

`Object.freeze` — поверхностная операция, что означает, что она не замораживает вложенные объекты. В листинге 2-35, например, вложенный объект, присвоенный свойству `c`, не зафиксирован.

Листинг 2-35.

```
const a = Object.freeze({
  b: 1,
  c: {
    d: 2
  }
});
a.c.d = 3; // OK
a.c.e = 4; // OK
a.b = 2;    // ошибка - невозможно изменить существующее свойство
a.e = 3;    // ошибка - нельзя добавить новое свойство
```

Вы можете явно заморозить **c**, но это становится многословным и подверженным ошибкам, как показано в листинге 2.36.

Листинг 2-36.

```
const a = Object.freeze({
  b: 1,
  c: Object.freeze({
    d: 2
  })
});
```

Поскольку замораживание объектов помогает оптимизировать использование памяти на встроенных устройствах, движок XS JavaScript, используемый в Moddable SDK, расширяет `Object.freeze` необязательным вторым аргументом, который включает глубокую заморозку, то есть рекурсивную заморозку всех вложенных объектов (см. листинг 2-37).

Листинг 2-37.

```
const a = Object.freeze({  
    b: 1,  
    c: {  
        d: 2  
    }  
}, true);  
a.c.d = 3; // / ошибка - невозможно изменить существующее свойство  
a.c.e = 4; // ошибка - нельзя добавить новое свойство
```

Обратите внимание, что это расширение `Object.freeze` не является частью стандарта языка JavaScript, поэтому оно не работает в большинстве сред. Однако он удовлетворяет общие потребности в разработке встраиваемых систем. Возможно, будущая версия языка JavaScript будет поддерживать эту возможность.

Если вашему коду необходимо знать, заморожен ли объект, вы можете использовать `Object.isFrozen`. Как и `Object.freeze`, это неглубокая операция, поэтому она не сообщает вам, заморожены ли какие-либо вложенные объекты.

```
if (!Object.isFrozen(a)) {  
    a.b = 2;  
    a.c = 3;  
}
```

Заморозка объекта — это односторонняя операция: нет `Object.unfreeze`. Это связано с тем, что `Object.freeze` иногда используется в качестве меры безопасности, чтобы предотвратить вмешательство ненадежного клиентского кода в объект. Если ненадежный код сможет разморозить объект, это позволит нарушить меры безопасности.

null

Подобно C и C++, код JavaScript использует значение `null`. В C и C++ это записывается как `NULL`, чтобы указать, что оно определено с помощью макроса; в JavaScript значение `null` является встроенным.

C использует `NULL` в качестве значения для указателей, которые в настоящий момент ни на что не ссылаются. В JavaScript нет указателей, поэтому это значение не имеет смысла. В JavaScript `null` — это значение, указывающее на отсутствие ссылки на объект. Значение `null` рассматривается как специальный нулевой объект и, следовательно, имеет тип `Object`.

Легко спутать `null` и `undefined`. Они похожи, но не идентичны: `undefined` означает, что значение не задано; `null` явно указывает на отсутствие ссылки на объект, что означает, что переменная или свойство будут содержать объект в какой-то момент во время его выполнения. Как правило, когда локальная переменная или свойство объекта предназначены для ссылки на объект, присваивайте значение `null`, когда объекта нет.

Сравнения

Сравнивать два значения в C по большей части просто, потому что вы обычно сравниваете два значения одного и того же типа. В некоторых случаях язык C применяет преобразование типов перед сравнением. Это позволяет вам, например, сравнивать значение `uint8_t` со значением `uint32_t` без необходимости явного преобразования типа любого значения. C++ значительно расширяет возможности сравнений, предоставляя перегрузку операторов, что позволяет программистам создавать свои собственные реализации операторов сравнения для определяемых ими типов. В этом отношении JavaScript больше похож на C, чем на C++; он не поддерживает перегрузку операторов, поэтому поведение сравнений полностью определяется языком JavaScript.

Как и C, JavaScript неявно преобразует определенные типы при выполнении сравнения с оператором равенства (==). В листинге 2-38 показано несколько примеров.

Листинг 2-38.

```
let a = 1 == "1";      // true
let b = 0 == "";       // true
let c = 0 == false;    // true
let d = "0" == false;  // true
let e = 1 == true;     // true
let f = 2 == true;     // false
let g = Infinity == "Infinity"; // истина
```

Как видите, правила преобразования типов при сравнении не всегда соответствуют вашим ожиданиям. По этой причине программисты JavaScript часто избегают неявных преобразований, используя вместо этого оператор строгого равенства (===), как показано в листинге 2.39. Оператор строгого равенства никогда не выполняет преобразование типов; если два значения имеют разные типы, они всегда не равны.

Листинг 2-39.

```
let a = 1 === "1";     // ложь
let b = 0 === "";      // false
let c = 0 === false;   // false
let d = "0" === false; // false
let e = 1 === true;    // false
let f = 2 === true;    // false
let g = Infinity === "Infinity"; // false
```

JavaScript также предоставляет оператор строгого неравенства (!==), который можно использовать вместо оператора неравенства (!=), чтобы избежать преобразования типов:

```
let a = 1 !== "1";      // истина
let b = 0 !== "";       // истина
let c = 0 !== false;    // true
```

Во многих случаях нет ничего плохого в использовании == and != вместо строгих версий. Однако крайние случаи, когда поведение отличается, могут привести к ошибкам, которые трудно отследить. Таким образом, наилучшая практика программирования на JavaScript — всегда использовать строгие версии операторов.

В некоторых примерах этой главы, предшествующих знакомству с операторами строгого сравнения, используются == и !=. Теперь, когда вы знаете о строгих версиях этих операторов и почему они предпочтительны, в примерах в оставшейся части этой книги будут использоваться только строгие операторы.

Сравнение объектов

Когда два объекта сравниваются в JavaScript, они равны, только если ссылаются на один и тот же экземпляр. Обычно это то, что вы ожидаете, хотя иногда разработчики ошибочно ожидают, что если все свойства двух разных экземпляров равны, результат сравнения на равенство будет верным. Такое глубокое сравнение не обеспечивается непосредственно JavaScript, хотя при необходимости оно может быть реализовано в вашем приложении.

```
let a = {b: 1};
let b = a === {b: 1};    // ложь
let c = a;
let d = a === c;         // истина
```

В C++ поведение по умолчанию для сравнения объектов такое же, как в JavaScript. Используя перегрузку операторов, программисты на C++ могут выполнять глубокие сравнения, если класс реализует поддержку.

Ошибки и исключения

JavaScript включает встроенный тип `Error`, который используется для сообщения о проблемах, возникающих во время выполнения. Ошибки почти всегда используются вместе с механизмом исключений JavaScript, который во многом похож на исключения C++. Язык C не включает исключений, хотя аналогичная функциональность часто создается с помощью `setjmp` и `longjmp` в стандартной библиотеке C.

Чтобы создать ошибку, вызовите конструктор `Error`. Чтобы помочь с отладкой, вы можете указать необязательное сообщение об ошибке.

```
let a = new Error;
let b = new Error("invalid value");
```

Существуют и другие виды ошибок, которые используются для обозначения конкретной проблемы. К ним относятся `RangeError`, `TypeError` и `ReferenceError`. Вы используете их так же, как `Error`. Чаще всего используется просто `Error`, но вы можете использовать и другие, если они подходят для вашей ситуации.

Если у вас есть ошибка, вы сообщаете об этом с помощью оператора `throw` (листинг 2-40).

Листинг 2-40.

```
function setTemperature(value) {
    if (value < 0)
        throw new RangeError("too cold");
    ...
}
```

Вы можете указать любое значение после оператора `throw`, хотя по соглашению это значение обычно является экземпляром ошибки.

Когда возникает исключение, текущий путь выполнения завершается. Выполнение возобновляется с первого блока `catch` в стеке. Если в стеке нет блоков `catch`, исключение считается необработанным. Необработанные исключения игнорируются, то есть хост не пытается обрабатывать исключение. Чтобы перехватить исключение, вы пишете блоки `try` и `catch` так же, как в C++; Листинг 2-41 следует из предыдущего примера, чтобы проиллюстрировать это.

Листинг 2-41.

```

try {
    setTemperature(-1); // throws an exception
    // Execution never reaches here
    displayMessage("Temperature set to -1\n");
}
catch (e) {
    trace(`setTemperature failed: ${e}\n`);
}

```

Когда `setTemperature` генерирует исключение в этом примере, выполнение переходит к блоку `catch`, пропуская вызов `displayMessage`. локальная переменная с именем `e`, указанная в скобках после ключевого слова `catch`. Если ваш блок `catch` не использует это значение, вы можете опустить круглые скобки после `catch`, как показано в листинге 2.42.

Листинг 2-42.

```

try {
    setTemperature(-1); //генерирует исключение
    // Выполнение никогда не достигает
    reaches heredisplayMessage("Temperature set to -1\n");
}
catch {
    trace("setTemperature failed\n");
}

```


После обнаружения ошибки у вас есть возможность распространить ее, как если бы она не была обнаружена. Это полезно, если вы хотите выполнить очистку при возникновении ошибки, и ошибка также должна быть обработана кодом, находящимся выше по стеку вызовов. Чтобы распространить исключение, используйте оператор `throw` внутри блока `catch` (листинг 2-43).

Листинг 2-43.

```
try {
    setTemperature(-1); // throws an exception
    // Execution never reaches here
    displayMessage("Temperature
    set to -1");
}
catch (e) {
    trace(`setTemperature failed: ${e}\n`);
    throw e;
}
```

Ваша обработка исключений может также включать блок `finally`, как показано в листинге 2-44. (Стандартный C++ не предоставляет `finally`, но это часть диалекта Microsoft C++.) Блок `finally` вызывается всегда, независимо от того, как обрабатывается исключение или даже если оно не перехватывается блоком `catch`.

Листинг 2-44.

```
try {
    setTemperature(-1);
}
catch (e) {
    trace(`setTemperature failed: ${e}\n`);
    throw e;
}
```

```
finally {
    displayMessage(`Temperature set to ${getTemperature()}\n`);
    // всегда выполняется
}
```

В листинге 2-44 вызов `displayMessage` происходит независимо от того, генерирует ли `setTemperature` исключение. При использовании `finally` вы можете опустить блок `catch` (листинг 2-45), и в этом случае исключение будет продолжать распространяться вверх по стеку после выполнения блока `finally`.

Листинг 2-45.

```
try {
    setTemperature(-1);
}
finally {
    displayMessage(`Temperature set to ${getTemperature()}`);
}
```

Когда исключение не обрабатывается — например, когда `setTemperature` генерирует исключение в листингах 2-44 и 2-45 — предупреждение отслеживается до консоли отладки. Оставление перехваченного исключения не обязательно является ошибкой, но это может быть признаком проблемы. Предупреждение может включать имя функции, обнаружившей неперехваченное исключение; это нативная функция, часто являющаяся частью среды выполнения Moddable SDK, поэтому имя может быть незнакомым.

Хотя в этих примерах в блоках `try` всего несколько строк кода, в реальном коде часто есть большие блоки кода в одном блоке `try`. Это позволяет вам сделать код для обработки ошибок небольшим и изолированным, а не включать его в каждый вызов функции, как это может быть в случае C.

Комбинация блоков `try`, `catch` и `finally` дает вам большая гибкость в том, как ваш код реагирует или не реагирует на исключения. Не беспокойтесь слишком много об их использовании, когда вы начинаете. Обычно код пишут без обработки исключений, а затем добавляют его позже, когда вы устраняете случаи сбоя.

Классы

Подобно C++, JavaScript позволяет создавать собственные типы объектов путем определения классов. В JavaScript вы используете ключевое слово `class` для определения и реализации ваших классов. Классы в JavaScript немного проще, чем в C++. Даже если вы не планируете создавать свои собственные классы, вам следует ознакомиться с классами JavaScript, чтобы вы могли понимать код, написанный другими.

До этого разработчики JavaScript создавали классы, используя подходы более низкого уровня, включая `Object.create`, или напрямую манипулировали свойством прототипа объекта. Хотя эти методы все еще работают и распространены в устаревшем коде в Интернете, в этом разделе основное внимание уделяется современному JavaScript, где класс делает код более читабельным и не влияет на производительность во время выполнения.

Конструктор класса и методы

В листинге 2-46 показан простой класс `Bulb`, представляющий лампочку, которая может быть либо включена, либо выключена.

Листинг 2-46.

```

class Bulb {
    constructor(name) {
        this.name = name;
        this.on = false;
    }
    turnOn() {
        this.on = true;
    }
    turnOff() {
        this.on = false;
    }
    toString() {
        return `${this.name}" is ${this.on ? "on" : "off"}`;
    }
}

```

В отличие от C++, здесь нет объявления класса; есть только реализация. Синтаксис, используемый для определения функций в классе, такой же, как вы уже видели для функций вне класса (в разделе «Сокращение объектов»). Однако, в отличие от того, когда функции определяются как свойства в обычном объекте, в классе между функциями нет запятых.

Как видно из листинга 2.46, класс Bulb представляет собой набор функций. Функция с именем конструктор в классе особенная; он вызывается автоматически при создании объекта. Конструктор выполняет всю необходимую инициализацию, прежде чем новый экземпляр будет возвращен создателю. Следующий код создает экземпляр Bulb:

```

let wallLight = new Bulb("wall light");
wallLight.turnOn();

```

Еще одна специальная функция в классах JavaScript — `toString`. Эта функция вызывается автоматически в ситуациях, когда JavaScript требует строкового представления объекта. Метод `toString` класса `Bulb` предоставляет сводку текущего состояния, что полезно для отладки.

```
let wallLight = new Bulb("wall light");
wallLight.turnOn();
trace(wallLight); // вывод: "настенный светильник" включен
```

Поскольку функция трассировки выводит строки, она преобразует свой аргумент в строку, которая вызывает метод `toString`. Вы также можете вызвать `toString` напрямую, как в `wallLight.toString()`.

Метод `toString` — это особый случай в JavaScript; нет других функций преобразования, таких как `toNumber`.

Примечание Вызов конструктора класса должен происходить после определения класса. Это означает, что вы можете вызывать `New Bulb` только после определения класса в листинге 2-46, но не раньше. Вызов его до этого вызывает исключение времени выполнения с сообщением `GET Bulb: NOT INITIALIZED` еще!.

Статические методы

Как и в C++, класс JavaScript может включать статические методы, то есть функции, доступ к которым осуществляется через класс, а не через экземпляр. Простым примером статического метода является метод, возвращающий версию реализации (листинг 2.47).

Листинг 2-47.

```
class Bulb {
    ...      // как раньше
    static getVersion() {
        return 1.2;
    }
}
```

Статические методы присоединяются к классу и поэтому могут быть вызваны еще до создания экземпляра.

```
if (Bulb.getVersion() < 1.5)
    throw new Error("incompatible version");
```

Подклассы

Большая часть силы классов исходит из возможности создавать подклассы. В JavaScript вы используете ключевое слово `extends` для создания подкласса. Код в листинге 2-48 реализует `DimmableBulb` как подкласс класса `Bulb`, определенного в листинге 2-46.

Листинг 2-48.

```
class DimmableBulb extends Bulb {
    constructor(name) {
        super(name);
        this.dimming = 100;
    }
    setDimming(value) {
        if ((value < 0) || (value > 100))
            throw new RangeError("bad dimming value");
        this.dimming = value;
    }
}
```

Как и следовало ожидать от подкласса, класс `DimmableBulb` наследует методы `TurnOff` и `TurnOn` от `Bulb`. Функция конструктора требует некоторого пояснения. Он немедленно вызывает `super` с тем же аргументом, который был ему передан. В классе JavaScript `super` — это ссылка на конструктор суперкласса, в данном случае на конструктор `Bulb`. Таким образом, первая задача, которую выполняет конструктор `DimmableBulb`, — создание его суперкласса `Bulb`.

Хотя конструктор подкласса может выполнять вычисления перед вызовом конструктора своего суперкласса, в конечном итоге он должен вызвать его. Пока это не произойдет, это не определено, поэтому любая попытка получить или установить свойства экземпляра завершится неудачно. Например, изменение конструктора `DimmableBulb`, как показано в листинге 2-49, создает исключение при попытке установить свойство затемнения, поскольку оно еще недоступно.

Листинг 2-49.

```
class DimmableBulb extends Bulb {
    constructor(name) {
        this.dimming = 100; // throws an exception
        super(name);
    }
    ...
}
```

Реализация `DimmableBulb` также наследует метод `toString` от `Bulb`. Реализация `toString` для `Bulb` не выводит уровень затемнения; реализация `toString` для `DimmableBulb` (листинг 2-50) добавляет уровень затемнения, сначала вызывая метод `toString` в `Bulb` (как указано `super`), а затем добавляя уровень затемнения к этому результату.

Листинг 2-50.

```
class DimmableBulb extends Bulb {
    ...
    toString() {
        return super.toString() +
            ` with dimming ${this.dimming}`;
    }
}
```

Встроенный класс `Object` является конечным надклассом всех классов JavaScript. Класс `Bulb` наследуется непосредственно от `Object`. Это подразумевается отсутствием предложения `extends` в его реализации, но его также можно указать явно, как показано в листинге 2-51.

Листинг 2-51.

```
class Bulb extends Object {
    constructor(name) {
        super();
        this.name = name;
        this.on = false;
    }
    ...
}
```

Обратите внимание, что поскольку `Bulb` теперь явно расширяет `Object`, конструктор `Bulb` должен вызывать конструктор класса, который он расширяет, вызывая `super`. Если вызов `super` опущен, доступ к нему вызывает исключение с сообщением `Bulb: this еще не инициализировано!`.

Классы, происходящие непосредственно от `Object`, обычно не пишутся таким образом, чтобы исходный код оставался кратким. Но этот пример намекает на еще одну особенность классов JavaScript: возможность создавать подклассы для встроенных объектов. Пример в листинге 2-52 является подклассом встроенного класса `Array` (о котором вы скоро узнаете больше), чтобы добавить методы для нахождения общего и среднего значений в массиве.

Листинг 2-52.

```

class MyArray extends Array {
    sum() {
        let total = 0;
        for (let i = 0; i < this.length; i++)
            total += this[i];
        return total;
    }
    average() {
        return this.sum() / this.length;
    }
}

let a = new MyArray;
a[0] = 1;
a[1] = 2;
let b = a.sum();           // 3
let c = a.average();       // 1.5

```

При создании продукта у вас может быть более одного экземпляра `Bulb`. Например, вы можете сделать выключатель, который управляет несколькими лампочками, и вы можете хранить этот список источников света в виде массива. Для этой цели вы можете создать подкласс `Array` с подклассом (`Bulbs` в примере из листинга 2-53), обеспечивающим пакетные операции над лампочками.

Листинг 2-53.

```

class Bulbs extends Array {
    allOn() {
        for (let i = 0; i < this.length; i++)
            this[i].turnOn();
    }
    allOff() {
        for (let i = 0; i < this.length; i++)
            this[i].turnOff();
    }
}

let bulbs = new Bulbs;
bulbs[0] = new Bulb("hall light");
bulbs[1] = new DimmableBulb("wall light");
bulbs[2] = new DimmableBulb("floor light");
bulbs.allOn();

```

Было бы неплохо иметь метод `dimAll` в `Bulbs`, но он будет работать только для экземпляров `DimmableBulb`; вызов `setDimming` для экземпляра `Bulb` вызывает исключение, потому что метод не существует. Здесь помогает оператор JavaScript `instanceof`, позволяющий определить, соответствует ли экземпляр определенному классу (листинг 2-54).

Листинг 2-54.

```

let a = new Bulb("hall light");
let b = new DimmableBulb("wall light");

let c = a instanceof Bulb;      // true
let d = b instanceof Bulb;      // true
let e = a instanceof DimmableBulb; // false
let f = b instanceof DimmableBulb; // true

```

Как видите, `instanceof` проверяет указанный класс, включая его суперклассы. В примере в листинге 2-54 это означает, что `b` является экземпляром как `DimmableBulb`, так и `Bulb`, поскольку `Bulb` является надклассом `DimmableBulb`. С этим знанием теперь возможна реализация `dimAll` (листинг 2-55).

Листинг 2-55.

```
class Bulbs extends Array {
    ...
    dimAll(value) {
        for (let i = 0; i < this.length; i++) {
            if (this[i] instanceof DimmableBulb)
                this[i].setDimming(value);
        }
    }
}
```

Свойства экземпляра `Bulb` являются обычными свойствами JavaScript, что делает их доступными как для реализации класса, так и для кода, использующего класс:

```
let wallLight = new Bulb("wall light");
wallLight.turnOn();
trace(`Light on: ${wallLight.on}\n`);
```

Это полезно, но иногда вы должны использовать представление значения внутри реализации, отличное от того, которое вы используете в API. Например, метод `setDimming` принимает значения от 0 до 100, потому что проценты — это естественный способ описания уровня затемнения; однако реализация может предпочесть хранить значение от 0 до 1,0, потому что это более эффективно для ее внутренних вычислений. Классы JavaScript поддерживают геттеры и сеттеры, полезные для подобных преобразований. Реализация в листинге 2-56 заменяет метод `setDimming` геттером и сеттером для свойства затемнения.

Листинг 2-56.

```

class DimmableBulb extends Bulb {
    constructor(name) {
        super(name);
        this._dimming = 1.0;
    }
    set dimming(value) {
        if ((value < 0) || (value > 100))
            throw new RangeError("bad dimming value");
        this._dimming = value / 100;
    }
    get dimming() {
        return this._dimming * 100;
    }
}

let a = new DimmableBulb("hall light");
a.dimming = 50;
a.dimming = a.dimming / 2;

```

Пользователи класса получают доступ к свойству затемнения как к обычному свойству JavaScript. Однако, когда свойство установлено, вызывается метод `set` затемнения класса, а когда свойство считывается, вызывается метод получения затемнения.

Приватные поля

Геттер и сеттер в листинге 2-56 сохраняют значение в свойстве с именем `_dimming`. Код JavaScript уже давно использует подчеркивание (`_`) в начале имен свойств, чтобы указать, что они предназначены только для внутреннего использования. В отличие от C++, в JavaScript не предусмотрены приватные поля в классах. Работа по добавлению приватных полей в стандарт JavaScript почти завершена; в этом разделе представлены приватные поля в том виде, в каком они должны быть

в стандарте JavaScript. Частные поля поддерживаются движком XS JavaScript для использования в вашей встроенной разработке.

Частные поля в JavaScript обозначаются префиксом имени поля с символом решетки (#). Приватное поле должно быть объявлено в теле класса. В листинге 2-57 показана версия DimmableBulb из листинга 2-56, переписанная для использования частного поля с именем #dimming вместо _dimming.

Листинг 2-57.

```
class DimmableBulb extends Bulb {
    #dimming = 1.0;

    set dimming(value) {
        if ((value < 0) || (value > 100))
            throw new RangeError("bad dimming value");
        this.#dimming = value / 100;
    }
    get dimming() {
        return this.#dimming * 100;
    }
}

let a = new DimmableBulb("hall light");
a.dimming = 50;
a.dimming = a.dimming / 2;
a.#dimming = 100;    // error
```

Обратите внимание, что приватное поле #dimming инициализировано значением 1.0 в его объявлении в теле класса. Это необязательно; вместо этого его можно инициализировать в конструкторе. Пока он не инициализирован, он имеет значение undefined.

Обратите также внимание на то, что в примере в листинге 2-57 конструктор полностью исключен. Это возможно здесь, потому что #dimming уже инициализирован. Поскольку DimmableBulb наследуется от Bulb, когда в DimmableBulb нет конструктора, конструктор Bulb

автоматически вызывается при создании экземпляра. Как и следовало ожидать от C++, код вне класса не имеет доступа к закрытым полям; следовательно, последняя строка примера, которая пытается присвоить значение `#dimming`, выдает ошибку.

JavaScript не поддерживает друзей C++ или защищенные функции классов. Частные свойства класса напрямую доступны только коду внутри тела класса. Частные поля являются действительно частными, оставаясь невидимыми даже для подклассов и суперклассов.

Приватные методы

Вместе с приватными полями стандарт языка JavaScript добавляет приватные методы — функции, которые можно вызывать только из реализации класса. Например, класс `DimmableBulb` в листинге 2-58 имеет закрытый метод `#log`.

Листинг 2-58.

```
class DimmableBulb extends Bulb {
    #dimming = 1.0;

    set dimming(value) {
        if ((value < 0) || (value > 100))
            throw new RangeError("bad dimming value");
        this.#dimming = value / 100;

        this.#log(`set dimming ${this.#dimming}`);
    }
    get dimming() {
        this.#log("get dimming");
        return this.#dimming * 100;
    }
}
```

```

    #log(msg) {
        trace(msg);
    }
}
let a = new DimmableBulb("hall light"); // "свет в холле"
a.#log("test");    // error

```

Использование функций обратного вызова в классах

Бывают случаи, когда реализация класса передает функцию API в качестве обратного вызова. Обычный пример — когда API использует таймер для отсрочки действия на будущее. Веб-разработчики JavaScript обычно используют `setTimeout` для этой цели; во встроенном JavaScript эквивалентом является `Timer.set`. Пример в листинге 2-59 добавляет к классу `Bulb` метод для включения или выключения света по истечении заданного интервала времени.

Листинг 2-59.

```

class Bulb {
    ...
    setOnAfter(value, delayInMS) {
        let bulb = this;
        Timer.set(function() {
            if (value)
                bulb.turnOn();
            else
                bulb.turnOff();
        }, delayInMS);
    }
}

```

Метод `setOnAfter` вызывает `Timer.set` с двумя аргументами: анонимная функция, которая должна выполняться после истечения времени таймера, и время ожидания в миллисекундах. Функция обратного вызова использует замыкание для доступа к лампочке; это необходимо, потому что значение `this` в обратном вызове — это не `instanceof bulb`, с которым был вызван `setOnAfter`, а скорее глобальный объект (то есть `globalThis`). Этот код работает, но в JavaScript есть лучшие инструменты для реализации этой функциональности.

Как и в современном C++, в современном JavaScript есть лямбда-функции, обычно называемые стрелочными функциями из-за синтаксиса `=>`, используемого для их объявления. Как и замыкания, стрелочные функции немного сложны для понимания, но просты в использовании. Когда вызывается стрелочная функция, ее значение `this` совпадает со значением `this` функции, в которой определена стрелочная функция. Эта особенность стрелочных функций называется лексическим `this`, поскольку значение `this` внутри стрелочной функции берется из объемлющей функции.

Стрелочные функции популярны, потому что они сохраняют значение `this` и более лаконичны в исходном коде. В примерах в листинге 2-60 показаны те же функции, использующие ключевое слово `function` и синтаксис стрелочной функции.

Листинг 2-60.

```
function randomTo100() {
    return Math.random() * 100;
}
let randomTo100 = () => Math.random() * 100;

function cube(a) {
    return a * a * a;
}
let cube = a => a * a * a;

function add(a, b) {
    return a + b;
}
let add = (a, b) => a + b;
```



```
function upperFirst(str) {
    let first = str[0].toUpperCase();
    return first + str.slice(1);
}
let upperFirst = str => {
    let first = str[0].toUpperCase();
    return first + str.slice(1);
};
```

Все пары примеров в листинге 2-60 функционально эквивалентны, за исключением значения `this` в функциях; однако в примерах это не используется. Код в листинге 2-61 использует функцию стрелки для улучшения реализации `setOnAfter` (в листинге 2-59) за счет использования лексического выражения `this` для устранения локальной переменной лампочки. Используя этот подход, код обратного вызова может использовать это так же, как и методы класса.

Листинг 2-61.

```
class Bulb {
    ...
    setOnAfter(value, delayInMS) {
        Timer.set(
            () => value ? this.turnOn() : this.turnOff(),
            delayInMS
        );
    }
}
```

Важно быть знакомым со стрелочными функциями, потому что они очень распространены в JavaScript. Вы встретите их в некоторых примерах этой книги. Имейте в виду, что стрелочные функции — это не просто альтернативный способ написания исходного кода функций; они также изменяют значение `this` внутри функции.

Модули

Модули — это механизм в JavaScript для упаковки библиотеки кода. Есть некоторое сходство между модулями JavaScript и разделяемыми или динамическими библиотеками в C и C++: оба определяют экспорт для совместного использования ограниченного числа классов, функций и значений; и обе могут импортировать классы, функции и значения из других библиотек. Как и динамические библиотеки в C, модули JavaScript загружаются во время выполнения. Есть также много отличий, в том числе отсутствие JavaScript-эквивалента статически связанной библиотеки C.

Импорт из модулей

Чтобы использовать возможности, предоставляемые модулем, вы должны сначала импортировать соответствующие классы, функции или значения. Существует множество различных способов импорта из модуля с гибкостью, которая дает вам контроль над тем, что вы импортируете, и как вы называете эти импорты.

В примерах из предыдущего раздела использовался класс `Timer` без указания его происхождения. Класс `Timer` содержится в модуле `timer`. Чтобы импортировать из модуля, используется оператор импорта.

```
import Timer from "timer";

Timer.set(() => trace("done"), 1000);
```

Оператор `import` является особенным в JavaScript, поскольку он выполняется перед всем остальным кодом. Обычно оператор импорта помещается вверху исходного кода, как операторы `include` в C, но даже если они не первые, они все равно выполняются первыми.

Предыдущая форма оператора импорта состоит из двух частей:

- Имя переменной, в которой будет храниться импорт. Здесь это `timer`, но вы можете использовать любое имя. Возможность выбора имени может помочь избежать конфликтов имен, особенно когда вы работаете со многими модулями.
- После ключевого слова `from` следует спецификатор модуля. Вот он, `"timer"`.

Спецификатор модуля, который, как и «`timer`», не является путем, называется спецификатором голого модуля. Для встроенного JavaScript они более распространены; на самом деле, в этой книге используются только голые спецификаторы модулей. Одна из причин этого заключается в том, что во встроенном устройстве часто нет файловой системы для разрешения пути. В отличие от этого, JavaScript в Интернете в настоящее время использует пути только для спецификаторов модулей, поэтому там вы увидите операторы импорта с предложением `from`, например `from "/modules/timer.js"`.

Проиллюстрированная ранее форма оператора импорта импортирует экспорт модуля таймера по умолчанию. Каждый модуль имеет экспорт по умолчанию. Некоторые модули имеют дополнительные экспорты; например, модуль `http`, использованный в главе 3, экспортирует как класс `Request`, так и класс `Server`. В листинге 2-62 показаны различные способы импорта этих нестандартных экспортов из `http`.

Листинг 2-62.

```
import {Server} from "http";    // только сервер
new Server;

import {Request} from "http";   // только клиент
new Request;

import {Server, Request} from "http";
new Server;
new Request;
```

Если вы предпочитаете удобочитаемость, вы можете использовать один и тот же спецификатор модуля в нескольких операторах импорта:

Листинг 2-63.

```
import {Server as HTTPServer, Request as HTTPClient} from "http";
new HTTPServer;
new HTTPClient;
```

```
new Request;      // ошибка, запрос не определен
new Server;       // ошибка, сервер не определен
```

Если вы предпочитаете удобочитаемость, вы можете использовать один и тот же спецификатор модуля в нескольких операторах импорта:

```
import {Server as HTTPServer} from "http";
import {Request as HTTPClient} from "http";
```

Вы также можете импортировать все экспорты из модуля. Когда вы делаете это, вы назначаете импорт объекту. Избегая конфликтов имен, эта функция JavaScript служит той же цели, что и пространства имен в C++.

```
import * as HTTP from "http";
new HTTP.Server;
new HTTP.Request;
```

Как только вы импортируете класс из модуля, вы можете использовать его как класс, объявленный в том же исходном файле, или как встроенный класс JavaScript. Как вы видели, вы можете создать экземпляр класса с помощью оператора `new`. Вы также можете создавать подклассы импортированного класса:

```
import {Request} from "http";
class MyRequest extends Request {
    ...
}
```

Экспорт из модулей

Когда вы начнете писать свои собственные классы, вы должны упаковать их в свои собственные модули; эти модули должны экспортировать свои классы, чтобы их можно было использовать в другом коде (также можно экспортировать функции и значения). В следующей строке оператор экспорта используется для предоставления класса `Bulb` в качестве экспортируемого модуля по умолчанию:

```
export default Bulb;
```

При желании вы можете поместить оператор экспорта перед объявлением класса, что здесь означает, что вы можете комбинировать экспорт по умолчанию с определением класса `Bulb` следующим образом:

```
export default class Bulb {
    ...
}
```

Этот подход действителен, но менее распространен. Текущие рекомендации по использованию JavaScript рекомендуют размещать все операторы импорта вместе в начале исходного файла, а все операторы экспорта — в конце, что упрощает чтение и обслуживание кода.

В следующем примере показаны два способа обеспечить экспорт `Bulb` и `DimmableBulb` не по умолчанию:

```
export {Bulb};
export {DimmableBulb};

export {Bulb, DimmableBulb};
```

Как и оператор импорта, оператор экспорта может выполнять переименование с помощью `as`. Это полезно, когда вы хотите экспортировать имя, отличное от используемого в вашей реализации.

```
export {Bulb as BULB, DimmableBulb as DIMMABLEBULB};
```

Единственный способ, которым модуль может получить доступ к содержимому другого модуля, — это его экспорт. К классам, функциям и значениям, которые не экспортируются, нельзя обращаться напрямую; они эквивалентны классам, функциям и значениям, определенным с помощью ключевого слова `static` в C и C++, но с одним важным отличием: по умолчанию в C и C++ экспортируется все, кроме объявленного статического, тогда как в JavaScript ничего не экспортируется, кроме как указано заявление об экспорте. Подход JavaScript — белый список вместо черного списка экспортов, как в C — помогает с безопасностью и удобством сопровождения, избегая непреднамеренного экспорта.

Модули ECMAScript и модули CommonJS

Модули, используемые в этой книге, являются частью спецификации языка JavaScript. Их иногда называют модулями ECMAScript или ESM. До того, как модули были добавлены в официальную спецификацию, в некоторых средах, особенно в Node.js, использовалась модульная система CommonJS. Из-за этой истории вы все еще можете видеть документацию и модули CommonJS. Однако они не работают на хостах, используемых в этой книге, и большинство сред (включая Node.js) переходят на стандартные модули JavaScript.

Globals

Подобно C и C++, JavaScript имеет глобальные переменные. Вы уже использовали некоторые из них, такие как `Object`, `Array` и `ArrayBuffer`. Эти встроенные классы назначаются глобальным переменным с тем же именем, что и класс. Вы получаете доступ к этим глобальным переменным, просто используя их имя. Если это имя не находится в текущей области, используется глобальная переменная. Если глобальная переменная с таким именем недоступна, генерируется ошибка. Это похоже на ошибку ссылки, возникающую при доступе к несуществующему глобальному объекту в C.

```
function example() {
    let a = Date;          // ОК, дата встроена
    let b = DateTime;      // ошибка, DateTime не определен
}
```

В С и С++ вы создаете глобальную переменную, объявляя переменную в области верхнего уровня файла исходного кода. Если переменная не помечена как статическая, она видна всему коду, статически связанному с этим файлом. В JavaScript вы должны четко указать создание глобальной переменной. Чтобы определить новую глобальную переменную в JavaScript, вы добавляете ее в объект с именем `globalThis`. Следующая строка создает глобальную переменную с именем `AppName` и устанавливает ее начальное значение:

```
globalThis.AppName = "light bulb";
```

Как только глобальная переменная определена, вы можете получить к ней доступ либо неявно, указав только ее имя, либо явно, прочитав свойство из `globalThis`:

```
AppName = "Light Bulb";
globalThis.AppName += " App";
```

Если вы хотите узнать, определена ли уже конкретная глобальная переменная, используйте ключевое слово `in`, представленное в разделе «Objects» этой главы.

```
if ("AppName" in globalThis)
    trace(`AppName is ${AppName}\n`);
else
    trace("AppName not available");
```

Точно так же, как вы удаляете свойства объекта с помощью оператора удаления, вы можете удалять глобальные переменные:

```
delete globalThis.AppName;
```

Обратите внимание, что исходное имя объекта `globalThis` было глобальным, и его легче запомнить и ввести; он был изменен по соображениям совместимости. Некоторые среды поддерживают `global` как псевдоним для `globalThis`.

Когда вы работаете с модулями, может показаться, что они имеют глобальные переменные. Рассмотрим пример модуля в листинге 2-64.

Листинг 2-64.

```
let counter = 0;

function count() {
    return ++counter;
}

export default count;
```

То, как переменная-счетчик объявляется в области видимости верхнего уровня, похоже на объявление глобальной переменной в C или C++, но это не так. Переменная счетчика является частной для модуля, поскольку она не экспортируется явно. Такие переменные являются локальными для модуля. В C или C++ аналогичный результат достигается, если перед объявлением переменной ставится `static`, чтобы ограничить ее видимость текущим файлом исходного кода.

Arrays (Массивы)

В C и C++ любой указатель на тип (такой как `char *`) или на структуру (такую как `struct DataRecord *`) может обращаться к одному элементу (как в `*ptr`) или массиву (как в `ptr[0]`, `ptr[1]`). Такое использование указателя может привести к ошибкам, например к записи в индекс за пределами конца памяти, зарезервированной для массива. Чтобы избежать некоторых опасностей работы с массивами, C++ предоставляет шаблон класса `std::array`, который также предоставляет итераторы и другие распространенные вспомогательные функции. Встроенный в JavaScript объект `Array` больше похож на `std::array` в C++, поскольку он спроектирован так, чтобы быть безопасным, и предоставляет множество вспомогательных функций для выполнения общих операций.

В отличие от C и C++, JavaScript не устанавливает число элементов на постоянной основе при создании экземпляра массива; массивы могут содержать переменное количество элементов. Вы можете дополнительно указать количество элементов при вызове конструктора.

```
let a = new Array;          // пустой массив
let b = new Array(10);     // массив из 10 элементов
```

Как вы могли догадаться, все элементы массива инициализируются как `undefined`. Обратите внимание, что при создании массива не указывается тип данных, которые он будет содержать. Это потому, что каждый элемент массива может содержать любое значение; значения не обязательно должны быть одного типа.

Сокращение массива

Поскольку создание массивов так распространено, JavaScript предлагает более короткий путь:

```
let a = []; // пустой массив
```

Используя этот сокращенный синтаксис, вы можете указать начальные значения массива:

```
let a = [0, 1, 2];
let b = [undefined, 1, "two", {three: 3}, [4]];
```

Доступ к элементам массива

Для доступа к элементам массива используется тот же синтаксис, что и в C и C++. Элементы нумеруются, начиная с 0.

```
let a = [0, 1, 2];
a[0] += 1;
trace(a[0]);
a[1] = a[2];
```

Чтение значения массива за пределами конца массива возвращает неопределенное значение. Запись значения за конец массива создает значение, увеличивая длину массива.

```
let sparse = [0, 1, 2];  
sparse[3] = 3;  
sparse[1_000_000] = "big array";
```

Вы можете ожидать, что это присвоение миллионному элементу массива не удастся выполнить на микроконтроллере с ограниченной памятью: у ESP8266 всего около 64 КБ ОЗУ, так как же он может хранить массив с миллионом элементов? Тем не менее, назначение выполняется успешно, и доступ к `sparse[1_000_000]` возвращает «большой массив». Как это работает?

Массив в JavaScript может быть разреженным, что означает, что не все элементы должны присутствовать. Любые отсутствующие элементы имеют значение `undefined`. В данном случае с разреженным массивом имеется только пять элементов, которые как раз имеют индексы 0, 1, 2, 3 и 1 000 000.

Массивы имеют свойство длины, которое указывает количество элементов в массиве. Длина используется, например, для перебора элементов в массиве. Для разреженного массива длина — это не количество элементов с присвоенным значением, а на 1 больше, чем самый высокий индекс, которому присвоено значение. В данном случае с разреженным массивом длина равна 1 000 001, несмотря на то, что имеется только пять элементов с присвоенными значениями.

Установка свойства `length` изменяет массив. Установка меньшего значения усекает массив. Следующее усекает предыдущий разреженный массив до четырех элементов:

```
sparse.length = 4; // [0, 1, 2, 3]
```

Установка для свойства длины массива большего значения не изменяет содержимое массива.

Итерация по массивам

Как показано в листинге 2-65, вы можете использовать свойство `length` для перебора элементов массива с помощью цикла `for`, как в C и C++.

Листинг 2-65.

```
let a = [0, 1, 2, 3, 4, 5];
let total = 0;
for (let i = 0; i < a.length; i++)
    total += a[i];
```

Вместо использования цикла `for` в стиле C вы можете использовать цикл `for-of` в JavaScript.

```
for (let value of a)
    total += value;
```

Подход цикла `for-of` является более компактным, исключая код для управления значением `i` и поиск значения в массиве `a[i]`. И цикл `for` в стиле C, и цикл `for-in` перебирают все значения от индекса 0 до длины массива, даже для разреженных массивов, в которых есть неназначенные значения. Поскольку неназначенные значения имеют значение `undefined`, значение `total` имеет значение `NaN` в конце кода в листинге 2-66.

Листинг 2-66.

```
let a = [0, 1, 2, 3, 4, 5];
a[1_000_000] = 6;
let total = 0;
for (let i in a)
    total += a[i];
```

Вы можете изменить этот код, чтобы игнорировать элементы массива со значением `undefined`, следующим образом:

```
for (let i in a)
    total += (undefined === a[i]) ? 0 : a[i];
```

Альтернативное решение состоит в переборе значений в массиве с использованием цикла `for-of`, который включает только элементы массива, которым присвоено значение, как показано в листинге 2-67; значение `total` в конце этого кода равно 21, а не NaN, как в листинге 2-66.

Листинг 2-67.

```
let a = [0, 1, 2, 3, 4, 5];
a[1_000_000] = 6;
let total = 0;
for (let value of a)
    total += value;
```

Объект `Array` также имеет методы для перебора массива различными способами, каждый из которых использует функцию обратного вызова. Метод `forEach` аналогичен циклу `for-in` (см. листинг 2.68). Как и цикл `for-of`, этот метод пропускает элементы массива, которым не присвоено значение.

Листинг 2-68.

```
let a = [0, 1, 2, 3, 4, 5];
let total = 0;
a.forEach(function(value) {
    total += value;
});
```

Использование стрелочных функций сокращает код итерации до одной строки:

```
let a = [0, 1, 2, 3, 4, 5];  
let total = 0;  
a.forEach(value => total += value);
```

Как вы могли догадаться, не все из множества способов перебора массива в JavaScript одинаково эффективны. Подход `forEach`, например, является наиболее компактным кодом, но требует вызова функции для каждого элемента, что может увеличить нагрузку. Для небольших массивов используйте наиболее удобный подход; для больших массивов может быть целесообразно измерить производительность различных подходов, чтобы найти самый быстрый.

Метод карты полезен, когда вам нужно выполнить операцию над каждым элементом массива. Он вызывает обратный вызов для каждого элемента и возвращает новый массив, содержащий результаты. В следующем примере создается массив, содержащий квадрат значений исходного массива. Функция `arrow`, вызываемая для каждого элемента, использует оператор возведения в степень (`**`) для вычисления квадрата.

```
let a = [-2, -1, 0, 1, 2];  
let b = a.map(value => value ** 2); // [4, 1, 0, 1, 4]
```

Добавление и удаление элементов массива

Поскольку массивы JavaScript не имеют фиксированной длины, их можно использовать не только в виде простого упорядоченного списка. Функции `push` и `pop` позволяют использовать массив в качестве стека (последний вошел, первый вышел), как показано в листинге 2.69.

Листинг 2-69.

```
let stack = [];  
stack.push("a");  
stack.push("b");  
stack.push("c");
```

```
let c = stack.pop();    // "c"
stack.push("d");
let d = stack.pop();    // "d"
let b = stack.pop();    // "b"
```

С функциями `unshift` и `pop` вы можете использовать массив в качестве очереди (первым пришел, первым вышел). Функция `unshift` добавляет значения в начало массива; см. листинг 2.70. (Есть также сдвиг, который удаляет первый элемент из массива.)

Листинг 2-70.

```
let queue = [];
queue.unshift("first");
queue.unshift("second");

let a = queue.pop();    // "первая"
queue.unshift("third");
let b = queue.pop();    // "вторая"
```

Использование `unshift` и `pop` для добавления и удаления элементов очереди полезно, но не совсем интуитивно понятно. Эти функции было бы проще использовать, если бы у них были имена, более понятные для очереди; вы можете сделать это, создав подкласс `Array`, как показано в листинге 2-71.

Листинг 2-71.

```
class Queue extends Array {
  add(element) {
    this.unshift(element);
  }
  remove(element) {
    return this.pop();
  }
}
```

```

let queue = new Queue;
queue.add("first");
queue.add("second");
let a = queue.remove(); // "первая"
queue.add("third");
let b = queue.remove(); // "вторая"

```

Чтобы извлечь часть массива в другой массив, используйте функцию `slice`. Как и при использовании `slice` для извлечения частей строк, она принимает два аргумента: начальный и конечный индексы (где конечный индекс — это индекс, перед которым нужно закончить извлечение). Если конечный индекс опущен, используется длина строки. Функция `slice` никогда не изменяет содержимое массива, над которым она работает.

```

let a = [0, 1, 2, 3, 4, 5];
let b = a.slice(0, 2); // [0, 1]
let c = a.slice(2, 4); // [2, 3]

```

Чтобы удалить часть массива, используйте `splice`. Имя `splice` очень похоже на `slice`, и они работают одинаково: они принимают одни и те же аргументы, и обе функции возвращают массив, содержащий раздел массива, идентифицированный аргументами. Однако объединение также удаляет элементы из исходного массива.

```

let a = [0, 1, 2, 3, 4, 5];
let b = a.splice(0, 2); // [0, 1]
let c = a.splice(0, 2); // [2, 3]
// a = [4, 5] here

```

Поиск массивов

Поиск определенного значения в массиве является распространенным явлением, и есть несколько функций, помогающих в этом. Как показано в листинге 2-72, вы можете использовать `indexOf` для поиска с начала массива или `lastIndexOf` для поиска с конца.

Первый параметр — это значение для поиска; необязательный второй параметр указывает индекс, с которого следует начать поиск в массиве. Если значение не найдено, обе функции возвращают -1.

Листинг 2-72.

```
let a = [0, 1, 2, 3, 2, 1, 0];
let b = a.indexOf(1);           // 1
let c = a.lastIndexOf(1);      // 5
let d = a.indexOf(1, 3);       // 5
let e = a.lastIndexOf(1, 3);   // 1
let f = a.indexOf("one");      // -1
```

Функции `indexOf` и `lastIndexOf` используют оператор строгого равенства, чтобы проверить, найдено ли совпадение. Если вы хотите применить другой тест, используйте функцию `findIndex`, которая вызывает функцию обратного вызова для проверки соответствия. В следующем примере выполняется поиск без учета регистра:

```
let a = ["Zero", "One", "Two"];
let search = "one";
let b = a.findIndex(value =>
    value.toLowerCase() === search); // 1
```

Сортировка массивов

Сортировка — еще одна распространенная операция над массивами. Функция `sort` для массивов похожа на функцию `qsort` в С и С++, хотя может быть реализована с использованием другого алгоритма сортировки. Подобно `qsort`, сортировка в JavaScript работает на месте, поэтому новый массив не создается. По умолчанию встроенная функция сортировки сравнивает значения массива как строки.

```
let a = ["Zero", "One", "Two"];
a.sort();
// ["One", "Two", "Zero"]
```


Чтобы реализовать другие варианты поведения, вы предоставляете функцию обратного вызова для выполнения сравнения. Сравнение аналогично функции обратного вызова в функции `qsort` C и C++, которая получает два значения для сравнения и возвращает отрицательное число, 0 или положительное число, в зависимости от результата сравнения. Например, следующий код сортирует массив чисел:

```
let a = [0, 1, 2, 3, 2, 1, 0];
a.sort((x, y) => x - y);
// [0, 0, 1, 1, 2, 2, 3]
```

Пример в листинге 2-73 использует более сложную функцию сравнения для выполнения сортировки строк без учета регистра.

Листинг 2-73.

```
let a = ["Zero", "zero", "two", "Two"];
a.sort();
// ["Two", "Zero", "two", "zero"]
a.sort((x, y) => {
    x = x.toLowerCase();
    y = y.toLowerCase();
    if (x > y)
        return +1;
    if (x < y)
        return -1;
    return 0;
});
// ["Two", "two", "Zero", "zero"]
```

Двоичные данные

JavaScript не всегда поддерживал двоичные данные, в отличие от C, который с самого начала поддерживал буферы памяти, содержащие собственные целочисленные типы. В C одной из первых вещей, которую вы изучаете, является то, как выделять память с помощью `malloc` и как заполнять эту память массивами и другими структурами данных. Возможность работать с буферами памяти напрямую важна для многих видов разработки встраиваемых систем, например, при работе с двоичными сообщениями в различных сетевых и аппаратных протоколах. JavaScript поддерживает те же виды операций, к которым вы привыкли при написании кода на C и C++, хотя способ выполнения этих операций совершенно другой.

Еще одним преимуществом использования двоичных данных в JavaScript является то, что они могут уменьшить использование памяти вашим проектом. Одной из фундаментальных характеристик JavaScript является то, что любое значение может содержать любой тип, но эта мощная функция имеет свою цену: для каждого значения требуется дополнительная память для хранения типа значения. В то время как логическое значение в C составляет всего один байт (или бит, используя битовые поля), логическое значение в JavaScript может быть намного больше — например, 8 или 16 байтов. Используя двоичные данные в JavaScript, вы можете хранить логическое значение в байтах (или даже в битах), приложив немного усилий. Если ваш проект поддерживает большой объем данных в памяти, рассмотрите возможность использования функций двоичных данных JavaScript, так как экономия памяти может быть значительной. Для создания массива из 1000 логических элементов JavaScript с использованием стандартного объекта `Array` может потребоваться 16 КБ ОЗУ, больше, чем может быть свободно на ESP8266, но для его создания с использованием объекта `Uint8Array` требуется всего 1 КБ ОЗУ — точно так же, как в C.

ArrayBuffer

Эквивалентом `calloc` в JavaScript является класс `ArrayBuffer`. `ArrayBuffer` — это блок памяти с фиксированным количеством байтов. Память изначально установлена на 0, чтобы избежать неожиданностей с неинициализированной памятью.

```
let a = new ArrayBuffer(10);    // 10 байт
```

Если буфер не может быть выделен из-за недостатка свободной памяти, конструктор `ArrayBuffer` выдает исключение.

Чтобы получить количество байтов в `ArrayBuffer`, получите свойство `byteLength`. Количество байтов, содержащихся в экземпляре `ArrayBuffer`, фиксируется на момент его создания. Нет эквивалента `realloc`; вы не можете установить свойство `byteLength` для `ArrayBuffer`.

```
let a = new ArrayBuffer(16);
let b = a.byteLength;    // 16
```

```
a.byteLength = 20;      // сгенерировано исключение
```

Как и в случае с массивом, вы используете метод `slice` для извлечения части буфера в новый экземпляр `ArrayBuffer`:

```
let a = new ArrayBuffer(16);
let b = a.slice(0, 8);      // копируем первую половину
let c = a.slice(8, 16);    // копируем вторую половину
let d = a.slice(0);        // клонировать весь буфер
```

Вы можете ожидать, что сможете получить доступ к содержимому `ArrayBuffer`, используя синтаксис массива (например, `a[0]`), но это не так. `ArrayBuffer` — это всего лишь буфер байтов. Поскольку с данными не связан тип, JavaScript не знает, как интерпретировать байты — например, являются ли значения байтов знаковыми или беззнаковыми. Чтобы получить доступ к данным в `ArrayBuffer`, вы заключаете их в представление. В следующих разделах представлены два вида представления: типизированный массив и представление данных.

Типизированные массивы

Типизированные массивы JavaScript — это набор классов, которые позволяют работать с массивами целых чисел и значений с плавающей запятой, хранящихся в `ArrayBuffer`. Вы работаете не с классом `TypedArray` напрямую, а с его подклассами для конкретных типов, таких как `Int8Array`, `Uint16Array` и `Float32Array`. Использование типизированного массива похоже на создание буфера памяти в C с помощью `calloc` и присвоение результата указателю на целочисленный тип или тип с плавающей запятой.

Вы можете создать типизированный массив, который обертывает существующий `ArrayBuffer`. В следующем примере `ArrayBuffer` помещается в `Uint8Array`:

```
let a = new ArrayBuffer(16);
let b = new Uint8Array(a);
```

Теперь, когда у вас есть представление о буфере, вы можете получить доступ к содержимому, используя синтаксис скобок массива, как и ожидалось:

```
b[0] = 12;
b[1] += b[0];
```

Типизированные массивы, такие как `Uint8Array` в предыдущем примере, имеют свойство `byteLength`, как и `ArrayBuffer`, но у них также есть свойство `length`, указывающее количество элементов в массиве. Когда элементами являются байты, эти два значения равны, но для больших типов они различаются (см. листинг 2.74).

Листинг 2-74.

```
let a = new ArrayBuffer(24);
let b = new Uint8Array(a);
let c = new Uint16Array(a);
let d = new Uint32Array(a);
let e = b.length;    // 24
let f = c.length;    // 12
let g = d.length;    // 6
```

Здесь один `ArrayBuffer` обернут несколькими представлениями; это разрешено. В C это называется «алиасинг» и опасно, потому что мешает некоторым оптимизациям компилятора. В JavaScript это безопасно, хотя вы должны использовать его с осторожностью, чтобы избежать неожиданных сюрпризов при чтении и записи в перекрывающихся представлениях.

Вы можете создать представление типизированного массива, которое ссылается на подмножество буфера, включив смещение в байтах до начала представления и количество элементов в представлении. Это похоже на присвоение целочисленному указателю значения в середине буфера памяти. В JavaScript, однако, нет непредсказуемого результата, когда вы читаете дальше конца буфера; который всегда возвращает `undefined` (см. листинг 2-75).

Листинг 2-75.

```
let a = new ArrayBuffer(18)
let b = new Int16Array(a);
b[0] = 0;
b[1] = 1;
b[2] = 2;
b[3] = 3;
let c = new Int16Array(a, 6, 1);
    // c начинается с 6 байтов в a и имеет один элемент
let d = c[0];    // 3
let e = c[1];    // undefined (чтение после конца представления)
```

Представление `Int16Array`, созданное в листинге 2-75 для переменной `c`, начинается со смещения 6, но оно может начинаться с любого смещения, включая нечетное. Доступ к 16-битным значениям в этом массиве требует неправильного чтения. Не все микроконтроллеры поддерживают операции чтения и записи со смещением; ESP8266 — это один из микроконтроллеров, который не поддерживает несогласованный доступ к памяти. Когда код C выполняет неправильное чтение или запись, генерируется аппаратное исключение, вызывающее сброс микроконтроллера. Код JavaScript не имеет этой проблемы, потому что язык гарантирует, что неправильно выровненные операции дадут тот же результат, что и выровненные операции — еще один способ, которым JavaScript немного упрощает кодирование встроенных продуктов.

Сокращенная запись типизированного массива

Обычно создаются небольшие целочисленные массивы. В С и С++ вы можете легко объявлять статические массивы в стеке.

```
static uint16_t values[] = {0, 1, 2, 3};
```

В JavaScript вы можете добиться того же результата, используя статический метод `of` для типизированных массивов:

```
let a = Uint16Array.of(0, 1, 2, 3);  
let b = a.byteLength;    // 8  
let c = a.length;        // 4
```

Функция `of` автоматически создает `ArrayBuffer` размера, необходимого для хранения значений. Вы можете получить доступ к `ArrayBuffer`, созданному с помощью `of`, получив свойство буфера типизированного массива. Этот буфер может использоваться с другими представлениями, такими как представления данных.

```
let a = Uint16Array.of(0, 1, 2, 3);  
let b = a.buffer;  
let c = b.byteLength;    // 8
```

Копирование типизированных массивов

В С и С++ вы используете `memcpy` и `memmove` для копирования значений данных в пределах одного буфера или между двумя буферами. Вы уже видели, как использовать `slice` для `ArrayBuffer` в JavaScript для копирования части или всего буфера в новый буфер; вы можете использовать `copyWithin` для копирования значений в одном буфере и `set` для копирования значений из одного буфера в другой. В С вам нужно проявлять особую осторожность при копировании в пределах одного буфера, когда источник и место назначения перекрываются, тогда как метод `copyWithin` в JavaScript гарантирует предсказуемость и правильность результатов. Первый аргумент для `copyWithin` — это целевой индекс, а второй и третий аргументы — это начальный и конечный исходные индексы для копирования (где конечный индекс — это индекс, перед которым нужно закончить).

```
let a = Uint16Array.of(0, 1, 2, 3, 4, 5, 6);
a.copyWithin(4, 1, 3);
// [0, 1, 2, 3, 1, 2, 6]
```

Метод `set` записывает один типизированный массив в другой. Первый аргумент — это исходные данные для записи, а второй аргумент — это индекс, с которого нужно начать запись данных.

```
let a = Int16Array.of(0, 1, 2, 3, 4, 5, 6);
let b = Int16Array.of(-2, -3);
a.set(b, 2);
// [0, 1, -2, -3, 4, 5, 6]
```

Чтобы записать только подмножество исходных данных, вам нужно создать другое представление. Для этого удобен метод подмассива, как показано в листинге 2-76. Учитывая начальный и конечный индексы типизированного массива, `subarray` возвращает новый типизированный массив, который ссылается только на эти индексы. Обратите внимание, что подмассив не выделяет новый `ArrayBuffer`; он просто ссылается на тот же `ArrayBuffer`.

Листинг 2-76.

```
let a = Int16Array.of(0, 1, 2, 3, 4, 5, 6);
let b = Int16Array.of(0, -1, -2, -3, -4, -5, -6);
let c = b.subarray(2, 4);
a.set(c, 2);
// [0, 1, -2, -3, 4, 5, 6]
```

Вы можете использовать срез вместо подмассива, чтобы скопировать подмножество в новый `Int16Array`, но это временно использует дополнительную память, поэтому в этом случае предпочтительнее использовать подмассив.

Классы `TypedArray` не являются подклассами `Array`; это полностью независимые классы, но они предназначены для использования общих API. Например, метод `copyWithin`, о котором вы узнали для типизированных массивов, доступен в `Array`. Точно так же многие методы `Array`, включая `map`, `forEach`, `indexOf`, `lastIndexOf`, `findIndex` и `sort`, также доступны для типизированных массивов.

Заполнение типизированных массивов

Другой полезный метод, доступный как для Array, так и для TypedArray, — это fill, который похож на memset в С и С++. Но в то время как memset работает только с байтовыми значениями, fill работает со значениями типа типизированного массива. Как показано в листинге 2-77, первый аргумент для заполнения — это значение, которое нужно присвоить, а необязательные второй и третий аргументы — это начальный и конечный индексы для заполнения (где конечный индекс — это индекс, перед которым нужно закончить заполнение). Если необязательные аргументы не указаны, заполняется весь массив.

Листинг 2-77.

```
let a = new Uint16Array(4);
a.fill(0x1234);
// [0x1234, 0x1234, 0x1234, 0x1234]

a.fill(0, 1, 3);
// [0x1234, 0, 0, 0x1234]

let b = new Uint32Array(2);
b.fill(0x12345678);
// [0x12345678, 0x12345678]
```

Запись значений типизированного массива

Запись значений в типизированный массив обычно происходит так же, как и в С. Например, если вы записываете 16-битное значение в 8-битный типизированный массив, используются младшие 8 бит (см. листинг 2.78).

Листинг 2-78.

```
let a = new Uint32Array(1);
a[0] = 0x12345678; // 0x12345678

let b = new Uint16Array(1);
b[0] = 0x12345678; // 0x5678
```



```
let c = new Uint8Array(1);  
c[0] = 0x12345678; // 0x78
```

В JavaScript также есть `Uint8ClampedArray`, который реализует другое поведение: вместо того, чтобы брать младшие значащие биты, он прикрепляет входное значение к значению между 0 и максимальным значением, которое может хранить экземпляр типизированного массива.

```
let a = new Uint8ClampedArray(1);  
a[0] = 5; // 5  
a[0] = 256; // 255  
a[0] = -1; // 0
```

Типизированные массивы с плавающей запятой

Существует два массива с плавающей запятой: `Float32Array` и `Float64Array`. Поскольку числовые значения в JavaScript представляют собой 64-битные числа с плавающей запятой IEEE 754, `Float64Array` может хранить эти значения без потери точности. `Float32Array` уменьшает точность и диапазон значений, которые могут быть сохранены, но этого достаточно для некоторых ситуаций.

Примечание Классы типизированных массивов не гарантируют порядок следования байтов при хранении значений (т. Реализация движка JAVAScRIPT может хранить значения любым способом по своему выбору до тех пор, пока сохраняется точность значения. Обычно он хранит их в том же порядке, что и основной микроконтроллер, для максимальной эффективности. Для управления порядком байтов значений используйте представление данных (обсуждается далее).

Представления данных

Класс `DataView` предоставляет другой вид представления `ArrayBuffer`. В отличие от типизированных массивов, в которых все значения имеют один и тот же тип, представления данных используются для чтения и записи в буфер целых чисел разного размера и значений с плавающей запятой. Вы можете использовать `DataView` для доступа к двоичным данным, которые соответствуют структуре C или C++, содержащей значения разных типов.

Вы создаете экземпляр представления данных, передавая конструктору `DataView` `ArrayBuffer` для переноса представления точно так же, как вы можете передать `ArrayBuffer` конструктору типизированного массива:

```
let a = new ArrayBuffer(16);
let b = new DataView(a);
```

Также, как и в случае с типизированными массивами, вы можете передать смещение и размер конструктору `DataView`, чтобы ограничить представление подмножеством всего буфера. Эта возможность полезна для доступа к структурам данных, внедренным в больший буфер памяти.

```
let a = new ArrayBuffer(16);
let b = new DataView(a, 4, 12);
// b может получить доступ только к байтам с 4 по 12
```

Доступ к значениям представления данных

Экземпляр `DataView` может получать и устанавливать все те же типы, что и типизированный массив, как показано в листинге 2-79. Все методы получения и установки имеют смещение в представлении в качестве первого аргумента. Второй аргумент методов установки указывает устанавливаемое значение.

Листинг 2-79.

```
let a = new DataView(new ArrayBuffer(8));
a.setUint8(0, 0);
a.setUint8(1, 1);
```

```
a.setUint16(2, 0x1234);
a.setUint32(4, 0x01020304);
```

Поскольку методы DataView по умолчанию записывают многобайтовые значения в порядке байтов с обратным порядком байтов, буфер a содержит следующие шестнадцатеричные байты после выполнения примера в листинге 2-79:

```
00 01 12 34 01 02 03 04
```

Вы считываете значения обратно, используя соответствующие методы получения. В следующем примере предполагается экземпляр DataView, показанный ранее:

```
let b = a.getUint8(0);      // 0
let c = a.getUint8(1);      // 1
let d = a.getUint16(2);     // 0x1234
let e = a.getUint32(4);     // 0x01020304
```

Методы DataView имеют необязательный конечный параметр для управления порядком байтов. Если параметр опущен или имеет значение false, порядок байтов — bigendian; если правда, то обратный порядок байтов (см. листинг 2-80).

Листинг 2-80.

```
let a = new DataView(new ArrayBuffer(8));
a.setUint8(0, 0);
a.setUint8(1, 1);
a.setUint16(2, 0x1234, true);
a.setUint32(4, 0x01020304, true);
```

Поскольку setUint8 записывает однобайтовое значение, порядок байтов отсутствует, поэтому третий параметр не нужен. Вызовы setUint16 и setUint32 в листинге 2-80 устанавливают для параметра порядка байтов значение true, поэтому на выходе используется обратный порядок байтов.

```
00 01 34 12 04 03 02 01
```

Чтобы прочитать значения, хранящиеся в порядке с прямым порядком байтов, передайте `true` в качестве последнего параметра методам получения:

```
let b = a.getUint16(2, true); // 0x1234 (с прямым порядком байтов)
let c = a.getUint16(2);      // 0x3412 (обратный порядок байтов)
```

Класс `DataView` включает методы получения и установки, соответствующие всем типам, доступным в `TypedArray`: `Int8`, `Int16`, `Int32`, `Uint8`, `Uint16`, `Uint32`, `Float32` и `Float64`.

Класс `DataView` — очень гибкий способ манипулирования двоичными структурами данных, но его код не особенно удобочитаем. Вместо того, чтобы писать `a.value` для доступа к полю, как в C, вы должны написать что-то вроде `a.getUint16(6, true)`. Один из способов улучшить читаемость и уменьшить вероятность ошибок — создать подкласс `DataView` для структуры данных. Представьте, что у вас есть структура данных C, показанная в листинге 2-81, для заголовка сетевого пакета, который вы хотите использовать из JavaScript. Для простоты предположим, что между полями нет отступов.

Листинг 2-81.

```
typedef struct Header {
    uint8_t    kind;
    uint8_t    priority;

    uint16_t   sequenceNumber;
    uint32_t   value;
}
```

Класс заголовка JavaScript в листинге 2-82 является подклассом `DataView` для реализации простого доступа к структуре заголовка C. Поскольку сетевые пакеты обычно используют порядок байтов с обратным порядком байтов, многобайтовые значения записываются в порядке байтов с обратным порядком байтов.

Листинг 2-82.

```

class Header extends DataView {
    constructor(buffer = new ArrayBuffer(8)) {
        super(buffer);
    }
    get kind() {return this.getUint8(0);}
    set kind(value) {this.setUint8(0, value);}
    get priority() {return this.getUint8(1);}
    set priority(value) {this.setUint8(1, value);}
    get sequenceNumber() {return this.getUint16(2);}
    set sequenceNumber(value) {this.setUint16(2, value);}
    get value() {return this.getUint32(4);}
    set value(value) {this.setUint32(4, value);}
}

```

Поскольку в классе используются геттеры и сеттеры, результирующий код для пользователей класса аналогичен коду C. Пример в листинге 2-83 использует класс Header для чтения значений из пакета, полученного в переменной **p**.

Листинг 2-83.

```

let a = new Header(p);
let b = a.kind;
let c = a.priority;
let d = a.sequenceNumber;
let e = a.value;

```

В листинге 2-84 создается новый пакет, инициализируются значения и вызывается функция `asend` для передачи буфера `ArrayBuffer`, используемого экземпляром заголовка для хранения.

Листинг 2-84.

```

let a = new Header;
a.kind = 1;
a.priority = 2;
a.sequenceNumber = 3;
a.value = 4;
send(a.buffer);

```

Как видите, определение класса, представляющего двоичную структуру данных, делает код, работающий с этой структурой данных, более понятным. Работа с двоичными данными — это одна из областей, где C имеет преимущество в компактности кода; тем не менее, можно добиться того же результата с помощью читаемого кода в JavaScript. Здесь у JavaScript тоже есть свои преимущества: учтите, что код, который считывает данные, полученные по сети, часто бывает ненадежным. В этом примере, если полученный пакет имеет только четыре байта вместо необходимых восьми байтов, чтение поля значения имеет неопределенный результат, что может привести к утечке личных данных или даже к сбою. Если такая ситуация возникает в JavaScript, попытка чтения значения с помощью `getUint32` завершится ошибкой с исключением, поскольку чтение выходит за пределы допустимого диапазона.

Управление памятью

Управление памятью — это то место, где JavaScript значительно отличается от C и C++. В C и C++ вы явно выделяете память с помощью `malloc`, `calloc` и `realloc` и освобождаете ее с помощью `free`. Эти функции выделения и освобождения памяти находятся не в самом языке, а в стандартной библиотеке. В C++ вы также выделяете память, когда создаете экземпляр класса с помощью `new`, и освобождаете эту память, когда используете `delete` для вызова деструктора класса.

JavaScript встраивает управление памятью в язык. Когда вы создаете объект, строку, `ArrayBuffer` или любой другой встроенный объект, требующий памяти, эта память прозрачно выделяется движком JavaScript. Как и следовало ожидать, язык также освобождает память; однако вместо того, чтобы требовать от вашего кода выполнения вызова типа `free` или

использования оператора удаления C++, JavaScript автоматически освобождает память, когда определяет, что это безопасно. Такой подход к управлению памятью реализуется с помощью сборщика мусора. В определенные моменты времени движок JavaScript запускает сборщик мусора, который сканирует всю память, выделенную движком, идентифицирует все выделения, на которые больше нет ссылок, и освобождает все блоки памяти, на которые нет ссылок.

Рассмотрим этот код:

```
let a = "this is a test";  
a = {};  
a = new ArrayBuffer(16);
```

Этот пример делает следующее:

1. Первая строка выделяет строку и присваивает ее `a`. Поскольку на строку ссылается `a`, она не может быть удалена сборщиком мусора.
2. Вторая строка присваивает пустой объект `a`, удаляя ссылку на строку. Поскольку никакие другие переменные или свойства не ссылаются на строку, она может быть удалена сборщиком мусора.
3. После назначения `ArrayBuffer` в третьей строке пустой объект становится пригодным для сборки мусора.

Язык JavaScript не определяет, когда запускается сборщик мусора. Сборщик мусора в движке XS, используемом в Moddable SDK, запускается всякий раз, когда ему не хватает памяти; это может быть никогда, один раз в час или много раз в секунду, в зависимости от выполняемого кода.

Сборщик мусора хорошо работает для управления памятью. Это уменьшает объем кода, который вам нужно написать, потому что и выделение, и освобождение происходят автоматически. Устранена ошибка забывания освободить память, приводившая к утечкам памяти; это является серьезной проблемой для встраиваемых систем, многие из которых должны работать месяцами или годами, поскольку периодически возникающая небольшая утечка памяти в конечном итоге приводит к сбою системы. Сборщик мусора также устраняет ошибку чтения памяти, которая была освобождена, поскольку память не освобождается, если код все еще может ссылаться.

При всех своих преимуществах сборщик мусора не является универсальным решением для управления ресурсами. Рассмотрим листинг 2-85, в котором файл открывается дважды, сначала в режиме записи, а затем в режиме только для чтения.

Листинг 2-85.

```
let f = new File("/foo.txt", 1);    // 1 для записи
f.write("this is a test");
f = undefined;

...

let g = new File("/foo.txt");      // только для чтения
```

Когда в этом примере `f` присваивается `undefined`, экземпляр класса `File`, соответствующий файлу, открытому для записи, имеет право на сборку мусора. В большинстве файловых систем, когда файл открывается для записи, доступ является эксклюзивным, то есть файл нельзя открыть во второй раз. Поскольку сборщик мусора может запуститься в любой момент, вызов открытия файла в режиме только для чтения может завершиться успешно, а может и не завершиться, в зависимости от того, был ли уже собран файловый объект с доступом для записи. По этой причине объекты, используемые для представления ресурсов, не связанных с памятью, таких как открытый файл, обычно предоставляют способ явного освобождения ресурса. В Moddable SDK для освобождения ресурсов используется метод `close`, аналогичный использованию оператора удаления в C++.


```
let f = new File("/foo.txt", 1);    // 1 для записи
f.write("this is a test");
f.close();
```

Вызов `close` немедленно закрывает файл. Любая дальнейшая попытка записи в экземпляр `f` потерпит неудачу. Теперь файл можно снова открыть в режиме чтения или записи.

Класс Date

Стандартная библиотека C предоставляет функции `gettimeofday` и `localtime` для определения текущей даты, времени, часового пояса и смещения летнего времени. Функция `strftime` в той же библиотеке преобразует дату и время в текстовый формат, используя строки формата. JavaScript предоставляет аналогичную функциональность во встроенном классе `Date`.

Следующий код создает экземпляр класса `Date`. Экземпляр содержит значение времени, которое инициализируется текущим временем, когда конструктор `Date` вызывается без аргументов.

```
let now = new Date;
trace(now.toString());
// Вт, 24 сентября 2019 г., 11:18:26 GMT-0700 (PDT)
```

Конструктор `Date` принимает аргументы для инициализации значения, отличного от текущего времени. Вы можете инициализировать его из строки, хотя это не рекомендуется из-за легкости ошибок в формате строки.

```
let d = new Date("Tue Sep 24 2019 11:18:26 GMT-0700 (PDT)");
```

Вместо этого вы можете передать компоненты времени (часы, минуты, год и т. д.) в качестве аргументов конструктору:

```
let d = new Date(2019, 8, 24);  
    // Сентябрь 24 2019 полночь  
let e = new Date(2019, 8, 24, 11, 18, 26);  
    // Сентябрь 24 2019 11:18:26
```

Обратите внимание, что значение сентября равно 8, а не 9, как можно было бы ожидать. Это связано с тем, что номера месяцев в JavaScript Date API начинаются с 0, а не с 1; это было решено на ранней стадии разработки JavaScript, чтобы соответствовать объекту `java.util.Date` языка Java. Также обратите внимание, что время, указанное во втором объявлении, является местным временем, а не UTC (Всемирное координированное время). Чтобы указать время UTC, используйте функцию `Date.UTC` вместе с конструктором `Date`.

```
let d = new Date(Date.UTC(2019, 8, 24));  
    //24 сентября 2019 г., полночь UTC
```

Экземпляр `Date` хранит значение времени в миллисекундах и всегда в формате UTC. Чтобы получить это значение, вызовите метод `getTime`.

```
let now = new Date;  
let utcTimeInMS = now.getTime();
```

Если ваш код должен часто получать время, предыдущий пример неэффективен, так как он создает новый экземпляр `Date` каждый раз, когда требуется текущее время. Для таких ситуаций статический метод теперь возвращает текущее время UTC в миллисекундах в виде числа.

```
let utcTimeInMS = Date.now();
```

Класс `Date` предоставляет доступ ко всем частям, из которых состоят дата и время (см. листинг 2.86).

Листинг 2-86.

```
let now = new Date;  
let ms = now.getMilliseconds();    // 0 to 999  
let seconds = now.getSeconds();    // 0 to 59  
let minutes = now.getMinutes();    // 0 to 59  
let hours = now.getHours();        // 0 to 23  
let day = now.getDay();             // 0 (Sunday) to 6 (Saturday)  
let date = now.getDate();           // 1 to 31  
let month = now.getMonth();         // 0 (January) to 11 (December)  
let year = now.getFullYear();
```

Значения, возвращаемые в листинге 2-86, представляют собой местное время с учетом часового пояса и смещения летнего времени. Также доступны версии тех же функций для значений UTC; они начинаются с `getUTC`, например, `getUTCMilliseconds`, `getUTCSeconds` и т. д.

Существуют также методы установки, соответствующие всем методам получения. Листинг 2-87 создает объект даты и изменяет его так, чтобы он был полночью следующего новогоднего дня.

Листинг 2-87.

```
let d = new Date;  
d.setMilliseconds(0);  
d.setSeconds(0);  
d.setMinutes(0);  
d.setHours(0);  
d.setDate(1);  
d.setMonth(0);  
d.setFullYear(d.getFullYear() + 1);
```

Методы `setHours` и `setFullYear` поддерживают дополнительные параметры, что позволяет записать пример в листинге 2-87 более компактно:

```
let d = new Date;  
d.setHours(0, 0, 0, 0);  
d.setFullYear(d.getFullYear() + 1, 0, 1);
```

Чтобы получить текущее смещение часового пояса от времени UTC, вызовите метод `getTimezoneOffset`. Возвращаемое значение находится в минутах и имеет текущее смещение летнего времени.

```
let timeZoneOffset = d.getTimezoneOffset();  
// timeZoneOffset = 420 (смещение в минутах от UTC)
```

Как показано ранее в этом разделе, метод `toString` объекта `Date` предоставляет строку, представляющую местное время с примененными смещениями часового пояса и перехода на летнее время. В некоторых ситуациях — например, в сети — полезно иметь текстовое представление строки во времени UTC. Используйте метод `toUTCString` для создания строки, представляющей время в формате UTC.

```
let d = new Date;  
trace(d.toUTCString());  
// "Tue, 24 Sep 2019 18:18:26 GMT"
```

Другой формат времени и даты, используемый во многих стандартах, — это ISO 8601. Метод `toISOString` предоставляет совместимую с ISO 8601 версию даты в виде строки.

```
let d = new Date;  
trace(d.toISOString());  
// "2019-09-24T18:18:26.000Z"
```

Хотя `toUTCString` и `toISOString` удобны, вы можете использовать свои знания о датах и строках JavaScript для создания строк в любом формате, необходимом вашему проекту.

Программирование, управляемое событиями

Встроенные программы, особенно те, которые работают на менее мощных устройствах, часто организованы вокруг одного цикла, который выполняется непрерывно. Перечисление 2-88 показывает тривиальный пример.

Листинг 2-88.

```
while (true) {  
    if (readButton())  
        lightOn();  
    else  
        lightOff();  
}
```

Этот стиль программирования подходит для очень простых встроенных устройств. Однако он плохо работает для более крупных систем с множеством различных входов и выходов; для таких систем предпочтительнее программирование, управляемое событиями. Программы, управляемые событиями, ожидают возникновения событий, таких как нажатие кнопки. Когда происходит событие, вызывается обратный вызов для ответа на него. JavaScript разработан для использования с программами, управляемыми событиями, потому что именно так работают веб-браузеры.

В листинге 2-89 показана управляемая событиями версия бесконечного цикла из предыдущего примера. Здесь обратный вызов `onRead` вызывается при изменении кнопки, поэтому коду не нужно постоянно опрашивать состояние кнопки.

Листинг 2-89.

```
let button = new Button;  
button.onRead = function(value) {  
    if (value)  
        lightOn();  
}
```

```
else  
    lightOff();  
}
```

Как правило, обратные вызовы, доставляющие события, вызываются только тогда, когда микроконтроллер простаивает. Когда код JavaScript выполняется, обратные вызовы откладываются до завершения кода. В листинге 2-88, поскольку цикл бесконечен, никакие обратные вызовы не могут быть вызваны. Следовательно, как правило, невозможно использовать один цикл в качестве основы для вашего приложения JavaScript; вы должны принять стиль программирования, управляемый событиями.

Если вы раньше не занимались программированием, управляемым событиями, не волнуйтесь. Все примеры в этой книге написаны для того, чтобы показать вам, как использовать встроенные API-интерфейсы JavaScript в стиле программирования, управляемом событиями. С небольшой практикой это должно стать второй натурой.

Заключение

Имея за плечами это введение в JavaScript, вы готовы двигаться дальше по этой книге. Остальные главы посвящены тому, как использовать JavaScript во встроенных системах для создания продуктов IoT с использованием функций, предоставляемых Moddable SDK.

Спецификация языка JavaScript огромна — более 750 страниц. Эта книга не может объяснить все особенности и нюансы языка, но доступно множество отличных ресурсов, которые помогут вам узнать больше. Веб-документы Mozilla MDN (developer.mozilla.org) фактически являются справочником по языку JavaScript. Он соответствует последнему стандарту, содержит множество примеров и чрезвычайно подробен. Это отличный ресурс для разработчиков встраиваемых систем, потому что многие из представленных в нем примеров можно понять, даже если вы не являетесь веб-разработчиком.

ГЛАВА 3

Сеть

Существует так много разных типов устройств IoT — от термостатов до дверных замков, от умных часов до умных лампочек, от стиральных машин до камер видеонаблюдения, — что легко забыть, что у них у всех есть что-то общее: сеть. Что отличает устройство IoT от обычного повседневного устройства, так это его подключение к сети. Эта глава посвящена этому соединению, начиная с различных способов подключения к сети.

Как только ваше устройство подключено к сети, оно может общаться разными способами. В этой главе показано, как общаться, используя тот же сетевой протокол HTTP, который используется веб-браузером на вашем компьютере и телефоне. Также показано, как использовать протокол WebSocket для интерактивной двусторонней связи и протокол MQTT, используемый для публикации и подписки.

Защита связи важна для многих продуктов, поэтому вы также узнаете, как устанавливать безопасные соединения с помощью TLS (безопасность транспортного уровня) в сочетании с такими протоколами, как HTTP, WebSocket и MQTT.

Глава завершается двумя дополнительными темами. Во-первых, как превратить ваше устройство в базовую станцию Wi-Fi — метод, используемый многими коммерческими продуктами IoT для простой настройки. Вы можете подключить свой компьютер, телефон и другие устройства к этой частной базовой станции Wi-Fi без установки какого-либо специального программного обеспечения. Вторая расширенная тема — как использовать промисы JavaScript с сетевыми API.

О сети

В этой книге основное внимание уделяется оборудованию, которое подключается к сети с помощью Wi-Fi. Ваша точка доступа Wi-Fi, также называемая базовой станцией или маршрутизатором, соединяет вашу сеть Wi-Fi с Интернетом. Точка доступа также создает локальную сеть, которая позволяет подключенным к ней устройствам взаимодействовать друг с другом. Протоколы HTTP, MQTT и WebSocket используются для связи с серверами в Интернете, но их также можно использовать для связи между устройствами в вашей локальной сети Wi-Fi. Прямое общение между устройствами происходит быстрее и может быть более конфиденциальным, поскольку ваши данные никогда не покидают вашу сеть Wi-Fi. Это устраняет стоимость облачного сервиса. Использование сетевого протокола mDNS позволяет устройствам в вашей локальной сети напрямую взаимодействовать друг с другом.

Все сетевые примеры в этой главе неблокирующие (или асинхронные). Это означает, например, что когда вы запрашиваете данные из сети по протоколу HTTP, ваше приложение продолжает работать, пока выполняется запрос. Точно так же работает сеть, когда вы используете JavaScript в Интернете, но отличается от большинства сетевых реализаций во встроенных средах. По разным причинам многие встроенные среды разработки вместо этого используют блокировку сети; это оставляет устройство невосприимчивым к пользовательскому вводу во время работы сети, если только не используется более сложный и ресурсоемкий метод, такой как потоки.

Классы в Moddable SDK, которые реализуют сетевые возможности, используют функции обратного вызова для предоставления состояния и доставки сетевых данных. Обратные вызовы просты в реализации и эффективно работают даже на оборудовании с относительно небольшой вычислительной мощностью и памятью. В Интернете разработчики уже давно используют обратные вызовы для сетевых операций. Совсем недавно функция JavaScript, называемая промисами, стала популярной альтернативой обратным вызовам в некоторых ситуациях. Поскольку промисы требуют больше ресурсов, здесь они используются экономно. Промисы поддерживаются в движке XS, на котором работает Moddable SDK. Сетевые возможности, представленные в этой главе, могут быть адаптированы для использования промисов; пример включен в раздел обещаний в конце этой главы.

Подключение к Wi-Fi

Вы уже знаете, как подключить свой компьютер и телефон (и, возможно, даже телевизор!) к Интернету, и этот опыт поможет вам при написании кода для подключения вашего устройства. Вам также нужно будет узнать несколько новых вещей, потому что устройства IoT не всегда имеют экран, а без экрана пользователь не может просто нажать на название сети Wi-Fi для подключения.

В этом разделе описываются три различных способа подключения к Wi-Fi:

- Из командной строки
- С помощью простого кода для подключения к известной точке доступа Wi-Fi
- Путем сканирования открытой точки доступа Wi-Fi.

Каждый из них оптимален для разных ситуаций в своих проектах. Использование командной строки отлично подходит для разработки, но два других подхода необходимы, когда вы переходите от экспериментов к созданию сложных прототипов и реальных продуктов.

Примечание. В этом разделе используется шаблон установки, отличный от того, который вы изучили в главе 1: вместо установки хоста с помощью `msconfig` и последующей установки примеров с помощью `msrun` вы устанавливаете примеры с помощью `msconfig`.

Подключение из командной строки

В главе 1 вы научились использовать инструмент командной строки `mcconfig` для сборки и установки хоста. Команда `mcconfig` может определять переменные. Как показано в следующей команде, вы можете подключиться к точке доступа Wi-Fi, определив переменную `ssid` со значением имени точки доступа Wi-Fi. SSID расшифровывается как идентификатор набора услуг и является техническим термином для удобочитаемого имени сети Wi-Fi, предоставляемого базовой станцией Wi-Fi.

```
> mcconfig -d -m -p esp ssid="my wi-fi"
```

Определение `ssid` таким образом приводит к тому, что в ваше приложение добавляется переменная конфигурации, которая используется базовой сетевой прошивкой устройства для автоматического подключения к Wi-Fi при включении устройства. После того, как соединение Wi-Fi установлено, ваше приложение запускается. Это удобно, потому что это означает, что ваше приложение может предполагать, что сеть всегда доступна.

Если для вашей точки доступа Wi-Fi требуется пароль, укажите его в командной строке в качестве значения переменной пароля:

```
> mcconfig -d -m -p esp ssid="my wi-fi" password="secret"
```

Во время процесса подключения к сети Wi-Fi в консоли отладки отображаются сообщения диагностической трассировки. Просмотрите сообщения, чтобы помочь диагностировать проблемы с подключением. Вот пример успешного подключения:

```
Wi-Fi connected to "Moddable"  
IP address 10.0.1.79
```

Если пароль Wi-Fi отклонен точкой доступа Wi-Fi, отображается следующее сообщение:

```
Wi-Fi password rejected // Пароль Wi-Fi отклонен
```

Все другие неудачные попытки подключения отображают следующее сообщение:

```
Wi-Fi disconnected // Wi-Fi отключен
```

Установите \$EXAMPLES/ch3-network/wifi-command-lineexample на свое устройство, чтобы протестировать этот метод подключения.

СВЯЗЬ С КОДОМ

Использование параметров командной строки для определения учетных данных Wi-Fi удобно для разработки, но для проектов, которыми вы делитесь с другими, вместо этого часто требуется хранить учетные данные Wi-Fi в настройках. В этом разделе рассматривается код для подключения к точке доступа Wi-Fi, определенной в вашем приложении (управление настройками описано в главе 5).

Wi-Fi module содержит класс JavaScript, используемый для управления соединениями Wi-Fi network. Чтобы использовать модуль WiFi в своем коде, сначала импортируйте из него класс WiFi:

```
import WiFi from "wifi";
```

Используйте метод статического подключения класса WiFi для подключения к сети Wi-Fi. В примере \$EXAMPLES/ch3-network/wifi-code SSID и пароль передаются конструктору как свойства в словаре (листинг 3-1).

Листинг 3-1.

```
WiFi.connect({  
    ssid: "my wi-fi",  
    password: "secret"  
})  
);
```

Этот вызов запускает процесс установления соединения. Вызов является асинхронным, что означает, что фактическая работа по подключению происходит в фоновом режиме; приложение продолжает работать, пока устанавливается соединение. Это так же, как на вашем телефоне, где вы можете продолжать использовать приложения, пока устанавливается соединение Wi-Fi. В устройстве IoT вам часто нужно знать, когда доступно сетевое подключение, чтобы вы знали, когда ваше приложение может подключаться к другим устройствам и Интернету.

Чтобы отслеживать состояние подключения, создайте экземпляр класса WiFi и предоставьте функцию обратного вызова мониторинга (листинг 3-2), которая будет вызываться при каждом изменении состояния подключения.

Листинг 3-2.

```
let wifiMonitor = new WiFi({
    ssid: "my wi-fi",
    password: "secret"
},
function(msg) {
    switch (msg) {
        case WiFi.gotIP:
            trace("network ready\n");
            break;

        case WiFi.connected:
            trace("connected\n");
            break;

        case WiFi.disconnected:
            trace("connection lost\n");
            break;
    }
});
```

Функция обратного вызова вызывается с одним из этих трех сообщений, в зависимости от состояния соединения:

connected (подключено) — ваше устройство подключилось к точке доступа Wi-Fi. Однако он еще не готов к использованию, поскольку еще не получил свой IP-адрес. Когда вы видите это сообщение, вы знаете, что SSID и пароль действительны.

gotIP — ваше устройство получило свой IP-адрес и теперь готово к обмену данными с другими устройствами в локальной сети и Интернете.

disconnected (отключено) — ваше устройство потеряло сетевое соединение. На некоторых устройствах вы получаете это сообщение до получения сообщения о подключении.

В некоторых проектах объект WiFi остается активным все время, чтобы отслеживать сетевые отключения. Если вам не нужно отслеживать обрыв сетевого соединения, вам следует закрыть объект WiFi, чтобы освободить используемую им память.

```
wifiMonitor.close();
```

Заккрытие объекта WiFi не приводит к отключению от сети Wi-Fi, это просто означает, что ваша функция обратного вызова больше не будет вызываться с уведомлениями о статусе обратного вызова.

Чтобы отключиться от сети Wi-Fi, вызовите метод статического отключения класса WiFi:

```
WiFi.disconnect();
```

Чтобы проверить этот метод подключения, выполните следующие действия:

1. Откройте \$EXAMPLES/ch3-network/wifi-code/main.js в текстовом редакторе.
2. Измените строки 4 и 5, чтобы ssid и пароль соответствовали вашим сетевым учетным данным.

3. Установите пример `$EXAMPLES/ch3-network/wifi-code` на свое устройство из командной строки с помощью `mcconfig`.

Если соединение установлено успешно, вы увидите следующие сообщения в консоли отладки:

```
connectednetwork ready // подключенная сеть готова
```

Если соединение установить не удалось, вместо этого вы будете постоянно видеть, что соединение потеряно.

Подключение к любой открытой точке доступа

Иногда вам нужно, чтобы ваше IoT-устройство подключалось к любой доступной точке доступа openWi-Fi (например, к той, для которой не требуется пароль). Подключение к неизвестной сети — плохая идея с точки зрения безопасности, но в некоторых ситуациях удобство важнее.

Чтобы подключиться к открытой точке доступа, первым делом нужно ее найти. Класс `WiFi` предоставляет метод статического сканирования для поиска точек доступа. Код в листинге 3-3 выполняет однократное сканирование точек доступа, записывая результаты в консоль отладки. Он получает мощность сигнала из свойства `rssi` точки доступа. RSSI расшифровывается как индикация мощности принимаемого сигнала и является мерой мощности сигнала, полученного от точки доступа Wi-Fi. Его значения — отрицательные числа, а более сильные сигналы имеют значение RSSI, близкое к 0.

Листинг 3-3.

```
WiFi.scan({}, accessPoint => {  
  if (!accessPoint) {  
    trace("scan complete\n");  
    return;  
  }  
})
```

```

let name = accessPoint.ssid;
let open = "none" === accessPoint.authentication;
let signal = accessPoint.rssi;
trace(`${name}: open=${open}, signal=${signal}\n`);
});

```

Вот пример вывода этого кода:

```

ESP_E5C7AF: open=true, signal=-62
Large Conf.: open=false, signal=-85
Expo 2.4: open=false, signal=-74
PAB: open=true, signal=-77
Kanpai: open=false, signal=-66
Moddable: open=false, signal=-70
scan complete

```

Продолжительность сканирования обычно составляет менее 5 секунд, в зависимости от устройства. Во время сканирования пример отслеживает имя точки доступа, открыта ли она и мощность ее сигнала. По завершении сканирования вызывается функция обратного вызова сканирования с аргументом `accessPoint`, для которого задано значение `undefined`, и выполняется трассировка завершения сканирования сообщения.

Если вы находитесь в месте с множеством точек доступа, одно сканирование может не обнаружить каждую доступную точку доступа. Чтобы составить полный список, ваше приложение может объединить результаты нескольких сканирований. См. пример `wifiscancontinuous` в Moddable SDK.

Пользователь, выбирающий точку доступа Wi-Fi для подключения, обычно выбирает ту, которая обладает наибольшей силой. Пример `$EXAMPLES/ch3-network/wifi-open-ap` выполняет тот же процесс выбора, используя код из листинга 3-4.

Листинг 3-4.

```

let best;

WiFi.scan({}, accessPoint => {
  if (!accessPoint) {
    if (!best) {
      trace("no open access points found\n");
      return;
    }
    trace(`connecting to ${best.ssid}\n`);
    WiFi.connect({ssid: best.ssid});
    return;
  }

  if ("none" !== accessPoint.authentication)
    return; // not open

  if (!best) {
    best = accessPoint; // first open access point found
    return;
  }

  if (best.rssi < accessPoint.rssi)
    best = accessPoint; // new best
});

```

Этот код использует переменную `best` для отслеживания открытой точки доступа с самым сильным сигналом во время сканирования. После завершения сканирования код подключается к этой точке доступа.

Чтобы протестировать этот метод, установите пример `wifi-open-ap` на свое устройство.

Установка сетевого хоста

Хост находится в каталоге \$EXAMPLES/ch3-network/host. Перейдите в этот каталог из командной строки и установите его с помощью `mcconfig`.

Примеры установки

Примеры в этой главе работают правильно, только если устройство подключено к точке доступа Wi-Fi. Ранее в этой главе вы узнали, как указать SSID и пароль точки доступа, определив переменные в команде `mcconfig`. Вы можете использовать эти же переменные в команде `mcrun` для подключения вашего устройства к Wi-Fi перед запуском примера.

```
> mcrun -d -m -p esp ssid="my wi-fi"
```

```
> mcrun -d -m -p esp ssid="my wi-fi" password="secret"
```

Получение информации о сети

При работе с сетью вам может понадобиться информация о сетевом интерфейсе или сетевом соединении для целей отладки или реализации функций. Эта информация доступна из сетевого модуля.

```
import Net from "net";
```

Информация извлекается из объекта `Net` с помощью его статического метода `get`. В этом примере извлекается имя точки доступа Wi-Fi, к которой подключено устройство:

```
let ssid = Net.get("SSID");
```

Вот некоторые другие фрагменты информации, которые вы можете получить:

- IP – IP-адрес сетевого подключения; например, 10.0.1.4
- MAC – MAC-адрес сетевого интерфейса; например, A4:D1:8C:DB:C0:20
- SSID – имя точки доступа Wi-Fi
- BSSID – MAC-адрес точки доступа Wi-Fi; например, 18:64:72:47:d4:32
- RSSI – мощность сигнала Wi-Fi

Выполнение HTTP-запросов

Наиболее часто используемым протоколом в Интернете является HTTP, и его популярность объясняется многими вескими причинами: он относительно прост, широко поддерживается, хорошо работает с небольшими и большими объемами данных, доказал свою исключительную гибкость и легкость, может поддерживаться на широком спектре устройств, в том числе относительно недорогих, которые можно найти во многих продуктах IoT. В этом разделе показано, как выполнять различные типы HTTP-запросов к HTTP-серверу. (В следующем разделе будет показано, как защитить эти соединения.)

ОСНОВЫ

Модуль `http` содержит поддержку выполнения HTTP-запросов и создания HTTP-сервера. Чтобы сделать HTTP-запрос, сначала импортируйте класс `Request` из модуля:

```
import {Request} from "http";
```

Класс Request использует словарь для настройки запроса. В словаре всего два обязательных свойства:

- Либо свойство хоста, либо свойство адреса для определения сервера для подключения, где хост указывает сервер по имени (например, `www.example.com`), а адрес определяет сервер по IP-адресу (например, `10.0.1.23`).
- Свойство пути для указания пути к ресурсу HTTP для доступа (например, `/index.html` или `/data/lights.json`).

Все остальные свойства являются необязательными; тип HTTP-запроса, который вы делаете, определяет, присутствуют ли они и каковы их значения. Многие необязательные свойства представлены в следующих разделах.

В дополнение к словарю конфигурации каждый HTTP-запрос имеет функцию обратного вызова, которая вызывается на различных этапах запроса. Обратный вызов получает сообщение, соответствующее текущему этапу. Вот полный список этапов HTTP-запроса:

- `requestFragment` — обратный вызов запрашивается для предоставления следующей части тела запроса.
- `status` – получена строка состояния HTTP-ответа. Доступен код состояния HTTP (например, 200, 404 или 301. Код состояния указывает на успех или неудачу запроса.
- `header` – был получен заголовок ответа HTTP. Это сообщение повторяется для каждого полученного заголовка HTTP.
- `headersComplete` – это сообщение получено между получением окончательного заголовка ответа HTTP и получением тела ответа.

- `responseFragment` — это сообщение предоставляет фрагмент ответа HTTP и может быть получено несколько раз.
- `responseComplete` — это сообщение получено после всех фрагментов ответа HTTP.
- `error` — Произошел сбой при обработке HTTP-запроса.

Если это выглядит ошеломляющим, не волнуйтесь; многие HTTP-запросы используют только одно или два таких сообщения. Два сообщения, `requestFragment` и `responseFragment`, используются только для работы с данными HTTP, которые слишком велики для размещения в памяти устройства. В следующих разделах показано, как использовать многие доступные сообщения.

GET

Наиболее распространенным HTTP-запросом является GET, который извлекает фрагмент данных. Код в листинге 3-5 из примера `$EXAMPLES/ch3-network/http-get` выполняет запрос HTTP GET для получения домашней страницы с веб-сервера `www.example.com`.

Листинг 3-5.

```
let request = new Request({
  host: "www.example.com",
  path: "/",
  response: String
});

request.callback = function(msg, value) {
  if (Request.responseComplete === msg)
    trace(value, "\n");
}
```

Свойство ответа в вызове конструктора запроса указывает, как вы хотите, чтобы тело ответа возвращалось. В этом случае вы указываете, что он должен быть возвращен как строка JavaScript. Обратный вызов получает сообщение завершения ответа, когда получен ответ — вся веб-страница. Веб-страница хранится в параметре `value`. Вызов `trace` отображает исходный HTML-код в консоли отладки.

Вы можете использовать этот подход в своих проектах для извлечения текстовых данных. Если вы хотите получить двоичные данные, вы можете сделать это, передав значение `ArrayBuffer` вместо `String` для свойства ответа, как в листинге 3-6.

Листинг 3-6.

```
let request = new Request({  
  host: "httpbin.org",  
  path: "/bytes/1024",  
  response: ArrayBuffer  
});
```

Получение всего HTTP-ответа сразу работает отлично, если на устройстве достаточно памяти для его хранения. Если памяти недостаточно, запрос завершается с сообщением об ошибке. В следующем разделе объясняется, как извлекать источники, объем которых превышает доступную память.

Streaming GET

В ситуациях, когда ответ на HTTP-запрос может не уместиться в доступной памяти, вместо этого можно выполнить потоковый HTTP-запрос GET. Это немного сложнее, как показано в листинге 3-7 из примера `$EXAMPLES/ch3-network/http-streaming-get`.

Листинг 3-7.

```

let request = new Request({
    host: "www.bing.com",
    path: "/"
});

request.callback = function(msg, value, etc) {
    if (Request.responseFragment === msg)
        trace(this.read(String), "\n");
    else if (Request.responseComplete === msg)
        trace(`\n\nTransfer complete.\n\n`);
}

```

Обратите внимание, что в вызове конструктора свойство `response` отсутствует. Отсутствие этого свойства указывает классу HTTP-запроса доставлять каждый фрагмент тела ответа обратному вызову по мере его получения с сообщением `responseFragment`. В этом примере обратный вызов затем считывает данные в виде строки для трассировки в консоль отладки, но он также может считывать данные как `ArrayBuffer`. Вместо трассировки в консоль отладки обратный вызов может записывать данные в файл; вы узнаете, как это сделать, в главе 5.

При потоковой передаче HTTP-запроса тело ответа не предоставляется в аргументе значения с сообщением завершения ответа.

Класс `Request` поддерживает функцию кодирования передачи по частям протокола HTTP. Эта функция часто используется для доставки больших ответов. Класс HTTP-запроса декодирует фрагменты перед вызовом функции обратного вызова. Следовательно, вашей функции обратного вызова не нужно анализировать заголовки фрагментов, что упрощает код.

GET JSON

Продукты IoT обычно не запрашивают веб-страницы, если только они не очищают страницу для извлечения данных; вместо этого они используют REST API, которые очень часто отвечают JSON. Поскольку JSON — это очень небольшое подмножество JavaScript, предназначенное только для данных, его чрезвычайно удобно использовать в коде JavaScript. В листинге 3-8 приведен пример запроса к службе погоды REST. Идентификатор приложения, используемый в \$EXAMPLES/ch3-network/http-get-json, является только примером; вы должны зарегистрировать свой собственный идентификатор приложения (APPID) на openweathermap.org и использовать его вместо этого.

Листинг 3-8.

```
const APPID = "94de4cda19a2ba07d3fa6450eb80f091";
const zip = "94303";
const country = "us";

let request = new Request({
  host: "api.openweathermap.org",
  path: `/data/2.5/weather?appid=${APPID}&` +
    `zip=${zip},${country}&units=imperial`
  response: String
});

request.callback = function(msg, value) {
  if (Request.responseComplete === msg) {
    value = JSON.parse(value);
    trace(`Location: ${value.name}\n`);
    trace(`Temperature: ${value.main.temp} F\n`);
    trace(`Weather: ${value.weather[0].main}\n`);
  }
}
```

Обратите внимание, что в словаре, переданном конструктору Request, для ответа задано значение String, как и в предыдущем примере GET. Ответ запрашивается как строка, поскольку JSON — это текстовый формат. Как только ответ доступен, обратный вызов получает сообщение responseComplete, а затем использует JSON.parse для преобразования полученной строки в объект JavaScript. Наконец, он отслеживает три значения из ответа на консоль отладки.

Если вы хотите узнать все доступные значения, возвращаемые погодным сервисом, вы можете либо прочитать их документацию, либо посмотреть ответ прямо в консоли отладки. Чтобы посмотреть в отладчике, установите точку останова на первом вызове трассировки; при остановке в точке останова разверните свойство value, чтобы увидеть значения, как показано на рис. 3-1.

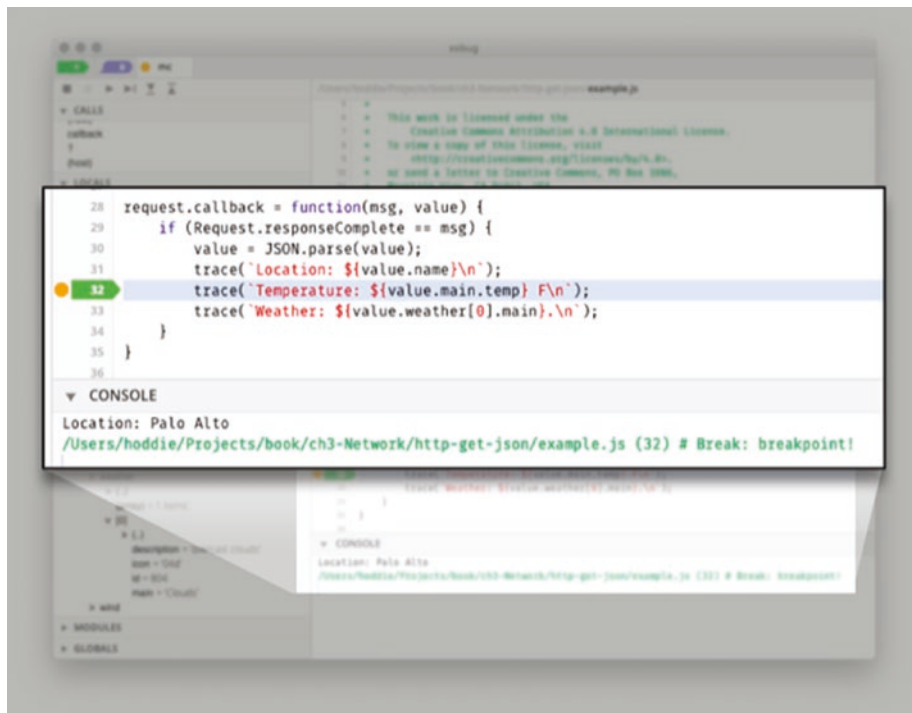


Рис. 3-1. Расширенный ответ погоды JSON, показанный в xsbug

Как вы можете видеть на этом рисунке, JSON, возвращаемый сервером, содержит множество свойств, которые не использует код JavaScript, например облака и видимость. В некоторых ситуациях на устройстве достаточно памяти для хранения всего текста JSON, но недостаточно памяти для хранения объекта JavaScript, созданного путем вызова `JSON.parse`. Объект может использовать больше памяти, чем текст, из-за способа хранения объектов JavaScript в памяти. Чтобы помочь решить эту проблему, движок XS JavaScript поддерживает необязательный второй параметр для вызова `JSON.parse`. Если второй параметр является массивом, из JSON анализируются только имена свойств в массиве. Это может значительно сократить объем используемой памяти, а синтаксический анализ также выполняется быстрее. Вот как можно изменить вызов `JSON.parse` в предыдущем примере, чтобы декодировать только те свойства, которые используются в примере:

```
value = JSON.parse(value, ["main", "name", "temp", "weather"]);
```

Подкласс HTTP-запроса

Класс HTTP-запроса — это низкоуровневый класс, предоставляющий множество функциональных возможностей с высокой степенью эффективности, что придает ему мощность и гибкость, необходимые для широкого спектра сценариев IoT. Тем не менее, в любой конкретной ситуации функциональное назначение кода может быть затемнено деталями, относящимися к протоколу HTTP. Рассмотрим код в листинге 3-8 из предыдущего раздела: входные данные — это почтовый индекс и страна, а выходные данные — текущие погодные условия, а все остальное — детали реализации.

Хороший способ упростить код — создать подкласс. Хорошо спроектированный подкласс предоставляет сфокусированный, простой в использовании API, который принимает только соответствующие входные данные (например, почтовый индекс) и предоставляет только желаемые выходные данные (например, погодные условия). В примере `$EXAMPLES/ch3-network/http-get-subclass` (листинг 3-9) показан дизайн подкласса для запроса погоды из предыдущего раздела.

Листинг 3-9.

```

const APPID = "94de4cda19a2ba07d3fa6450eb80f091";

class WeatherRequest extends Request {
  constructor(zip, country) {
    super({
      host: "api.openweathermap.org",
      path: `/data/2.5/weather?appid=${APPID}&` +
        `zip=${zip},${country}&units=imperial`,
      response: String
    });
  }
  callback(msg, value) {
    if (Request.responseComplete === msg) {
      value = JSON.parse(value,
        ["main", "name", "temp", "weather"]);
      this.onReceived({
        temperature: value.main.temp,
        condition: value.weather[0].main
      });
    }
  }
}

```

Использовать этот подкласс WeatherRequest легко (листинг 3-10), так как все детали протокола HTTP, API openweathermap.org и синтаксического анализа JSON скрыты в реализации подкласса.

Листинг 3-10.

```
let weather = new WeatherRequest(94025, "us");

weather.onReceived = function(result) {
    trace(`Temperature is ${result.temperature}\n`);
    trace(`Condition is ${result.condition}\n`);
}
```

Настройка заголовков запроса

Протокол HTTP использует заголовки для передачи дополнительной информации о запросе на сервер. Например, принято включать название и версию продукта, отправляющего HTTP-запрос, в заголовок User-Agent, один из стандартных заголовков HTTP. Вы также можете включить нестандартные заголовки HTTP с запросом на передачу информации в конкретную облачную службу.

В листинге 3-11 показано, как добавить заголовки к HTTP-запросу. Он добавляет стандартный заголовок User-Agent и настраиваемый заголовок X-Custom. Заголовки предоставляются в виде массива, где за именем каждого заголовка следует его значение.

Листинг 3-11.

```
let request = new Request({
    host: "api.example.com",
    path: "/api/status",
    response: String,
    headers: [
        "User-Agent", "my_iot_device/0.1 example/1.0",
        "X-Custom", "my value"
    ]
});
```

Указание заголовков в массиве, а не в словаре или объекте Map несколько необычно. Это сделано здесь, потому что это более эффективно и сокращает ресурсы, необходимые на устройстве IoT.

Получение заголовков ответа

Протокол HTTP использует заголовки для передачи дополнительной информации об ответе клиенту. Распространенным заголовком является Content-Type, который указывает тип данных ответа (например, text/plain, application/json или image/png). Заголовки ответа доставляются в функцию обратного вызова вместе с сообщением заголовка. За один раз доставляется один заголовок, чтобы уменьшить использование памяти, избегая необходимости одновременного хранения всех полученных заголовков в памяти. Когда все заголовки ответов получены, обратный вызов вызывается с сообщением завершения заголовков.

В листинге 3-12 проверяются все полученные заголовки на наличие заголовка Content-Type. Если он найден, его значение сохраняется в переменной contentType. После того, как все заголовки получены, код проверяет, был ли получен заголовок Content-Type (то есть, contentType не является неопределенным) и что тип содержимого — text/plain.

Листинг 3-12.

```
let contentType;

request.callback = function(msg, value, etc) {
  if (Request.header === msg) {
    if ("content-type" === value)
      contentType = etc;
  }
  else if (Request.headersComplete === msg) {
    trace("all headers received\n");
```

```

        if ((undefined === contentType) ||
            !contentType.toLowerCase().startsWith("text/plain"))
            this.close();
    }
}

```

Имена заголовков HTTP по определению нечувствительны к регистру, поэтому Content-Type, content-type и CONTENT-TYPE ссылаются на один и тот же заголовок. Класс HTTP Request преобразует имя заголовка в нижний регистр, поэтому обратный вызов всегда может использовать строчные буквы при сравнении имен заголовков.

POST

Все примеры HTTP-запросов до сих пор использовали метод HTTP-запроса по умолчанию GET и имели пустое тело запроса. Класс HTTP-запроса поддерживает установку для метода запроса любого значения, например POST, и предоставление тела запроса.

В примере \$EXAMPLES/ch3-network/http-post (листинг 3-13) выполняется вызов POST на веб-сервер с телом запроса JSON. Свойство `method` словаря определяет метод HTTP-запроса, а свойство `body` определяет содержимое тела запроса. Тело запроса может быть либо строкой, либо массивом `ArrayBuffer`. Запрос отправляется на сервер, который возвращает ответ JSON. Функция обратного вызова отслеживает отраженные значения JSON в консоль отладки.

Листинг 3-13.

```

let request = new Request({
  host: "httpbin.org",
  path: "/post",
  method: "POST",
  body: JSON.stringify({string: "test", number: 123}),
  response: String
});

```

```
request.callback = function(msg, value) {
  if (Request.responseComplete === msg) {
    value = JSON.parse(value);
    trace(`string: ${value.json.string}\n`);
    trace(`number: ${value.json.number}\n`);
  }
}
```

В этом примере все тело запроса хранится в памяти. В некоторых случаях недостаточно свободной памяти для хранения тела запроса, например при загрузке большого файла. Класс HTTP Request поддерживает потоковую передачу тела запроса; пример см. в примере `examples/network/http/httppoststreaming` в Moddable SDK.

Обработка ошибок

Иногда HTTP-запрос завершается со сбоем, возможно, из-за сбоя сети. Теперь возникла проблема с запросом. Во всех случаях отказ является неустраняемым. Поэтому вам необходимо решить, как обработать ошибку способом, подходящим для вашего продукта IoT, например, сообщить об этом пользователю, повторить попытку немедленно, повторить попытку позже или просто проигнорировать ошибку. Если вы еще не готовы добавить в свой проект обработку ошибок, хорошим началом будет добавление диагностической трассировки ошибок, так как это поможет вам увидеть сбой во время разработки.

Когда сбой происходит из-за сетевой ошибки — сбой сети, сбой DNS или сбой сервера — ваш обратный вызов вызывается с сообщением об ошибке. В следующем примере показан обратный вызов, который отслеживает сбой в консоли отладки:

```
request.callback = function(msg, value) {
  if (Request.error === msg)
    trace(`http request failed: ${value}\n`);
}
```

Если сбой вызван проблемой с запросом — он был неправильно сформирован, неверный путь или вы не авторизованы должным образом — сервер отвечает ошибкой в коде состояния HTTP. Класс HTTP-запроса предоставляет код состояния для обратного вызова в сообщении о состоянии. Для многих веб-служб код состояния от 200 до 299 означает, что запрос выполнен успешно, в то время как другие указывают на сбой. Перечисление 3-14 демонстрирует обработку кодов состояния HTTP.

Листинг 3-14.

```
request.callback = function(msg, value) {
    if (Request.status === msg) {
        if ((value < 200) || (value > 299))
            trace(`http status error: ${value}\n`);
    }
}
```

Защита соединений с помощью TLS

Безопасная связь является важной частью большинства продуктов IoT. Это помогает поддерживать конфиденциальность данных, генерируемых продуктом, и предотвращает подделку данных при их перемещении с устройства на сервер. В Интернете большая часть связи защищена с помощью безопасности транспортного уровня или TLS, который заменяет уровень защищенных сокетов (SSL). TLS — это низкоуровневый инструмент для защиты связи, который работает с множеством различных протоколов. В этом разделе объясняется, как использовать TLS с протоколом HTTP. Тот же подход применяется к протоколам WebSocket и MQTT, описанным ниже.

Работа с TLS на встроенном устройстве немного сложнее, чем на компьютере, сервере или мобильном устройстве, из-за меньшего объема памяти, вычислительной мощности и хранилища. На самом деле, установка безопасного TLS-соединения является самой сложной вычислительной задачей, которую выполняют многие продукты IoT.

Использование TLS с классом SecureSocket

Класс SecureSocket реализует TLS таким образом, что его можно использовать с различными сетевыми протоколами. Чтобы использовать SecureSocket, вы должны сначала импортировать его:

```
import SecureSocket from "securesocket";
```

Чтобы сделать безопасный HTTP-запрос (HTTPS), добавьте свойство Socket со значением SecureSocket, которое указывает классу HTTP-запроса использовать защищенный сокет вместо стандартного сокета по умолчанию. Листинг 3-15 — это выдержка из примера \$EXAMPLES/ch3-network/https-get, в котором показан словарь из более раннего примера HTTP GET (листинг 3-5), измененный для выполнения HTTPS-запроса.

Листинг 3-15.

```
let request = new Request({  
  host: "www.example.com",  
  path: "/",  
  response: String,  
  Socket: SecureSocket  
});
```

Обратный вызов не отличается от исходного примера.

Публичные сертификаты

Сертификаты являются важной частью обеспечения безопасности TLS: они позволяют клиенту проверить подлинность сервера. Сертификаты встроены в программное обеспечение продукта IoT так же, как они встроены в веб-браузер, с одним отличием: в то время как веб-браузер может хранить сотни сертификатов — достаточно для проверки подлинности всех общедоступных серверов в Интернете — У продукта IoT недостаточно места для хранения такого количества сертификатов.

К счастью, продукт IoT обычно связывается только с несколькими серверами, поэтому вы можете включить только те сертификаты, которые вам нужны.

Сертификаты — это данные, поэтому они хранятся в ресурсах, к которым приложения могут получить доступ, а не в коде. Манифест для примера HTTPS GET включает сертификат, необходимый для проверки подлинности `www.example.com` (листинг 3-16).

Листинг 3-16.

```
"resources": {
  "*": [
    "${MODULES}/crypt/data/ca107"
  ]
}
```

Если вы пытаетесь получить доступ к веб-сайту, а ресурс сертификата недоступен, реализация TLS выдает ошибку, подобную той, что показана на рис. 3-2.

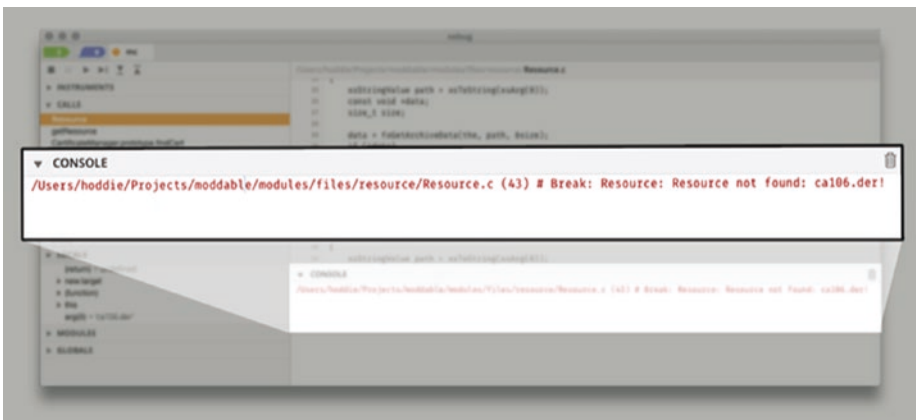


Рис. 3-2. Сообщение об ошибке сертификата TLS в xdebug

Ошибка показывает номер отсутствующего ресурса, поэтому вы можете изменить манифест, чтобы включить этот ресурс (листинг 3-17).

Листинг 3-17.

```
"resources": {
  "*": [
    "$(MODULES)/crypt/data/ca106"
  ]
}
```

Это работает, потому что Moddable SDK включает сертификаты для большинства общедоступных веб-сайтов. В следующем разделе описывается, как подключиться к серверу, который использует частный сертификат.

Частные сертификаты

Частные сертификаты обеспечивают дополнительную безопасность, гарантируя, что только продукты IoT, имеющие частный сертификат, могут подключаться к серверу. Частный сертификат обычно предоставляется в файле с расширением .der. Чтобы использовать частный сертификат в своем проекте, сначала поместите сертификат в том же каталоге, что и ваш манифест, и измените манифест, чтобы включить его (листинг 3-18). Обратите внимание, что манифест не включает расширение имени файла .der.

Листинг 3-18.

```
"resources": {
  "*": [
    "./private_certificate"
  ]
}
```

Далее, как показано в листинге 3-19, ваше приложение загружает сертификат из ресурса и передает его HTTP-запросу в безопасном свойстве словаря конструктора.

Листинг 3-19.

```
import Resource from "resource";

let cert = new Resource("private_certificate.der");
let request = new Request({
  host: "iot.privateserver.net",
  path: "/",
  response: String,
  Socket: SecureSocket,
  secure: {
    certificate: cert
  }});
```

Создание HTTP-сервера

Включение HTTP-сервера в ваш продукт IoT открывает множество возможностей, например позволяет вашему продукту делать следующее:

- Предоставление веб-страниц пользователям в той же сети, что является отличным способом предоставления пользовательского интерфейса для продуктов без дисплея.
- Предоставьте REST API для приложений и других устройств, с которыми можно взаимодействовать.

ОСНОВЫ:

Чтобы создать HTTP-сервер, сначала импортируйте класс `Server` из модуля `http`:

```
import {Server} from "http";
```

Как и класс HTTP-запроса, класс HTTP-сервера настраивается с помощью объекта словаря. В словаре нет обязательных свойств. Также как HTTP-запрос, HTTP-сервер использует функцию обратного вызова для доставки сообщений на различных этапах ответа на HTTP-запрос. Вот полный список этапов HTTP-запроса:

- `connection` – сервер принял новое соединение.
- `status` – получена строка состояния HTTP-запроса. Путь запроса и метод запроса доступны.
- `header` – получен заголовок HTTP-запроса. Это сообщение повторяется для каждого полученного заголовка HTTP.
- `headersComplete` – это сообщение получено между получением окончательного заголовка HTTP-запроса и получением тела запроса.
- `requestFragment` – (только для потокового тела запроса) Доступен фрагмент тела запроса.
- `requestComplete` – все тело запроса получено.
- `prepareResponse` – сервер готов начать доставку ответа. Обратный вызов возвращает словарь, описывающий ответ.
- `responseFragment` – (только для потоковых ответов) Обратный вызов отвечает на это сообщение, предоставляя следующий фрагмент ответа.
- `responseComplete` – весь ответ был успешно доставлен.

`error` – сбой произошел до того, как HTTP-ответ был полностью доставлен.

Следующие примеры показывают, как использовать многие из этих сообщений. Большинство приложений, работающих с классом HTTP Server, используют лишь некоторые из них.

Ответ на запрос

HTTP-сервер отвечает на всевозможные запросы. Листинг 3-20 — это выдержка из примера `$EXAMPLES/ch3-network/http-server-get`, который отвечает на каждый запрос простым текстом, указывающим метод HTTP, используемый для ответа (обычно GET), и путь к ресурсу HTTP. просил. И метод, и путь предоставляются обратному вызову с сообщением о состоянии. Обратный вызов сохраняет эти значения, чтобы вернуть их в тексте при получении сообщения `prepareResponse`.

Листинг 3-20.

```
let server = new Server({port: 80});
server.callback = function(msg, value, etc) {
  if (Server.status === msg) {
    this.path = value;
    this.method = etc;
  }
  else if (Server.prepareResponse === msg) {
    return {
      headers: ["Content-Type", "text/plain"],
      body: `hello. path "${this.path}".
            method "${this.method}".`
    };
  }
}
```

При запуске этого примера IP-адрес устройства отображается в консоли отладки следующим образом:

```
Wi-Fi connected to "Moddable"  
IP address 10.0.1.5
```

После отображения IP-адреса вы можете использовать веб-браузер в той же сети для подключения к веб-серверу. Когда вы вводите `http://10.0.1.5/test.html` в адресную строку браузера, вы получаете следующий ответ:

```
hello. path "/test.html". method "GET".
```

Обратите внимание, что обратный вызов не устанавливает поле `Content-Length`. Когда вы используете свойство `body`, реализация сервера автоматически добавляет заголовок `Content-Length`.

Свойство `body` в этом примере — это строка, но оно также может быть `ArrayBuffer` для ответа двоичными данными.

Ответ на JSON PUT

Часто REST API получает входные данные в виде JSON в теле запроса и предоставляет выходные данные в виде JSON в теле ответа. Пример `$EXAMPLES/ch3-network/http-server-put` представляет собой эхо-сервер JSON, который отвечает на каждое полученное сообщение, отправляя это сообщение обратно. В примере предполагается, что клиент будет использовать метод `PUT` для отправки объекта JSON. Ответ встраивает этот объект JSON в более крупный объект JSON, который также включает свойство ошибки.

Когда сообщение о состоянии получено, сервер проверяет, что это метод `PUT`; в противном случае сервер закрывает соединение, чтобы отклонить запрос. Обратный вызов возвращает `String`, когда он получает сообщение о состоянии, чтобы указать, что он хочет получить все тело запроса за один раз в виде строки. Чтобы вместо этого получить тело запроса в виде двоичных данных, он может вернуть `ArrayBuffer`.

В ответ на сообщение `requestComplete` сервер анализирует ввод JSON и внедряет его в объект, используемый для генерации ответа. Когда получено сообщение `prepareResponse`, сервер в листинге 3-21 возвращает тело ответа JSON в виде строки и устанавливает для заголовка `Content-Type` значение `application/json`.

Листинг 3-21.

```
let server = new Server;

server.callback = function(msg, value, etc) {
  switch (msg) {
    case Server.status:
      if ("PUT" !== etc)
        this.close();
      return String;

    case Server.requestComplete:
      this.json = {
        error: "none",
        request: JSON.parse(value)
      };
      break;

    case Server.prepareResponse:
      return {
        headers: ["Content-Type", "application/json"],
        body: JSON.stringify(this.json)
      };
  }
}
```

Поскольку в этом примере словарь не передается конструктору сервера, по умолчанию используется порт 80.

Вы можете использовать следующую команду, чтобы попробовать пример `http-server-put` с помощью инструмента командной строки `curl`. Вам нужно будет изменить `<IP_address>`, чтобы он соответствовал IP-адресу вашей платы разработки (например, 192.168.1.45). Команда отправляет простое сообщение JSON в аргументе `--data` на сервер и отображает результат в консоли отладки.

```
> curl http://<IP_address>/json
--request PUT
--header "Content-Type: application/json"
--data '{"example": "data", "value": 101}'
```

Получение запроса на потоковую передачу

Когда на HTTP-сервер отправляется большое тело запроса, оно может оказаться слишком большим для размещения в памяти. Это может произойти, например, когда вы загружаете датированный магазин в файл. Решение состоит в том, чтобы получать тело запроса фрагментами, а не все сразу. Перечисление 3-22 из примера `$EXAMPLES/ch3-network/http-server-streaming-put` регистрирует произвольно большой текстовый запрос на консоль отладки. Чтобы попросить класс HTTP-сервера доставить тело запроса фрагментами, обратный вызов возвращает `true` в сообщение `prepareRequest`. Фрагменты доставляются с сообщением `requestFragment` и отслеживаются до консоли отладки. Сообщение `requestComplete` указывает, что все фрагменты тела запроса доставлены.

Листинг 3-22.

```
let server = new Server;

server.callback = function(msg, value) {
  switch (msg) {
    case Server.status:
      trace("\n ** begin upload to ${value} **\n");
      break;
```



```

    case Server.prepareRequest:
        return true;

    case Server.requestFragment:
        trace(this.read(String));
        break;

    case Server.requestComplete:
        trace("\n ** end of file **\n");
        break;
}
}

```

Вы можете адаптировать этот пример для записи полученных данных туда, где они нужны вашему приложению, а не в консоль отладки. Например, в главе 5 вы изучите API для записи данных в файл.

Чтобы попробовать этот пример, используйте инструмент командной строки `curl`, как показано ниже. Вам потребуется изменить `<directory_path>` и `<IP_address>` для вашей конфигурации.

```

> curl --data-binary "@/users/<directory_path>/test.txt"
    http://<IP_address>/test.txt -v

```

Отправка потокового ответа

Если ответ на HTTP-запрос слишком велик и не помещается в памяти, ответ может быть передан в потоковом режиме. Этот подход подходит для загрузки файлов. Как показано в листинге 3-23, пример `$EXAMPLES/ch3-network/http-server-streaming-get` генерирует ответ произвольной длины, содержащий случайные целые числа от 1 до 100. Чтобы указать, что тело ответа должно быть передано в потоковом режиме, `callback` устанавливает для свойства `body` значение `true` в словаре, возвращенном из сообщения `prepareResponse`. Сервер повторно вызывает обратный вызов с сообщением `responseFragment`, чтобы получить следующую часть ответа. Обратный вызов возвращает `undefined`, чтобы указать конец ответа.

Листинг 3-23.

```

let server = new Server;

server.callback = function(msg, value) {
  if (Server.prepareResponse === msg) {
    return {
      headers: ["Content-Type", "text/plain"],
      body: true
    };
  }
  else if (Server.responseFragment === msg) {
    let i = Math.round(Math.random() * 100);
    if (0 === i)
      return;
    return i + "\n";
  }
}

```

Этот пример возвращает строковые значения для тела ответа, но также может возвращать значения `ArrayBuffer` для предоставления двоичных данных. Когда получено сообщение `responseFragment`, аргумент `value` обратного вызова указывает максимальное количество байтов, которое сервер готов принять для этого фрагмента. Когда вы выполняете потоковую передачу файла, это можно использовать как количество байтов для чтения из файла для фрагмента.

Класс HTTP-сервера отправляет потоковые тела ответов, используя кодирование передачи по частям. Для тела ответа, длина которого известна, сервер использует кодировку удостоверения по умолчанию для отправки тела без заголовка кодировки передачи и включает заголовок `Content-Length`.

mDNS

Многоадресный DNS, или mDNS, представляет собой набор возможностей, упрощающих совместную работу устройств в локальной сети. Вы, вероятно, знакомы с протоколом DNS (система доменных имен), потому что с его помощью ваш веб-браузер находит сетевой адрес веб-сайта, который вы вводите в адресную строку (например, именно так браузер преобразует `www.example.com` в `93.184.216.34`). DNS предназначен для использования во всем Интернете. Напротив, mDNS предназначен для работы только в вашей локальной сети, например, для всех устройств, подключенных к вашей точке доступа Wi-Fi. DNS — это централизованный дизайн, который зависит от авторитетных серверов для сопоставления имен с IP-адресами, тогда как mDNS полностью децентрализован, и каждое отдельное устройство отвечает на запросы о сопоставлении своего имени с IP-адресом.

В этом разделе вы узнаете, как использовать mDNS для присвоения вашему IoT-устройству имени, например `porch-light.local`, чтобы другие устройства могли найти его по имени, а не знать его IP-адрес. Вы также научитесь использовать другую часть mDNS, DNS-SD (обнаружение службы DNS), для поиска служб, предоставляемых устройствами (например, для поиска всех принтеров или всех веб-серверов), и для рекламы служб вашего устройства в локальной сети.

Модуль `mdns` содержит классы JavaScript, которые вы используете для работы с mDNS и DNS-SD из своего приложения. Чтобы использовать модуль `mdns` в своем коде, сначала импортируйте его следующим образом:

```
import MDNS from "mdns";
```

Notes mDNS хорошо поддерживается в macOS, Android, iOS и Linux. Windows 10 еще не полностью поддерживает mDNS, поэтому вам может потребоваться установить дополнительное программное обеспечение, чтобы использовать его там.

Заявление на имя

mDNS обычно используется для присвоения имени устройству для использования в локальной сети. Имена mDNS всегда находятся в домене .local, как и в термостате.local. Вы можете выбрать любое имя для устройства. Устройство должно проверить, не используется ли уже это имя, поскольку несколько устройств не будут отвечать на одно и то же имя. Процесс проверки называется претензией. Процесс подачи заявки длится несколько секунд. Если обнаружен конфликт, mDNS определяет процесс согласования. В конце согласования только одно устройство имеет запрошенное имя, а другое выбирает неиспользуемое имя. Например, если вы попытаетесь запросить `iotdevice` безуспешно, вы можете получить `iotdevice-2`.

В примере `$EXAMPLES/ch3-network/mdns-claim-name` показан процесс запроса имени (см. листинг 3-24). Конструктор MDNS вызывается со словарем, который содержит свойство `hostName` со значением желаемого имени. Существует функция обратного вызова, которая получает сообщения о ходе выполнения заявки. Когда сообщение об имени получено с ненулевым значением, заявленное имя отслеживается до консоли отладки.

Листинг 3-24.

```
let mdns = new MDNS({
    hostName: "iotdevice"
},
function(msg, value) {
    if ((MDNS.hostName === msg) && value)
        trace(`Claimed name ${value}.\n`);
}
);
```

Как только устройство заявило имя, вы можете использовать это имя для доступа к устройству. Например, вы можете использовать инструмент командной строки `ping`, чтобы убедиться, что устройство подключено к сети.

```
> ping iotdevice.local
```

Поиск службы

Утверждая имя, с вашим устройством становится легче общаться, но в лучшем случае имя дает лишь небольшую подсказку о том, что делает устройство. Было бы полезно знать, что устройство является источником света, термостатом, динамиком или веб-сервером. что вы можете написать код, который работает с ним без какой-либо настройки. Это проблема, которую решает DNS-SD: это способ рекламировать возможности вашего продукта IoT в локальной сети.

Каждый вид службы DNS-SD имеет уникальное имя. Например, служба веб-сервера имеет имя `http`, а сетевая файловая система — имя `nfs`. В примере `$EXAMPLES/ch3-network/mdns-discover` показано, как искать все рекламные веб-серверы в вашей локальной сети. В вашей сети могут быть веб-серверы, о которых вы не знаете, поскольку многие принтеры имеют встроенный веб-сервер для настройки и управления.

Как показано в листинге 3-25, пример `mdns-discover` создает экземпляр MDNS без запроса имени. Он устанавливает функцию обратного вызова для мониторинга, чтобы получать уведомления при обнаружении службы `http`. Для каждой найденной службы он делает HTTP-запрос домашней страницы устройства и отслеживает ее заголовки HTTP до консоли отладки.

Листинг 3-25.

```
let mdns = new MDNS;
mdns.monitor("_http_tcp", function(service, instance) {
    trace(`Found ${service}: "${instance.name}" @ ` +
        `${instance.target} ` +
        `${instance.address}:${instance.port}}\n`);
```

```

let request = new Request({
  host: instance.address,
  port: instance.port,
  path: "/"
});
request.callback = function(msg, value, etc) {
  if (Request.header === msg)
    trace(` ${value}: ${etc}\n`);
  else if (Request.responseComplete === msg)
    trace("\n\n");
  else if (Request.error === msg)
    trace("error \n\n");
};
});

```

Аргумент экземпляра функции обратного вызова имеет несколько свойств для работы с устройством:

- `name` — удобочитаемое имя устройства
- `mDNS-имя` устройства (например, `lightbulb.local`)
- `address` – IP-адрес устройства
- `port` – порт, используемый для подключения к сервису

Вот вывод из примера, когда он находит принтер HP со службой http:

```

Found _http._tcp: "HP ENVY 7640 series"
                @hpprinter.local (192.168.1.223:80)
server: HP HTTP Server; HP ENVY 7640 series - E4W44A;
content-type: text/html

```

```
last-modified: Mon, 23 Jul 2018 10:53:51 GMT
content-language: en
content-length: 658
```

Реклама услуги

Ваше устройство может использовать DNS-SD для рекламы предоставляемых им услуг, что позволяет другим устройствам в той же сети находить и использовать эти услуги.

Пример `$EXAMPLES/ch3-network/mdns-advertise` определяет сервис, который он предоставляет, в объекте JavaScript, хранящемся в переменной `httpService`. В описании службы сказано, что пример поддерживает службу `http` и делает ее доступной через порт 80. В листинге 3-26 определяется служба HTTP для DNS-SD.

Листинг 3-26.

```
let httpService = {
  name: "http",
  protocol: "tcp",
  port: 80
};
```

Затем в примере создается экземпляр MDNS для запроса сервера имен. После того, как имя было заявлено, сценарий в листинге 3-27 добавляет службу `http`. Службу нельзя добавить до того, как будет заявлено имя, поскольку DNS-SD требует, чтобы каждая служба была связана с именем mDNS.

Листинг 3-27.

```
let mdns = new MDNS({
  hostName: "server"
},
```

```
function(msg, value) {
    if ((MDNS.hostName === msg) && value)
        mdns.add(httpService);
}
);
```

После добавления службы ее могут найти другие устройства, как показано ранее в разделе «Поиск службы».

Полный пример mdns-advertise также содержит простой веб-сервер, который прослушивает порт 80. При запуске примера вы можете ввести server.local в свой веб-браузер, чтобы просмотреть ответ веб-сервера.

Веб-сокеты

Протокол WebSocket — хорошая альтернатива HTTP, когда вам нужна частая двусторонняя связь между устройствами. Когда два устройства обмениваются данными с помощью WebSocket, сетевое соединение между ними остается открытым, что позволяет эффективно обмениваться короткими сообщениями, такими как отправка показаний датчика или команда на включение света. В HTTP одно устройство является клиентом, а другое — сервером; только клиент может сделать запрос, а сервер всегда отвечает. WebSocket, с другой стороны, является одноранговым протоколом, позволяющим обоим устройствам отправлять и получать сообщения. Часто это хороший выбор для продуктов IoT, которым необходимо отправлять много небольших сообщений. Однако, поскольку он постоянно поддерживает соединение между двумя устройствами, для него обычно требуется больше памяти, чем для HTTP.

Протокол WebSocket реализуется модулем websocket, который содержит поддержку как клиента WebSocket, так и сервера WebSocket. Ваш проект может импортировать один или оба по мере необходимости.

```
import {Client} from "websocket";
import {Server} from "websocket";
import {Client, Server} from "websocket";
```


Because WebSocket is a peer-to-peer protocol, the code for a client and a server is very similar. The primary difference is in the initial setup.

Подключение к серверу WebSocket

В примере \$EXAMPLES/ch3-network/websocket-client используется эхо-сервер WebSocket, который отвечает на каждое полученное сообщение, отправляя это сообщение обратно. Конструктор класса WebSocket Client использует словарь конфигурации. Единственное обязательное свойство — host, имя сервера. Если свойство порта не указано, предполагается значение WebSocket по умолчанию 80.

```
let ws = new Client({  
  host: "echo.websocket.org"  
});
```

Вы можете установить безопасное соединение с помощью TLS, передав SecureSocket для свойства Socket, как объяснялось ранее в разделе «Использование TLS с классом SecureSocket».

Вы предоставляете функцию обратного вызова для получения сообщений от класса WebSocket Client. Протокол WebSocket проще, чем HTTP, поэтому обратный вызов также проще. В примере с веб-клиентом сообщения о подключении и закрытии просто отслеживают сообщение. Процесс соединения по протоколу WebSocket состоит из двух этапов: сообщение о соединении получено, когда сетевое соединение установлено между клиентом и сервером, и сообщение о рукопожатии получено, когда клиент и сервер соглашаются обмениваться данными с использованием WebSocket, указывая, что соединение готово для использовать.

Когда пример получает сообщение рукопожатия, он отправляет первое сообщение, строку JSON со свойствами счетчика и переключателя. Когда эхо-сервер отправляет этот JSON обратно, обратный вызов в листинге 3-28 вызывается с сообщением получения. Он анализирует строку обратно в JSON, изменяет значения счетчика и переключения и отправляет измененный JSON обратно на эхо-сервер. Этот процесс повторяется бесконечно, с каждым разом увеличиваясь.

Листинг 3-28.

```

ws.callback = function(msg, value) {
    switch (msg) {
        case Client.connect:
            trace("connected\n");
            break;

        case Client.handshake:
            trace("handshake success\n");
            this.write(JSON.stringify({
                count: 1,
                toggle: true
            }));
            break;

        case Client.receive:
            trace(`received: ${value}\n`);
            value = JSON.parse(value);
            value.count += 1;
            value.toggle = !value.toggle;
            this.write(JSON.stringify(value));
            break;

        case Client.disconnect:
            trace("disconnected\n");
            break;
    }
}

```

Вот вывод этого кода:

```

connected
handshake success
received: {"count":1,"toggle":true}

```

```
received: {"count":2,"toggle":false}
received: {"count":3,"toggle":true}
received: {"count":4,"toggle":false}
...
```

Каждый вызов записи отправляет одно сообщение WebSocket. Вы можете отправить сообщение в любое время после получения сообщения рукопожатия, а не только внутри обратного вызова:

```
ws.write("hello");ws.write(Uint8Array.of(1, 2, 3).buffer);
```

Сообщения представляют собой либо строку, либо массив `ArrayBuffer`. Когда вы получаете сообщение WebSocket, это либо строка, либо `ArrayBuffer`, в зависимости от того, что было отправлено. В листинге 3-29 показано, как проверить тип значения полученного сообщения.

Листинг 3-29.

```
if (typeof value === "string")

    ...;    // строка

if (value instanceof ArrayBuffer)
    ...;    // буфер массива, двоичные данные
```

Создание сервера веб-сокеты

Пример `$EXAMPLES/ch3-network/websocket-server` реализует эхо-сервер WebSocket (опять же, это означает, что всякий раз, когда сервер получает сообщение, он отправляет обратно то же самое сообщение). Класс `WebSocket Server` настроен со словарем, который не имеет обязательных свойств. Необязательное свойство порта указывает порт для прослушивания новых подключений; по умолчанию он равен 80.

```
let server = new Server;
```

Функция обратного вызова сервера в листинге 3-30 получает те же сообщения, что и клиент. В этом примере все сообщения просто отслеживают состояние до консоли отладки, за исключением получения, которое возвращает полученное сообщение.

Листинг 3-30.

```
server.callback = function(msg, value) {
    switch (msg) {
        case Server.connect:
            trace("connected\n");
            break;

        case Server.handshake:
            trace("handshake success\n");
            break;

        case Server.receive:
            trace(`received: ${value}\n`);
            this.write(value);
            break;

        case Server.disconnect:
            trace("closed\n");
            break;
    }
}
```

Этот сервер поддерживает несколько одновременных подключений, каждое из которых имеет уникальное значение `this` при вызове обратного вызова. Если вашему приложению необходимо поддерживать состояние соединения, оно может добавить к нему свойства. Когда устанавливается новое соединение, принимается сообщение о соединении; когда соединение завершается, принимается сообщение об отключении.

MQTT

Протокол передачи телеметрии очереди сообщений, или MQTT, представляет собой протокол публикации и подписки, разработанный для использования облегченными клиентскими устройствами IoT. Сервер (иногда называемый «брокером» в MQTT) более сложен и, следовательно, обычно не реализуется на устройствах с ограниченными ресурсами. Сообщения на сервер MQTT и с него организованы по темам. Конкретный сервер может поддерживать множество тем, но клиент получает сообщения только для тех тем, на которые он подписан.

Клиент для протокола MQTT реализован модулем `mqtt`:

```
import MQTT from "mqtt";
```

Подключение к серверу MQTT

Конструктор MQTT настраивается словарем с тремя обязательными параметрами: свойство `host` указывает сервер MQTT для подключения, `port` — номер порта для подключения, а `id` — уникальный идентификатор для этого устройства. Подключение двух устройств с одинаковым идентификатором к серверу MQTT является ошибкой, поэтому позаботьтесь о том, чтобы они были действительно уникальными. Выдержка из примера `$EXAMPLES/ch3-network/mqtt` в листинге 3-31 использует MAC-адрес устройства в качестве уникального идентификатора.

Листинг 3-31.

```
let mqtt = new MQTT({
  host: "test.mosquitto.org",
  port: 1883,
  id: "iot_" + Net.get("MAC")
});
```

Если сервер MQTT требует аутентификации, свойства пользователя и пароля добавляются в словарь конфигурации. Пароль всегда представляет собой двоичные данные, поэтому в листинге 3-32 используется метод `ArrayBuffer.fromString` для преобразования строки в `ArrayBuffer`.

Листинг 3-32.

```
let mqtt = new MQTT({
  host: "test.mosquitto.org",
  port: 1883,
  id: "iot_" + Net.get("MAC"),
  user: "user name",
  password: ArrayBuffer.fromString("secret")
});
```

Чтобы использовать зашифрованное соединение MQTT, используйте TLS, как описано ранее в разделе «Защита соединений с помощью TLS», добавив в словарь свойство `Socket` и необязательное свойство `secure`.

Некоторые серверы используют протокол WebSocket для передачи данных MQTT. Если вы используете сервер, который делает это, вам нужно указать свойство пути, чтобы указать классу MQTT конечную точку для подключения, как показано в листинге 3-33. Передача MQTT через соединение WebSocket не дает никаких преимуществ и использует больше памяти и пропускной способности сети, поэтому ее следует использовать только в том случае, если это требуется удаленному серверу.

Листинг 3-33.

```
let mqtt = new MQTT({
  host: "test.mosquitto.org",
  port: 8080,
  id: "iot_" + Net.get("MAC"),
  path: "/"
});
```

Клиент MQTT имеет три функции обратного вызова (листинг 3-34). Обратный вызов `onReady` вызывается, когда соединение с сервером успешно установлено, `onMessage` — при получении сообщения и `onClose` — при потере соединения.

Листинг 3-34.

```
mqtt.onReady = function() {  
    trace("connection established\n");  
}  
  
mqtt.onMessage = function(topic, data) {  
    trace("message received\n");  
}  
  
mqtt.onClose = function() {  
    trace("connection lost\n");  
}
```

После вызова обратного вызова `onReady` ваш MQTT-клиент готов подписаться на темы сообщений и публиковать сообщения.

Подписка на тему

Чтобы подписаться на тему, отправьте серверу название темы, на которую хотите подписаться. Ваш клиент может подписаться на несколько клиентов, вызвав подписку более одного раза.

```
mqtt.subscribe("test/string");  
mqtt.subscribe("test/binary");  
mqtt.subscribe("test/json");
```

Сообщения доставляются в функцию обратного вызова `onMessage` для всех тем, на которые подписан ваш клиент. Аргумент темы — это название темы, а аргумент данных — полное сообщение.

```
mqtt.onMessage = function(topic, data) {
    trace(`received message on topic "${topic}"\n`);
}
```

Аргумент данных всегда предоставляется в двоичной форме, как `ArrayBuffer`. Если вы знаете, что сообщение представляет собой строку, вы можете преобразовать ее в строку; если вы знаете, что это строка JSON, вы можете преобразовать ее в объект JavaScript.

```
data = String.fromArrayBuffer(data);
data = JSON.parse(data);
```

`String.fromArrayBuffer` — это функция XS, упрощающая работу приложений с двоичными данными. Существует функция `parallelArrayBuffer.fromString`. Они не являются частью стандарта языка JavaScript.

Публикация в теме

Чтобы отправить сообщение в тему, вызовите `publish` с помощью строки или `ArrayBuffer`:

```
mqtt.publish("test/string", "hello");
mqtt.publish("test/binary", Uint8Array.of(1, 2, 3).buffer);
```

Чтобы опубликовать JSON, сначала преобразуйте его в строку:

```
mqtt.publish("test/json", JSON.stringify({
    message: "hello",
    version: 1
})));
```


SNTP

Простой протокол сетевого времени, или SNTP, — это упрощенный способ получения текущего времени. Ваш компьютер, вероятно, использует SNTP (или его родителя, NTP) для установки времени за кулисами. В отличие от вашего устройства Интернета вещей, ваш компьютер также имеет часы реального времени с резервным питанием от батареи, поэтому он всегда знает текущее время. Если вам нужно текущее время на устройстве IoT, вам нужно получить его. Если вы используете метод подключения к Wi-Fi из командной строки, текущее время извлекается после установления соединения Wi-Fi, если вы укажете сервер времени в командной строке.

```
> mcconfig -d -m -p esp ssid="my wi-fi" sntp="pool.ntp.org"
```

При подключении к Wi-Fi с помощью кода вам также необходимо написать код для установки часов вашего IoT-устройства. Вы получаете текущее время с помощью протокола SNTP, который реализован в модуле `sntp`, и устанавливаете время устройства с помощью модуля времени.

```
import SNTP from "sntp";
import Time from "time";
```

В листинге 3-35 показан пример `$EXAMPLES/ch3-network/sntp`, запрашивающий текущее время с сервера времени по адресу `pool.ntp.org`. Когда время получено, время устройства устанавливается и отображается в формате UTC (Всемирное координированное время). в консоли отладки. Экземпляр SNTP закрывается, чтобы освободить ресурсы, которые он использует, поскольку он больше не нужен.

Листинг 3-35.

```
new SNTP({
  host: "pool.ntp.org"
}),
function(msg, value) {
  if (SNTP.time !== msg)
    return;
```

```

        Time.set(value);
        trace("UTC time now: ",
              (new Date).toUTCString(), "\n");
    }
);

```

Большинство продуктов IoT хранят список из нескольких серверов SNTP для ситуаций, когда один из них недоступен. Класс SNTP поддерживает этот сценарий без необходимости создания дополнительных экземпляров класса SNTP. См. пример `examples/network/sntp` в Moddable SDK, чтобы узнать, как использовать эту функцию аварийного переключения.

Расширенные темы

В этом разделе представлены две сложные темы: как превратить ваше устройство в частную базовую станцию Wi-Fi и как использовать промисы JavaScript с сетевыми API.

Создание точки доступа Wi-Fi

Иногда вы не хотите подключать свой продукт IoT ко всему Интернету, но хотите, чтобы люди подключались к вашему устройству, чтобы настроить его или проверить его состояние. В других случаях вы хотите подключить свое устройство к Интернету, но у вас еще нет имени и пароля для точки доступа Wi-Fi. В обеих этих ситуациях решением может стать создание частной точки доступа Wi-Fi. Многие микроконтроллеры IoT (включая ESP32 и ESP8266) могут быть не только клиентом Wi-Fi, который подключается к другим точкам доступа, но и точкой доступа.

Вы можете превратить свое IoT-устройство в точку доступа с помощью вызова статического метода `accessPoint` класса `WiFi`:

```
WiFi.accessPoint({
    ssid: "South Village"
});
```

Свойство `ssid` определяет имя точки доступа и является единственным обязательным свойством. Как показано в листинге 3-36, дополнительные свойства позволяют вам установить пароль, выбрать канал Wi-Fi для использования и скрыть точку доступа, чтобы она не появлялась при сканировании Wi-Fi.

Листинг 3-36.

```
WiFi.accessPoint({
    ssid: "South Village",
    password: "12345678",
    channel: 8,
    hidden: false
});
```

Устройство является либо точкой доступа, либо клиентом точки доступа. Это не может быть и то, и другое одновременно, поэтому, как только вы войдете в режим точки доступа, вы не сможете получить доступ к Интернету.

Вы можете предоставить веб-сервер на своей точке доступа, как показано ранее в разделе «Ответ на запрос». В листинге 3-37 из примера `$EXAMPLES/ch3-network/accesspoint` импорт класса `HTTP Server` немного отличается, потому что он переименовывает или присваивает классу `HTTPServer`, чтобы избежать конфликта имен с сервером DNS (вводится по этому примеру).

Листинг 3-37.

```
import {Server as HTTPServer} from "http";

(new HTTPServer).callback = function(msg, value) {
  if (HTTPServer.prepareResponse === msg) {
    return {
      headers: ["Content-Type", "text/plain"],
      body: "hello"
    };
  }
}
```

Как другие устройства узнают адрес вашего веб-сервера, чтобы подключиться к нему? Вы можете запросить локальное имя с помощью mDNS. Но поскольку ваш продукт IoT является точкой доступа, он теперь также является маршрутизатором для сети, поэтому он может разрешать запросы DNS. Это означает, что всякий раз, когда устройство в сети ищет имя, например `www.example.com`, ваше приложение может направить запрос на ваш HTTP-сервер. В листинге 3-38 показан простой DNS-сервер, который делает именно это.

Листинг 3-38.

```
import {Server as DNSServer} from "dns/server";

new DNSServer(function(msg, value) {
  if (DNSServer.resolve === msg)
    return Net.get("IP");
});
```

Конструктор класса DNS-сервера принимает функцию обратного вызова в качестве единственного параметра. Функция обратного вызова вызывается с сообщением о разрешении всякий раз, когда любое устройство, подключенное к точке доступа, пытается разрешить имя aDNS. В ответ обратный вызов предоставляет свой собственный IP-адрес. Когда большинство компьютеров или телефонов

подключаются к новой точке Wi-Fi, они проверяют, подключены ли они к Интернету и требуется ли вход в систему. Когда эта проверка выполняется на вашей точке доступа, она вызывает вызов точки доступа вашего веб-сервера для отображения веб-страницы. В этом примере будет просто приветствие, но вы можете изменить это, чтобы показать статус устройства, настроить Wi-Fi или что-то еще, что вам нравится.

Promises и асинхронные функции

Промисы (Обещания — это функция JavaScript, упрощающая программирование с помощью функций обратного вызова. Функции обратного вызова просты и эффективны, поэтому они используются во многих местах. Промисы могут улучшить читаемость кода, выполняющего последовательность шагов с использованием функций обратного вызова.

Этот раздел не предназначен для полного введения в промисы и асинхронные функции. Если вы не знакомы с этими функциями JavaScript, прочтите этот раздел, чтобы узнать, могут ли они оказаться полезными для ваших проектов; если это так, в Интернете доступно множество отличных ресурсов, которые помогут вам узнать больше, например <https://javascript.info/promise-basics>.

Выдержка из примера \$EXAMPLES/ch3-network/http-get-with-promise в листинге 3-39 основана на классе HTTP Request для реализации функции выборки, которая возвращает полный HTTP-запрос в виде строки.

Листинг 3-39.

```
function fetch(host, path = "/") {
  return new Promise((resolve, reject) => {
    let request = new Request({host, path, response: String});
    request.callback = function(msg, value) {
      if (Request.responseComplete === msg)
        resolve(value);
      else if (Request.error === msg)
        reject(-1);
    }
  });
}
```

Реализация функции выборки сложна и требует глубокого понимания того, как промисы работают в JavaScript. Но использовать функцию выборки легко (листинг 3-40).

Листинг 3-40.

```
function httpTrace(host, path) {  
    fetch(host, path)  
        .then(body => trace(body, "\n"))  
        .catch(error => trace("http get failed\n"));  
}
```

Читая код `httpTrace`, вы можете представить, что HTTP-запрос происходит синхронно, но это не так, поскольку все сетевые операции неблокируются. Стрелочные функции, переданные в вызовы `.then` и `.catch`, выполняются после завершения запроса — `.then`, если вызов выполнен успешно, или `.catch`, если он завершился неудачно.

Последние версии JavaScript предоставляют другой способ написания этого кода: как асинхронную функцию. В листинге 3-41 показан вызов `fetch`, переписанный в виде асинхронной функции. Код выглядит как обычный JavaScript, за исключением ключевых слов `async` и `await`.

Листинг 3-41.

```
async function httpTrace(host, path) {  
    try {  
        let body = await fetch(host, path);  
        trace(body, "\n");  
    }  
    catch {  
        trace("http get failed\n");  
    }  
}
```

Функция `httpTrace` является асинхронной, поэтому при вызове она возвращается немедленно. Ключевое слово `await` перед вызовом `fetch` сообщает языку JavaScript, что когда `fetch` возвращает обещание, выполнение `httpTrace` должно быть приостановлено до тех пор, пока обещание не будет готово (разрешено или отклонено).

Промисы и асинхронные функции — мощные инструменты, и они используются в коде JavaScript для гораздо более мощных систем, включая веб-серверы и компьютеры. Они доступны для ваших IoT-проектов даже на устройствах с ограниченными ресурсами, поскольку вы используете движок XS JavaScript. Тем не менее, функции обратного вызова предпочтительнее в большинстве ситуаций, поскольку они требуют меньше кода, выполняются быстрее и используют меньше памяти. При создании проекта вам нужно будет решить, перевешивает ли удобство их использования дополнительные используемые ресурсы.

Заключение

В этой главе вы узнали о различных способах взаимодействия вашего IoT-устройства по сети. Различные протоколы, описанные в этой главе, следуют одному и тому же базовому шаблону API:

- Класс протокола предоставляет конструктор, который принимает словарь для настройки соединения.
- Функции обратного вызова доставляют информацию из сети в ваше приложение.
- Коммуникация всегда асинхронна, чтобы избежать блокировки, что является важным фактором для продуктов IoT, которые не всегда могут позволить себе роскошь нескольких потоков выполнения.
- Обратные вызовы можно превратить в обещания с помощью функций `smallhelper`, чтобы приложения могли использовать асинхронные функции в современном JavaScript.

Вы, как разработчик продукта IoT, должны решить, какие методы связи он поддерживает. Необходимо учитывать множество факторов. Если вы хотите, чтобы ваше устройство обменивалось данными с облаком, HTTP, WebSocket и MQTT — все возможные варианты, и все они поддерживают безопасную связь с использованием TLS. Для прямой связи между устройствами mDNS является хорошей отправной точкой, позволяющей устройствам рекламировать свои услуги, а HTTP — это упрощенный способ обмена сообщениями между устройствами.

Конечно, ваш продукт не должен выбирать только один сетевой протокол для связи. Начиная с примеров в этой главе, вы готовы попробовать различные инструменты `protools`, чтобы найти то, что лучше всего подходит для нужд вашего устройства.

ГЛАВА 4

Bluetooth с низким энергопотреблением (BLE)

Есть много способов включить беспроводную связь между устройствами. В главе 3 представлено множество протоколов, которые работают через соединение Wi-Fi для связи с устройствами в любой точке мира. В этой главе основное внимание уделяется Bluetooth с низким энергопотреблением, или BLE, беспроводной связи, широко используемой между двумя устройствами, находящимися в непосредственной близости друг от друга. Продукты выбирают использование BLE вместо Wi-Fi, если особенно важно минимизировать потребление энергии, например, в продуктах с батарейным питанием. , и когда прямая связь с другим устройством, например мобильным телефоном, является приемлемой альтернативой доступу в Интернет. Многие продукты IoT используют BLE, от мониторов сердечного ритма до электрических зубных щеток и духовок. Производители продуктов часто предлагают мобильное приложение или настольное приложение-компаньон для мониторинга или управления этими продуктами.

BLE — это версия 4 стандарта Bluetooth, впервые представленная в 2010 году. Оригинальный Bluetooth был стандартизирован в 2000 году для отправки потоков данных на короткие расстояния. BLE значительно снижает энергопотребление оригинального Bluetooth, позволяя ему работать намного дольше на одном заряде батареи. BLE достигает этого частично за счет уменьшения объема передаваемых данных. Передача на более короткие расстояния также требует меньше энергии; Устройства BLE обычно имеют радиус действия не более 100 метров, тогда как Wi-Fi имеет гораздо больший радиус действия. Более низкое энергопотребление и стоимость BLE делают его подходящим для многих продуктов IoT.

Используя информацию в этой главе, вы можете создавать свои собственные устройства BLE, работающие на микроконтроллере.

Примечание Примеры в этой главе относятся к ESP32. Если вы попытаетесь собрать их для ESP8266, сборка завершится ошибкой, потому что у ESP8266 нет аппаратного обеспечения BLE. Однако примеры работают на других устройствах со встроенным BLE, поддерживаемым MObdable SDK, включая QUalcOmm QCA4020 и BUIe GeckO от SilicON Labs.

Основы BLE

Если вы новичок в работе с BLE, информация в этом разделе важна, так как в ней объясняются концепции, используемые в остальной части этой главы. Если вы знакомы с BLE, подумайте о том, чтобы быстро просмотреть этот раздел, чтобы ознакомиться с терминологией, используемой в этой книге, и с тем, как она связана с BLE API в MObdable SDK.

Центральные и периферийные устройства GAP

Общий профиль доступа, или *GAP*, определяет, как устройства рекламируют себя, как они устанавливают соединения друг с другом и безопасность. *GAP* определяет две основные роли: центральную и периферийную.

Центральный сканирует устройства, действующие как периферийные устройства, и инициирует запросы на установление нового соединения с периферийными устройствами. Устройство, выступающее в роли центрального устройства, обычно имеет относительно высокую вычислительную мощность и большой объем памяти (например, смартфон, планшет или компьютер), в то время как периферийные устройства часто бывают небольшими и питаются от батареи. Периферийные устройства рекламируют себя и принимают запросы на установление соединения.

Спецификация BLE позволяет центральному устройству подключаться к нескольким периферийным устройствам, а периферийное устройство — к нескольким центральным устройствам. Центральное устройство обычно подключается к нескольким периферийным устройствам одновременно. Например, вы можете использовать свой смартфон для подключения к пульсометру, умным часам и фонарям. Периферийное устройство редко может подключаться более чем к одному центральному устройству одновременно; большинство периферийных устройств не допускают множественных одновременных подключений. API BLE модифицируемого SDK позволяет периферийному устройству одновременно подключаться к одному центральному устройству.

Клиенты и серверы GATT

Общий профиль атрибутов, или GATT, определяет способ передачи данных устройствами BLE туда и обратно после того, как между ними установлено соединение — отношения клиент-сервер.

Клиент GATT — это устройство, которое получает доступ к данным с удаленного сервера GATT, отправляя запросы на чтение/запись. Сервер GATT — это устройство, которое хранит данные локально, получает запросы на чтение/запись и уведомляет удаленного клиента GATT об изменении значений его характеристик. В этой главе термин «Сервер» используется для обозначения сервера GATT, а «Клиент» означает клиента GATT.

GAP против GATT

Во многих руководствах по BLE термины «Центральный» и «Клиент» используются как синонимы, а термины «Периферийный» и «Сервер» — как синонимы. Это связано с тем, что центральные устройства обычно берут на себя роль клиента, а периферийные устройства обычно берут на себя роль сервера. Однако спецификация BLE говорит, что либо центральные, либо периферийные устройства могут брать на себя роль клиента, сервера или и того, и другого.

Центральный и периферийный — это термины, определенные GAP, которые говорят вам, как управляется соединение BLE. Клиент и сервер — это термины, определенные GATT, которые сообщают вам о хранении и потоке данных после установления соединения. GATT вступает в игру только после того, как процесс рекламы и подключения, определенный GAP, ¹⁸⁷ завершен.

Профили, услуги и характеристики

GATT также определяет формат данных с иерархией профилей, услуг и характеристик. Как показано на рис. 4-1, верхний уровень иерархии — это профиль.

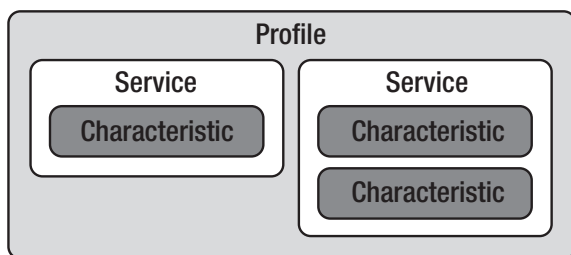


Рис. 4-1. Иерархия профилей GATT

Профили

Профиль определяет конкретное использование BLE для связи между несколькими устройствами, включая роли задействованных устройств и их общее поведение. Например, стандартный профиль термометра здоровья определяет роли термометра (датчика) и коллектора; устройство Термометр измеряет температуру, а устройство Коллектор получает измерение температуры и другие данные от Термометра. В профиле указаны службы, экземпляры которых должен создавать термометр (служба Health Thermometer и служба информации об устройстве), и указано, что предполагаемое использование профиля — приложения для здравоохранения.

Профили не существуют на устройстве BLE; скорее, это спецификации, реализованные устройством BLE. Список официально принятых профилей BLE доступен на странице bluetooth.com/specifications/gatt. Когда вы внедряете свое собственное устройство BLE, рекомендуется проверить, доступен ли стандартный профиль, соответствующий потребностям вашего продукта. Если да, вы получите выгоду от взаимодействия с другими продуктами, поддерживающими стандарт, что сэкономит вам время на разработку нового профиля.

Службы

Служба — это набор характеристик, описывающих поведение части устройства BLE. Например, служба Health Thermometer предоставляет данные о температуре с устройства Thermometer. Услуга может иметь одну или несколько характеристик и отличается UUID. Официально принятые сервисы BLE имеют 16-битные UUID. Службе Health Thermometer присвоен номер 0x1809. Вы можете создавать свои собственные сервисы, присваивая им 128-битный UUID.

Услуги рекламируются устройством BLE. Список официально принятых сервисов BLE доступен по адресу bluetooth.com/specifications/gatt/services.

Характеристики

Характеристика — это отдельное значение или точка данных, которая предоставляет информацию об услуге GATT. Формат данных зависит от характеристики; например, характеристика измерения частоты сердечных сокращений, используемая службой частоты сердечных сокращений, обеспечивает измерение частоты сердечных сокращений в виде целого числа, а характеристика имени устройства предоставляет имя устройства в виде строки.

Список официально принятых характеристик BLE доступен по адресу bluetooth.com/specifications/gatt/characteristics. Как и в случае с сервисами, официально принятые характеристики BLE имеют 16-битные UUID, и вы можете создать свои собственные с 128-битным UUID.

The BLE API of the Moddable SDK

Moddable SDK не имеет отдельных классов в своем BLE API для ролей, определенных GAP и GATT. Вместо этого он предоставляет функции для центральных устройств и клиентов GATT в одном классе BLEClient, а также функции для периферийных устройств и серверов GATT в одном классе BLEServer. Эта организация класса отражает две наиболее распространенные конфигурации устройств BLE: устройства, которые действуют как центральные и берут на себя роль клиента, и устройства, которые действуют как периферийные устройства и берут на себя роль сервера.

Класс клиента BLE

Класс `BLEClient` предоставляет функции, которые вы используете для создания `Centrals` и клиентов `GATT`. Функции для `Centrals` выполняют следующие операции:

1. Поиск периферийных устройств.
2. Инициировать запросы на установление соединения с периферийными устройствами.

Функции для клиентов `GATT` выполняют следующие операции:

1. Найдите интересующие вас услуги `GATT`
2. Найдите интересующие характеристики в каждой услуге.
3. Читайте, записывайте и включайте уведомления для характеристик в каждой службе.

Вы создаете подкласс класса `BLEClient` для реализации определенного устройства BLE, которое поддерживает операции, необходимые вашему устройству. Подклассы вызывают методы класса `BLEClient`, чтобы инициировать предыдущие операции. Все операции BLE, выполняемые `BLEClient`, являются асинхронными, чтобы избежать блокировки выполнения на неопределенный период времени. Следовательно, экземпляры класса `BLEClient` получают результаты через обратные вызовы. Например, класс `BLEClient` имеет метод `startScanning`, который вы вызываете, чтобы начать сканирование периферийных устройств, и обратный вызов `onDiscovered`, который автоматически вызывается при обнаружении периферийного устройства.

Вам нужно только реализовать обратные вызовы, необходимые для работы с периферийными устройствами, службами и характеристиками, которые требуются вашему устройству.

Класс BLE-сервера

Класс `BLEServer` предоставляет функции, которые вы используете для создания периферийных устройств и серверов GATT. Функции для периферийных устройств выполняют следующие операции:

1. Рекламируйте так, чтобы `entrals` мог обнаружить `Peripheral` (периферийное устройство).
2. Принимать запросы на подключение от `Central`.

Функции серверов GATT выполняют следующие операции:

1. Развертывание сервисов.
2. Отвечать на характерные запросы на чтение и запись от клиента.
3. Принимать запросы на уведомление об изменении значения характеристики от клиента.
4. Уведомлять Клиента об изменении значения характеристики.

Вы можете внедрить стандартные профили BLE, такие как частота сердечных сокращений, или собственный настраиваемый профиль для поддержки уникальных возможностей вашего продукта. В обоих случаях вы сначала определяете службы GATT в файлах JSON, а затем подклассируете класс `BLEServer` для реализации конкретных устройств BLE. Подклассы вызывают методы класса `BLEServer` для инициирования предыдущих операций. Все операции BLE, выполняемые `BLEServer`, являются асинхронными, чтобы избежать блокировки выполнения на неопределенный период времени. Следовательно, экземпляры класса `BLEServer` получают результаты через обратные вызовы.

Установка BLE-хоста

Примеры в этой главе устанавливаются по шаблону, описанному в главе 1: вы устанавливаете хост на свое устройство с помощью `mcconfig`, затем устанавливаете примеры приложений с помощью `mcrun`.

Хост находится в каталоге `$EXAMPLES/ch4-ble/host`. Перейдите в этот каталог из командной строки и установите его с помощью `mcconfig`.

Создание BLE-сканера

Пример \$EXAMPLES/ch4-ble/scanner реализует Central, который сканирует близлежащие периферийные устройства и отслеживает их имена до консоли. Он реализован с использованием класса BLEClient. В листинге 4-1 показана большая часть исходного кода для этого примера.

Листинг 4-1.

```
class Scanner extends BLEClient {
  onReady() {
    this.startScanning();
  }
  onDiscovered(device) {
    let scanResponse = device.scanResponse;
    let completeName = scanResponse.completeName;
    if (completeName)
      trace(`${completeName}\n`);
  }
}
```

Класс Scanner реализует два обратных вызова BLEClient:

- Обратный вызов onReady вызывается, когда стек BLE готов к использованию. В этом примере обратный вызов onReady вызывает startScanning, чтобы включить сканирование ближайших периферийных устройств.
- Обратный вызов onDiscovered вызывается один или несколько раз для каждого обнаруженного периферийного устройства. В этом примере обратный вызов onDiscovered отслеживает имя обнаруженного периферийного устройства до консоли.

В этом простом примере ваш центральный обнаруживает периферийные устройства вокруг вас и сообщает вам их имена. Теперь вы готовы сделать еще один шаг: следующий пример демонстрирует, как использовать другие функции класса BLEClient для создания устройства BLE, которое взаимодействует с виртуальным периферийным устройством.

Создание двусторонней связи

Пример \$EXAMPLES/ch4-ble/text-client реализует Central, который подключается к Peripheral и получает текстовые данные через уведомления об изменении значения характеристики.

Чтобы увидеть, как работает пример, вам понадобится периферийное устройство, которое предоставляет характеристику текстовых данных. Вы можете создать его с помощью Bluefruit, мобильного приложения, доступного бесплатно на устройствах iOS и Android. Чтобы создать периферийное устройство, выполните следующие шаги, показанные на рисунках 4-2 и 4-3:

1. Загрузите и откройте Bluefruit и войдите в периферийный режим. В разделе РЕКЛАМНАЯ ИНФОРМАЦИЯ измените поле Local Name на esp.

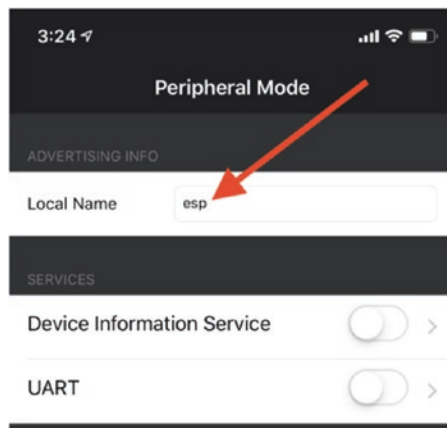


Рис. 4-2. Периферийный режим в Bluefruit

ГЛАВА 4 BLUETOOTH LOW ENERGY (BLE)

2. Убедитесь, что служба UART включена.

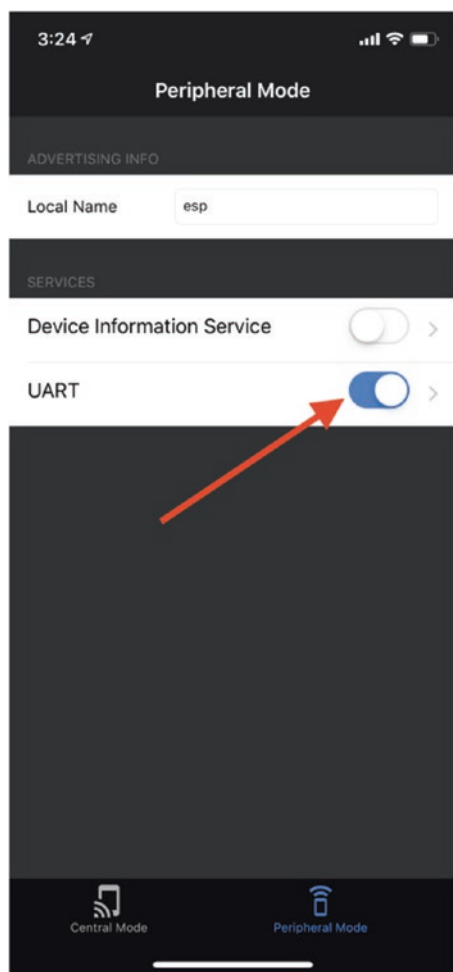


Рис. 4-3. Служба UART включена

В следующих разделах объясняется код, который выполняется на ESP32 для реализации центрального устройства.

Подключение к периферийному устройству

Константы в верхней части приложения соответствуют имени устройства, которое вы установили в Bluefruit, а также службе и характерному UUID, используемому службой UART:

```
const PERIPHERAL_NAME = 'esp';
const SERVICE_UUID = uuid`6E400001B5A3F393E0A9E50E24DCCA9E`;
const CHARACTERISTIC_UUID = uuid`6E400003B5A3F393E0A9E50E24DCCA9E`;
```

Как и в примере со сканером, в этом примере реализованы обратные вызовы `onReady` и `onDiscovered`, как показано в листинге 4-2. Но вместо того, чтобы просто отслеживать имена устройств на консоли, этот пример проверяет имя каждого обнаруженного периферийного устройства, чтобы увидеть, соответствует ли оно константе `PERIPHERAL_NAME`. Если это так, он останавливает поиск периферийных устройств и вызывает метод подключения, который инициирует соединение. запрос между `BLEClient` и целевым периферийным устройством.

Листинг 4-2.

```
class TextClient extends BLEClient {
  onReady() {
    this.startScanning();
  }
  onDiscovered(device) {
    if (PERIPHERAL_NAME ===
        device.scanResponse.completeName) {
      this.stopScanning();
      this.connect(device);
    }
  }
  ...
}
```

Аргумент для подключения — это экземпляр класса `Device`, представляющий одно периферийное устройство. `BLEClient` автоматически создает экземпляры класса `Device` при обнаружении периферийного устройства; приложения не создают их экземпляры напрямую. Однако приложения взаимодействуют с экземплярами класса `Device` напрямую, например, вызывая методы для выполнения службы GATT и обнаружения характеристик.

Обратный вызов `onConnected`

Метод `onConnected` — это обратный вызов, который вызывается, когда центральный сервер подключается к периферийному устройству. В этом примере вызывается метод `discoverPrimaryService` объекта устройства для получения основной службы GATT от периферийного устройства. Аргумент `DiscoverPrimaryService` — это UUID службы, которую необходимо обнаружить.

```
onConnected(device) {  
    device.discoverPrimaryService(SERVICE_UUID);  
}
```

Вы можете обнаружить все основные службы периферийного устройства, используя метод `DiscoverAllPrimaryServices`. Например, обратный вызов `onConnected` может быть записан следующим образом:

```
onConnected(device) {  
    device.discoverAllPrimaryServices();  
}
```

Обратный вызов `onServices`

Метод `onServices` — это обратный вызов, который вызывается после завершения обнаружения службы. Аргумент `services` представляет собой массив объектов службы — экземпляров класса `Service`, — каждый из которых обеспечивает доступ к одной службе. Если `discoverPrimaryService` был вызван для поиска одной службы, массив `services` будет содержать только одну найденную службу.

Как показано в листинге 4-3, в этом примере проверяется, предоставляет ли периферийное устройство сервис с UUID, совпадающим с UUID, определенным константой `SERVICE_UUID`. Если это так, он вызывает метод `discoverCharacteristic` объекта службы для поиска характеристики службы с UUID, совпадающим с тем, который определен константой `CHARACTERISTIC_UUID`.

Листинг 4-3.

```
onServices(services) {
  let service = services.find(service =>
    service.uuid.equals(SERVICE_UUID));
  if (service) {
    trace(`Found service\n`);
    service.discoverCharacteristic(CHARACTERISTIC_UUID);
  }
  else
    trace(`Service not found\n`);
}
```

Вы можете обнаружить все характеристики службы, используя метод `DiscoverAllCharacteristics`. Например, обратный вызов `onServices` может заменить строку, вызывающую `discoverCharacteristic`, следующей строкой:

```
service.discoverAllCharacteristics();
```

Обратный вызов `onCharacteristics`

Метод `onCharacteristics` — это обратный вызов, который вызывается после завершения обнаружения характеристик. Аргумент характеристик представляет собой массив объектов-характеристик — экземпляров класса `Characteristic`, — каждый из которых обеспечивает доступ к одной характеристике службы.

Если метод `discoverCharacteristic` был вызван для поиска одной характеристики, массив характеристик содержит эту единственную найденную характеристику.

Когда желаемая характеристика найдена, пример вызывает метод `enableNotifications` объекта характеристики, чтобы включить уведомления при изменении значения характеристики, как показано в листинге 4-4.

Листинг 4-4.

```
onCharacteristics(characteristics) {  
    let characteristic = characteristics.find(characteristic =>  
        characteristic.uuid.equals(CCHARACTERISTIC_UUID));  
    if (characteristic) {  
        trace(`Enabling notifications\n`);  
        characteristic.enableNotifications();  
    }  
    else  
        trace(`Characteristic not found\n`);  
}
```

Если вы правильно настроили периферийное устройство, вы увидите следующие сообщения в консоли отладки при запуске текстового клиентского приложения:

Found service

Enabling notifications

Получение уведомлений

После включения уведомлений вы можете отправлять уведомления Клиенту, изменяя значение характеристики Периферийного устройства со своего смартфона. Чтобы изменить значение, коснитесь кнопки UART. Это приведет вас к экрану, показанному на рис. 4-4. Введите текст в поле ввода в нижней части экрана и нажмите «Send - Отправить», чтобы обновить значение характеристики.

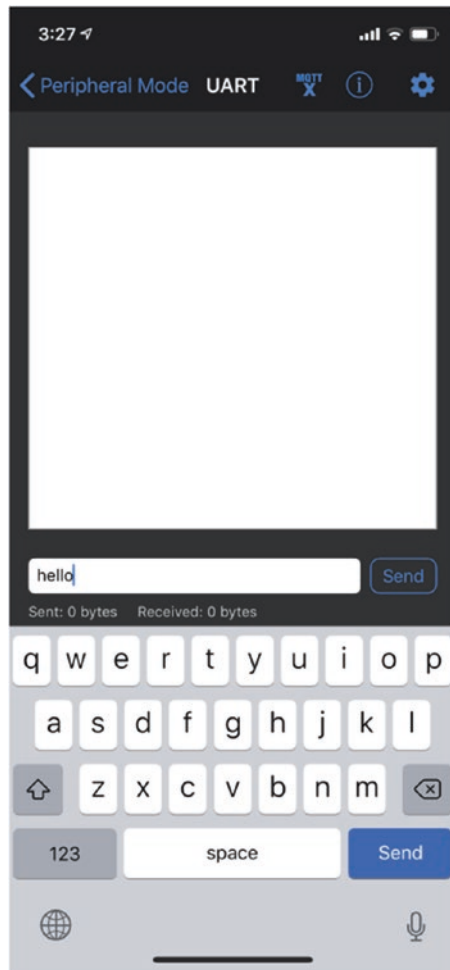


Рис. 4-4. Экран UART в Bluefruit

Значение характеристики доставляется клиенту посредством обратного вызова `onCharacteristicNotification` в `ArrayBuffer`. В этом примере предполагается, что значение является строкой, поэтому он преобразует `ArrayBuffer` в строку с помощью `String.fromArrayBuffer` (функция XS, упрощающая работу приложений с двоичными данными). Есть параллельный `ArrayBuffer.fromString`. Они не являются частью стандарта языка JavaScript.

```
onCharacteristicNotification(characteristic, buffer) {
    trace(String.fromArrayBuffer(buffer)+"\n");
}
```

Создание пульсометра

Теперь, когда вы знаете основы реализации клиента, который получает уведомления от сервера, этот пример покажет вам, как использовать функции класса `BLEServer` для реализации периферийного устройства, которое берет на себя роль сервера после подключения к центральному.

Пример `$EXAMPLES/ch4-ble/hrm` рекламирует стандартные сервисы Heart Rate и Battery, принимает запросы на подключение от Centrals, отправляет уведомления для смоделированных значений сердечного ритма и отвечает на запросы чтения от клиента для смоделированного уровня заряда батареи. Следующие несколько разделов объясняют, как это реализовано с использованием класса `BLEServer`.

Определение и развертывание сервисов

Сервисы GATT определяются в файлах JSON, расположенных в каталоге `bleservices` хоста. JSON автоматически преобразуется в собственный код для конкретной платформы во время сборки, а скомпилированный объектный код связывается с приложением.

Каждая служба GATT определяется в собственном файле JSON. В листинге 4-5 показана стандартная служба сердечного ритма.

Листинг 4-5.

```
{
  "service": {
    "uuid": "180D",
    "characteristics": {
      "bpm": {
```



```

        "uuid": "2A37",
        "maxBytes": 2,
        "type": "Array",
        "permissions": "read",
        "properties": "notify"
    }
}
}
}

```

Ниже приведены пояснения некоторых важных свойств:

- Свойство `uuid` объекта сервиса — это номер, присвоенный сервису спецификацией GATT. Служба сердечного ритма имеет UUID 180F.
- Объект характеристик описывает каждую характеристику, поддерживаемую службой. Каждое непосредственное свойство является именем характеристики. В этом примере есть только одна характеристика: `bpm`, что означает количество ударов в минуту.
- Свойство `uuid` объекта характеристики — это уникальный номер, присвоенный характеристике спецификацией GATT. Характеристика ударов в минуту службы сердечного ритма имеет UUID 2A37.
- Свойство `type` указывает тип значения характеристики, используемого в вашем коде JavaScript. Класс `BLEServer` использует значение свойства `type` для преобразования двоичных данных, передаваемых клиентом, в типы JavaScript. Это избавляет ваш серверный код от работы по преобразованию между различными типами данных (`ArrayBuffer`, `String`, `Number` и т. д.).

- Свойство разрешений определяет, доступна ли характеристика только для чтения, только для записи или для чтения/записи, а также требуется ли для доступа к характеристике зашифрованное соединение. Свойство ударов в минуту доступно только для чтения, поскольку в пульсометре количество ударов в минуту определяется показаниями датчика и, следовательно, не может быть записано клиентом. Разрешение на чтение указывает, что Клиент может читать характеристику через незашифрованное или зашифрованное соединение; используйте `readEncrypted`, когда значение доступно только через зашифрованное соединение. Точно так же используйте `write` или `writeEncrypted` для разрешений на запись. Чтобы указать, что характеристика поддерживает как чтение, так и запись, включите значение чтения и записи в строку разрешений, разделенную запятой, например, «`readEncrypted,writeEncrypted`».
- Свойство `properties` определяет свойства характеристики. Это может быть чтение, запись, уведомление, указание или их комбинация (через запятую). Значения чтения и записи разрешают чтение и запись значения характеристики, уведомление позволяет Серверу уведомлять Клиента об изменениях значения характеристики без его запроса и без подтверждения того, что уведомление было получено, а значение указания аналогично уведомлению, за исключением того, что оно требует подтверждения того, что уведомление был получен до того, как может быть отправлено другое указание.

Как только стек BLE завершает свою инициализацию, он вызывает обратный вызов `onReady`. Реализация `onReady` в примере `hrm` иницирует рекламу, позволяя клиентам обнаружить ее услуги. В следующем разделе объясняется, как подкласс управляет, когда реклама активна.

Реклама

Периферийные устройства транслируют рекламные данные, чтобы заявить о себе. Класс `BLEServer` имеет метод `startAdvertising` для начала широкоэмитательной рассылки рекламных пакетов и метод `stopAdvertising` для остановки.

Пример `hrm` вызывает `startAdvertising`, когда стек BLE готов к использованию, а также при потере соединения с `Central`. Когда вызывается `startAdvertising`, периферийное устройство передает значение своих флагов типа рекламных данных, свое имя и свои службы (`Heart Rate` и `Battery`), как показано в листинге 4-6. UUID для служб `Heart Rate` и `Battery` взяты из спецификации GATT.

Листинг 4-6.

```
this.startAdvertising({
  advertisingData: {
    flags: 6,
    completeName: this.deviceName,
    completeUUID16List: [uuid`180D`, uuid`180F`]
  }
});
```

Когда подключение к центральному успешно установлено, периферийное устройство прекращает отправку рекламных пакетов, так как поддерживает только одно соединение в каждый момент времени:

```
onConnected() {
  this.stopAdvertising();
}
```

При потере связи `Peripheral` снова запускает рекламу.

Объявление BLE может содержать дополнительные данные, например, для реализации маяка BLE. Маяки BLE рекламируют данные для просмотра многими центральными устройствами без подключения к ним. Код в листинге 4-7 взят из примера `examples/network/ble/uri-beacon` в `Moddable SDK`, который реализует `UriBeacon`, рекламирующий

модифицируемый веб-сайт. UUID здесь взят из спецификации Assigned Numbers (см. bluetooth.com/specifications/assigned-numbers/16-bit-uuids-for-members). Метод `encodeData` кодирует URI в формате, указанном в спецификации UriBeacon. Исходный код см. в примере `uri-beacon`.

Листинг 4-7.

```
this.startAdvertising({
  advertisingData: {
    completeUUID16List: [uuid`FED8`],
    serviceDataUUID16: {
      uuid: uuid`FED8`,
      data: this.encodeData("http://www.moddable.com")
    }
  }
});
```

Рекламные данные только передают данные; нет возможности ответить. Двусторонняя связь требует, чтобы одно устройство выполняло роль клиента GATT, а другое устройство — роль сервера GATT. Прежде чем это может произойти, должен быть завершен процесс подключения, определенный GAP.

Установление соединения

Как только периферийное устройство сердечного ритма начинает рекламировать, оно ожидает запроса Centralto на подключение к нему. Вы можете использовать любое из множества мобильных приложений для создания центра, который делает это. В этом разделе вы будете использовать LightBlue, который доступен бесплатно на устройствах iOS и Android. LightBlue имеет центральный режим, который позволяет вам сканировать, подключаться и отправлять запросы на чтение/запись на периферийные устройства. Вы можете использовать его в качестве клиента для своего периферийного устройства, выполнив следующие действия:

1. Запустите пример на вашем ESP32.
2. Загрузите и откройте программу Light Blue и подождите, пока не появится периферийное устройство пульсометра, как показано на рис. 4-5.

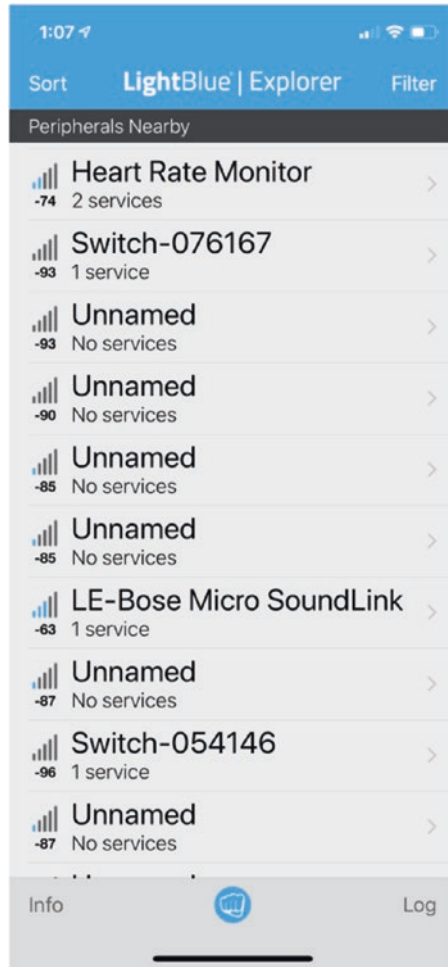


Рис. 4-5. Heart Rate Monitor - Периферийные устройства в списке периферийных устройств голубым цветом

3. Коснитесь **Rate Monitor Peripheral** (пульсометра), чтобы установить с ним соединение.

Листинг 4-8.

```
class HeartRateService extends BLEServer {  
    ...  
    onConnected() {  
        this.stopAdvertising();  
    }  
    ...  
}
```

Отправка уведомлений

Клиенты BLE могут запрашивать уведомления для характеристик, имеющих свойство уведомления, таких как характеристика ударов в минуту в этом примере. Когда уведомления включены, Серверы уведомляют Клиента об изменении значения характеристики, при этом Серверу не нужно запрашивать значение. Уведомления экономят энергию, что является ключевой особенностью проекта BLE, поскольку устраняют необходимость для клиента опрашивать сервер на предмет изменений характеристик.

Чтобы получать уведомления о смоделированной частоте сердечных сокращений в LightBlue, выполните следующие действия (как показано на рисунках 4-6, 4-7 и 4-8):

1. Коснитесь **Heart Rate Measurement** - характеристики измерения сердечного ритма.

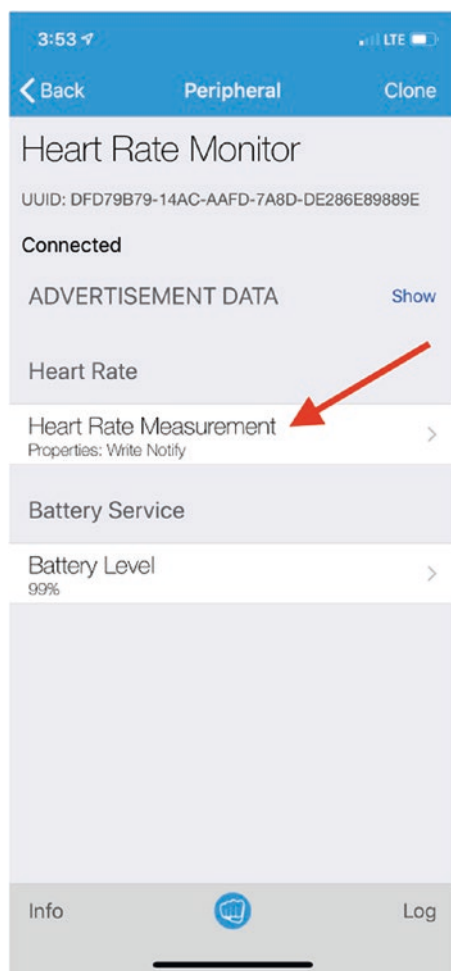


Рис. 4-6. Экран характеристик монитора сердечного ритма с кнопкой **Heart Rate Measurement** (измерение сердечного ритма).

2. Нажмите **Listen for notifications** ,чтобы включить уведомления.

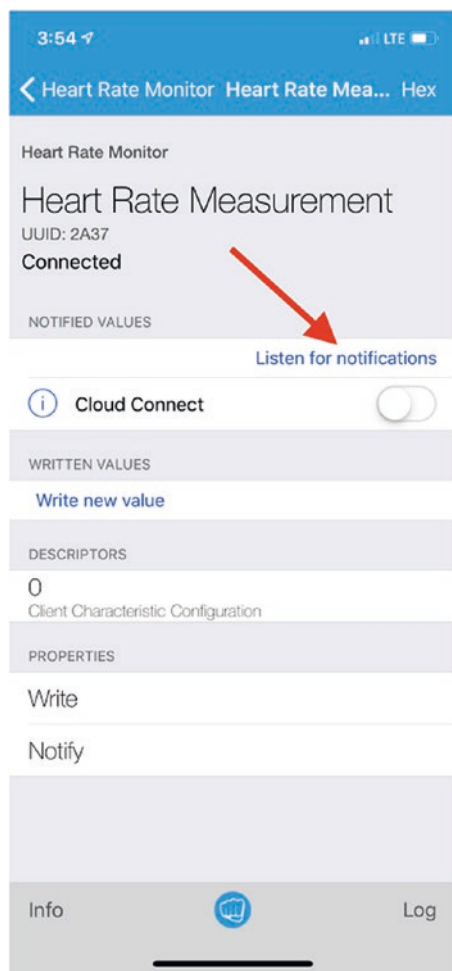


Рис. 4-7. Экран измерения сердечного ритма с кнопкой прослушивания уведомлений

- Посмотрите, как появляются смоделированные значения частоты сердечных сокращений.

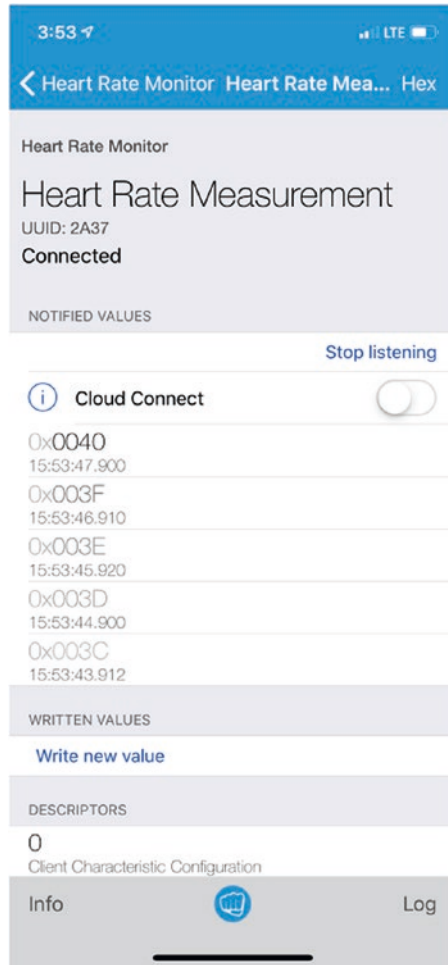


Рис. 4-8. Значения частоты сердечных сокращений, отображаемые в разделе NOTIFIED VALUES - УВЕДОМЛЕННЫЕ ЗНАЧЕНИЯ

Теперь давайте взглянем на код, который реализует уведомления для службы Heart Rate на стороне сервера. Метод `onCharacteristicNotifyEnabled` — это обратный вызов, который вызывается, когда клиент включает уведомления о характеристике. Метод `onCharacteristicNotifyDisabled` — это обратный вызов, который вызывается, когда клиент отключает

уведомления о характеристике. Аргумент характеристики для обоих является экземпляром класса `Characteristic`, который обеспечивает доступ к одной служебной характеристике.

Метод `onCharacteristicNotifyEnabled` (показанный в листинге 4-9) вызывает метод `notifyValue`, который отправляет уведомление об изменении значения характеристики подключенному клиенту с интервалом в 1000 миллисекунд (1 секунду). Он имитирует датчик сердечного ритма, хотя реальный пульсометр не будет отправлять периодические обновления; скорее, он отправит уведомление, когда значение действительно изменится.

Листинг 4-9.

```
onCharacteristicNotifyEnabled(characteristic) {
    this.bump = +1;
    this.timer = Timer.repeat(id => {
        this.notifyValue(characteristic, this.bpm);
        this.bpm[1] += this.bump;
        if (this.bpm[1] === 65) {
            this.bump = -1;
            this.bpm[1] = 64;
        }
        else if (this.bpm[1] === 55) {
            this.bump = +1;
            this.bpm[1] = 56;
        }
    }, 1000);
}
```

Метод `onCharacteristicNotifyDisabled` (листинг 4-10) завершает отправку уведомлений, вызывая метод `stopMeasurements`.

Листинг 4-10.

```

onCharacteristicNotifyDisabled(characteristic) {
    this.stopMeasurements();
}
...
stopMeasurements() {
    if (this.timer) {
        Timer.clear(this.timer);
        delete this.timer;
    }
    this.bpm = [0, 60]; // flags, beats per minute
}

```

Ответы на запросы на чтение

Клиенты могут запросить значение характеристик, которые поддерживают свойство чтения, например службу батареи в этом примере. Чтобы отправить запросы на чтение значения смоделированного уровня заряда батареи в LightBlue, выполните следующие действия (как показано на рисунках 4-9, 4-10 и 4-11):

1. Коснитесь характеристики уровня заряда батареи.

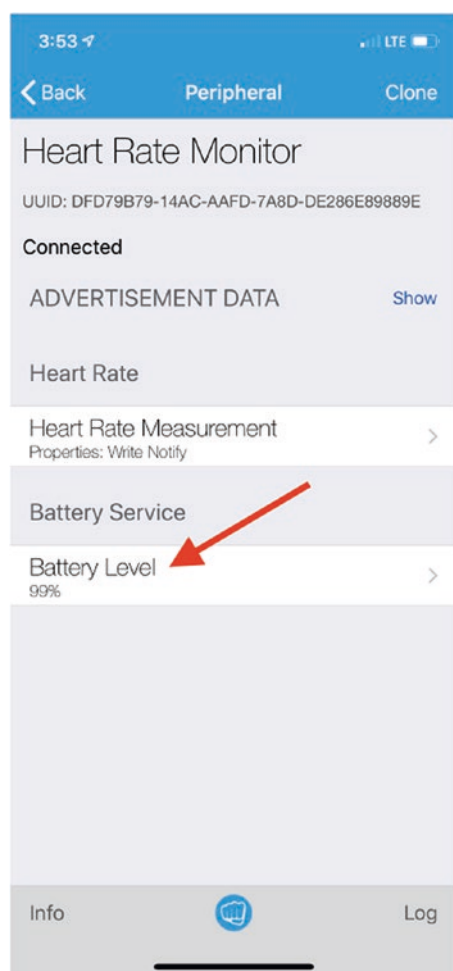


Рис. 4-9. Экран характеристик монитора сердечного ритма с кнопкой уровня заряда батареи

2. Нажмите «Read - Читать» еще раз.

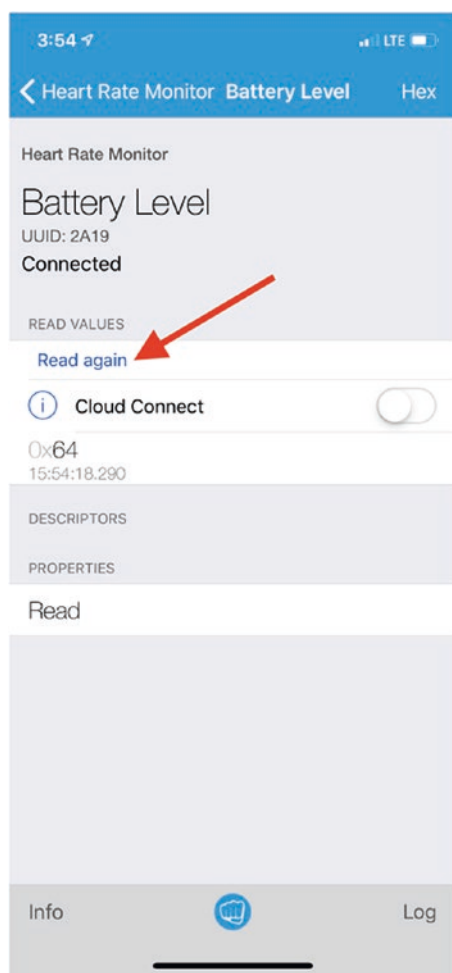


Рис. 4-10. Экран уровня заряда батареи с кнопкой «Read again - Повторить чтение»

3. Посмотрите, как появляется смоделированный уровень заряда батареи.

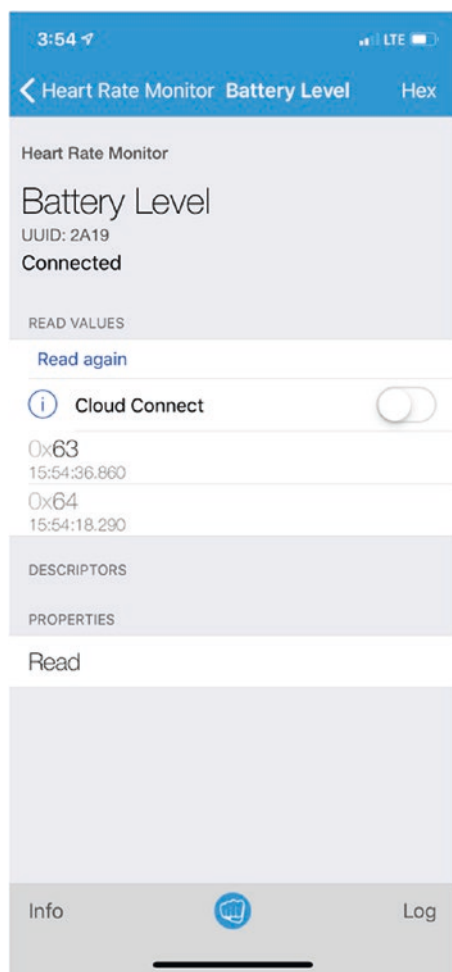


Рис. 4-11. Значение уровня заряда батареи отображается в разделе *READ VALUES - ЧИТАТЬ ЗНАЧЕНИЯ*

Теперь давайте посмотрим на код, который обрабатывает служебные уведомления об уровне заряда батареи. Метод `onCharacteristicRead` (листинг 4-11) представляет собой обратный вызов, который вызывается, когда клиент считывает значение характеристики службы по запросу. Экземпляр `BLEServer` отвечает за обработку запроса на чтение.

В этом примере уровень заряда батареи начинается со 100; каждый раз, когда он читается, обратный вызов возвращает значение и уменьшает его на 1.

Листинг 4-11.

```
onCharacteristicRead(params) {
    if (params.name === "battery") {
        if (this.battery === 0)
            this.battery = 100;
        return this.battery--;
    }
}
```

Установление безопасной связи

Moddable SDK поддерживает расширенные функции безопасности, представленные в версии 4.2 базовой спецификации Bluetooth: безопасные соединения LE с числовым сравнением, ввод пароля и методы сопряжения Just Works. Оба класса BLEClient и BLEServer имеют необязательное свойство securityParameters, которое запрашивает, чтобы устройства установили безопасное соединение LE. Используемый метод сопряжения зависит от возможностей и параметров устройств, установленных в свойстве securityParameters. Функции обратного вызова безопасности размещаются в классах BLEClient и BLEServer. В следующем разделе рассматривается простой пример.

Безопасный монитор сердечного ритма

Пример \$EXAMPLES/ch4-ble/hrm-secure — это безопасная версия примера \$EXAMPLES/ch4-ble/hrm, требующая ввода пароля для сопряжения.

Опять же, вы можете использовать LightBlue в качестве клиента. Выполните те же действия, что и раньше, и когда вам будет предложено ввести код от пульсометра (как показано на рис. 4-12), введите ключ доступа, отслеживаемый до консоли в xdebug (рис. 4-13).

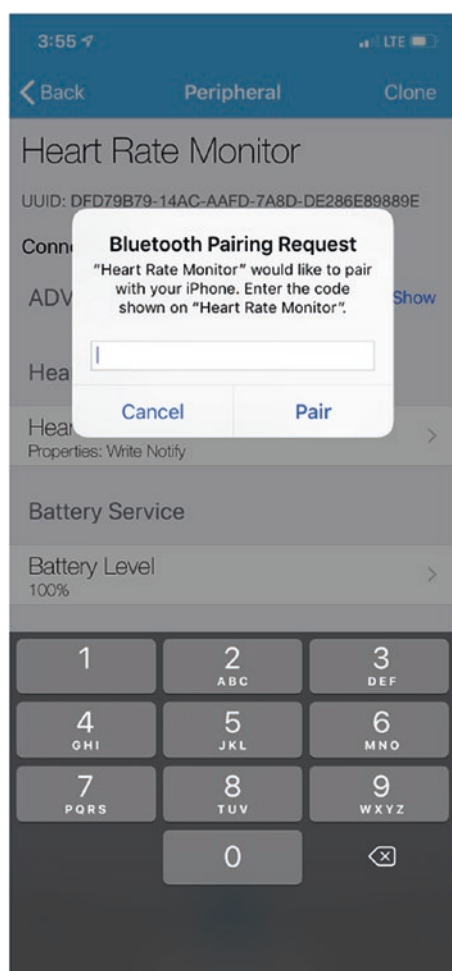


Рис. 4-12. Предлагает ввести код в *LightBlue*

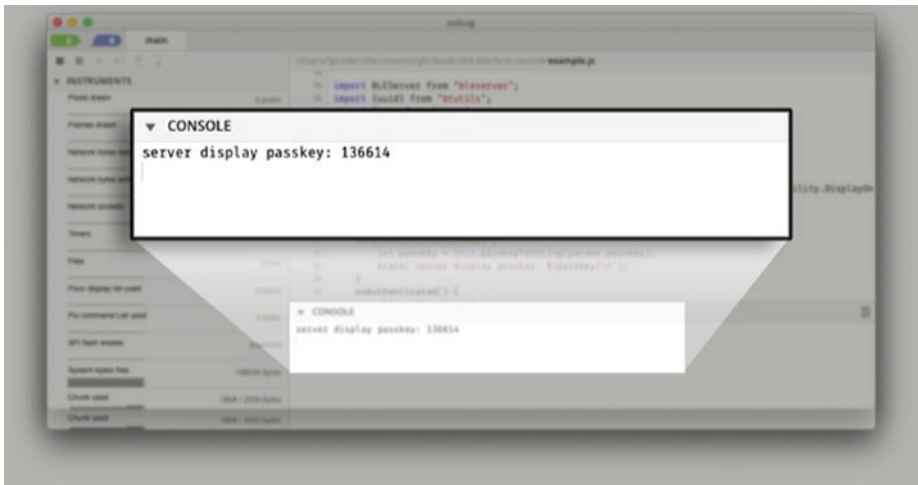


Рис. 4-13. Пароль в консоли отладки

Теперь вы можете включить уведомления для значения частоты сердечных сокращений и считывать значение батареи по запросу, как и раньше, но соединение между Сервером и Клиентом защищено.

Код имеет лишь несколько отличий от примера \$EXAMPLES/ch4-ble/hrm. Как показано в листинге 4-12, обратный вызов `onReady` включает дополнительный код для настройки требований безопасности устройства и возможностей ввода-вывода периферийного устройства.

Листинг 4-12.

```
this.securityParameters = { bonding: true,
                             mitm: true,ioCapability: IOCapability.DisplayOnly};
```

Свойства в этом коде указывают следующее:

- Свойство связывания позволяет связывание, что означает, что оба устройства будут хранить и использовать ключи, которыми они обмениваются, при следующем подключении. Если соединение не включено, устройствам придется выполнять сопряжение каждый раз при подключении.
- Свойство `mitm` требует защиты от посредника, что означает, что данные, которыми обмениваются два сопряженных устройства, шифруются для предотвращения прослушивания ненадежным устройством.
- Свойство `ioCapability` указывает возможности пользовательского интерфейса для устройства, связанные с подтверждением ключа доступа. У этого устройства нет дисплея, но у него есть возможность отображения, поскольку оно может отслеживать консоль отладки. Другие периферийные устройства могут иметь больше возможностей ввода/вывода (например, устройство с клавиатурой) или меньше возможностей ввода/вывода (например, устройство без возможности отображения текста). Свойства `ioCapability` обоих устройств используются для определения метода сопряжения. Например, если ни на одном устройстве нет клавиатуры или дисплея, используется метод сопряжения `Just Works`.

Реализованы два дополнительных обратных вызова класса `BLEServer` (см. листинг 4-13):

- Обратный вызов `onPasskeyDisplay` вызывается, когда вы пытаетесь установить соединение с периферийным устройством. В этом случае он вызывается, когда вы касаетесь имени периферийного устройства в `LightBlue`. Как вы видели ранее, в этом примере ключ доступа отслеживается до консоли отладки.

- Обратный вызов `onAuthenticated` вызывается после успешного сопряжения устройств. В этом примере просто изменяется свойство `authentication`, чтобы указать, что безопасное соединение установлено.

Листинг 4-13.

```
onPasskeyDisplay(params) {
    let passkey = this.passkeyToString(params.passkey);
    trace(`server display passkey: ${passkey}\n`);
}
onAuthenticated() {
    this.authenticated = true;
}
```

Сервер проверяет, установлено ли свойство «authenticated», когда Клиент разрешает уведомления. Код внутри блока `if` выглядит так же, как метод `onCharacteristicNotifyEnabled` из примера `hrm`.

```
onCharacteristicNotifyEnabled(characteristic) {
    if (this.authenticated) {
        ...
    }
}
```

Сервер также определяет дополнительный вспомогательный метод с именем `passkeyToString`. Значения пароля являются целыми числами и всегда должны включать шесть цифр при отображении пользователю. Этот метод дополняет пароль ведущими нулями, когда это необходимо для отображения.

```
passkeyToString(passkey) {
    return passkey.toString().padStart(6, "0");
}
```

Заключение

Теперь, когда вы понимаете суть этих примеров, вы можете многое сделать с помощью BLE на своем ESP32. Вместо подключения к виртуальным периферийным устройствам, которые вы создаете в LightBlue, вы можете подключаться к продуктам BLE у себя дома. Вместо того, чтобы отправлять смоделированные данные, как в примере с пульсометром, вы можете отправлять фактические данные с ваших любимых готовых датчиков.

Если вы хотите попробовать другие примеры BLE, см. каталог `examples/network/ble` в Moddable SDK на GitHub. Доступны примеры, которые позволяют вашему устройству стать маяком, передающим URI, связать ваше устройство с приложением iPhone Music и т. д. Если вы хотите узнать больше о BLE API для Moddable SDK, см. каталог `document/network/ble`.

ГЛАВА 5

Файлы и данные

Почти у каждого продукта есть некоторые данные, которые необходимо обеспечить доступностью при перезапуске устройства, даже в случае отключения питания. В микроконтроллерах флэш-память обычно используется для этой энергонезависимой памяти (NVS). Та же флэш-память, в которой хранится код вашего приложения, также хранит данные, используемые вашим приложением, и данные, которые оно создает. Вот некоторые виды данных, которые может хранить ваше приложение:

- Данные только для чтения, такие как изображения, составляющие пользовательский интерфейс вашего продукта, или файлы, содержащие статические веб-страницы, обслуживаемые встроенным веб-сервером.
- Небольшие фрагменты данных, которые одновременно считываются и записываются, например пользовательские настройки и другое долгосрочное состояние.
- Большие наборы данных, которые создаются по мере того, как ваш продукт отслеживает операции, например, при сборе данных с его датчиков.

На компьютерах и мобильных устройствах файловая система обычно используется для большинства, если не для всех, потребностей в хранении данных. Однако из-за ограничений встраиваемых систем — ограничений по размеру кода, ограниченного объема ОЗУ и серьезных ограничений производительности — прошивка часто даже не включает файловую систему.

В этой главе объясняются три различных способа работы с хранимыми данными во встроенных системах: файлы, настройки и ресурсы. В последнем разделе рассказывается о прямом доступе к флэш-памяти — передовой технологии, обеспечивающей наибольшую гибкость.

При создании продукта выберите методы хранения данных, которые лучше всего соответствуют вашим потребностям. Прежде чем предположить, что файлы — это правильный выбор, рассмотрите предпочтения и ресурсы, которые являются более легкими способами работы с сохраненными данными.

Установка хоста файлов и данных

Вы можете запустить все примеры, упомянутые в этой главе, следуя схеме, описанной в главе 1: установите хост на свое устройство с помощью `mcconfig`, затем установите примеры приложений с помощью `mcrun`.

Хост находится в каталоге `$EXAMPLES/ch5-files/host`. Перейдите в этот каталог из командной строки и установите его с помощью `mcconfig`.

Файлы

ESP32 и ESP8266 используют SPIFFS (Serial Peripheral Interface Flash File System) для своей файловой системы во флэш-памяти. SPIFFS разработан специально для работы с флэш-памятью NOR (НЕ ИЛИ), используемой во многих микроконтроллерах. Хотя SPIFFS далеко не так полнофункциональна, как файловые системы на компьютере, она предоставляет все основные функции, которые вам могут понадобиться.

При использовании файлов на встроенном устройстве важно помнить об ограничениях реализации файловой системы:

- SPIFFS — это плоская файловая система, что означает отсутствие реальных каталогов. Все файлы находятся в корневом каталоге SPIFFS.
- Имена файлов ограничены 32 байтами.

- Нет никаких прав доступа к файлам или блокировок. Все файлы можно читать, записывать и удалять.
- Продолжительность записи непредсказуема. Часто это происходит быстро, но когда файловой системе необходимо консолидировать блоки, она может заблокироваться на некоторое время.

Этот раздел посвящен доступу к файлам с помощью SPIFFS, который доступен без добавления какого-либо оборудования и имеет относительно небольшой размер кода. На ESP32 эти же API также можно использовать для доступа к карте памяти SD, отформатированной с использованием файловой системы FAT32.

Классы файлов

Весь доступ к файловой системе осуществляется с помощью классов в файловом модуле:

```
import {File, Iterator, System} from "file";
```

Файловый модуль экспортирует эти три класса, как подробно описано в следующих разделах:

- Класс File выполняет операции с отдельными файлами, включая чтение, запись, удаление и переименование.
- Класс Iterator возвращает содержимое каталога. В плоской файловой системе, такой как SPIFFS, Iterator доступен только для корневого каталога.
- Продолжительность записи непредсказуема. Часто это происходит быстро, но когда файловой системе необходимо консолидировать блоки, она может заблокироваться на некоторое время.

Пути к файлам

Пути к файлам — это строки, которые вы используете для идентификации файлов и каталогов. Файловый модуль использует символ косой черты (/) для разделения частей пути, как в /spiffs/data.txt.

Хотя SPIFFS — это прямая файловая система без подкаталогов, доступ к ней осуществляется с помощью корня `/spiffs/` вместо `/` для поддержки встроенных устройств, имеющих более одной файловой системы, например, встроенную файловую систему флэш-памяти и внешнюю SD-карту.

В настольном симуляторе корень зависит от хост-платформы. Например, в macOS корневой каталог файловой системы по умолчанию — `/Users/Shared/`. Когда вы пишете код, предназначенный для работы более чем в одной среде, вы можете использовать предопределенное значение в модуле `mc/config`, чтобы найти корень для вашей хост-платформы.

```
import config from "mc/config";  
  
const root = config.file.root;
```

Поскольку файловых систем может быть несколько, этот корень — просто удобное место для файлов по умолчанию, а не единственная доступная файловая система.

Каждая файловая система может иметь разные ограничения на длину имени файла или каталога. Используйте статический метод `System.config` для получения максимальной длины имен в указанном корне.

```
const spiffsConfig = System.config("/spiffs/");  
let name = "this is a very long file name.txt";  
if (name.length > spiffsConfig.maxPathLength)  
    throw new Error("file name too long");
```

Файловые операции

В этом разделе описываются методы, выполняющие операции с файлом, включая удаление, создание и открытие. Не существует методов для чтения или записи всего содержимого файла, так как это часто приводит к сбою из-за нехватки памяти; более поздние разделы знакомят с методами чтения и письма.

Определение существования файла

Используйте статический метод `exists` класса `File`, чтобы определить, существует ли файл:

```
if (File.exists(root + "test.txt"))trace("file exists\n");
else trace("files does not exist\n");
```

Удаление файла

Чтобы удалить файл, используйте статический метод удаления класса

`File`:

```
File.delete(root + "goaway.txt");
```

Метод удаления возвращает `true` в случае успеха и `false` в противном случае. Если файл не существует, `delete` возвращает `true`, а не выдает ошибку, поэтому нет необходимости окружать его вызов блоком `try/catch`. Этот метод выдает ошибку в случае сбоя операции удаления, но это происходит только в редких случаях, например, когда флэш-память изношена или структуры данных файловой системы повреждены.

Переименование файла

Чтобы переименовать файл, используйте метод статического переименования класса `File`. Первый аргумент — это полный путь к файлу, который нужно переименовать, тогда как второй аргумент — это только новое имя.

```
File.rename(root + "oldname.txt", "newname.txt");
```

Примечание Метод переименования предназначен только для переименования файла. В файловых системах, поддерживающих подкаталоги, переименование нельзя использовать для перемещения файла из одного каталога в другой.

Открытие файла

Чтобы открыть файл, создайте экземпляр класса `File`. Первый параметр конструктора `File` — это полный путь к открываемому файлу.

Необязательный второй параметр имеет значение `true` для открытия в режиме записи (создание файла, если он не существует) и либо отсутствует, либо `false` для открытия в режиме чтения. Вот пример открытия файла в режиме чтения:

```
let file = new File(root + "test.txt");
```

Следующий пример открывает файл в режиме записи, создавая файл, если он не существует:

```
let file = new File(root + "test.txt", true);
```

Конструктор `File` выдает ошибку, если возникает ошибка при открытии файла, например при попытке открыть несуществующий файл в режиме чтения.

Когда вы закончите доступ к файлу, закройте экземпляр файла, чтобы освободить системные ресурсы, которые он использует:

```
file.close();
```

Запись в файл

В этом разделе представлены методы записи данных в файл. Вы можете использовать класс `File` для записи как текстовых, так и двоичных данных. Файл должен быть открыт в режиме записи, иначе операции записи вызовут ошибку. Чтобы открыть в режиме записи, передайте `true` в качестве второго аргумента конструктору `File`.

Файловая система автоматически увеличивает размер файла, когда вы записываете данные, превышающие текущий размер. Нет поддержки для усечения файла. Чтобы уменьшить размер файла, создайте другой файл и скопируйте в него необходимые данные из исходного файла.

Написание текста

Метод записи класса `File` определяет тип данных, которые вы хотите записать, исходя из типа объекта JavaScript, который вы передаете вызову. Чтобы написать текст, передайте строку. Следующий код из примера `$EXAMPLES/ch5-files/files` записывает в файл одну строку:

```
let file = new File(root + "test.txt", true);
file.write("this is a test");
file.close();
```

Строки всегда записываются как данные UTF-8.

Запись двоичных данных

Чтобы записать двоичные данные в файл, передайте `ArrayBuffer` для записи. Следующий код из примера `$EXAMPLES/ch5-files/files` записывает в файл пять 32-битных целых чисел без знака. Значения находятся в `Uint32Array`, который использует для хранения `ArrayBuffer`. Вызов записи получает `ArrayBuffer` из свойства буфера массива байтов.

```
let bytes = Uint32Array.of(0, 1, 2, 3, 4);
let file = new File(root + "test.bin", true);
file.write(bytes.buffer);
file.close();
```

Чтобы записать байты (8-битные значения без знака), передайте целочисленное значение в качестве аргумента (см. листинг 5-1).

Листинг 5-1.

```
let file = new File(root + "test.bin", true);
file.write(1);
file.write(2);
file.write(3);
file.close();
```

Получение размера файла

Чтобы определить размер файла в байтах, вы сначала открываете файл, а затем проверяете его свойство длины:

```
let file = new File(root + "test.txt");
let length = file.length;
trace(`test.txt is ${length} bytes\n`);
file.close();
```

Свойство `length` доступно только для чтения. Невозможно настроить изменение размера файла.

Написание смешанных типов

Метод записи позволяет передавать несколько аргументов для записи нескольких фрагментов данных за один вызов. Это выполняется немного быстрее и делает ваш код немного меньше. В следующем примере записывается `ArrayBuffer`, четыре байта и одна строка за один вызов записи:

```
let bytes = Uint32Array.of(0x01020304, 0x05060708);
let file = new File(root + "test.bin", true);
file.write(bytes.buffer, 9, 10, 11 12, "ONE TWO!");
file.close();
```

Шестнадцатеричный дамп файла после записи выглядит так:

```
04 03 02 01 08 07 06 05      .... ....
09 0A 0B 0C 79 78 69 32      .... ONE
84 87 79 33                  TWO!
```

Вы можете ожидать, что первые четыре байта будут 01 02 03 04, но помните, что экземпляры `TypedArray`, включая `Uint32Array`, хранятся в порядке байтов хост-платформы, а микроконтроллеры ESP32 и ESP8266 являются устройствами с прямым порядком байтов.

Чтение из файла

В этом разделе представлены методы извлечения данных из файла. Вы можете использовать класс `File` для чтения как текстовых, так и двоичных данных. Большинство файлов имеют одно или другое значение — все двоичные или все текстовые данные — хотя это не обязательно.

Класс `File` поддерживает чтение файла по частям, что позволяет контролировать максимальный объем памяти, используемый при чтении из файла.

Чтение текста

Иногда полезно получить все содержимое файла в виде одной текстовой строки. Вы делаете это, вызывая метод чтения с одним аргументом `String`, который указывает экземпляру файла читать от текущей позиции до конца файла и помещать результат в строку. Следующий код из примера `$EXAMPLES/ch5-files/files` считывает содержимое из созданного ранее файла `test.txt`:

```
let file = new File(root + "test.txt");
let string = file.read(String);
trace(string + "\n");
file.close();
```

Метод чтения всегда начинает чтение с текущей позиции. В этом случае, поскольку файл только что был открыт, текущая позиция равна 0, началу файла.

Чтение текста по частям

Вы также можете использовать метод чтения для извлечения частей файла, чтобы свести к минимуму пиковое использование памяти. Необязательный второй аргумент для чтения указывает максимальное количество байтов для чтения. Это количество прочитанных байтов, за одним исключением: если при чтении запрошенного количества байтов будет пропущен конец файла, читается текст от текущей позиции до конца файла.

В примере в листинге 5-2 файл считывается фрагментами по десять байтов и отслеживается до консоли. Он сравнивает свойство `position` со свойством `length`, чтобы определить, когда были прочитаны все данные из файла.

Листинг 5-2.

```
let file = new File(root + "test.txt");
while (file.position < file.length) {
    let string = file.read(String, 10);
    trace(string + "\n");
}
file.close();
```

На компьютере вы можете отобразить файл в память, чтобы упростить доступ к данным; однако этот подход обычно недоступен для микроконтроллеров, поскольку в них обычно отсутствует MMU (блок управления памятью) для выполнения отображения. Если вы хотите отобразить в памяти данные, предназначенные только для чтения, ресурсы — хорошая альтернатива, как объясняется далее в этой главе.

Чтение двоичных данных

Чтобы прочитать весь файл как двоичные данные, вызовите `read` с одним аргументом `ArrayBuffer`. Следующий код из примера `$EXAMPLES/ch5-files/files` считывает содержимое из созданного ранее файла `test.bin`:

```
let file = new File(root + "test.bin");
let buffer = file.read(ArrayBuffer);
file.close();
```

Как и при чтении текста, двоичное чтение начинается с текущей позиции, которая равна 0 при открытии файла, и продолжается до конца файла. Данные возвращаются в `ArrayBuffer`. В следующем примере возвращаемый буфер помещается в массив `Uint8Array` и отображаются шестнадцатеричные значения байтов на консоли:

```
let bytes = new Uint8Array(buffer);
for (let i = 0; i < bytes.length; i++)
    trace(bytes[i].toString(16).padStart(2, "0"), "\n");
```

Чтение двоичных данных по частям

Метод чтения также можно использовать для извлечения двоичных данных из произвольных мест в файле. Пример в листинге 5-3 считывает последние четыре байта файла и отображает результат в виде 32-разрядного целого числа без знака. Место чтения указывается установкой свойства `position` равным четырём байтам от конца файла.

Листинг

5-3.

```
let file = new File(root + "test.bin");
file.position = file.length - 4;
let buffer = file.read(ArrayBuffer, 4);
file.close();
let value = (new Uint32(buffer))[0];
```

Каталоги

Файловая система SPIFFS реализует только один каталог, корневой каталог. Другие файловые системы, такие как FAT32, поддерживают произвольное количество подкаталогов. Во всех случаях вы используете класс `Iterator` файлового модуля для вывода списка файлов и подкаталогов, содержащихся в каталоге.

Итерация по каталогам

Получение списка элементов в каталоге — это двухэтапный процесс. Сначала вы создаете экземпляр класса `Iterator` для каталога, по которому нужно выполнить итерацию; затем вы вызываете следующий метод итератора для извлечения каждого элемента.

Когда все элементы возвращены, итератор возвращает значение `undefined`. Листинг 5-4 из примера `$EXAMPLES/ch5-files/files` отслеживает файлы и каталоги, содержащиеся в корневом каталоге, до консоли.

Листинг 5-4.

```
let iterator = new Iterator(root);
let item;
while (item = iterator.next()) {
  if (undefined === item.length)
    trace(`${item.name.padEnd(32)} directory\n`);
  else
    trace(`${item.name.padEnd(32)} file ${item.length}` +
          "bytes\n");
}
```

Следующий метод возвращает объект со свойствами, описывающими элемент. Свойство `name` присутствует всегда. Свойство `length` присутствует только для файлов и указывает количество байтов в файле. Нет отдельного свойства, указывающего, является ли элемент файлом или каталогом, поскольку для этой цели достаточно наличия свойства `length`.

Экземпляр итератора имеет метод `close`, который можно вызвать для освобождения системных ресурсов, используемых итератором. Однако обычно в этом нет необходимости, поскольку реализация итератора автоматически освобождает любые системные ресурсы, когда достигает конца элементов.

Класс `Iterator` возвращает по одному элементу за раз, а не список всех элементов, чтобы свести использование памяти к минимуму. Порядок, в котором возвращаются элементы, зависит от базовой реализации файловой системы. В общем случае вы не можете предположить, например, что элементы возвращаются в алфавитном порядке или что каталоги возвращаются перед файлами.

Итерация с помощью итераторов JavaScript

Язык JavaScript предоставляет функцию итератора, упрощающую написание кода, использующего итераторы. Например, вы можете использовать синтаксис цикла `for-of` для перебора элементов. Эта функция языка работает с любым экземпляром, который реализует протокол итератора, что и делает класс `Iterator` файлового модуля. Этот подход немного более лаконичен для вашего кода за счет использования немного большего количества памяти и процессорного времени. Листинг 5-5 адаптирует листинг 5-4 для использования итератора JavaScript.

Листинг 5-5.

```
for (let item of new Iterator(root)) {
  if (undefined === item.length)
    trace(`${item.name.padEnd(32)} directory\n`);
  else
    trace(`${item.name.padEnd(32)} file ${item.length}` +
          "bytes\n");
}
```

Где итераторы действительно хороши, так это в качестве входных данных для функций, которые работают с итераторами. Например, если вам нужен массив, содержащий все элементы, содержащиеся в каталоге, вы можете просто передать экземпляр итератора в `Array.from`.

```
let items = Array.from(new Iterator(root));
```

Получение информации о файловой системе

Объект `System` файлового модуля содержит метод `info` для предоставления информации о каждом корне файловой системы. Этот метод используется для определения общего количества байтов доступного хранилища и количества байтов, используемых в настоящее время.

```
let info = System.info(root);
trace(`Used ${info.used} of ${info.total}\n`);
```

Настройки

Настройки — это еще один инструмент для хранения данных на микроконтроллере вашего продукта IoT. Они намного эффективнее файлов, но и гораздо более ограничены. Файл хорошо подходит для хранения большого количества информации, тогда как параметр хранит только небольшие фрагменты информации. Часто в вашем продукте вам нужно отслеживать только несколько пользовательских настроек, и предпочтения — это все, что вам нужно для таких ситуаций; вы даже можете полностью исключить файловую систему из вашего продукта.

Еще одним преимуществом использования предпочтений является их надежность. Реализации настроек для ESP32 и ESP8266 предпринимают шаги, чтобы гарантировать, что данные о настройках не будут повреждены, даже если отключится питание во время обновления настроек. Такого уровня надежности труднее достичь в файловой системе, поскольку структуры данных более сложны.

Класс предпочтения

Модуль предпочтений обеспечивает доступ к настройкам. Чтобы использовать настройки в своем коде, импортируйте класс `Preference` из модуля предпочтений.

```
import Preference from "preference";
```

API предпочтений JavaScript, представленный в этой главе, одинаков для всех микроконтроллеров; однако базовая реализация отличается. Например, на ESP32 предпочтения реализуются с помощью библиотеки NVS в ESP32 IDF SDK, тогда как на ESP8266 предпочтения реализуются с помощью Moddable SDK, поскольку нет системного эквивалента. Поскольку реализации разные, есть и различия в поведении. В следующих разделах отмечены различия, которые необходимо учитывать.

Имена предпочтений

Каждое предпочтение идентифицируется двумя значениями: доменом и именем. Они похожи на простой путь к файловой системе: домен похож на имя каталога, а имя — на имя файла. Например, рассмотрим индикатор Wi-Fi, где вы хотите сохранить пользовательские настройки для восстановления при включении питания. Вы можете использовать светлый домен для всех настроек состояния освещения, включая включение, яркость и цвет для имен. Свет может хранить статистические данные (например, количество включений света) в другом домене, например stats.

Значения домена и имени предпочтения всегда являются строками. Имена ограничены 15 байтами на ESP32 и 31 байтами на ESP8266.

Данные о предпочтениях

Настройки не предназначены для замены файловой системы; Попытка использовать их таким образом является распространенной ошибкой. Поскольку размер каждой отдельной настройки ограничен, как и общее хранилище, доступное для всех настроек, они гораздо менее универсальны, чем файловая система.

Каждое предпочтение имеет тип данных: логическое значение, 32-разрядное целое число со знаком, строка или буфер массива. Числовые значения с плавающей запятой не поддерживаются. Строковый тип часто наиболее удобен в использовании, но также часто наименее эффективно использует пространство для хранения. Если вам нужно объединить несколько значений в одном параметре, рассмотрите возможность использования `ArrayBuffer`.

Когда вы записываете значение, тип значения устанавливается на основе предоставленных данных. Чтобы изменить тип, запишите значение еще раз. Когда вы читаете значение, возвращаемое значение имеет тот же тип, что и записанное значение.

Обратите внимание на эти различия между данными о предпочтениях на ESP32 и на ESP8266:

- На ESP32 пространство данных предпочтений настраивается и установлено на 16 КБ на хостах, используемых в этой книге. На ESP8266 место для данных настроек составляет 4 КБ.
- На ESP32 каждое предпочтение может содержать до 4000 байт данных; на ESP8266 это значение ограничено 64 байтами. Если вы пишете код, который, как предполагается, будет работать на нескольких разных платформах микроконтроллеров, вам необходимо разработать свои предпочтительные значения для размера данных 64 байта.

Чтение и запись предпочтений

Поскольку предпочтения — это просто небольшие фрагменты данных определенного типа, их гораздо легче читать и записывать, чем файл. Листинг 5-6 из примера \$EXAMPLES/ch5-files/preferences записывает четыре предпочтения в домен примера. Тип каждого значения используется в качестве имени предпочтения. Реализация набора определяет тип предпочтения на основе значения, переданного в третьем аргументе.

Листинг 5-6.

```
Preference.set("example", "boolean", true);
Preference.set("example", "integer", 1);
Preference.set("example", "string", "my value");
Preference.set("example", "arraybuffer",
               Uint8Array.of(1, 2, 3).buffer);
```

Используйте статический вызов `get` для получения значений предпочтений, как показано в листинге 5-7. Тип возвращаемого значения соответствует типу значения, используемого в вызове `set`.

Листинг 5-7.

```

let a = Preference.get("example", "boolean");    // true
let b = Preference.get("example", "integer");    // 1
let c = Preference.get("example", "string");     // "my value"
let d = Preference.get("example", "arraybuffer");
           // ArrayBuffer of [1, 2, 3]

```

Если предпочтения с указанным доменом и именем не найдены, вызов `get` возвращает `undefined`:

```

let on = Preference.get("light", "on");
if (undefined === on)
    on = false;

```

Удаление настроек

Используйте метод удаления, чтобы удалить предпочтение:

```
Preference.delete("example", "integer");
```

Ошибка не выдается, если не удастся найти предпочтение с указанным доменом и именем. Если при обновлении флэш-памяти для удаления предпочтения возникает ошибка, удаление вызывает ошибку.

Не используйте JSON

При создании продуктов на JavaScript для Интернета или компьютеров настройки обычно сохраняются с использованием JSON — этот подход чрезвычайно прост в кодировании и является гибким. Заманчиво сделать то же самое при создании встроенного продукта с использованием JavaScript; однако, хотя он и работает в некоторых продуктах, его не рекомендуется использовать, так как это с большей вероятностью приведет к сбоям на более поздних этапах процесса разработки. Рассмотрим следующее:

- Хранение настроек в файле JSON требует, чтобы ваш проект включал файловую систему — большой объем кода, который занимает часть ограниченного пространства вашей флэш-памяти.
- Объект JSON должен быть загружен в память сразу, а это означает, что для доступа к одному значению предпочтения требуется достаточно памяти для хранения всех значений предпочтения.
- Загрузка строковых данных JSON из файла и их последующий анализ в объектах JavaScript занимает значительно больше времени, чем простая загрузка одного значения из предпочтения.
- Файловые системы, как правило, менее устойчивы к ошибкам при сбоях питания, чем предпочтения. Следовательно, выше вероятность того, что пользовательские настройки будут потеряны.

Использование JSON также может показаться хорошим способом хранения нескольких значений в одном параметре. Это действительно работает, но имеет два ограничения, которые во многих случаях делают его нецелесообразным выбором. Во-первых, поскольку данные о предпочтениях на некоторых устройствах ограничены всего 64 байтами, вы не можете комбинировать многие значения таким образом. Во-вторых, накладные расходы формата JSON почти наверняка означают, что данные о предпочтениях занимают больше места, чем другие методы. Например, следующий код использует 24 байта памяти для хранения трех небольших целочисленных значений в формате JSON:

```
Preference.set("example", "json",JSON.stringify({a: 1, b: 2, c: 3}));
```

Напротив, в этом примере требуется всего три байта при использовании Uint8Array:

```
Preference.set("example", "bytes",Uint8Array.of(1, 2, 3).buffer);
```

Чтение значений из версии JSON проще:

```
let pref = JSON.parse(Preference.get("example", "json"));
```

Чтение значений из более эффективной версии требует дополнительной строки кода:

```
let pref = new Uint8Array(Preference.get("example", "bytes"));
pref = {a: pref[0], b: pref[1], c: pref[2]};
```

Безопасность

Модуль предпочтений не дает никаких гарантий безопасности данных о предпочтениях. Домен, имя и значение могут храниться «в открытом виде» без какого-либо шифрования или обфускации. Как и в случае пользовательских данных в файлах, вы должны предпринять соответствующие шаги в своем продукте, чтобы обеспечить надлежащую защиту пользовательских данных. Примерами конфиденциальных пользовательских данных, которые обычно хранятся в продуктах IoT, являются пароли Wi-Fi и идентификаторы учетных записей облачных служб. Как минимум, вы должны рассмотреть возможность применения некоторой формы шифрования к этим значениям, чтобы они не могли быть прочитаны злоумышленником, сканирующим флэш-память устройства.

Некоторые хосты предоставляют зашифрованное хранилище для данных о предпочтениях. С дополнительной настройкой это доступно, например, на ESP32.

Ресурсы

Ресурсы — это инструмент для работы с данными только для чтения. Это наиболее эффективный способ встраивания больших фрагментов данных в ваш проект. Доступ к ресурсам обычно осуществляется на месте во флэш-памяти, где они хранятся, и, следовательно, не использует оперативную память, независимо от того, насколько велики данные ресурса. Moddable SDK использует ресурсы для самых разных целей, включая сертификаты TLS, изображения и аудио, но нет ограничений на тип данных, которые вы можете хранить в ресурсе.

Пример \$EXAMPLES/ch5-files/resources содержит простую веб-страницу, определенную в mydata.dat, которая включена в качестве ресурса. После запуска примера откройте веб-браузер и введите IP-адрес вашего устройства, и вы увидите веб-страницу с надписью «Hello, world».

Добавление ресурсов в проект

Включение ресурса в ваш проект требует двух шагов:

1. Добавляйте файл, содержащий данные ресурса, в свой проект. Часто файлы ресурсов размещаются в подкаталогах, таких как активы, данные или ресурсы, но вы можете хранить их где угодно.
2. Добавляйте файл в раздел ресурсов вашего манифеста, чтобы указать инструментам сборки скопировать данные файла в ресурс.

Листинг 5-8 взят из манифеста примера ресурсов. Он включает только один ресурс, mydata.dat, из каталога, содержащего манифест.

Листинг 5-8.

```
"resources": {
  "*": [
    "./mydata"
  ],
}
```

Файл данных должен иметь расширение .dat. Однако имя файла в манифесте не должно включать расширение; инструменты сборки автоматически находят ваш файл с расширением .dat. Важно, чтобы вы не включали несколько файлов с одинаковым именем, но разными расширениями (например, mydata.dat и mydata.bin), так как инструменты могут не найти тот, который вы ожидаете первым.

В этой главе описываются данные ресурсов, которые копируются непосредственно из входного файла в выходной двоичный файл без каких-либо изменений. Инструменты сборки также имеют возможность применять преобразования к данным, такие как преобразование изображений в формат, оптимизированный для вашего целевого микроконтроллера; Глава 8 объясняет, как использовать преобразования ресурсов.

Доступ к ресурсам

Чтобы получить доступ к ресурсу, импортируйте класс `Resource` из модуля ресурсов:

```
import Resource from "resource";
```

Используйте конструктор класса `Resource` с путем к ресурсу из манифеста. Обратите внимание, что путь всегда включает расширение файла — в данном случае `.dat`.

```
let data = new Resource("mydata.dat");
```

Конструктор ресурсов выдает ошибку, если не может найти запрошенный ресурс. Если вы хотите проверить, существует ли ресурс перед вызовом конструктора, используйте статический метод `exists`:

```
if (Resource.exists("mydata.dat")) {
  let data = new Resource("mydata.dat");    ...
}
```

Использование ресурсов

Конструктор `Resource` возвращает двоичные данные в виде `HostBuffer`. `HostBuffer` похож на `ArrayBuffer`, но, в отличие от `ArrayBuffer`, данные `HostBuffer` могут быть доступны только для чтения и, следовательно, могут располагаться во флэш-памяти.

Чтобы получить количество байтов в ресурсе, используйте свойство `byteLength`, как и в случае с `ArrayBuffer`:

```
let r1 = new Resource("mydata.dat");
let length = r1.byteLength;
```

Также, как и в случае с `ArrayBuffer`, вы не можете получить доступ к данным `HostBuffer` напрямую, но должны обернуть их в типизированный массив или представление данных. В следующем примере ресурс помещается в `Uint8Array` и отслеживаются значения на консоли:

```
let r1 = new Resource("mydata.dat");
let bytes = new Uint8Array(r1);
for (let i = 0; i < bytes.length;
i++)trace(bytes[i], "\n");
```

В этом примере ресурс помещается в объект `DataView` для доступа к его содержимому в виде 32-разрядных целых чисел без знака с обратным порядком байтов:

```
let r1 = new Resource("mydata.dat");
let view = new DataView(r1);
for (let i = 0; i < view.byteLength;
i += 4)trace(view.getUint32(i, false), "\n");
```

Иногда вам нужно изменить данные в ресурсе. Поскольку данные доступны только для чтения, вам необходимо сделать копию. `HostBuffer`, возвращаемый конструктором `Resource`, имеет метод среза, который можно использовать для копирования данных ресурса так же, как метод среза в экземпляре `ArrayBuffer`. Например, вы можете скопировать весь ресурс в `ArrayBuffer` в ОЗУ следующим образом:

```
let r1 = new Resource("mydata.dat");
let clone = r1.slice(0);
```

Первым аргументом `slice` является начальное смещение копируемых данных. Необязательный второй аргумент — это конечное смещение для копирования; если он опущен, данные копируются в конец ресурса. В следующем примере извлекается десять байтов данных ресурса, начиная с 20-го байта:

```
let r1 = new Resource("mydata.dat");
let fragment = r1.slice(20, 30);
```

Метод `slice` поддерживает необязательный третий аргумент, который не предоставляется `ArrayBuffer`. Этот аргумент определяет, будут ли данные копироваться в ОЗУ. Если установлено значение `false`, `slice` возвращает `HostBuffer`, ссылающийся на фрагмент данных ресурса, что полезно, когда вы хотите связать только часть ресурса с объектом, не копируя его данные в ОЗУ. Например, если есть массив из пяти беззнаковых 16-битных данных по смещению 32 ресурса, вы можете создать `Uint16Array`, который ссылается на него, следующим образом:

```
let r1 = new Resource("mydata.dat");
let values = new Uint16Array(r1.slice(32, 10, false));
```

Аналогичного результата можно добиться, используя необязательные параметры `byteOffset` и `length` конструктора `Uint16Array`:

```
let r1 = new Resource("mydata.dat");
let values = new Uint16Array(r1, 32, 10);
```

Преимущество использования среза заключается в том, что он гарантирует, что весь ресурс недоступен для ненадежного кода с доступом к массиву значений. В первом из двух предыдущих примеров `values.buffer` имеет доступ ко всему ресурсу, тогда как во втором примере он может использоваться только для доступа к пяти значениям в массиве `Uint16Array`.

Прямой доступ к флэш-памяти

Все описанные в этой главе модули для хранения и извлечения данных — файлов, настроек и ресурсов — используют флэш-память, подключенную к контроллеру, для хранения данных. Каждый подход к работе с данными во флэш-памяти имеет свои преимущества и ограничения.

В большинстве случаев один из этих подходов хорошо подходит для нужд вашего продукта; в некоторых ситуациях более эффективным может быть более специализированный подход. Флэш-модуль обеспечивает прямой доступ к флэш-памяти. Для его правильного использования требуется больше работы, но в некоторых случаях оно того стоит.

Предупреждение Это сложная тема. Прямой доступ к флэш-памяти опасен. Вы можете привести к сбою устройства или повреждению данных. Вы даже можете повредить флэш-память, что сделает ваше устройство непригодным для использования.

Основы аппаратной части флэш-памяти

Чтобы иметь возможность использовать API, предоставляемый флэш-модулем, важно понимать основы аппаратного обеспечения флэш-памяти.

Флэш-память, используемая с микроконтроллерами ESP32 и ESP8266, подключается с помощью шины SPI (Serial Peripheral Interface). Хотя доступ к ним достаточно быстрый, он все же во много раз медленнее, чем доступ к данным в оперативной памяти.

Флэш-память организована в виде блоков (также называемых «секторами»). Размер блока зависит от используемого компонента флэш-памяти. Обычное значение составляет 4096 байт. Когда вы читаете и записываете флэш-память, вам обычно не нужно знать размер блока. Однако размер блока важен при инициализации флэш-памяти.

Флэш-память использует технологию NOR для хранения данных. Это имеет любопытное следствие: в стертом байте флэш-памяти все биты установлены в 1, в то время как обычно считается, что стертая память установлена в 0. Вы можете подумать, что можете просто установить только что стертые байты во все нули, но, поскольку вы увидите, что это не очень хорошая идея с флэш-памятью NOR.

Когда вы записываете во флэш-память NOR, вы записываете только 0 бит. Поскольку флэш-память стирается до всех 1 бит, это не имеет значения при первой записи. Рассмотрим два байта (16 бит) флэш-памяти. Они начинают стираться до всех 1 бит.

```
11111111 11111111
```

Запишите туда два байта, 1 и 2, и результат будет простым:

```
00000001 00000010
```

Следующий шаг, когда результат является неожиданным. Вот что происходит, когда вы затем записываете два байта 2 и 1 в одно и то же место:

```
00000000 00000000
```

В результате оба байта равны 0. Почему? Помните, что с флэш-памятью NOR запись устанавливает только 0 бит. Любые биты во флэш-памяти, которые уже установлены на 0, не могут быть изменены обратно на 1 с помощью записи.

- Flash 0. Write 0 => Flash 0.
- Flash 0. Write 1 => Flash 0.
- Flash 1. Write 0 => Flash 0.
- Flash 1. Write 1 => Flash 1.

Если запись может изменить только биты с 1 на 0, как биты изменяются с 0 на 1? Для этого вы используете метод стирания флэш-памяти. В отличие от чтения и записи, которые могут обращаться к любому байту во флэш-памяти напрямую, стирание — это массовая операция, которая устанавливает все биты в блоке флэш-памяти в 1. Вы стираете блоки, выровненные по границе размера блока, что означает, что байты с 0 по 4095 или байты от 4096 до 8191 — не от 1 до 4096, потому что они не выровнены по началу блока, и не байты от 1 до 2, потому что это не полный блок.

Если вы хотите изменить один бит, вы можете прочитать весь блок в ОЗУ, стереть блок, изменить бит в ОЗУ, а затем записать блок обратно. Это работает, но медленно, потому что стирание — относительно

медленная операция — во много раз медленнее, чем чтение и запись. Этот подход также требует достаточного объема оперативной памяти для хранения полного блока, а на микроконтроллере с ограниченными ресурсами не всегда имеется столько памяти. Однако самая большая проблема заключается в том, что флэш-память изнашивается. Каждый блок может быть стерт только определенное количество раз, после чего этот блок больше не надежно хранит данные; чтобы сохранить устройство, вам нужно свести к минимуму количество стираний каждого блока.

Хорошая новость заключается в том, что флэш-память вашего ESP32 или ESP8266 поддерживает тысячи, если не десятки тысяч, операций стирания. Реализации модулей предпочтений и файлов знают об ограничениях и характеристиках флэш-памяти NOR и предпринимают шаги для минимизации стираний. Если вы получаете доступ к флэш-памяти непосредственно в продукте, который предназначен для использования в течение многих лет, вам нужно сделать то же самое.

Одной из часто используемых стратегий является добавочная запись. При таком подходе текущие значения обнуляются, а новые значения записываются после нулей в блоке. Это позволяет многократно обновлять одно значение без стирания. Этот подход используется модулем предпочтений. В приведенном ниже в этом разделе примере часто обновляемого целого числа подробно рассматриваются добавочные записи.

Другой распространенной стратегией является выравнивание износа. Этот подход пытается стереть каждый блок флэш-памяти одинаковое количество раз в течение срока службы продукта, чтобы гарантировать, что ни один блок (например, первый блок) не изнашивается намного раньше, чем другие, из-за более частого доступа. Файловая система SPIFFS, лежащая в основе файлового модуля, использует этот метод.

Доступ к разделам Flash

Доступ к флэш-памяти, доступной вашему микроконтроллеру, осуществляется с помощью класса Flash из модуля flash:

```
import Flash from "flash";
```

Флэш-память разделена на сегменты, называемые разделами. Например, один раздел содержит код вашего проекта, другой — данные о настройках, а третий — хранилище для файловой системы SPIFFS. Каждый раздел идентифицируется именем.

Чтобы получить доступ к байтам в разделе, создайте экземпляр класса `Flash` с именем раздела. Когда вы устанавливаете примеры приложений с помощью `mcrun`, как описано в главе 1, байтовый код приложения сохраняется в разделе `xs`. Следующая строка создает экземпляр класса `Flash` для доступа к нему:

```
let xsPartition = new Flash("xs");
```

Разделы, доступные для вашего кода, зависят от микроконтроллера и реализации хоста. Раздел `xs`, содержащий приложения, установленные с помощью `mcrun`, всегда доступен. Область, используемая для файловой системы SPIFFS, называемая хранилищем, также обычно всегда доступна; если вы не используете файловую систему SPIFFS в своем проекте, вы можете использовать ее для других целей. Хотя оба этих раздела присутствуют, их размеры различаются в зависимости от устройства.

На ESP32 разделы определяет ESP32 IDF от Espressif. IDF предоставляет гибкий механизм разделов, который позволяет вам определять свои собственные разделы. На ESP8266 Moddable SDK определяет разделы, и их нельзя легко переконфигурировать.

В ESP32 конструктор `Flash` ищет в карте разделов IDF соответствие запрошенному имени раздела. Следовательно, вы можете получить доступ к разделу, содержащему настройки ESP32, которые реализованы в библиотеке NVS, с именем `nvs`, как указано в карте раздела (файл `partitions.csv` в проекте IDF).

```
let nvsPartition = new Flash("nvs");
```

Получение информации о разделе

Экземпляр класса `Flash` имеет два свойства только для чтения, которые предоставляют важную информацию о разделе: `blockSize` и `byteLength`.

Свойство `blockSize` указывает количество байтов в одном блоке флэш-памяти. Это значение часто равно 4096, но для надежности следует использовать свойство `blockSize`, а не жестко задавать постоянное значение в коде. Таким образом, ваш код может работать без изменений на оборудовании, которое включает в себя другой аппаратный компонент флэш-памяти.

```
let storagePartition = new Flash("storage");  
let blockSize = storagePartition.blockSize;
```

Свойство `blockSize` важно, потому что оно сообщает вам как выравнивание, так и размер операций стирания в разделе.

Свойство `byteLength` предоставляет общее количество байтов, доступных в разделе. В следующем примере вычисляется количество блоков в разделе:

```
let blocks = storagePartition.byteLength / blockSize;
```

Значение свойства `byteLength` всегда является целым числом, кратным значению свойства `blockSize`, поэтому количество блоков всегда является целым числом.

Чтение из флэш-раздела

Используйте метод чтения для извлечения байтов из раздела флэш-памяти. Метод чтения принимает два аргумента: смещение в разделе и количество байтов для чтения. Результатом вызова чтения является `ArrayBuffer`. Ниже приведен отрывок из примера `$EXAMPLES/ch5-files/flash-readwrite`:

```
let buffer = partition.read(0, 10);  
let bytes = new Uint8Array(buffer);
```



```
for (let i = 0; i < bytes.byteLength; i++)
  trace(bytes[i] + "\n");
```

Этот код извлекает первые десять байтов из раздела. Он заключает возвращенный `ArrayBuffer` в массив `Uint8Array` для трассировки значений байтов на консоль.

Нет никаких ограничений на смещение и количество байтов для чтения, кроме требования, чтобы они находились внутри раздела. В частности, одиночный вызов чтения может пересекать границу блока.

Вызов `read` копирует запрошенные данные из раздела в новый `ArrayBuffer`. Следовательно, вы должны считывать флэш-память небольшими фрагментами, чтобы использовать как можно меньше оперативной памяти.

Стирание флэш-раздела

Используйте метод стирания, чтобы сбросить все биты в разделе флэш-памяти до 1. Метод принимает один аргумент — количество блоков для сброса. Эта строка стирает первый блок раздела:

```
partition.erase(0);
```

Следующий код сбрасывает весь раздел. Операция стирания выполняется относительно медленно; для большого раздела — например, раздела хранилища на ESP8266 — эта операция занимает несколько секунд.

```
let blocks = partition.byteLength / partition.blockSize;
for (let block = 0; block < blocks; block++)
  partition.erase(block);
```

Запись на флэш-раздел

Используйте метод записи, чтобы изменить значения, хранящиеся в разделе флэш-памяти. Этот метод принимает три аргумента: смещение для записи данных в раздел, количество записываемых байтов и `ArrayBuffer`, содержащий данные. Когда количество байтов для записи меньше размера `ArrayBuffer`, записывается только это количество байтов.

В следующем примере первые десять байтов раздела задаются целыми числами от 1 до 10:

```
let buffer = Uint8Array.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10).buffer;  
partition.write(0, 10, buffer);
```

Имейте в виду, что запись устанавливает только биты 0, как объяснялось ранее в разделе «Основы аппаратного обеспечения флэш-памяти». Поэтому перед вызовом записи может потребоваться выполнить стирание.

Отображение раздела Flash

На ESP32 у вас есть возможность сопоставления памяти раздела, что дает вам доступ только для чтения к содержимому раздела с использованием типизированного массива или конструктора представления данных. Чтобы отобразить раздел в память, вызовите метод `map`. Следующий код взят из примера `$EXAMPLES/ch5-files/flash-map`:

```
let partition = new Flash("storage");  
let buffer = partition.map();  
let bytes = new Uint8Array(buffer);
```

Свойство карты возвращает `HostBuffer`, который может быть передан конструктору типизированного массива или представления данных для доступа к данным. Разделы с отображением памяти в некоторых ситуациях являются более удобным способом доступа к данным, чем вызов чтения. Кроме того, поскольку данные в разделе не копируются в ОЗУ методом карты, использование ОЗУ сведено к минимуму.

Метод сопоставления недоступен на ES8266 из-за аппаратных ограничений, позволяющих отображать только первый мегабайт флэш-памяти, область, зарезервированную для хранения прошивки.

Пример: часто обновляемое целое число

В этом разделе представлен пример прямого доступа к флэш-памяти для более эффективного обслуживания 32-битного значения, чем это возможно при использовании файла или настройки. Пример относится к ситуации, когда вашему продукту необходимо часто обновлять значение во флэш-памяти, чтобы гарантировать его надежное сохранение при перезагрузке продукта.

В примере используется один блок флэш-памяти. Обычно это 4096 байт, что в 1024 раза больше, чем сохраняемое 32-битное (четырехбайтовое) значение. В примере используется дополнительная память для уменьшения количества операций стирания, что продлевает срок службы флэш-памяти. Для удобства используемый блок является первым блоком раздела хранилища, что предотвращает использование этого примера с файловой системой SPIFFS.

Полный пример часто обновляемого целого числа доступен по адресу `$EXAMPLES/ch5-files/flash-frequentupdate`.

Инициализация блока

Первый шаг — открыть раздел хранилища:

```
let partition = new Flash("storage");
```

Как показано в листинге 5-9, следующим шагом является проверка того, инициализирован ли блок. Это делается путем поиска уникальной подписи в начале блока. Если подпись не найдена, блок стирается и подпись записывается.

Листинг 5-9.

```
const SIGNATURE = 0xa82aa82a;

let signature = partition.read(0, 4);
signature = (new Uint32Array(signature))[0];
```

```

if (signature !== SIGNATURE)
    initialize(partition);

function initialize(partition) {
    let signature = Uint32Array.of(SIGNATURE);

    partition.erase(0);
    partition.write(0, 4, signature.buffer);
}

```

Обновление значения

После подписи в блоке остается место для хранения 1023 копий счетчика. В листинге 5-10 показана функция записи, которая обновляет значение счетчика. Он ищет первое неиспользуемое 32-битное целое число в блоке и записывает туда значение. Напомним, что когда блок стирается, все биты устанавливаются в 1. Это означает, что любые неиспользуемые записи содержат значение 0xFFFFFFFF (32-битное целое число со всеми битами, установленными в 1). Если блок заполнен, он повторно инициализирует блок и записывает значение в первую свободную позицию.

Листинг 5-10.

```

function write(partition, newValue) {
    for (let i = 1; i < 1024; i++) {
        let currentValue = partition.read(i * 4, 4);
        currentValue = (new Uint32Array(currentValue))[0];
        if (0xFFFFFFFF === currentValue) {
            partition.write(i * 4, 4,
                            Uint32Array.of(newValue).buffer);
            return;
        }
    }
    initialize(partition);
    partition.write(4, 4, Uint32Array.of(newValue).buffer);
}

```

Чтение значения

Последняя часть — это функция чтения, показанная в листинге 5-11. Как и функция записи, она ищет первую свободную запись. Как только это найдено, чтение возвращает значение предыдущей записи. Если поиск достигает конца блока, возвращается последнее значение в блоке.

Листинг 5-11.

```
function read(partition) {
    let i;

    for (i = 1; i < 1024; i++) {
        let currentValue = partition.read(i * 4, 4);
        currentValue = (new Uint32Array(currentValue))[0];
        if (0xFFFFFFFF === currentValue)
            break;
    }

    let result = partition.read((i - 1) * 4, 4);
    return (new Uint32Array(result))[0];
}
```

Преимущества и будущая работа

В этом примере целочисленное значение эффективно сохраняется во флэш-памяти. Значение может быть обновлено 1023 раза, прежде чем потребуется стереть блок. Чтобы понять последствия этого, рассмотрим продукт, который обновляет это значение раз в минуту. Получается 514 операций стирания в год.

($60 * 24 * 365$, что составляет 525 600 минут в год, разделенное на 1023 обновления на цикл стирания до 514). При использовании флэш-чипа с поддержкой 10 000 операций стирания (по самым скромным оценкам) срок службы продукта составляет около 19,5 лет. Если бы каждая операция записи требовала стирания, тот же продукт изнашивался бы всего за 7 дней ($60 * 24 * 7$ — это 10 080 операций записи в неделю).

Внимательный читатель заметил два ограничения этого примера: если в функции записи пропадает питание после стирания и перед записью, текущее значение будет потеряно; и значение не может быть установлено на 0xffffffff, потому что это значение используется для идентификации неиспользуемых записей в блоке. Решения этих недостатков возможны и оставлены в качестве упражнений для читателя.

Заключение

В этой главе вы узнали о нескольких различных способах хранения информации во встроенном продукте. Файлы, настройки и ресурсы — это три основных способа хранения данных, и каждый из них оптимизирован для различного использования хранилища. Вы можете использовать любую комбинацию этих подходов в своем продукте. При разработке вашего продукта учитывайте потребности вашего хранилища, чтобы определить, какие подходы использовать для оптимального использования доступного хранилища.

Некоторые ситуации настолько специфичны, что ни один из этих стандартных методов хранения не является оптимальным; для решения таких случаев в этой главе показано, как работает флэш-память, чтобы вы могли создавать свои собственные методы хранения.

ГЛАВА 6

Аппаратная часть

Датчики и приводы являются неотъемлемой частью почти каждого продукта IoT. Датчики собирают данные из окружающей среды, такие как температура, влажность и уровень освещенности, и преобразуют их в электрические сигналы, на которые может реагировать микроконтроллер или другая система. Актуаторы делают обратное: они принимают электрические сигналы и переводят их в физические действия, например включают двигатель или свет, или воспроизводят звук.

Так же, как существуют разные сетевые протоколы, которые определяют, как данные передаются по сети, существуют разные аппаратные протоколы, которые определяют, как датчики и исполнительные механизмы взаимодействуют с микроконтроллером, к которому они подключены. Moddable SDK включает API-интерфейсы JavaScript для различных аппаратных протоколов, включая цифровые, аналоговые, PWM, сервоприводы и I2C. Эти API позволяют вам взаимодействовать с готовым оборудованием или вашими собственными схемами с вашего ESP32 или ESP8266.

В этой главе вы узнаете, как приступить к написанию собственного кода JavaScript для взаимодействия с оборудованием. Глава включает множество примеров, для которых требуется всего несколько простых, широко доступных и недорогих датчиков и исполнительных механизмов.

Код в этой главе взаимодействует с оборудованием напрямую, используя различные аппаратные протоколы. Как только вы научитесь работать с несколькими распространенными аппаратными протоколами, у вас будут знания, необходимые для включения множества аппаратных компонентов, использующих эти протоколы, в ваши собственные проекты. При подключении нового оборудования к компьютеру часто необходимо установить программный драйвер, то есть программное обеспечение, которое знает, как взаимодействовать с оборудованием через низкоуровневые аппаратные протоколы; по сути, в этой главе вы узнаете,

как писать программные драйверы для различных аппаратных компонентов. Продукты Интернета вещей, которые напрямую управляют оборудованием таким образом, имеют много преимуществ, включая более точное управление, меньший объем кода и меньшую задержку. Конечно, для многих компонентов также доступны программные драйверы; в Moddable SDK вы найдете их в модулях/драйверах.

Установка аппаратного хоста

Примеры в этой главе устанавливаются по шаблону, описанному в главе 1: вы устанавливаете хост на свое устройство с помощью `mcconfig`, затем устанавливаете примеры приложений с помощью `mcrun`.

Хост находится в каталоге `$EXAMPLES/ch6-hardware/host`.

Перейдите в этот каталог из командной строки и установите его с помощью `mcconfig`.

Примечания по монтажу

В этой главе, в отличие от большинства других глав этой книги, перед запуском большинства примеров требуется выполнить дополнительную настройку устройства: вам необходимо подключить к устройству различные датчики и исполнительные устройства. Если вы новичок в этом, это может сбивать с толку, пр монтаже. Если вы делали это раньше, вы знаете, что легко допустить ошибку и что устранение неполадок иногда может занять время. В этом разделе содержится важная информация о подключении, которую вы должны знать перед запуском примеров.

Следуя инструкциям по подключению

В этой главе приведены таблицы и схемы подключения для большинства датчиков и исполнительных механизмов, используемых в примерах. На схемах подключения показано подключение плат NodeMCU и, следовательно, номера контактов NodeMCU, например D6 или D7. Эти метки не обязательно совпадают с номером GPIO, используемым в коде. Если вы используете другую плату для разработки, обязательно посмотрите таблицы проводки, в которых указан номер GPIO, например, GPIO12 или GPIO13, вместе с выводом NodeMCU. цифры в скобках. Все макетные платы помечают контакты по-разному, поэтому вам необходимо соответствующим образом сопоставить контакты.

Модифицируемые отладочные платы помечены буквой «GP», за которой следует номер GPIO, используемый в коде, например, GP12 или GP13, поэтому, если в таблице проводки указано, что контакт должен быть подключен к GPIO12, подключите его к контакту с маркировкой GP12 на модифицируемой плате.

Устранение неполадок с соединениями

Важно внимательно следовать инструкциям по подключению. Если вы допустили ошибку в соединениях, может произойти несколько вещей:

- Выдается ошибка. Это распространенная и, как правило, самая простая проблема для устранения. Например, если вы поменяете местами контакты SDA и SCL датчика I2C, вы получите ошибки при чтении и записи. Воспользуйтесь преимуществами `xsbug` и используйте сообщения об ошибках для диагностики вашей проблемы. Иногда вам просто нужно проверить соединение; в других случаях у вас может быть неисправный датчик или привод.
- Приложение работает, но выдает неожиданные результаты. Это также распространено, но его бывает трудно уловить. Например, если вы подключите цифровой контакт датчика к неправильному контакту на макетной плате, приложение будет считывать данные с контакта, который ни к чему не подключен; он не выдает ошибку, но дает неожиданные результаты. Если вы нажимаете кнопку, а приложение не отвечает должным образом, или вы пишете на вывод трехцветного светодиода, а цвет не обновляется, дважды проверьте схему соединений.

- Вы сожжете свой датчик или привод. Это менее распространено, чем первые две проблемы, но это может случиться. Например, питание датчика 5 В, когда он рассчитан на 3,3 В, может повредить электронику датчика.

Мигание светодиода

Самый простой физический вывод, который вы можете создать с помощью ESP32 или ESP8266, — это включать и выключать встроенный светодиод (рис. 6-1). Обе платы ESP32 и ESP8266 NodeMCU имеют встроенный светодиод, подключенный к контакту 2.

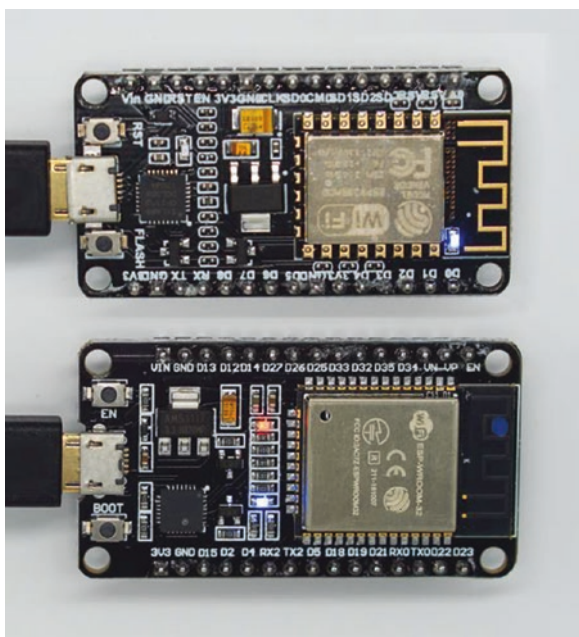


Рис. 6-1. Встроенный светодиод на ESP8266 (вверху) и ESP32 (внизу)

Класс `Digital` предоставляет доступ к контактам GPIO на вашем устройстве:

```
import Digital from "pins/digital";
```

Вы можете настроить цифровой контакт для ввода или вывода. После настройки контакт может принимать значение 1, что означает высокое напряжение, или 0, что означает низкое напряжение. Пример `$EXAMPLES/ch6-hardware/blink` использует класс `Digital` и таймер для мигания встроенного светодиода. Как показано в листинге 6-1, в примере используется метод статической записи класса `Digital`, который устанавливает контакт (указанный первым аргументом) в режим `Digital.Output` и устанавливает его значение равным 0 или 1 (второй аргумент).

Листинг 6-1.

```
let blink = 1;
Timer.repeat(() => {
  blink = blink ^ 1;
  Digital.write(2, blink);
}, 200);
```

в качестве альтернативы вы можете создать экземпляр класса `Digital` и вызвать метод записи экземпляра. Использование конструктора позволяет полностью настроить вывод. Когда вы вызываете конструктор, вы передаете словарь со свойствами вывода и режима. Для цифровых выходных контактов доступны следующие значения режима:

```
Digital.Output
Digital.OutputOpenDrain
```

В листинге 6-2 показан другой способ написания примера мигания с использованием конструктора `Digital`.

Листинг 6-2.

```
let led = new Digital({
  pin: 2,
  mode: Digital.Output
});

let blink = 1;
Timer.repeat(() => {
  blink = blink ^ 1;
  led.write(blink);
}, 200);
```

Использование экземпляра `Digital` более эффективно для записи, чем использование статического метода `Digital.write`; конструктор инициализирует контакт один раз, тогда как `Digital.write` должен инициализировать его при каждой записи. `Digital.write` удобен для нечастой записи, но если ваш проект часто записывает на цифровой выход, создайте экземпляр один раз и вместо этого записывайте в него.

Чтение кнопки

Кнопки — это простой способ добавить физический ввод в проекты. Модули NodeMCU ESP32 и ESP8266 имеют две встроенные кнопки. Одна из кнопок подключена к цифровому выводу 0 и может использоваться в качестве цифрового входа в ваших проектах; эта кнопка помечена как FLASH, BOOT или IO0, в зависимости от того, какой модуль вы используете.

Пример `$EXAMPLES/ch6-hardware/button` использует класс `Digital` и таймер для считывания встроенной кнопки. Как показано в листинге 6-3, в примере используется статический метод чтения класса `Digital`, который устанавливает контакт (указанный первым аргументом) в режим `Digital.Input` и считывает его значение, возвращая 0 или 1. Пример отслеживает до консоли отладки при каждом нажатии кнопки. Он также ведет подсчет количества нажатий кнопок и включает его в вывод.

Листинг 6-3.

```

let previous = 1;
let count = 0;
Timer.repeat(id => {
    let value = Digital.read(0);
    if (value !== previous) {
        if (value)
            trace(`button pressed: ${++count}\n`);
        previous = value;
    }
}, 100);

```

В качестве альтернативы вы можете создать экземпляр класса `Digital` и вызвать метод чтения экземпляра. Использование конструктора позволяет полностью настроить вывод. Когда вы вызываете конструктор, вы передаете словарь со свойствами вывода и режима. Для цифровых входных контактов доступны следующие значения режима:

```

Digital.Input
Digital.InputPullUp
Digital.InputPullDown
Digital.InputPullUpDown

```

В листинге 6-4 показано, как можно переписать пример кнопки для использования конструктора `Digital`.

Листинг 6-4.

```

let button = new Digital({
    pin: 0,
    mode: Digital.Input
});
let previous = 1;

```

```

let count = 0;
Timer.repeat(id => {
  let value = button.read();
  if (value !== previous) {
    if (value)
      trace(`button pressed: ${++count}\n`);
    previous = value;
  }
}, 100);

```

Другие режимы цифрового ввода

Режимы `Digital.InputPullUp`, `Digital.InputPullDown` и `Digital.InputPullUpDown` используются для включения подтягивающих и понижающих резисторов, встроенных в некоторые контакты GPIO на ESP32 и ESP8266. Это не всегда необходимо, но полезно для кнопок, подобных показанной на рис. 6-2, для которой требуется подтягивающий резистор, чтобы предотвратить получение случайного шума, когда она находится в ненажатом состоянии. Вы можете получить такие кнопки от SparkFun (идентификатор продукта COM-10302) и Adafruit (идентификатор продукта 1009).



Рис. 6-2. Тактильная кнопка

Пример `$EXAMPLES/ch6-hardware/external-button` имеет ту же функциональность, что и пример кнопки, но он работает с кнопкой, подобной той, что показана на рис. 6-2, а не со встроенной кнопкой. Если вы хотите запустить этот пример, сначала следуйте приведенным здесь инструкциям по подключению, чтобы подключить его к вашему ESP32 или ESP8266.

Инструкции по подключению ESP32

В Таб. 6-1 и на рис. 6-3 показано, как подключить кнопку к ESP32.

Таб. 6-1. Схема подключения кнопки к ESP32

Button	ESP32
PWR	3V3
DIN	GPIO16 (RX2)

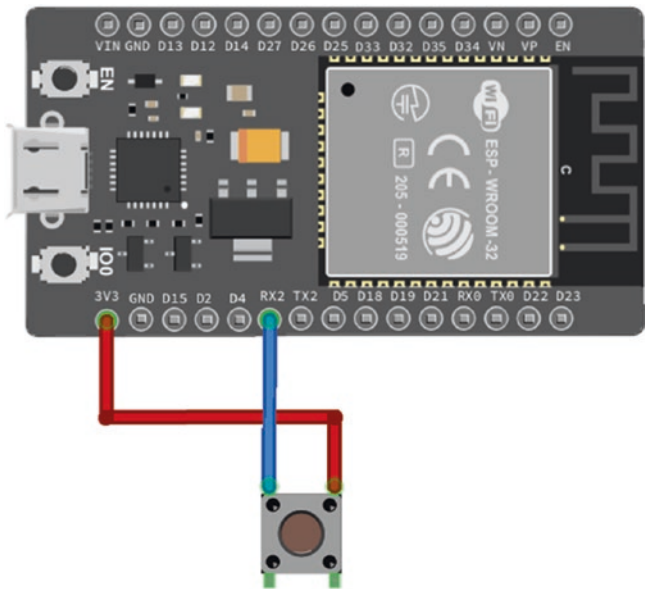


Рис. 6-3. Схема подключения кнопки к ESP32

Инструкции по подключению ESP8266

Таб. 6-2 и рис. 6-4 показывают, как подключить кнопку к ESP8266..

Таб. 6-2. *Схема подключения кнопки к ESP8266*

Button	ESP8266
PWR	3V3
DIN	GPI016 (D0)

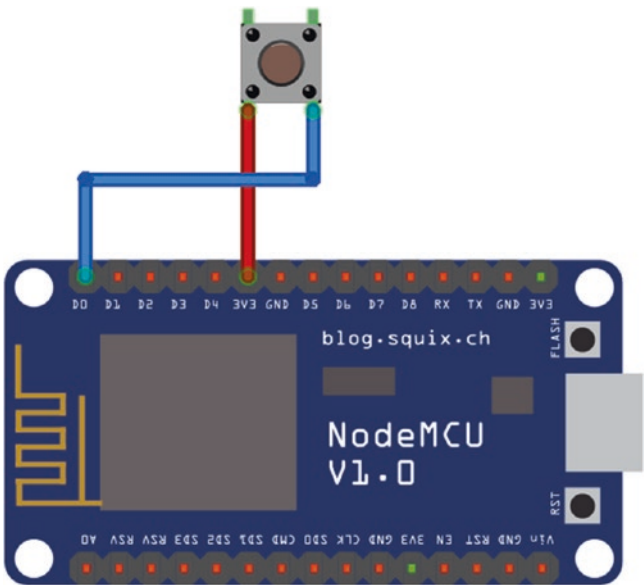


Рис. 6-4. *Схема подключения кнопки к ESP8266.*

Понимание кода внешней кнопки

В примере с внешней кнопкой используется конструктор `Digital`, как показано в следующем коде. Он настраивает контакт 16 в режиме `Digital.InputPullDown`, который включает встроенный подтягивающий резистор на контакте 16.

```
let button = new Digital({
  pin: 16,
  mode: Digital.InputPullDown
});
```

Остальная часть кода очень похожа на переписанный пример кнопки (листинг 6-4), за исключением нескольких небольших изменений, учитывающих использование подтягивающего резистора.

Подробнее о подтягивающих и ограничивающих резисторах

И ESP32, и ESP8266 имеют встроенный понижающий резистор на контакте 16, поэтому пример с внешней кнопкой выполняется на любом из них без каких-либо изменений в коде. Тем не менее, вы можете изменить его, чтобы использовать любой вывод со встроенным подтягивающим резистором, и другие приложения, которые вы создаете, также могут использовать другие выводы. ESP32 имеет встроенные подтягивающие резисторы на всех контактах GPIO, кроме контактов 34–39, тогда как ESP8266 имеет встроенный подтягивающий резистор только на контакте 16.

Для других датчиков может потребоваться подтягивающий резистор. ESP32 имеет встроенные подтягивающие резисторы на всех контактах GPIO, кроме контактов 34–39; ESP8266 имеет встроенные подтягивающие резисторы на контактах GPIO 1–15.

Вместо использования встроенных резисторов вы также можете добавить подтягивающие или подтягивающие резисторы непосредственно к датчикам. Если вы сделаете это, то сможете использовать любой GPIOpin, а не только пины со встроенным резистором. Также обратите внимание, что если вы делаете это, вы всегда должны использовать режим `Digital.Input`. Другими словами, не включайте встроенный подтягивающий резистор, если вы добавляете подтягивающий резистор к самому датчику, а также не включайте встроенный подтягивающий резистор, если вы добавляете подтягивающий резистор к датчику. сам датчик.

Мониторинг изменений

Вы можете более эффективно обнаруживать изменения значения цифрового входа, используя класс `digital Monitor`. Вместо периодического опроса он использует функцию микроконтроллера для отслеживания изменений. Экземпляр монитора настроен на запуск при изменении от 0 до 1 (то есть нарастающий фронт) и/или изменении от 1 до 0 (задний фронт). Когда оборудование обнаруживает триггерное событие, класс `Monitor` вызывает функцию обратного вызова.

В листинге 6-5 показано, как пример кнопки может использовать класс `Monitor`. Обратите внимание, что эта версия немного меньше.

Листинг 6-5.

```
let monitor = new Monitor({
  pin: 0,
  mode: Digital.Input,
  edge: Monitor.Rising
});
let count = 0;
monitor.onChanged = function() {
  trace(`button pressed: ${++count}\n`);
}
```

В исходном примере кнопки значение и предыдущие переменные используются для отслеживания состояния кнопки; использование класса `Monitor` значительно упрощает код, поскольку класс сам отслеживает состояние кнопки, уведомляя приложение только об изменении состояния.

Как и конструктор `Digital`, конструктор `Monitor` использует словарь со свойствами вывода и режима. Он также включает свойство края, указывающее события, запускающие обратный вызов `onChanged`; может быть `Monitor.Rising`, `Monitor.Falling` или `Monitor.Rising | Monitor.Falling` (Монитор.Падение). Приложение должно установить обратный вызов `onChanged` в экземпляре, который будет вызываться при возникновении указанных пограничных событий.

Использование класса `Monitor` вместо опроса имеет преимущества помимо упрощения кода. Поскольку класс использует встроенное оборудование микроконтроллера для обнаружения изменений, нет необходимости запускать какой-либо код для отслеживания изменений, освобождая циклы ЦП для другой работы. Кроме того, монитор немедленно обнаруживает изменения, в то время как метод опроса проверяет наличие изменений только каждые 100 миллисекунд. Конечно, вы можете опрашивать чаще, но это потребует еще больше циклов процессора. Кроме того, метод опроса пропускает очень быстрые нажатия кнопок, которые происходят между чтениями, в то время как монитор всегда активен и поэтому не пропускает нажатия кнопок.

Управление трехцветным светодиодом

В отличие от обычного одноцветного светодиода включения/выключения в предыдущем примере мигания, трехцветный светодиод (также называемый светодиодом RGB) объединяет три светодиода — красный, зеленый и синий — в одном корпусе, что позволяет вам точно контролировать оба цвета и яркость. Для управления тремя цветами трехцветного светодиода требуется четыре контакта: по одному для управления каждым из трех светодиодов, а также контакт питания, общий для всех цветов.

В примерах в этом разделе предполагается, что вы используете светодиод с общим анодом, подобный показанному на рис. 6-5.



Рис. 6-5. Трехцветный светодиод

Прежде чем запускать примеры, следуйте инструкциям по настройке трехцветного светодиода и инструкциям по подключению его к ESP32 или ESP8266.

Подключение светодиодов

Как показано на рис. 6-6, для светодиода требуется, чтобы ко всем контактам, кроме контакта питания, были добавлены токоограничивающие резисторы, для ограничения тока через светодиодов. Используйте резисторы 330 Ом.

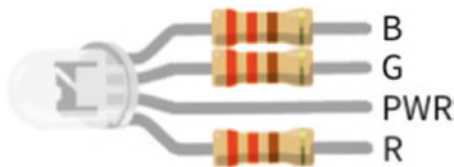


Рис. 6-6. Трехцветный светодиод с токоограничивающими резисторами

Таблица подключения к ESP32

В таб. 6-3 и на рис. 6-7 показано, как подключить светодиод к ESP32.

Таб. 6-3. Схема подключения светодиода к ESP32

LED	ESP32
PWR	3V3
R	GPIO12 (D12)
G	GPIO13 (D13)
B	GPIO14 (D14)

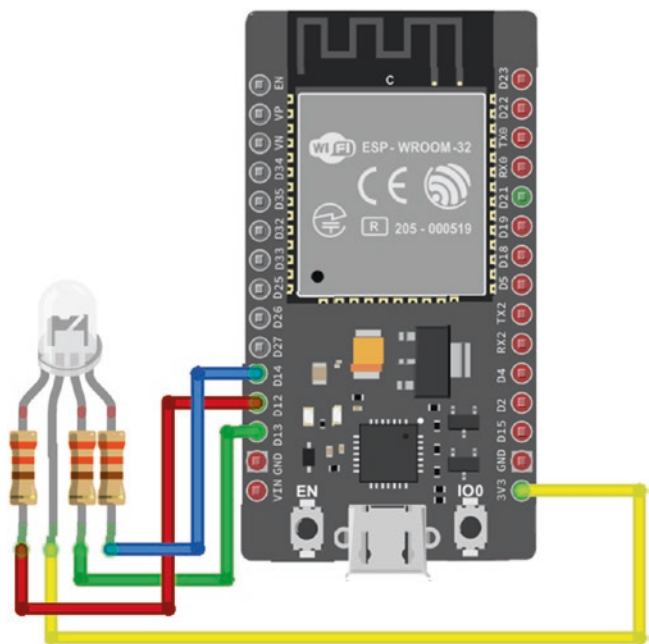


Рис. 6-7. Схема подключения светодиода к ESP32

Инструкции по подключению ESP8266

Таб. 6-4 и рис. 6-8 показывают, как подключить трехцветный светодиод к ESP8266.

Таб. 6-4.Схема подключения светодиода к ESP8266

LED	ESP8266
PWR	3V3
R	GPIO12 (D6)
G	GPIO13 (D7)
B	GPIO14 (D5)

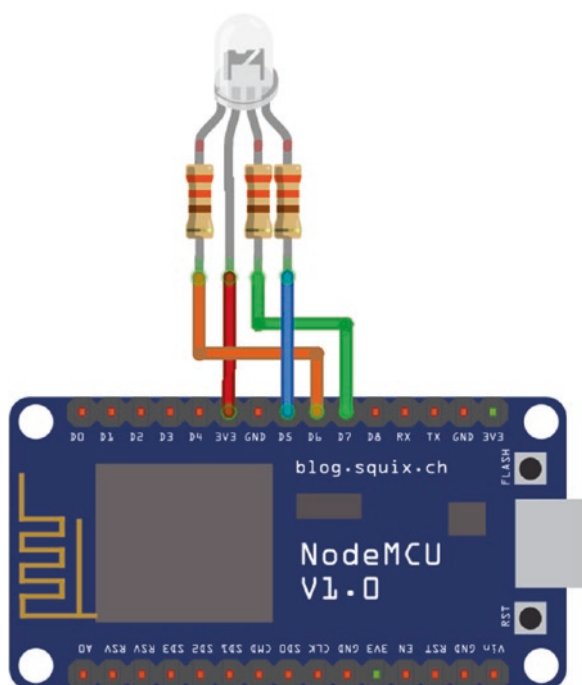


Рис. 6-8. Схема подключения светодиода к ESP8266

Использование Digital с трехцветным светодиодом

Красный, зеленый и синий контакты трехцветного светодиода подключены к цифровым выходам. В примере `$EXAMPLES/ch6-hardware/tricolor-led-digital` ими можно управлять по отдельности, используя тот же класс `Digital`, который вы использовали в примере с миганием для управления простым одноцветным светодиодом. Как показано в листинге 6-6, разница в том, что вы можете отображать восемь разных цветов, смешивая три основных цвета.

Листинг 6-6.

```
let r = new Digital(12, Digital.Output);
let g = new Digital(13, Digital.Output);
let b = new Digital(14, Digital.Output);
```

```

Timer.repeat(() => {
  // черный (все выключены)
  r.write(1);
  g.write(1);
  b.write(1);
  Timer.delay(100);

  // красный (red on)
  r.write(0);Timer.delay(100);

  // пурпурный (красный и синий включены)
  b.write(0);Timer.delay(100);

  // белый (включены все)
  g.write(0);
  Timer.delay(100);
}, 1);

```

Трехцветный светодиод может отображать не только первичные и вторичные цвета: черный, белый, красный, зеленый, синий, пурпурный, голубой и желтый. Чтобы достичь этого, вам нужно больше контроля, чем просто включение и выключение красного, зеленого и синего светодиодов; вы должны иметь возможность устанавливать для них значения между включенным и выключенным, то есть между 0 и 1. Цифровой выход не может этого сделать, так как его выход всегда либо 0, либо 1. В следующем разделе вы научитесь преодолевать это ограничение.

Использование ШИМ с трехцветным LED

Для отображения большего диапазона цветов и яркости трехцветный светодиод может вместо этого управляться с помощью широтно-импульсной модуляции или ШИМ, специального типа цифрового сигнала, обычно используемого в двигателях и светодиодах, включая трехцветные светодиоды. ШИМ примерно эквивалентен аналоговому выходу, но генерируется с использованием цифрового сигнала.

ГЛАВА 6 HARDWARE

В частности, цифровой вывод выводит прямоугольную волну с различной шириной высоких и низких значений. Усреднение этих высоких и низких импульсов во времени создает уровень мощности между высокими и низкими значениями, пропорциональный ширине импульса. В результате вместо того, чтобы ограничиваться 0 и 1 в качестве выходных значений, вы можете выводить любое промежуточное значение.

Класс PWM обеспечивает доступ к выходным контактам PWM. Пример \$EXAMPLES/ ch6-hardware/tricolor-led-pwm использует ШИМ и таймер для циклического переключения различных цветов.

```
import PWM from "pins/pwm";
```

В примере требуется три экземпляра класса PWM, по одному для каждого провода трехцветного светодиода, который управляет яркостью отдельного цвета. Конструктор ШИМ берет словарь, определяющий номер контакта.

```
let r = new PWM({pin: 12});  
let g = new PWM({pin: 13});  
let b = new PWM({pin: 14});
```

Метод записи устанавливает текущее значение вывода. Значение, которое вы передаете, представляет собой число от 0 до 1023, аналоговое значение для синтеза. Более низкие значения соответствуют более высокой яркости. Когда приложение запускается, светодиод становится зеленым. Значение ШИМ, равное 0, эквивалентно цифровому выходу, установленному на 0, а значение ШИМ, равное 1023, эквивалентно цифровому выходу, установленному на 1. Следующий код устанавливает трехцветный светодиод на зеленый, устанавливая зеленый светодиод на полную яркость и красный и синий светодиоды выключаются:

```
r.write(1023);  
g.write(0);  
b.write(1023);
```

Как показано в листинге 6-7, код циклически переключает цвета, регулируя значение яркости отдельных LED. Сначала он меняет цвет с зеленого на голубой, уменьшая значение яркости синего LED. Между вызовами записи метод задержки класса Timer используется для задержки выполнения на 50 миллисекунд.

Листинг 6-7.

```
while (bVal >= 21) {  
    bVal -= 20;  
    b.write(bVal);  
    Timer.delay(50);  
}  
b.write(1);
```

После перехода от зеленого к голубому цвет светодиода меняется от голубого к синему, от синего к пурпурному и, наконец, от пурпурного к красному (листинг 6-8).

Листинг 6-8.

```
while (gVal <= 1003) {  
    gVal += 20;  
    g.write(gVal);  
    Timer.delay(50);  
}  
g.write(1023);  
  
while (rVal >= 21) {  
    rVal -= 20;  
    r.write(rVal);  
    Timer.delay(50);  
}  
r.write(0);  
  
while (bVal <= 1003) {  
    bVal += 20;  
    b.write(bVal);  
    Timer.delay(50);  
}  
b.write(1023);
```

Вращение вала сервопривода

Сервоприводы — это двигатели, которые управляют вращающимся валом. Вал может быть точно повернут в указанное положение в пределах дуги, обычно на 180 градусов. Сервоприводы обычно используются в робототехнике для управления движениями роботов и для вращения объектов, таких как объектив камеры, для управления фокусом и масштабированием. На рис. 6-9 показан микросервопривод, доступный в Adafruit . Микро-сервоприводы, подобные этому, могут питаться от ESP32 или ESP8266. Существуют также более крупные и мощные сервоприводы для перемещения более крупных объектов; для работы этих сервоприводов требуется больше энергии, чем может обеспечить микроконтроллер, и поэтому требуется внешний источник питания.



Рис. 6-9. Микро сервопривод от Adafruit

Сервоприводы настроены с классом Servo, который использует цифровые контакты для управления серводвигателями.

Пример `$EXAMPLES/ch6-hardware/servo` вращает вал сервопривода от 0 до 180 градусов, по 2,5 градуса за раз. Прежде чем запускать пример, следуйте приведенным здесь инструкциям по подключению, чтобы подключить его к ESP32 или ESP8266.

Инструкции по подключению ESP32

В таб. 6-5 и на рис. 6-10 показано, как подключить сервопривод к ESP32.

Таб. 6-5. *Схема подключения сервопривода к ESP32*

Servo	ESP32
PWR	3V3
GND	GND
Servo (DOUT)	GPIO14 (D14)

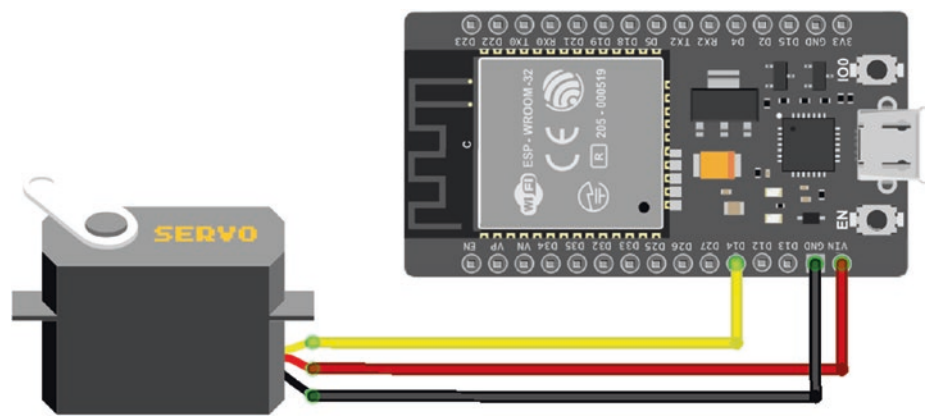


Рис. 6-10. *Схема подключения сервопривода к ESP32*

Инструкции по подключению ESP8266

В таб. 6-6 и на рис. 6-11 показано, как подключить сервопривод к ESP8266

Таб. 6-6. *Схема подключения сервопривода к ESP8266*

Servo	ESP8266
PWR	3V3
GND	GND
Servo (DOUT)	GPIO14 (D5)

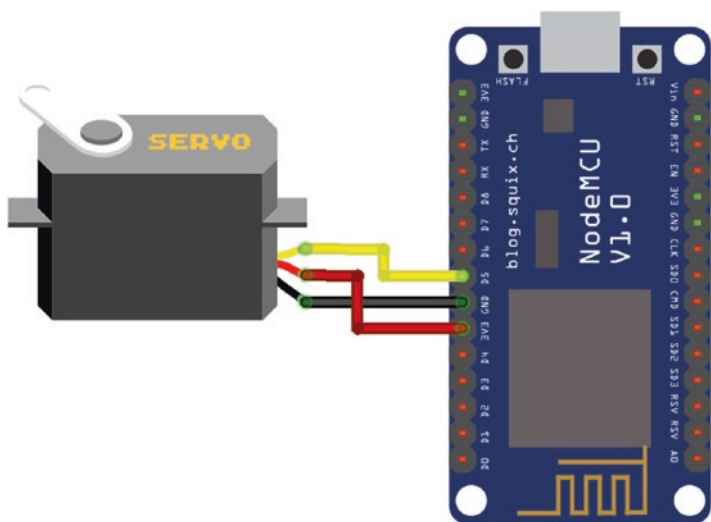


Рис. 6-11. Схема подключения сервопривода к ESP8266

Разбор кода сервопривода

Листинг 6-9.

```
let servo = new Servo({pin: 14});  
let angle = 0;  
Timer.repeat(() => {  
  angle += 2.5;
```

```

    if (angle > 180)
        angle -= 180;
    servo.write(angle);
}, 250);

```

Сервоприводу требуется время, чтобы повернуть вал в новое положение; количество времени зависит от используемого вами сервопривода. В зависимости от сервопривода использование более короткого интервала может не заставить сервопривод вращаться быстрее, а вместо этого может привести к запутанному поведению, поскольку сервопривод делает все возможное, чтобы не отставать от изменений, которые поступают быстрее, чем он может работать.

Класс Servo также имеет метод `writeMicroseconds`, который обеспечивает большую точность, позволяя указать количество микросекунд (вместо градусов) для импульса сигнала. Диапазон допустимых значений варьируется от сервопривода к сервоприводу; Установка длины импульса на слишком низкое или слишком высокое значение может привести к поломке сервопривода, поэтому обязательно проверьте техническое описание вашего сервопривода.

Измерение температуры

Измерение температуры является настолько распространенной задачей для продуктов IoT, что производители датчиков создали множество различных датчиков температуры. Эти датчики используют различные аппаратные протоколы для связи с микроконтроллером. В этом разделе описываются два простых в использовании и широко доступных датчика температуры:

- TMP36 (рис. 6-12) использует аналоговое значение для передачи температуры. Более простой из двух датчиков, он имеет только один выход — аналоговый выход, который подключается к аналоговому входу микроконтроллера — и не имеет параметров конфигурации.



Рис. 6-12. Датчик TMP36

- TMP102 (рис. 6-13) использует шину I2C для передачи данных о температуре. Он подключается с использованием аппаратного протокола I2C, с которым значительно сложнее работать, чем с аналоговым входом, но он позволяет датчику получить дополнительные функции и параметры конфигурации.



Рис. 6-13. датчик TMP102

В этом разделе также объясняется, как использовать техническое описание датчика, чтобы понять данные, предоставляемые датчиком, и преобразовать их в удобочитаемый формат.

TMP36

Пример `$EXAMPLES/ch6-hardware/tmp36` считывает температуру с датчика TMP36 и отслеживает значение в градусах Цельсия на консоли отладки. Прежде чем запускать пример, следуйте инструкциям по подключению, приведенным здесь, чтобы подключить TMP36 к вашему ESP32 или ESP8266.

Инструкции по подключению ESP32

В таб. 6-7 и на рис. 6-14 показано, как подключить TMP36 к ESP32..

Таб. 6-7. *Схема подключения TMP36 к ESP32*

TMP36	ESP32
PWR	3V3
Analog	ADC0 (VP) on NodeMCU board ADC7 (GP35) on Moddable Two
GND	GND

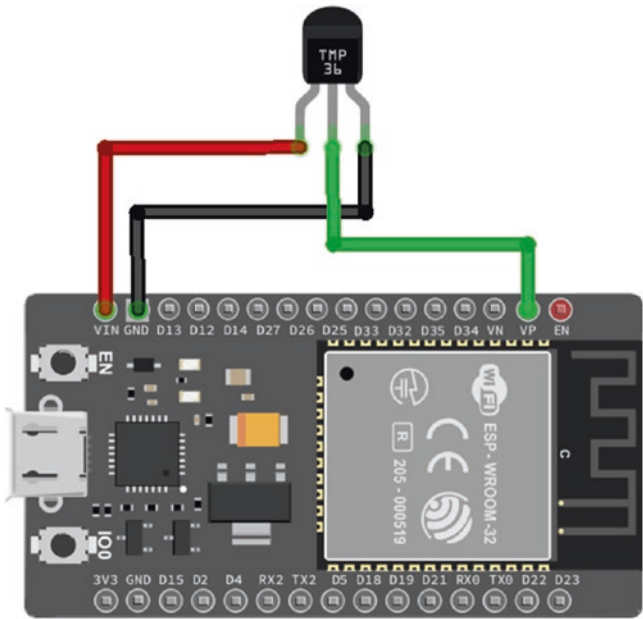


Рис. 6-14. *Схема подключения TMP36 к ESP32*

Инструкции по подключению ESP8266

В таб. 6-8 и на рис. 6-15 показано, как подключить TMP36 к ESP8266.

Таб. 6-8. Схема подключения TMP36 к ESP8266

TMP36	ESP8266
PWR	3V3
Analog	ADC0 (A0)
GND	GND

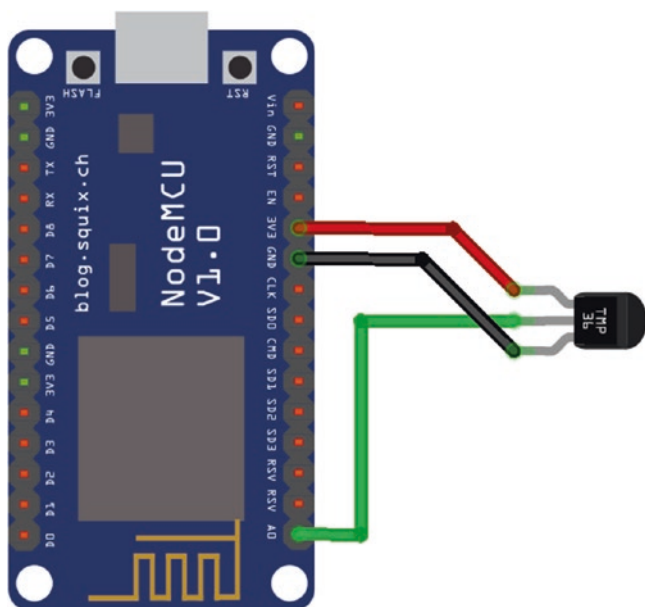


Рис. 6-15. Схема подключения TMP36 к ESP8266

Разбор кода tmp36

Аналоговый вывод на TMP36 выдает напряжение, пропорциональное температуре. Класс Analog предоставляет доступ к аналоговым входам на вашем устройстве:

```
import Analog from "pins/analog";
```

В отличие от других классов аппаратных протоколов, класс Analog никогда не создается. Он предоставляет только один статический метод: метод чтения, который производит выборку значения указанного вывода, возвращая значение от 0 до 1023. Пример tmp36 вызывает метод чтения и преобразует возвращенное напряжение в температуру.

В отличном учебном пособии Adafruit для TMP36 (learn.adafruit.com/tmp36-temperature-sensor/overview) приводится следующая формула для преобразования напряжения в температуру:

$$Temp\ in\ ^\circ C = [(V_{out\ in\ mV}) - 500] / 10$$

Пример tmp36 основан на этой формуле, как вы можете видеть здесь:

```
let value = (Analog.read(0) / 1023) * 330 - 50;
trace(`Celsius temperature: ${value}\n`);
```

Примечание Если вы используете Moddable Two, вам необходимо изменить номер контакта с 0 на 7 из-за разницы в распиновке.

TMP36 предназначен для точного измерения температуры от -40°C до $+125^{\circ}\text{C}$. Для температур за пределами этого диапазона он возвращает показания, но с меньшей точностью. Аналоговый вход имеет разрешение 10 бит, что позволяет получать показания с точностью около $0,25^{\circ}\text{C}$. Этой точности достаточно для многих целей, но не для всех; как описано далее, датчик температуры TMP102 обеспечивает более высокое разрешение при измерении температуры.

TMP102

Пример \$EXAMPLES/ch6-hardware/tmp102 считывает температуру с датчика TMP102 и отслеживает значение в градусах Цельсия до консоли отладки. Прежде чем запускать пример, следуйте инструкциям по подключению, приведенным здесь, чтобы подключить TMP102 к вашему ESP32 или ESP8266.

Этот раздел относится к схеме TMP102 и техническому описанию TMP102, которые можно найти на странице продукта TMP102 на веб-сайте SparkFun: sparkfun.com/products/13314.

Инструкции по подключению ESP32

В Таб. 6-9 и на рис. 6-16 показано, как подключить TMP102 к ESP32.

Таб. 6-9. *Схема подключения TMP102 к ESP32*

TMP102	ESP32
GND	GND
VCC	3V3
SDA	GPIO21 (D21)
SCL	GPIO22 (D22)

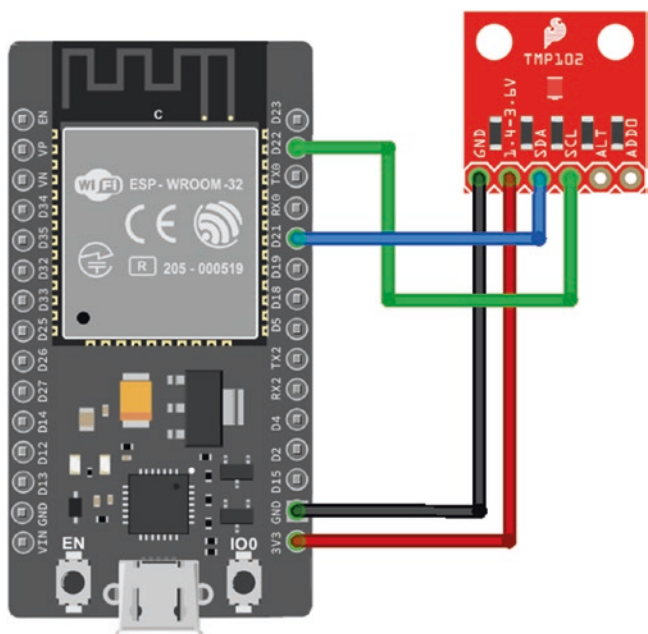


Рис. 6-16. Схема подключения TMP102 к ESP32.

Инструкции по подключению ESP8266

Таб. 6-10 и рис. 6-17 показывают, как подключить TMP102 к ESP8266.

Таб. 6-10. Подключение TMP102 к ESP8266.

TMP102	ESP8266
GND	GND
VCC	3V3
SDA	GPIO5 (D1)
SCL	GPIO4 (D2)

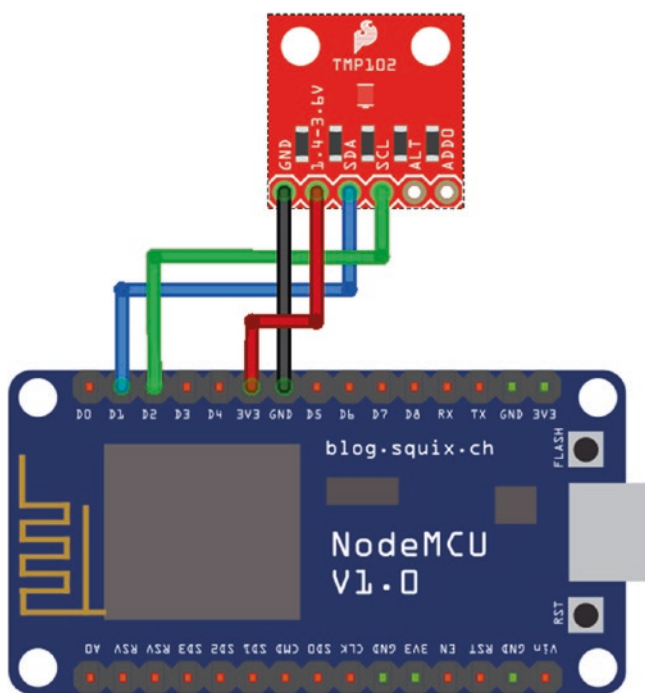


Рис. 6-17. Схема подключения TMP102 к ESP8266

Разбор кода tmp102

Пример tmp102 извлекает данные о температуре из TMP102 и преобразует их в градусы Цельсия для вывода на консоль отладки. Этот конкретный пример заслуживает внимательного рассмотрения, потому что он знакомит с аппаратным протоколом I2C, который используется в огромном количестве датчиков. I2C — это последовательный протокол для подключения нескольких устройств к одной двухпроводной шине.

После того, как вы изучите основы работы с I2C в JavaScript, вы сможете быстро написать код для связи с новым датчиком на основе обзора технического описания или примера реализации, такого как скетч Arduino. Альтернативный метод, использующий подмножество SMBus I2C, обсуждается в следующем разделе. Понимание того, как использовать I2C и SMBus, позволит вам изучить множество вариантов огромного набора доступных датчиков.

Обратите внимание, что в этой главе представлены многие, но не все возможности TMP102. Техническое описание — лучший способ узнать о возможностях любого датчика. Дальнейшее чтение о TMP102 показывает, что он включает в себя функции, предназначенные, например, для использования в термостатах.

Одной из особенностей, делающих I²C популярным, является то, что, поскольку это шина, она позволяет подключать несколько разных датчиков к одним и тем же двум контактам микроконтроллера. Каждый датчик имеет уникальный адрес, что позволяет получать к ним независимый доступ. Использование шины для этого аппаратного протокола уменьшает общее количество контактов, необходимых для подключения нескольких датчиков, что ценно, поскольку часто количество доступных контактов ограничено.

Класс I2C обеспечивает доступ к шине I²C, подключенной к паре контактов:

```
import I2C from "pins/i2c";
```

Пример tmp102 создает экземпляр класса I2C. Словарь, переданный конструктору, содержит I2C-адрес целевого устройства. Вы можете включить номера выводов в словарь, указав свойства sda и scl; в этом примере используются контакты по умолчанию для целевого устройства, поэтому номера контактов в словаре не указываются. Номера выводов по умолчанию для ESP32 или ESP8266 соответствуют разводке на предыдущих схемах. Адрес платы 0x48 указан на схеме TMP102.

```
let sensor = new I2C({address: 0x48});
```

Этот экземпляр класса I2C теперь может получить доступ к датчику на шине I²C по адресу 0x48. TMP102 поддерживает четыре 16-битных регистра, доступ к которым осуществляется через I2C с использованием операций чтения и записи. Регистры показаны в таблице 6-11. (См. также таблицы 1 и 2 таблицы данных TMP102.)

Таб. 6-11. Регистры TMP102

Регистр #	Имя регистра	Цель
0	Temperature	Прочитать самую последнюю температуру
1	Configuration	Чтение или настройка параметров скорости преобразования температуры, управления питанием и т. д.
2	T _{LOW}	Считать или установить низкую температуру при использовании встроенного компаратора
3	T _{HIGH}	Чтение или установка высокой температуры при использовании встроенного компаратора

Чтобы прочитать или записать в регистр, вы сначала записываете номер целевого регистра в устройство. После этого вы читаете или записываете значение регистра. В примере считывается температура (регистр 0), поэтому сначала в датчик записывается значение 0, а затем считываются два байта.

```
const TEMPERATURE_REG = 0;
sensor.write(TEMPERATURE_REG);
let value = sensor.read(2);
```

Метод чтения возвращает байты в именованном значении экземпляра Uint8Array. Он может считывать до 40 байтов с целевого устройства, хотя большинство операций чтения I2C занимают всего несколько байтов.

Из двух прочитанных байтов первый является самым значащим байтом. Второй байт, наименее значимый, имеет младшие 4 бита, установленные на 0, что дает разрешение 12 бит. Следующая строка кода объединяет два байта в массиве значений в 12-битное целое число:

```
value = (value[0] << 4) | (value[1] >> 4);
```

Это значение имеет формат, описанный в таблице 5 таблицы данных TMP102. Отрицательные значения представлены в формате дополнения до двух. Если первый бит значения равен 1, температура ниже 0°C, что требует дополнительных вычислений (листинг 6-10) для получения правильного отрицательного значения.

Листинг 6-10.

```

if (value & 0x800) {
    value -= 1;
    value = ~value & 0xFFF;
    value = -value;
}

```

Последний шаг — преобразовать значение в градусы Цельсия и отследить его до консоли отладки. Поскольку общее разрешение составляет 12 бит, а 4 из них используются для дробной части значения, TMP102 выдает значения температуры с точностью до 0,0625°C. Точный диапазон показаний температуры составляет от -55°C до +128°C.

```

value /= 16;
trace(`Celsius temperature: ${value}\n`);

```

Использование SMBus

Протокол System Management Bus, или SMBus, построен на основе I2C. Он использует подмножество методов, определенных I2C, для формализации соглашения, используемого многими устройствами I2C на основе регистров, включая TMP102. Как упоминалось ранее, TMP102 использует четыре регистра для чтения и записи значений между датчиком и микроконтроллером. Чтобы прочитать или записать в регистр, вы сначала отправляете номер регистра, а затем отправляете команду чтения или записи.

Вы можете использовать I2C для связи с устройствами SMBus, как и раньше, но, поскольку устройства SMBus довольно распространены, Moddable SDK для удобства включает класс SMBus:

```
import SMBus from "pins/smbus";
```

The SMBus class is a subclass of the I2C class, and its constructor accepts the same dictionary arguments. SMBus adds additional calls to I2C to read and write registers directly. In the tmp102 example, using I2C

directly to read a register requires two calls: a write to set the register to read and then the actual read. SMBus combines these two calls into a single `readWord` call.

```
let sensor = new SMBus({address: 0x48});
let value = sensor.readWord(TEMPERATURE_REG, true) >> 4;
```

Метод `readWord` принимает два аргумента: сначала регистр для чтения, а затем значение `true`, если два байта находятся в порядке от старшего к старшему, или `false`, если порядок от младшего к старшему (по умолчанию). Поскольку первый возвращаемый здесь байт является самым значащим байтом, значение имеет обратный порядок байтов, поэтому второй аргумент имеет значение `true`. Поскольку два байта уже объединены в целое число, все, что остается, — это сдвинуться вправо на 4 бита, чтобы сгенерировать 12-битное значение.

Класс `SMBus` предоставляет `readByte` для чтения одного байта и `readBlock` для чтения указанного количества байтов. Он также предоставляет соответствующие методы записи `writeByte`, `writeWord` и `writeBlock`.

Подключение TMP102

TMP102 может поддерживать различные варианты конфигурации, поскольку он обменивается данными по I²C, гибкому и расширяемому аппаратному протоколу. В этом разделе рассматриваются четыре таких варианта.

Обратите внимание, что для упрощения кода в примерах в этом разделе используется подкласс `SMBus I2C` вместо непосредственного `I2C`.

Чтение более высоких температур в расширенном режиме

TMP102 может измерять температуру до 150°C, но для этого необходимо увеличить разрешение с 12 бит по умолчанию до 13 бит, что достигается включением расширенного режима. Этот режим, как и большинство опций TMP102, управляется 16-битным регистром конфигурации, который является регистром 1. Чтобы включить расширенный режим, вы устанавливаете бит `EM` в регистре конфигурации на 1.

Поскольку регистр конфигурации управляет многими параметрами, во избежание непреднамеренного изменения параметра код (листинг 6-11) сначала считывает текущее значение регистра конфигурации, затем устанавливает бит ЕМ и, наконец, записывает значение обратно.

Листинг 6-11.

```
const CONFIGURATION_REG = 1;
const EM_MASK = 0b0000_0000_0001_0000;

let configuration = sensor.readWord(CONFIGURATION_REG, true);
sensor.writeWord(CONFIGURATION_REG, configuration | EM_MASK,
                  true);
```

В вашем собственном продукте IoT вы можете уже знать значение регистра конфигурации, не читая его. В этом случае вы можете установить его напрямую, без начального чтения.

При включенном расширенном режиме регистр температуры возвращает 13-битные значения вместо 12-битных, что требует небольшой корректировки при расчете градусов Цельсия. В версии SMBus значение сдвига вправо меняется с 4 на 3 и изменяются вычисления для отрицательных чисел. В листинге 6-12 показан измененный код.

Листинг 6-12.

```
let sensor = new SMBus({address: 0x48});

let configuration = sensor.readWord(CONFIGURATION_REG, true);
sensor.writeWord(CONFIGURATION_REG, configuration | EM_MASK,
                  true);

let value = sensor.readWord(TEMPERATURE_REG, true) >> 3;
if (value & 0x1000) {
    value -= 1;
```

```

    value = ~value & 0x1FFF;
    value = -value;
}

value /= 16;
trace(`Celsius temperature: ${value}\n`);

```

Установка коэффициента конверсии

Скорость преобразования — это количество раз в секунду, когда TMP102 выполняет измерение температуры и обновляет значение в регистре температуры. Для завершения измерения температуры TMP102 требуется около 26 миллисекунд. По умолчанию скорость преобразования составляет четыре раза в секунду. В течение 224 миллисекунд между снятием показаний и началом следующего измерения TMP102 переходит в режим пониженного энергопотребления, снижая энергопотребление примерно на 94 %, с 40 мкА до 2,2 мкА.

Знание коэффициента конверсии важно для вашего приложения. Если вы считываете температуру с датчика чаще, чем она обновляется, вы получаете одно и то же значение, без необходимости используя ограниченное количество циклов процессора. С другой стороны, если датчик выполняет показания температуры чаще, чем требуется вашему приложению, он потребляет больше энергии, чем необходимо, поскольку генерирует показания, которые не используются.

Скорость преобразования управляется двумя битами в регистре конфигурации, поэтому она имеет четыре возможных значения, как показано здесь (и в таблице 8 таблицы данных):

- **00** – раз в 4 секунды (0,25 Гц)
- **01** – раз в секунду (1 Гц)
- **10** – четыре раза в секунду (4 Гц, по умолчанию)
- **11** – восемь раз в секунду (8 Гц)

Код в листинге 6-13 устанавливает скорость преобразования восемь раз в секунду.

Листинг 6-13.

```

const CONVERSION_RATE_SHIFT = 6;
const CONVERSION_RATE_MASK = 0b0000_0000_1100_0000;

let configuration = sensor.readWord(CONFIGURATION_REG, true);
configuration &= ~CONVERSION_RATE_MASK;
sensor.writeWord(CONFIGURATION_REG,
    configuration | (0b11 << CONVERSION_RATE_SHIFT), true);

```

Экономия энергии в режиме отключения

Снижение частоты преобразования температуры позволяет экономить энергию. Однако самая низкая частота по-прежнему составляет одно преобразование каждые 4 секунды, что может быть чаще, чем требуется вашему продукту IoT. TMP102 обеспечивает режим выключения, который полностью отключает аппаратное обеспечение преобразования температуры, снижая потребление энергии до 0,5 мкА. Ваше приложение может перейти в режим отключения в промежутке между показаниями, а затем снова включить преобразование.

Код в листинге 6-14 переходит в режим отключения путем установки бита режима отключения в регистре конфигурации.

Листинг 6-14.

```

const SHUTDOWN_MODE_MASK = 0b0000_0001_0000_0000;

let configuration = sensor.readWord(CONFIGURATION_REG, true);
sensor.writeWord(CONFIGURATION_REG,
    configuration | SHUTDOWN_MODE_MASK, true);

```

Выход из режима выключения аналогичен входу, но сбрасывает бит режима выключения, а не устанавливает его:

```

let configuration = sensor.readWord(CONFIGURATION_REG, true);
sensor.writeWord(CONFIGURATION_REG, configuration & ~SHUTDOWN_MODE_MASK, true);

```

Одна важная деталь, о которой следует помнить при выходе из режима отключения, заключается в том, что, поскольку преобразования занимают около 26 миллисекунд, чтение регистра температуры сразу после выхода из режима отключения возвращает устаревшее значение. Чтобы дождаться завершения нового чтения температуры без блокировки выполнения, используйте таймер, как показано в листинге 6-15.

Листинг 6-15.

```
Timer.set(() => {
  let value = sensor.readWord(TEMPERATURE_REG);
  // Выполните преобразование в градусы Цельсия, как и
  раньше...
}, 27);
```

Однократное измерение температуры

До этого момента вы настроили датчик TMP102 на непрерывное снятие показаний температуры через равные промежутки времени. TMP102 также поддерживает однократный режим для получения только одного показания (см. листинг 6-16). Однократная функция доступна только тогда, когда устройство находится в режиме отключения, и после завершения чтения TMP102 возвращается в состояние отключения. Это делает его наиболее энергоэффективным способом снятия нечастых показаний — например, если ваш продукт снимает показания один раз в час или только в ответ на нажатие кнопки пользователем.

Листинг 6-16.

```
const ONESHOT_MODE_MASK = 0b1000_0000_0000_0000;

let configuration = sensor.readWord(CONFIGURATION_REG, true);
sensor.writeWord(CONFIGURATION_REG,
  configuration | ONESHOT_MODE_MASK, true);
```

После включения однократного режима необходимо дождаться готовности чтения — около 26 миллисекунд. Однако вместо того, чтобы ждать фиксированный интервал, вы можете использовать специальную функцию однократного режима, которая позволяет узнать, когда чтение будет готово. Это важно, потому что фактическое время, необходимое для снятия показаний, зависит от текущей температуры. После того, как вы установите одноразовый бит в 1 в регистре конфигурации, вы опрашиваете тот же самый бит, чтобы узнать, когда будет готово новое чтение; TMP102 возвращает 0 во время измерения температуры и 1, когда показания доступны. В листинге 6-17 показан код, ожидающий готовности чтения.

Листинг 6-17.

```
while (true) {
    let configuration = sensor.readWord(CONFIGURATION_REG, true);
    if (configuration & ONESHOT_MODE_MASK)
        break;
}
// теперь доступны новые показания температуры
```

Предыдущий код блокирует выполнение, ожидая показания температуры. Это приемлемо для некоторых продуктов, но не для других. Для выполнения неблокирующего опроса используйте таймер (листинг 6-18).

Листинг 6-18.

```
Timer.repeat(id => {
    let configuration = sensor.readWord(CONFIGURATION_REG, true);
    if (!(configuration & ONESHOT_MODE_MASK))
        return;
    Timer.clear(id);
    // теперь доступны новые показания температуры
}, 1);
```

Одноразовый режим имеет еще одно интересное применение. Поскольку считывание температуры занимает около 26 миллисекунд, теоретически может быть снято около 38 показаний в секунду. Однако помните, что максимальная скорость преобразования, поддерживаемая регистром конфигурации, составляет восемь раз в секунду. Использование непрерывных последовательных однократных показаний позволяет снимать показания температуры настолько быстро, насколько позволяет аппаратное обеспечение, что полезно в ситуациях, когда вы хотите точно зафиксировать изменение температуры во времени.

Заключение

Теперь, когда вы понимаете основы некоторых аппаратных протоколов и знаете, как взаимодействовать с некоторыми датчиками и исполнительными механизмами, вы можете многое сделать, чтобы сделать приведенные простые примеры более интересными. Например, вы можете заставить приводы реагировать на входные данные от датчиков или взять то, что вы узнали из главы 3 об общении с облачными службами, и использовать это для потоковой передачи данных от ваших датчиков в облако. В главах 8, 9 и 10 вы узнаете, как работать с сенсорным экраном, который отлично подходит для отображения данных датчиков и создания пользовательских интерфейсов, работающих с оборудованием.

Бесчисленное множество других датчиков и приводов доступны в Интернете и в магазинах электроники. В этой главе использовались некоторые из SparkFun и Adafruit, оба из которых являются отличными ресурсами для начинающих в области электроники. В дополнение к предложению множества датчиков и исполнительных механизмов и их спецификаций, они также предоставляют учебные пособия для многих своих продуктов, которые являются полезными отправными точками для написания собственных модулей JavaScript для взаимодействия с ними.

ГЛАВА 7

Аудио

Звук — отличный способ донести информацию до пользователя устройства. Вы можете использовать звук, чтобы обеспечить обратную связь для действий пользователя, таких как нажатие кнопки, чтобы предупредить пользователя о завершении фоновой задачи, такой как таймер или загрузка, и многое другое.

И ESP32, и ESP8266 поддерживают воспроизведение звука. Некоторые макетные платы, в том числе M5Stack FIRE, поставляются со встроенным динамиком. Если на вашей плате нет динамика, вы можете подключить его самостоятельно. В этой главе вы узнаете, как воспроизводить звуки с помощью недорогого динамика, который легко подключить непосредственно к ESP32 или ESP8266. Вы также узнаете, как добиться более высокого качества воспроизведения звука с помощью внешнего аудиодрайвера I2S и как выбрать для своего проекта оптимальный аудиоформат, сбалансировав качество и объем памяти.

Параметры динамика

Если вы не используете макетную плату со встроенным динамиком, перед запуском примеров вам необходимо подключить динамик к устройству.

На рис. 7-1 показан мини-динамик, который работает с ESP32 и ESP8266. Это простой аналоговый динамик с сопротивлением 8 Ом, который выдает 0,5 Вт мощности. Вы можете найти много других подобных с другими импедансом и мощностью. Динамик 8 Ом, 0,5 Вт — отличный девайс для начала, поскольку его можно использовать с тем же источником питания, что и ESP32 и ESP8266.



Рис. 7-1. Миниатюрный динамик

Миниатюрный динамик можно подключить напрямую к вашему устройству, и это простой способ быстро приступить к работе. Однако вы можете улучшить качество звука, добавив чип I²S. На рис. 7-2 показана микросхема I2S от Adafruit. Этот чип также усиливает звук.



Рис. 7-2. Плата I2S от Adafruit

Плата I2S сама по себе не воспроизводит звук; вам все равно придется добавить к ней динамик. Миниатюрный динамик работает с платой I2S; однако качество будет скомпрометировано недорогим динамиком. Для более качественного звука используйте динамик более высокого качества, например закрытый монофонический динамик от Adafruit (идентификатор продукта 3351), показанный на рис. 7-3.

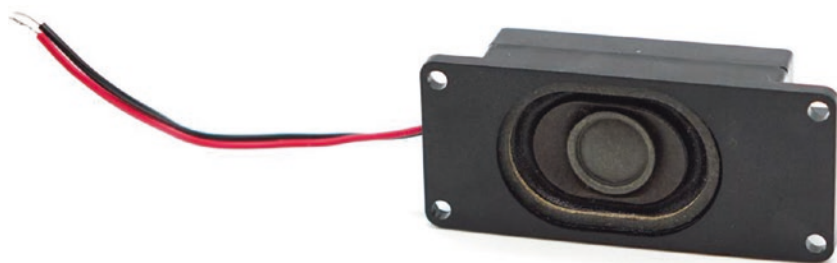


Рис. 7-3. Динамик в закрытом корпусе от Adafruit

Плата I2S требует дополнительных затрат, но она может быть необходима, если вашему продукту требуется высококачественный звук. Кроме того, использование платы I2S снижает нагрузку на ЦП на ESP8266, что также может оправдать затраты. Вы можете решить, какой вариант лучше всего подходит для вас.

Если вы просто хотите попробовать возможности воспроизведения звука в Moddable SDK, самый быстрый способ начать — с аналогового динамика. Если позже вы решите, что вам нужен звук более высокого качества, вы всегда можете переключиться на использование платы I2S и закрытого монофонического динамика. API-интерфейсы JavaScript для воспроизведения аудио идентичны независимо от того, какую настройку вы выберете, поэтому вам не придется изменять код приложения. Однако для каждого параметра необходимо настроить параметры звука по-разному. Хосты для этой главы позаботятся о конфигурации звука в своих файлах `manifest.json`. Предполагается, что вы используете динамик, показанный на рисунке 7-1, или плату I2S и динамик, показанные на рисунках 7-2 и 7-3.

Добавление аналогового динамика

В этом разделе объясняется, как подключить аналоговый динамик к ESP32 или ESP8266.

Инструкции по подключению ESP32

Таб. 7-1 и рис. 7-4 показывают, как подключить динамик к ESP32.

Таб. 7-1.Схема подключения динамика к ESP32

Speaker	ESP32
Wire 1	GPIO25 (D25)
Wire 2	GND

Неважно, какой провод динамика идет к GPIO25, а какой к GND на ESP32.

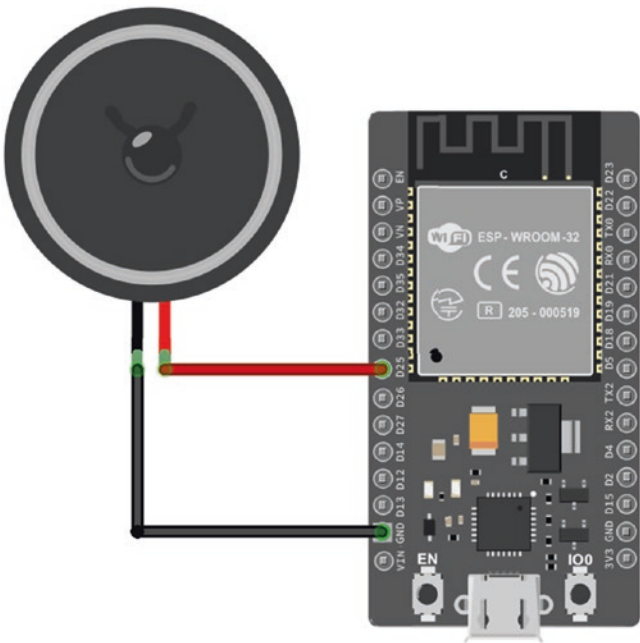


Рис. 7-4. Схема подключения динамика к ESP32.

Инструкции по подключению ESP8266

В Таб. 7-2 и на рис. 7-5 показано, как подключить динамик к ESP8266.

Таб.7-2. Схема подключения динамика к ESP8266

Speaker	ESP8266
Wire 1	GPIO3 (RX)
Wire 2	GND

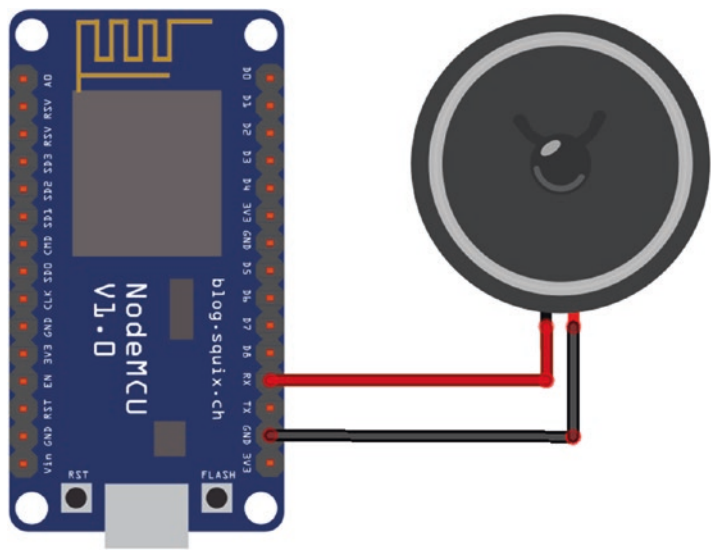


Рис. 7-5. Схема подключения динамика к ESP8266.

Обратите внимание, что GPIO3 на ESP8266 используется для последовательной связи с вашим компьютером как для установки, так и для отладки. Это означает, что вы не можете использовать xstbug для отладки аудиопримеров и что для установки аудиопримеров требуется несколько дополнительных шагов:

3. Отключите динамик от GPIO3.
4. Установите пример как обычно.
3. Снова подключите динамик к GPIO3.
4. Сбросьте ESP8266, чтобы запустить пример

Если вы используете Moddable One, GPIO3 находится на маленьком разъеме, к которому вы подключаете адаптер для программирования. После установки аудио примера отсоедините адаптер программирования, подключите динамик и используйте USB-кабель для питания Moddable One.

Неважно, какой провод динамика идет к GPIO3, а какой к GND на ESP8266.

Добавление платы I2S и динамика

В этом разделе объясняется, как подключить плату I2S к ESP32 или ESP8266 и динамик к плате I2S.

Инструкции по подключению ESP32

В Таблице 7-3 показано, как подключить плату I2S к ESP32.

Таб. 7-3. Схема подключения платы I2S к ESP32

I ² S Chip	ESP32
LRC	GPIO12 (D12)
BCLK	GPIO13 (D13)
DIN	GPIO14 (D14)
GND	GND
Vin	3V3

В таб. 7-4 показано, как подключить динамик к микросхеме I2S.

Таб. 7-4. Схема подключения динамика к чипу I²S

Speaker	I ² S Chip
Black wire	—
Red wire	+

Рис. 7-6 показывает схему соединений

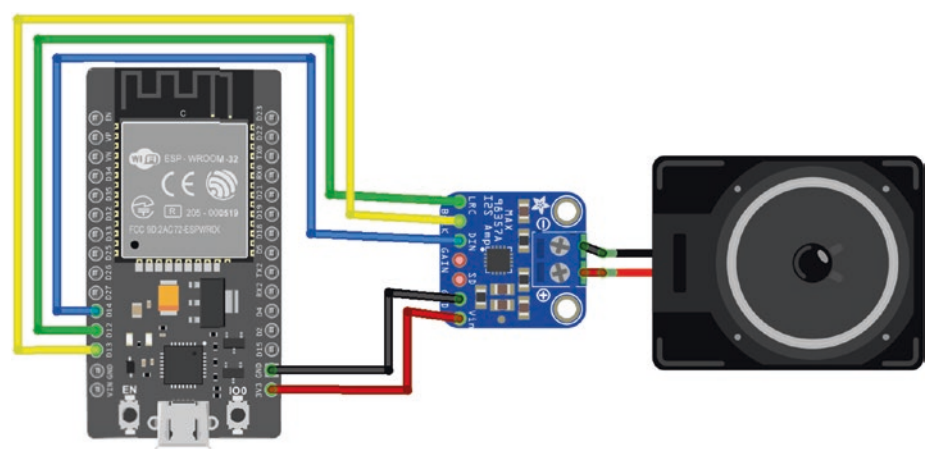


Рис. 7-6. Схема подключения динамика, платы I2S и ESP32

Инструкции по подключению ESP8266

В Таб. 7-5 показано, как подключить плату I2S к ESP8266. Обратите внимание, что GPIO2 и GPIO15 недоступны на Moddable One, поэтому вы не можете использовать I2S на Moddable One.

Таб. 7-5. Схема подключения платы I2S к ESP8266

I²S Chip	ESP8266
LRC	GPIO2 (D4)
BCLK	GPIO15 (D8)
DIN	GPIO3 (RX)
GND	GND
Vin	3V3

Таб. 7-6 показано, как подключить динамик к плате I2S.

Таб. 7-6. Схема подключения динамика к плате I2S

динамик	I ² S Chip
Черный провод	—
Красный провод	+

На рис. 7-7 показана схема соединений.

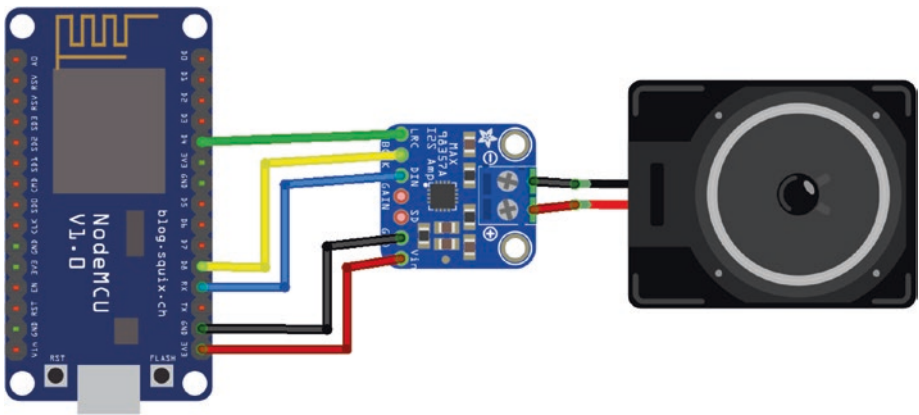


Рис. 7-7. Схема подключения динамика, платы I2S и ESP8266.

Установка аудио хоста

Примеры в этой главе устанавливаются по шаблону, описанному в главе 1: вы устанавливаете хост на свое устройство с помощью `mcconfig`, затем устанавливаете примеры приложений с помощью `mcrun`.

В каталогах `$EXAMPLES/ch7-audio/host-pdm` и `$EXAMPLES/ch7-audio/host-i2s` доступны два хост-приложения. Разница между ними заключается в том, как они настраивают параметры звука. Используйте `host-i2s`, если вы используете комбинацию чипа I2S и динамика, и `host-pdm`, если вы используете только динамик. Перейдите в каталог из командной строки и установите его с помощью `mcconfig`.

Класс AudioOut

Звук подается на динамики с помощью класса AudioOut:

```
import AudioOut from "pins/audioout";
```

Класс AudioOut поддерживает воспроизведение несжатого монофонического или стереозвuka с разрядностью 8 или 16 бит на выборку и воспроизведение монофонического звука, сжатого с использованием алгоритма IMA ADPCM. Встроенный микшер может объединять до четырех каналов звука для одновременного воспроизведения. Он может генерировать обратные вызовы в определенных точках во время воспроизведения звука, например, для синхронизации рисования на экране с воспроизведением звука. AudioOut генерирует 8-битный или 16-битный звук и отправляет его на псевдоаналоговый выход или на цифровой цифро-аналоговый преобразователь I2S.

При таком большом количестве функций работа со звуком требует понимания компромиссов между доступными параметрами, чтобы помочь вам принять решение о наилучшем способе воспроизведения звука в вашем продукте.

Конфигурация аудиовыхода

В этом разделе описываются аппаратные аудиопrotocolы, форматы данных и другие параметры конфигурации для класса AudioOut. Для примеров в этой главе параметры настраиваются в манифесте хоста.

Аппаратные аудиопrotocolы

Класс AudioOut поддерживает два различных аппаратных протокола, PDM и I²S, как описано в этом разделе.

Модуляция плотности импульсов (PDM)

Модуляция плотности импульсов, или PDM, представляет собой вариант ШИМ, который быстро переключает цифровой выходной контакт для создания уровней энергии, соответствующих желаемому выходному сигналу. Этот способ воспроизведения звука иногда называют аналоговым аудиовыходом, потому что преобразование PDM синтезирует сигнал, который при усреднении по времени соответствует уровням энергии, выводимым аналоговым сигналом.

Преимущество PDM заключается в том, что он работает только со встроенными компонентами цифрового вывода вашего микроконтроллера. Одним из недостатков PDM является более низкое качество звука; по этой причине звук PDM в первую очередь полезен для звуковых эффектов в пользовательском интерфейсе или игре, а не для музыки или произношении слов.

ESP32 имеет встроенные компоненты для преобразования аудиоданных в PDM, поэтому при использовании этого протокола нет накладных расходов на ЦП. Однако ESP8266 не имеет компонентов для преобразования PDM; преобразование происходит программно, используя некоторые циклы ЦП.

Раздел манифеста определяет параметры вывода PDM. Для ESP32 это выглядит так, как показано в листинге 7-1.

Листинг 7-1.

```
"defines": {
  "audioOut": {
    "i2s": {
      "DAC": 1
    }
  }
}
```

Если установлено значение 1, свойство DAC указывает реализации AudioOut использовать вывод PDM. Выходной контакт не указан, потому что только цифровой контакт 25 на ESP32 поддерживает вывод PDM.

Для ESP8266 раздел манифеста немного отличается (листинг 7-2).

Листинг 7-2.

```

"defines": {
  "audioOut": {
    "i2s": {
      "pdm": 32
    }
  }
}

```

Свойство `pdm` с ненулевым значением указывает на использование `PDMoutput`. Значение должно быть 32, 64 или 128. Значение 32 указывает, что при преобразовании не следует выполнять передискретизацию; это требует меньше времени и памяти, но приводит к более низкому качеству вывода. Большие значения обеспечивают лучшее качество.

I²S

Другой аппаратный протокол, поддерживаемый классом `AudioOut`, — это `I2S`, протокол, предназначенный для подключения цифровых аудиоустройств. `I2S` передает немодифицированные аудиосэмплы по цифровому соединению от микроконтроллера к выделенному компоненту аудиовыхода, который выполняет цифро-аналоговое преобразование с использованием специализированных алгоритмов и оборудования для получения высококачественного результата. И `ESP32`, и `ESP8266` имеют встроенную аппаратную поддержку для передачи аудиоданных с использованием `I2S`, поэтому микроконтроллер очень мало нагружает ЦП для воспроизведения звука.

Для использования `I2S` требуется внешний компонент, который требует дополнительных затрат и использует как минимум два, а часто и три цифровых контакта, тогда как выход `PDM` использует только один цифровой контакт. С другой стороны, аудиоаппаратура `I2S` генерирует звук очень высокого качества, поэтому ограничивающим фактором качества становится динамик, используемый для вывода, а не способ преобразования цифровых образцов в аналоговый сигнал.

Части I²S сильно различаются. У некоторых нет параметров конфигурации, в то время как другие включают соединение I²C для настройки части и не работают должным образом, пока они не будут настроены. В этом разделе предполагается, что вы повторно используете часть I²S, которая либо не требует настройки, либо уже настроена.

Раздел `defines` манифеста настраивает вывод I²S. Для ESP32 это выглядит так, как показано в листинге 7-3.

Листинг 7-3.

```
"defines": {
  "audioOut": {
    "i2s": {
      "bck_pin": 13,
      "lr_pin": 12,
      "dataout_pin": 14,
      "bitsPerSample": 16
    }
  }
}
```

Свойства `bck_pin`, `lr_pin` и `dataout_pin` соответствуют трем контактам аппаратного протокола I²S. Значения по умолчанию — 26, 25 и 22 соответственно. Свойство `bitsPerSample` указывает размер выборки в битах для передачи по соединению I²S. Для многих компонентов I²S это значение по умолчанию равно 16, но для других требуется 32 бита.

Для ESP8266 раздел манифеста намного проще, как показано в листинге 7-4, потому что контакты I²S определены аппаратно и не могут быть изменены. Установка для свойства `pdm` значения 0 отключает вывод PDM и вместо этого использует аппаратный протокол I²S. Выводы I²S: 15 (`bck_pin`), 2 (`lr_pin`) и 3 (`dataout_pin`).

Листинг 7-4.

```

"defines": {
  "audioOut": {
    "i2s": {
      "pdm": 0
    }
  }
}

```

Реализация ESP8266 поддерживает только вывод 16-битных выборок, поэтому свойство `bitsPerSample` отсутствует.

Форматы аудиоданных

Аудиоданные, которые воспроизводит ваше приложение, должны храниться в формате, совместимом с классом `AudioOut` и аппаратным обеспечением вывода звука, подключенным к микроконтроллеру. Для максимальной эффективности и простоты `AudioOut` использует специальный формат данных для хранения цифрового звука; этот формат называется MAUD, сокращение от Moddable Audio. Он состоит из простого заголовка, за которым следуют аудиосэмплы. Инструменты, которые вы используете для создания своего приложения, умеют преобразовывать стандартные аудиофайлы WAVE (файлы с расширением `.wav`), содержащие несжатый звук, в ресурсы MAUD, избавляя вас от необходимости создавать файлы MAUD самостоятельно. Инструмент преобразования называется `wav2maud` и автоматически вызывается `mcconfig` и `mcrun`. Если ваш звук хранится в другом формате, например MP3, вы должны сначала преобразовать его в файл WAVE; бесплатное приложение Audacity — хороший инструмент для этой задачи.

Для простоты класс `AudioOut` требует, чтобы все воспроизводимые аудиосэмплы имели те же биты на сэмпл, количество каналов и частоту дискретизации, что и аудиовыход. Это избавляет от необходимости выполнять преобразование формата программно на микроконтроллере. Эти свойства `AudioOut` настраиваются в манифесте, как показано в листинге 7-5.

Листинг 7-5.

```
"defines": {
    "audioOut": {
        "bitsPerSample": 16,
        "numChannels": 1,
        "sampleRate": 11025
    }
}
```

Свойство `bitsPerSample` может иметь значение 8 или 16, хотя 16 встречается чаще. Точно так же свойство `numChannels` может быть равно 1 (моно) или 2 (стерео); однако стереофонические звуки для взаимодействия с пользовательским интерфейсом на микроконтроллере воспроизводятся редко, поэтому значение обычно равно 1.

Чтобы включить аудиоданные в ваше приложение, вы добавляете их как ресурсы в манифест, как показано в листинге 7-6.

Листинг 7-6.

```
"resources": {
    "*": [
        ".bflatmajor"
    ]
},
```

Когда `mcconfig` и `mcrun` обрабатывают манифест, они вызывают `wav2maud` для преобразования файла `bflatmajor.wav` в ресурс в формате MAUD. соответствуют ресурсам, определенным в разделе `audioOut` манифеста. На основе предыдущего примера аудиосэмплы являются 16-битными моно с частотой дискретизации 11 025 Гц.

Сжатие аудио

Аудиоданные могут занимать много места для хранения. Десять секунд 16-битного монофонического звука с частотой 8 кГц занимают 160 000 байт памяти, или около 15% адресного пространства флэш-памяти ESP8266 объемом 1 МБ, и по-прежнему соответствуют качеству аналогового телефонного звонка. Сжатие аудио обычно используется для уменьшения размера аудио, хранящегося на цифровых устройствах и передаваемого через Интернет. Используемые там алгоритмы, включая MP3, AAC и Ogg, практически не работают на большинстве микроконтроллеров, поэтому здесь они непрактичны. Более простой формат сжатия аудио, IMA ADPCM (адаптивная дифференциальная импульсно-кодовая модуляция), обеспечивает сжатие 4:1 16-битных аудиовыборок и значительно менее сложен, чем MP3, AAC или Ogg, что делает его пригодным для использования в режиме реального времени на ESP32 и ESP8266.

Чтобы использовать IMA ADPCM, добавьте свойство формата в раздел `audioOut` вашего файла `manifest.json`:

```
"audioOut": {
  ... // other audioOut configuration
  "format": "ima"
}
```

Ваш звук автоматически сжимается во время сборки. 10 секунд 16-битного монофонического звука 8 кГц, упомянутые ранее, уменьшаются со 160 000 до 40 000 байт. Есть некоторое снижение качества, но для многих целей — например, для звуковых эффектов пользовательского интерфейса — разница может быть незаметной.

Установка длины аудио очереди

Длина очереди звука фиксируется во время сборки, чтобы повысить эффективность воспроизведения звука во время выполнения за счет устранения необходимости выделения памяти при изменении очереди. Длина очереди по умолчанию составляет восемь записей, что достаточно для большинства целей, включая все примеры в этой главе. Если вам нужно больше записей в очереди, вы можете изменить длину очереди, определив свойство `queueLength` в разделе `audioOut` манифеста.

```
"audioOut": {
  ... // other audioOut configuration
  "queueLength": 20
}
```

Каждая запись в очереди использует некоторую память (24 байта на момент написания этой статьи), поэтому не следует выделять больше, чем вам нужно. Если в вашем проекте просто используется аудио, вы можете уменьшить значение по умолчанию, чтобы восстановить эту память.

Playing Audio with AudioOut

Класс AudioOut предоставляет множество различных возможностей воспроизведения звука, которые помогут вам включить звуковую обратную связь в пользовательский интерфейс вашего проекта. Механизм воспроизведения способен плавно воспроизводить последовательности сэмплов. Он предоставляет механизм обратного вызова для синхронизации звука с другими частями взаимодействия с пользователем. Он даже поддерживает микширование нескольких аудиоканалов в реальном времени, что весьма необычно для микроконтроллеров. В этом разделе представлены эти и многие другие возможности.

Создание аудиовыхода

Конструктор AudioOut принимает словарь для настройки вывода звука. Пример \$EXAMPLES/ch7-audio/sound настраивает экземпляр AudioOut следующим образом:

```
let speaker = new AudioOut({streams: 1});
```

Количество потоков указывает количество звуков, которые могут воспроизводиться одновременно, максимум до четырех. Поскольку каждый поток использует некоторую дополнительную память, лучше настроить экземпляр `AudioOut` только на столько, сколько необходимо. В примере с базовым звуком воспроизводится один звук, поэтому ему нужен только один поток.

Частота дискретизации, количество битов на выборку и количество каналов определяются в манифесте, поэтому они не передаются в качестве свойств в словаре для настройки экземпляра `AudioOut`. Аудиоресурсы хранятся в том же формате, поскольку `mconfig`, `mcrun` и `wav2maud` выполняют любое необходимое преобразование формата.

Воспроизведение одного звука

Чтобы воспроизвести звук, вы сначала используете метод постановки в очередь, чтобы поставить образец звука в очередь в потоке экземпляра `AudioOut`. В примере `$EXAMPLES/ch7-audio/sound` звуковой ресурс `bflatmajor.maud` ставится в очередь в потоке 0 следующим образом:

```
speaker.enqueue(0, AudioOut.Samples,
                new Resource("bflatmajor.maud"));
```

Чтобы начать воспроизведение поставленных в очередь аудиосэмплов, вызовите метод запуска:

```
speaker.start();
```

Чтобы остановить воспроизведение всего аудио в экземпляре `AudioOut`, вызовите метод остановки:

```
speaker.stop();
```


Повторение звука

Если вы хотите воспроизвести звук более одного раза, вы можете передать необязательный параметр повторения в метод постановки в очередь. Вот как воспроизвести звук четыре раза:

```
speaker.enqueue(0, AudioOut.Samples,
                new Resource("bflatmajor.maud"), 4);
```

To repeat the sound indefinitely, pass Infinity for the repeat value:

```
speaker.enqueue(0, AudioOut.Samples,
                new Resource("bflatmajor.maud"), Infinity);
```

Использование обратных вызовов для синхронизации аудио

Метод постановки в очередь можно использовать для постановки в очередь не только звуков; вы можете, например, поставить в очередь обратные вызовы, которые будут вызываться в определенный момент воспроизведения потока. Постановка в очередь обратных вызовов полезна для запуска других событий в ответ на завершение звука, как в экранных анимациях. В листинге 7-7 обратный вызов просто ведет к консоли отладки и один раз мигает встроенным светодиодом, когда заканчивается воспроизведение звука.

Листинг 7-7.

```
speaker.enqueue(0, AudioOut.Samples,
                new Resource("bflatmajor.maud"));
speaker.callback = function() {
    trace("Sound finished\n");
    Digital.write(2, 1);
    Timer.delay(500);
    Digital.write(2, 0);
};
speaker.enqueue(0, AudioOut.Callback, 0);
```

Использование команд для изменения громкости

Вы также можете поставить в очередь команду для регулировки громкости отдельных звуков. Команда изменяет объем сэмплов, стоящих в очереди после нее; он не изменяет объем уже поставленных в очередь сэмплов. Код в листинге 7-8 воспроизводит звук три раза подряд: один раз на минимальной громкости (1), один раз на средней громкости (128) и один раз на полной громкости (256).

Листинг 7-8.

```
let bFlatMajor = new Resource("bflatmajor.maud");
speaker.enqueue(0, AudioOut.Volume, 1);
speaker.enqueue(0, AudioOut.Samples, bFlatMajor);
speaker.enqueue(0, AudioOut.Volume, 128);
speaker.enqueue(0, AudioOut.Samples, bFlatMajor);
speaker.enqueue(0, AudioOut.Volume, 256);
speaker.enqueue(0, AudioOut.Samples, bFlatMajor);
```

Воспроизведение последовательности звуков

пример \$EXAMPLES/ch7-audio/sound-sequence воспроизводит последовательность звуков. Поскольку одновременно воспроизводится только один звук, ему нужен только один поток, поэтому он настроен с теми же параметрами, что и экземпляр AudioOut в примере с базовым звуком.

```
let speaker = new AudioOut({streams: 1});
```

Затем каждый звук ставится в очередь с помощью метода постановки в очередь экземпляра AudioOut. Как показано в листинге 7-9, все звуки в примере звуковой последовательности поставлены в очередь в одном и том же потоке, заставляя их воспроизводиться последовательно в том порядке, в котором они поставлены в очередь.

Листинг 7-9.

```

speaker.callback = function() {
    speaker.enqueue(0, AudioOut.Samples,
                    new Resource("ding.maud"));
    speaker.enqueue(0, AudioOut.Samples,
                    new Resource("tick-tock.maud"));
    speaker.enqueue(0, AudioOut.Samples,
                    new Resource("tada.maud"));
    speaker.enqueue(0, AudioOut.Callback, 0);
}
speaker.callback();
speaker.start();

```

Обратный вызов ставится в очередь после образцов; как только все сэмплы воспроизведены, вызывается обратный вызов и снова ставит сэмплы в очередь, вызывая повторное воспроизведение последовательности.

Воспроизведение звуков одновременно

Пример \$EXAMPLES/ch7-audio/sound-simultaneous воспроизводит два звука одновременно, поэтому экземпляру AudioOut требуется два потока.

```
let speaker = new AudioOut({streams: 2});
```

Метод постановки в очередь экземпляра Audio Out вызывается один раз, чтобы поставить в очередь тикающий звук в потоке 0. Этот звук повторяется бесконечно.

```

speaker.enqueue(0, AudioOut.Samples,
                new Resource("tick.maud"), Infinity);
speaker.start();

```

Затем в примере задается повторяющийся таймер, обратный вызов которого ставит в очередь звук звона в потоке 1. Поскольку звук ставится в очередь в потоке, отличном от тикающего звука, оба звука воспроизводятся одновременно.

```
Timer.repeat(() => {
    speaker.enqueue(1, AudioOut.Samples,
                    new Resource("ding.maud"));
}, 5000);
```

Игра части звука

Пример \$EXAMPLES/ch7-audio/sound-clip демонстрирует, как воспроизводить части звука. Экземпляр AudioOut настроен с теми же параметрами, что и в примере с базовым звуком.

```
let speaker = new AudioOut({streams: 1});
```

Звуковой файл «тик-так» представляет собой запись часов. Сначала воспроизводится полный звук.

```
let tickTock = new Resource("tick-tock.maud");
speaker.enqueue(0, AudioOut.Samples, tickTock);
```

Затем первая половина секунды играется дважды. Чтобы воспроизвести части звуков, вы указываете необязательные аргументы повторения, смещения и подсчета метода постановки в очередь. В следующей строке повтор равен 2, поэтому звук воспроизводится дважды; смещение равно 0, поэтому оно начинается с начала звука; и count равен 11 025/2, поэтому играет полсекунды:

```
speaker.enqueue(0, AudioOut.Samples, tickTock, 2, 0, 11025 / 2);
```

Очистка очереди аудио

В некоторых ситуациях вы должны остановить воспроизведение звука на одном канале, продолжая воспроизведение на других. Это можно сделать, очистив аудиоочередь потока, который вы хотите остановить.

```
speaker.enqueue(0, AudioOut.Flush);
```

Одна из ситуаций, когда это полезно, — это когда у вас есть один канал, воспроизводящий фоновый звуковой эффект с бесконечным повторением, используя канал 0, а канал 1 — для интерактивных звуковых эффектов. Вы можете отключить фоновые звуковые эффекты, очистив канал 0, что позволяет каналу 1 продолжать воспроизведение без перерыва. Это отличается от вызова остановки экземпляра `AudioOut`, который немедленно прекращает воспроизведение на всех каналах.

Заключение

Теперь, когда вы понимаете, как настраивать параметры звука и использовать многие функции класса `AudioOut` для воспроизведения звука, вы должны приступить к добавлению звуков в свои проекты. Информация в этой главе наиболее полезна в сочетании с информацией из других глав:

- В главе 5 вы узнали, как взаимодействовать с датчиками и исполнительными механизмами. Теперь вы можете запускать звуковые эффекты в ответ на показания датчика или указывать, когда привод выполняет действие.
- В следующих нескольких главах вы узнаете, как работать с сенсорным экраном. Вы можете обеспечить звуковую обратную связь для действий пользователя на экране или добавить звуковые сигналы, чтобы привлечь внимание пользователя к дисплею. Сочетание звуковой и визуальной обратной связи обеспечивает более полное взаимодействие с пользователем.

ГЛАВА 8

Основы графики

В этой и двух последующих главах показано, насколько просто создавать современные пользовательские интерфейсы, используя только недорогой микроконтроллер и небольшой недорогой сенсорный экран. В этой главе сначала рассматривается, как добавление дисплея к вашему продукту IoT может улучшить взаимодействие с пользователем и стать гораздо более экономичным и практичным сейчас, чем в прошлом.

Последующие разделы охватывают основы графики на микроконтроллерах, включая важную информацию об оптимизации и ограничениях, информацию о том, как добавлять графические ресурсы в проекты, и введение в различные методы рисования. Более подробная информация представлена в следующих двух главах, в которых описывается следующее в Moddable SDK:

- Pico — механизм рендеринга для встроенных систем, который можно использовать для рисования на дисплеях.
- Picu, объектно-ориентированная среда пользовательского интерфейса, которая использует Pico для рисования и упрощает процесс создания сложных взаимодействий с пользователем.

Обладая этими знаниями, вы должны приступить к созданию продуктов IoT со встроенными дисплеями и объяснить своим друзьям и коллегам, что эта цель достижима для ваших продуктов.

Зачем добавлять дисплей?

Дисплеи с красиво оформленными пользовательскими интерфейсами сегодня воспринимаются как должное на компьютерах и мобильных телефонах; однако они остаются редкостью в продуктах IoT. Вы, вероятно, знакомы с тем, как сложно настроить и использовать продукты Интернета вещей с крайне ограниченным пользовательским интерфейсом, например, устройства с одной кнопкой и несколькими мигающими индикаторами. Кажется очевидным, что добавление дисплея ко многим из этих продуктов улучшит взаимодействие с пользователем и сделает продукт более ценным для покупателя. Вот лишь некоторые из преимуществ, которые следует учитывать:

- Дисплей передает гораздо больше информации, чем несколько пульсирующих индикаторов состояния или предупреждающие звуки. Дисплей подробно показывает пользователю, что делает продукт, и, если есть проблема, он сообщает пользователю, что пошло не так.
- Дисплей позволяет включить полные параметры конфигурации для всех возможностей продукта. Такой уровень точности настройки, как правило, недоступен при наличии нескольких кнопок и ручек.
- Дисплей позволяет пользователю выполнять сложные действия напрямую, без необходимости использования другого устройства. Сравните это с загрузкой и установкой пользователем мобильного приложения для взаимодействия с продуктом и сопряжением приложения с продуктом, прежде чем он сможет начать его настройку.
- Графическое богатство дисплея позволяет комбинировать изображения с анимацией, чтобы придать продукту стиль и характер, сделать его более приятным для пользователя и укрепить имидж бренда производителя.

С таким большим количеством преимуществ использования дисплея, почему бы больше продуктов IoT не включать его? Основная причина - стоимость. Продукты IoT, которые включают в себя дисплеи, как правило, представляют собой модели высокого класса, часто так называемые продукты-герои, которые предназначены для демонстрации бренда, но не предназначены для продажи в большом количестве. Но действительно ли добавление экрана к продукту непомерно дорого? В свое время ответ был утвердительным. Вот некоторые из распространенных причин, по которым производители не добавляют дисплеи в свои продукты:

- Сам дисплей дорогой. Небольшой сенсорный экран может стоить 20 долларов без добавления микроконтроллера и коммуникационных компонентов.
- Программа для взаимодействия с дисплеем требует увеличения объема оперативной памяти, а графические активы (изображения и шрифты) для создания пользовательского интерфейса требуют увеличения объема памяти.
- Для достижения приемлемой частоты кадров для анимации необходим специальный микропроцессор с аппаратным ускорением графики, то есть графический процессор.
- Программирование графики для микроконтроллеров требует узкоспециализированных навыков, что усложняет поиск квалифицированных инженеров и удорожает их найм.
- Затраты на лицензирование графических и пользовательских интерфейсов SDK для микроконтроллеров слишком высоки.
- Подготовка графических ресурсов для встроенных систем требует много времени и чревата ошибками.

Все это было вескими причинами в прошлом, но сегодня ситуация совсем другая. К сожалению, большинство планировщиков продуктов, дизайнеров и инженеров, работающих над продуктами IoT, не знают, что можно получить сенсорный экран, микроконтроллер, ОЗУ и ПЗУ для предоставления красиво оформленного современного пользовательского интерфейса менее чем за 10 долларов, даже для продуктов с очень

низкими ценами. Кроме того, этот же микроконтроллер также может обеспечивать поддержку Wi-Fi и Bluetooth. Проблемы с программным обеспечением и активами решают Pico и Pi+.

Преодоление аппаратных ограничений

Аппаратные ресурсы современных компьютеров и мобильных телефонов предназначено для выполнения чрезвычайно сложных графических операций с удивительной эффективностью. Эта замечательная производительность достигается за счет сочетания сложного аппаратного и программного обеспечения. Неудивительно, что типичный микроконтроллер не имеет такого же графического оборудования и ему не хватает скорости и памяти для выполнения тех же сложных графических алгоритмов.

Естественным следствием этих различий является то, что, когда продукты IoT на базе микроконтроллеров включают дисплей, предоставляемые ими пользовательские интерфейсы часто кажутся довольно примитивными, как у компьютеров и видеоигр на заре эпохи персональных компьютеров в 1980-х и начале 1990-х годов. В некотором смысле это имеет смысл, потому что, как и ранние персональные компьютеры и видеоигры, эти микроконтроллеры значительно менее мощные, чем современные компьютеры. Но сегодня ESP32 работает в шесть раз быстрее, чем микропроцессор в первом Macintosh IIx 1992 года, поэтому производительности современного микроконтроллера достаточно, чтобы соответствовать или превосходить результаты этого древнего продукта.

Moddable SDK обеспечивает отличные графические результаты на микроконтроллере, применяя методы, использовавшиеся в раннем оборудовании, перед современными высокоскоростными шинами дисплея, большим объемом памяти и графическими процессорами. Реализации вдохновлены классическими методами, которые успешно использовались для компьютерной анимации, видеоигр, шрифтов и многого другого. Современные микроконтроллеры по-прежнему ограничены в памяти, но они быстрее, поэтому можно выполнять больше вычислений. Это делает возможным использование некоторых методов, которых не было на старых устройствах.

Детали того, как работают эти техники, выходят за рамки этой книги. Весь код, который их реализует, доступен вам в Moddable SDK, если вы хотите узнать больше. В этой книге основное внимание уделяется тому, как использовать эти возможности для создания отличного пользовательского интерфейса для вашего продукта.

Скорость пикселей влияет на частоту кадров

В современных мобильных приложениях и веб-страницах частота кадров является основным показателем производительности графики. Более высокая частота кадров обеспечивает более плавную анимацию. Графический процессор в компьютерах и мобильных телефонах настолько мощный, что способен обновлять каждый пиксель дисплея в каждом кадре. По разным причинам микроконтроллер просто не может сделать то же самое; однако можно добиться анимации со скоростью 30 или даже 60 кадров в секунду (fps).

Поскольку микроконтроллер не может отображать высокую частоту кадров при обновлении всего дисплея, решение состоит в том, чтобы обновить только часть дисплея. Пользовательский интерфейс можно спроектировать таким образом, чтобы одновременно обновлялись только относительно небольшие части экрана. Это значительно сокращает работу микроконтроллера, поэтому пользователь видит плавную анимацию с высокой частотой кадров, как в мобильном приложении или на веб-странице.

Чтобы добиться высокой частоты кадров с помощью микроконтроллера, полезно думать о частоте пикселей — количестве пикселей, обновляемых в секунду. ESP32 и ESP8266 используют шину SPI для связи с дисплеем, и это соединение работает на частоте 40 МГц, обеспечивая скорость около 1 000 000 пикселей в секунду, около 15 кадров в секунду. Достижение полной теоретической скорости пикселей обычно невозможно из-за других факторов; тем не менее, если ваше приложение обновляет только около 40 % пикселей в каждом кадре (скорость пикселей составляет около 30 000 пикселей на кадр), оно может обеспечить надежную частоту кадров 30 кадров в секунду.

На дисплеях QVGA (320 x 240), используемых в этой книге, 30 000 пикселей составляют около 40% общей площади экрана, что более чем достаточно для создания плавного, убедительного пользовательского интерфейса. Обновление только 10 000 пикселей на кадр обеспечивает скорость 60 кадров в секунду.

Вы можете ожидать, что область экрана, обновляемая в каждом кадре, должна быть одним прямоугольником. Это ограничило бы возможности дизайна, ограничив движение одной областью дисплея. К счастью, это не так. Как вы скоро узнаете, у вас может одновременно обновляться несколько различных областей, что может создать у пользователя впечатление движения на полном экране, даже если изменяется лишь часть реальных пикселей.

Рамки для рисования

Большинство графических библиотек, используемых для микроконтроллеров, являются непосредственными API-интерфейсами режима, что означает, что средство визуализации выполняет запрошенную операцию рисования, когда вы вызываете функцию рисования. Росо, с другой стороны, представляет собой рендерер с сохраненным режимом, который работает следующим образом:

1. Ты сообщаем Поко, когда начинаешь рисовать.
2. Когда вы вызываете функции рисования, они не рисуются сразу, а добавляются в список команд рисования.
3. Когда вы сообщаете Росо, что закончили рисовать, он выполняет все команды рисования.

Большинство графических библиотек, предназначенных для микроконтроллеров, являются непосредственными интерфейсами API-интерфейса, что означает, что это требует выполнения запрошенной функции извлечения, когда вы вызываете функцию рисования. Росо, с другой стороны, представляет собой рендерер с сохраненным режимом, который работает следующим образом:

- Рендеринг в сохраненном режиме устраняет мерцание. Например, когда вы рисуете фон экрана в средстве визуализации непосредственного режима, все пиксели экрана рисуются фоновым цветом; когда вы затем рисуете элементы управления, текст и изображения, составляющие пользовательский интерфейс, пользователь может сначала увидеть фоновый экран без этих элементов пользовательского интерфейса, вызывая отвлекающее мгновенное мерцание

- Поскольку рендеринг в сохраненном режиме выполняет все команды рисования перед отправкой результата на экран, он сочетает стирание фона с отрисовкой элементов пользовательского интерфейса на микроконтроллере перед их передачей на дисплей, тем самым устраняя мерцание.
- Сохраненный режим повышает производительность за счет уменьшения количества пикселей, передаваемых с микроконтроллера на дисплей. Учтите, что в каждом пользовательском интерфейсе есть несколько перекрывающихся пикселей, например фон кнопки и ее текстовая метка. В режиме визуализации с анимацией перекрывающиеся пиксели дважды отправляются на дисплей, тогда как в режиме остаточной визуализации каждый пиксель отправляется только один раз за кадр. Поскольку рендерить пиксель гораздо быстрее, чем передавать его на дисплей, это увеличивает общую частоту пикселей.

Сохраненный режим улучшает качество рендеринга за счет эффективного смешивания пикселей. В современной компьютерной графике широко используется смешивание для сглаживания краев объектов — например, сглаживание шрифтов для устранения острых краев («неровностей»). Это одна из причин, по которой текст на современных компьютерах и мобильных телефонах выглядит намного четче, чем текст на экране в 1980-х годах. Смешивание в вычислительном отношении более сложное, и для этого достаточно производительности, потому что микроконтроллеры намного быстрее; однако для смешивания также требуется доступ к пикселю за пикселем, который вы рисуете в данный момент. В типичном аппаратном обеспечении микроконтроллера предыдущий пиксель хранится в памяти дисплея, а не в памяти микроконтроллера, что делает его либо полностью недоступным, либо непрактично медленным для доступа.

Рендерер с сохраненным режимом, поскольку он передает пиксели на дисплей только тогда, когда они полностью визуализированы, всегда имеет текущее значение пикселя, доступное в памяти, и поэтому может эффективно выполнять смешение.

Существуют и другие преимущества рендереров с сохраненным режимом, но этих трех должно быть достаточно, чтобы убедить вас в том, что затраты памяти и сложности оправдывают использование рендерера с сохраненным режимом, такого как Pico, вместо более распространенного рендерера с немедленным режимом. Качество рендеринга пользовательского интерфейса настолько выше, что у пользователей создается впечатление, что они используют более качественный продукт — тот, который принадлежит их компьютеру и телефону, а не выставлен в музее компьютерной истории.

Рендеринг строки сканирования

Дисплей QVGA имеет 76 800 пикселей, что означает, что дисплею с 16-битными пикселями требуется 153 600 байт памяти для хранения одного полного кадра. ESP8266 имеет около 80 КБ общей памяти — достаточно только для половины кадра, если ваш продукт IoT не использует любую другую память! У ESP32 гораздо больше, но тем не менее, хранение всего кадра в памяти использует 50% или более от общего объема свободной памяти при запуске. Дисплеи, используемые в этой книге, имеют память для одного кадра, поэтому микроконтроллеру не нужно хранить весь кадр, но ему нужна память для визуализации кадра. Чтобы свести к минимуму требуемый объем памяти, модуль визуализации Pico реализует визуализацию по линии развертки — метод, который делит кадр на горизонтальные полосы размером всего в один ряд пикселей; после рендеринга каждой полосы она сразу же передается на дисплей. Этот подход более сложен, чем одновременный рендеринг всего кадра, но он снижает требования к памяти для рендеринга для одного 16-битного дисплея QVGA со 153 600 байт до 480 байт — одна 240-пиксельная строка развертки с двумя байтами на пиксель — экономия памяти 99,68%!

Каждая визуализируемая полоса приводит к некоторому снижению производительности, поэтому есть преимущество в уменьшении количества полос за счет увеличения их размера, но, конечно, это также увеличивает объем необходимой памяти. Преимущество в производительности несколько снижается с каждой добавленной строкой в полосу, поэтому увеличение более восьми строк сканирования обычно нецелесообразно. Если в вашем проекте есть немного свободной памяти или требуется рендеринг с максимальной производительностью, вы можете захотеть, чтобы Росо рендерил несколько строк развертки за раз; в следующих главах объясняется, как это настроить.

Многие современные микроконтроллеры, в том числе ESP32 и ESP8266, могут асинхронно использовать SPI для передачи данных на дисплей, что означает, что микроконтроллер может выполнять другую работу во время передачи этих данных. Росо использует асинхронную передачу SPI для рендеринга следующей части дисплея, в то время как предыдущая часть передается на дисплей, и эта простая параллельная обработка позволяет значительно повысить производительность. Чтобы использовать этот метод, у Росо должно быть достаточно памяти, чтобы хранить в памяти как минимум две строки развертки: предыдущую строку развертки, которая сейчас передается, и текущую строку развертки, которая сейчас обрабатывается. Поскольку этот метод обеспечивает такое значительное увеличение производительности, Росо по умолчанию выделяет две строки сканирования.

Ограничение области рисования

Как вы видели в разделах «Частота пикселей влияет на частоту кадров» и «Сканлайн-рендеринг», ключевым методом, используемым с графикой на микроконтроллерах, является обновление дисплея по частям, а не все сразу. Обратите внимание на следующие аспекты этой техники у Росо и Пиу:

- **In Росо** – Функция механизма рендеринга Росо, позволяющая ограничивать отрисовку частями дисплея, называется отсечением. Росо использует один прямоугольник для описания области отсечения; часть каждой операции рисования, которая пересекает этот прямоугольник отсечения, рисуется, тогда как любая часть

операции, выходящая за пределы прямоугольника отсечения, не рисуется. Эта функция используется Росо для реализации рендеринга строки развертки (и `Piu` для частичного обновления кадров). Он также доступен для использования в ваших приложениях, например, для рисования подмножества изображения.

- **В `Piu`** — обновление минимально возможной площади дисплея повышает производительность рендеринга на микроконтроллерах; однако определить наименьшую возможную область для обновления в общем случае довольно сложно. Росо не может определить для вас оптимальные области рисования, потому что, будучи механизмом рендеринга, он не знает, что рисует ваш код.

`Piu`, с другой стороны, представляет собой структуру пользовательского интерфейса с полным знанием различных объектов, составляющих ваш экраный дисплей. В результате `Piu` может вычислить для вас наименьшие возможные области обновления автоматически, за кулисами.

Чтобы понять проблемы вычисления минимально возможной области обновления, давайте рассмотрим пример прыгающего мяча. В каждом кадре мяч перемещается на некоторое количество пикселей. На рис. 8.1 мяч перемещается на несколько пикселей вниз и вправо. Наименьший прямоугольник, который охватывает предыдущее и текущее положение мяча, является хорошей первой оценкой наименьшей возможной области экрана для обновления.

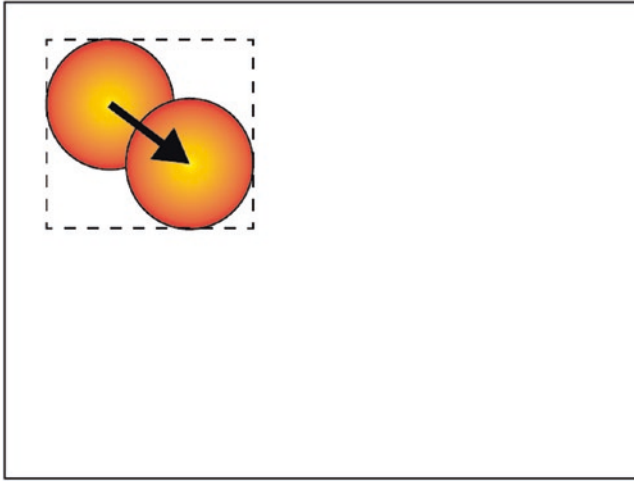


Рисунок 8-1. Мяч слегка движется, один прямоугольник обновления

Теперь рассмотрим случай, когда мяч перемещается на гораздо большее расстояние (рис. 8-2): наименьший прямоугольник, охватывающий предыдущую и текущую позиции, включает в себя множество пикселей, которые на самом деле не изменились, но они перерисовываются, потому что включены в области, которую нужно обновить.

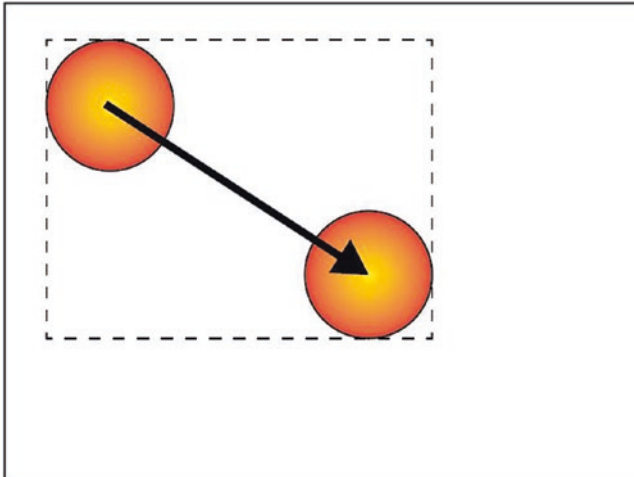


Рис. 8-2. Мяч движется дальше, один прямоугольник обновления

Как показано на рис. 8-3, в этом случае *Piu* распознает, что более эффективно обновлять две отдельные области: область, охватывающую предыдущее местоположение мяча, которая заполняется цветом фона, и область, охватывающую текущее местоположение мяча. .

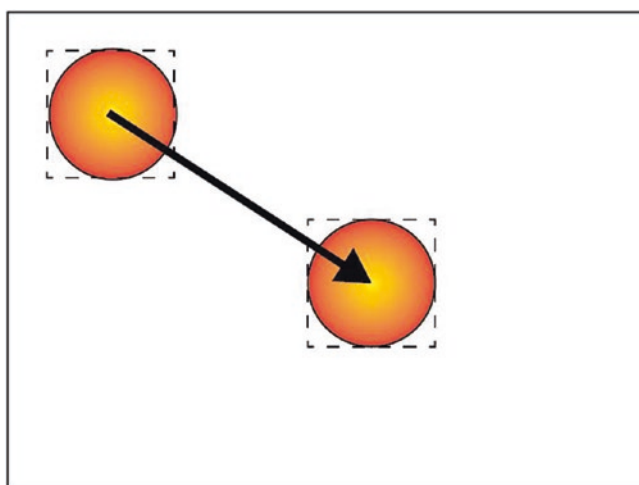


Рис.8-3. Мяч движется дальше, два прямоугольника обновления

Piu на самом деле идет еще дальше. В первом примере, где мяч перемещается лишь на небольшое расстояние — расстояние, при котором текущая позиция перекрывается с предыдущей позицией, — *Пиу* понимает, что один ограничивающий прямоугольник не является наименьшей возможной областью обновления; следовательно (как показано на рис. 8-4), в данном случае обновляются три отдельных прямоугольника, что позволяет избежать ненужного обновления множества неизменившихся фоновых пикселей.

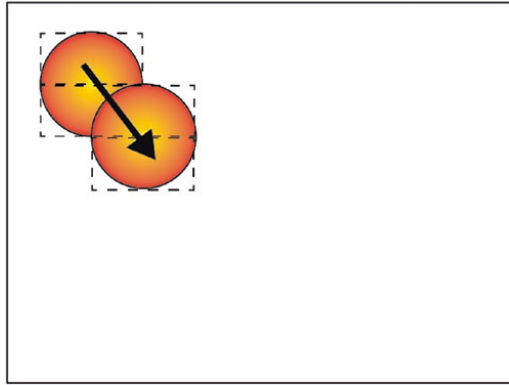


Рис. 8-4. Мяч слегка движется, три прямоугольника обновления

Вычисления, связанные с оптимизацией области рисования для одного прыгающего мяча, уже удивительно сложны, и они были бы еще более сложными в приложении с несколькими прыгающими мячами, которые иногда перекрываются. `Piu` автоматически вычисляет для вас минимальный набор прямоугольников; это требует времени и памяти, но повышение производительности, которое он дает, делает его стоящим. Это связано с тем, что производительность рендеринга в значительной степени ограничена частотой пикселей вашего приложения, а `Piu` автоматически минимизирует частоту пикселей вашего кода.

Пиксели

Каждый дисплей содержит пиксели, но не все дисплеи имеют одинаковые типы пикселей. Пиксели бывают разных размеров и цветов. Так было всегда, но об этом легко забыть, потому что почти все современные компьютеры и мобильные устройства поддерживают один и тот же формат 24-битных цветовых пикселей. Как и во многих других областях разработки встраиваемых систем, разнообразие форматов пикселей отчасти является следствием стремления снизить затраты на оборудование. Дисплей, способный отображать цвета, как правило, стоит дороже, но существуют и другие факторы, помимо стоимости, которые влияют на используемый формат пикселей.

Например, дисплей ePaper (часто называемый по названию компании-первопроходца, E Ink), в котором используется технология, способная отображать лишь несколько цветов — обычно черный, белый и несколько оттенков серого — не имеет нужен формат пикселей, который содержит более нескольких цветов.

Пиксельные форматы

Большинство дисплеев поддерживают один тип пикселей. Цветные дисплеи QVGA, используемые в большинстве примеров в этой книге, используют цветной 16-битный пиксель с 5 битами для красного, 6 битами для зеленого и 5 битами для синего. Ваш мобильный телефон, вероятно, имеет 24-битные цветные пиксели, по 8 бит для красного, зеленого и синего цветов. В то время как оба типа пикселей подходят для отображения полноцветных пользовательских интерфейсов, 24-битные цветные пиксели могут отображать в 256 раз больше цветов (16 777 216 против 65 536). Это различие означает, что изображения на встроенном устройстве могут иметь менее четкий вид, особенно в областях, заполненных похожими цветами, например на закате. Это может быть проблемой для фотографий, но, как правило, это не проблема для пользовательских интерфейсов, управляемых микроконтроллерами, если дизайн интерфейса учитывает это ограничение.

В дополнение к 16-битному цвету некоторые дисплеи поддерживают только 8-битный цвет. Это гораздо более ограничено, позволяя использовать только 256 цветов. Каждый пиксель содержит 3 бита для красного, 3 бита для зеленого и 2 бита для синего. Можно создать разумный пользовательский интерфейс с дисплеем, использующим этот тип пикселя, но требуется некоторая работа, чтобы тщательно выбрать цвета, которые хорошо выглядят в заданных пределах. В некоторых случаях может быть полезно использовать 8-битные цветные пиксели на дисплее, который поддерживает 16-битные пиксели. Это явно не улучшает качество, но уменьшает объем памяти, необходимый для ресурсов, и время, необходимое для рендеринга изображений. Если вы обнаружите, что ваш проект с трудом помещается в доступное пространство для хранения или если производительность рендеринга не совсем то, что вам нужно, использование 8-битных цветных пикселей на 16-битном дисплее может быть жизнеспособным решением.

Существуют также 4-битные цветные пиксели, но с ними настолько сложно добиться профессионального результата, что они здесь не рассматриваются. Однако 4-битные серые пиксели, которые могут отображать 14 уровней серого плюс черный и белый, очень полезны. Дисплей ePaper, который не может отображать цвета, нуждается только в серых пикселях; поскольку большинство дисплеев ePaper способны отображать только несколько уровней серого, достаточно 4-битного серого пикселя. Рендеринг в градациях серого выполняется даже быстрее, чем цветной. Вы можете использовать 4-битные серые пиксели с 16-битным цветным дисплеем, чтобы сэкономить еще больше места для хранения. Существуют также 8-битные серые пиксели, которые могут отображать 254 уровня серого плюс черный и белый; они обеспечивают превосходное качество, но для многих практических целей 4-битный рендеринг серых почти неотличим по качеству от 8-битных серых пикселей.

Некоторые дисплеи просто черно-белые. Эти дисплеи имеют тенденцию быть маленькими и низкокачественными и использоваться больше для промышленных продуктов IoT, чем для потребительских продуктов IoT. Для этих дисплеев достаточно 1-битного пикселя, однако хороший рендеринг при 1-битном пикселе очень сложен. Picosrenderer не поддерживает отображение 1-битных пикселей напрямую. Вместо этого драйвер дисплея получает 4-битные серые пиксели, а затем уменьшает изображение до 1-битного при передаче его на дисплей.

Настройка хоста для пиксельного формата

В главе 1 вы узнали, как создать хост с помощью инструмента командной строки `mcconfig`. В командной строке вы используете параметр `-p` для передачи имени аппаратной платформы, на которую вы ориентируетесь, например, `-p esp32` для сборки ESP32. Для целевых устройств, которые включают в себя дисплей, таких как макетные платы от Moddable и M5Stack, автоматически настраивается формат пикселей по умолчанию. Например, при сборке для Moddable One, Moddable Two или M5Stack FIRE формат пикселей устанавливается на `rgb565le` для пикселей с 16-битным цветом; для Moddable Three с дисплеем ePaper установлено значение `gray16` для 4-битных серых пикселей.

Наиболее распространенным драйвером дисплея для 16-битных пикселей является драйвер ILI9341, который реализует стандарт отображения MIPI, используемый контроллерами дисплея в макетных платах Moddable и M5Stack. Аппаратное обеспечение использует 16-битные пиксели, но драйвер также поддерживает другие форматы пикселей. Вы можете поэкспериментировать с различными форматами пикселей, указав формат в командной строке с помощью параметра `-f`. Например, чтобы использовать 4-битные серые пиксели:

```
> mcconfig -d -m -p esp32/moddable_two -f gray16
```

При такой настройке хоста драйвер ILI9341 преобразует 4-битные серые пиксели, визуализируемые POCO, в 16-битные цветные пиксели при передаче их на дисплей. Но изменений больше, чем это:

- Когда вы меняете формат пикселей, сам модуль визуализации POCO перекомпилируется. В этом примере вся поддержка рендеринга в 16-битные пиксели удалена и заменена поддержкой рендеринга в 4-битные серые пиксели. Это один из методов, который использует POCO, чтобы сохранить небольшой размер кода, но при этом поддерживать множество различных форматов пикселей.
- POCO требует, чтобы определенные графические ресурсы хранились в том же формате пикселей, что и дисплей, что обычно требует воссоздания графики в совместимом формате. Но поскольку это утомительно, отнимает много времени и чревато ошибками, `mcconfig` автоматически вызывает другие инструменты в Moddable SDK для преобразования ваших активов в совместимый формат. Это означает, что вы можете переключать форматы пикселей, просто указав другой формат, что делает его таким же простым, как перестройку вашего проекта, чтобы попробовать разные форматы и увидеть компромиссы.

Драйвер ILI9341 также поддерживает 8-битные цветные и 8-битные серые пиксели. Вы можете использовать их с `mcconfig`, указав `rgb332` и `gray256` соответственно в параметре командной строки `-f`.

Если вы обнаружите, что формат пикселей, который лучше всего подходит для вашего продукта, отличается от формата по умолчанию, вы можете указать предпочтительный формат в манифесте своего проекта. Таким образом, вам не нужно помнить о включении его в командную строку при каждой сборке. Для этого определите свойство формата в разделе конфигурации вашего манифеста:

```
"config": {  
  "format": "gray256"  
},
```

Свобода выбора дисплея

Хотя большое разнообразие доступных форматов пикселей может показаться запутанным, оно дает вам возможность выбора при создании продукта. Вы можете выбрать дисплей, максимально соответствующий вашим требованиям по качеству, стоимости и размеру. Pico может отображать пиксели, которые работают с вашим дисплеем, поэтому вам не нужно выбирать дисплей на основе ограничений вашего программного стека. В следующем разделе вы узнаете, как автоматически преобразовывать графические ресурсы в вашем проекте, чтобы они соответствовали используемому вами дисплею.

Графические активы

Пользовательские интерфейсы, созданные с использованием Pico и Piu, состоят из трех разных элементов: прямоугольников, растровых изображений и текста. Это все; нет графических операций для рисования линий, окружностей, круглых прямоугольников, дуг, сплайнов или градиентов. Сначала это может показаться слишком простым, и вы можете решить, что невозможно построить современный пользовательский интерфейс с таким небольшим количеством операций рисования. В следующих главах вы увидите, как объединить эти элементы для создания богатого пользовательского опыта, который хорошо работает на недорогих микроконтроллерах. В этом разделе основное внимание уделяется графическим ресурсам — изображениям и шрифтам, которые вы используете для создания пользовательского интерфейса.

Маски

Наиболее распространенным типом ресурсов, используемых для создания пользовательских интерфейсов с помощью Roco и Piu, является маска. Маска представляет собой изображение в градациях серого; вы можете думать об этом почти как о форме. Поскольку маска содержит серые пиксели, а не только черные и белые пиксели, она может иметь гладкие края. На рис. 8-5 показаны две версии круга, первая как маска в оттенках серого, а вторая как простая 1-битная маска, с увеличенными краями, чтобы показать разницу; обратите внимание на серые края при увеличении маски оттенков серого.

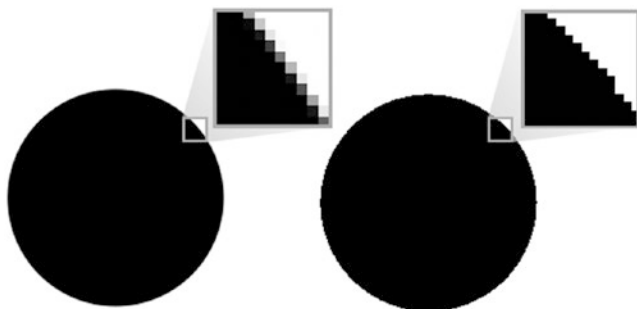


Рис. 8-5. Маска оттенков серого (слева) и 1-битная маска (справа)

Когда Roco визуализирует маску в градациях серого, она не рисуется как изображение. Если бы это было так, белые пиксели скрыли бы фон, как показано на рис. 8-6.

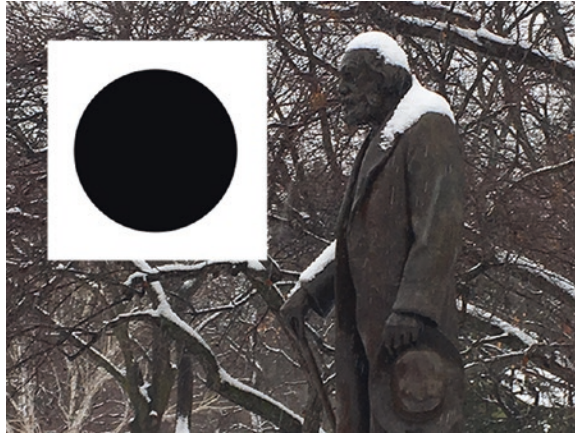


Рис. 8-6. Маска оттенков серого, если она нарисована как изображение

Вместо этого Росо визуализирует маски, обрабатывая черные пиксели как сплошные (полностью непрозрачные), белые пиксели как прозрачные (полностью невидимые), а уровни оттенков серого между ними — как разные уровни смешивания. Результат, соответствующий Рисунку 8-6, показан на Рисунке 8-7, где черный круг наложен на фон (который виден сквозь прозрачные белые пиксели), а серые края круга сливаются с фоном, устраняя любые неровности края.

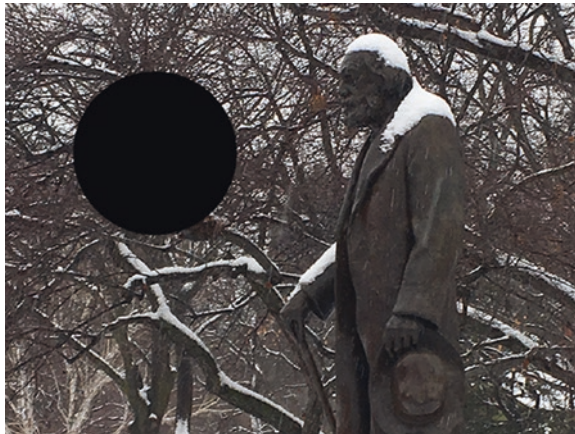


Рис. 8-7. Маска оттенков серого, нарисованная как маска

Вы, вероятно, захотите включить цвет в свой пользовательский интерфейс, и в этом случае серые изображения не кажутся очевидным решением. Однако Росо позволяет рисовать маски в градациях серого любым цветом. Черные пиксели заменяются выбранным вами цветом, а серые пиксели смешивают этот цвет с фоном. На рис. 8.8 показана та же круговая маска, нарисованная синим цветом (в печатных версиях этой книги она выглядит серой).



Рис. 8-8. Маска в оттенках серого, нарисованная как маска в цвете

Возможность рисовать одну маску в градациях серого в разных цветах очень эффективна, поскольку позволяет отображать один графический ресурс в разных цветах. Это уменьшает количество необходимых ресурсов, экономя место для хранения в вашем проекте.

На рис. 8-9 показаны некоторые примеры полутонных масок, используемых в качестве элементов пользовательского интерфейса.



Рис. 8-9. Маски оттенков серого, используемые в качестве элементов пользовательского интерфейса

Как вы знаете из раздела «Форматы пикселей», Росо определяет два разных типа пикселей в градациях серого: 4-битные и 8-битные. Все маски Росо представляют собой 4-битные оттенки серого, что обеспечивает наименьший размер хранилища и быструю визуализацию без особого ущерба для качества.

Добавление масок в ваш проект

Вы добавляете маски в свой проект в виде файлов PNG, таких же файлов изображений, которые используются настольными приложениями, мобильными приложениями и веб-страницами для элементов пользовательского интерфейса. Возможность использовать файлы PNG в своем проекте — это удобно; однако ESP32 и ESP8266 не могут эффективно работать с изображениями PNG из-за требований к памяти и накладных расходов ЦП, необходимых для декодирования изображений PNG. Вместо этого инструменты сборки преобразуют файлы PNG в форматы, которые могут эффективно обрабатываться этими микроконтроллерами. Из-за этого автоматического преобразования вам не нужно разбираться в деталях этих нестандартных форматов изображений (хотя подробности доступны в Moddable SDK).

Чтобы включить изображение маски PNG в свой проект, добавьте его в файл манифеста вашего проекта в разделе ресурсов, как показано в листинге 8-1.

Листинг 8-1.

```
"resources": {
  "*-mask": [
    "./assets/arrow",
    "./assets/thermometer"
  ]
}
```

Имейте в виду, что ресурсы, указанные в манифесте, не включают расширение файла. В примере в листинге 8-1 имена файлов изображений — `arrow.png` и `thermometer.png`.

Сжатие маски

Маски оттенков серого достаточно малы, чтобы их можно было использовать в продуктах, ориентированных на микроконтроллеры. Изображение термометра, показанное ранее на рис. 8-9, имеет размер 2458 байт при сохранении в виде 4-битной маски оттенков серого. Тем не менее, было бы неплохо, если бы он был меньше. У Росо есть решение: он включает алгоритм сжатия специально для 4-битных изображений в градациях серого. Алгоритм оптимизирован для использования на микроконтроллерах и поэтому не требует много процессорного времени или дополнительной памяти.

Для изображения термометра алгоритм сжатия уменьшает размер данных до 813 байт, что на 67% меньше, чем исходная несжатая версия. Коэффициенты сжатия различаются в зависимости от изображения. Коэффициент сжатия маски Росо улучшается для изображений, содержащих большие непрерывные белые и черные области.

Несжатые маски

При рисовании масок для пользовательского интерфейса часто бывает удобно сгруппировать несколько связанных элементов вместе в одном графическом файле. Многие графические дизайнеры предпочитают работать таким образом, так как это позволяет быстрее и проще изменять маски. Поскольку Росо поддерживает усеченный рендеринг, он может использовать только часть маски при рисовании, поэтому у вас есть возможность организовать свои графические файлы таким образом. Маски на рис. 8-10, показывающие несколько различных состояний соединения Wi-Fi, объединены в один графический файл.



Рис. 8-10. Объединение нескольких масок в один графический файл

Вы можете сжать эти комбинированные изображения маски, как описано ранее. Однако использование сжатия с маской, содержащей несколько изображений, снижает производительность. Это связано с тем, что для рендеринга части сжатого изображения декомпрессор должен пропустить части изображения выше и левее целевой области, что требует дополнительного времени. Для некоторых проектов преимущество сжатия в уменьшении размера хранилища важнее, чем снижение производительности. Вы можете оставить маску несжатой, добавив ее в свой манифест в разделе `*-alpha`, а не в разделе `*-mask` (см. листинг 8-2). Конечно, ваш манифест может включать как `*-маску`, так и `*-альфа`, чтобы сжать одни маски, а другие оставить несжатыми.

Листинг 8-2.

```
"resources": {
  "*-alpha": [
    "./assets/wifi-states"
  ]
}
```

Шрифты

Шрифты представляют собой уникальную проблему при разработке встраиваемых систем. В ваш компьютер и мобильный телефон встроены десятки, если не сотни, шрифтов. Один или несколько из этих шрифтов содержат почти все символы, определенные в стандарте Unicode, а это означает, что нет текстовых символов, которые ваши устройства не могут отображать. На микроконтроллере нет встроенных шрифтов; единственные шрифты, доступные для вашего проекта, — это шрифты, которые вы включаете в свой проект.

Для вашего компьютера доступно множество шрифтов, и было бы удобно иметь возможность использовать эти же шрифты в своих продуктах IoT. Большинство, если не все, шрифты на вашем компьютере хранятся в формате, основанном на технологии масштабируемых шрифтов TrueType, созданной Apple (формат шрифта OpenType является производным от TrueType). Рендеринг этих шрифтов на микроконтроллере возможен, но сложен, а объем кода, памяти и ресурсов ЦП, необходимых для

рендеринга, делает его непрактичным для многих проектов. В примерах в этой книге используется более простой формат шрифта, высококачественный растровый шрифт. Средство визуализации, совместимое с TrueType, доступно на ESP32 и представлено в этом разделе.

Преобразование шрифтов TrueType в растровые шрифты

Несмотря на то, что использование шрифтов TrueType во всех ваших проектах, вероятно, нецелесообразно, вы все же можете использовать шрифты на своем компьютере, используя компьютер для преобразования шрифтов TrueType в формат, который может легко обрабатываться микроконтроллером. Шрифт TrueType преобразуется в растровое изображение с определенным размером точек, при этом все символы сохраняются в одном растровом изображении. Растровое изображение использует 4-битные серые пиксели, а не черно-белые, чтобы сохранить сглаживание исходного шрифта. Кроме того, создается файл .fnt, который сопоставляет коды символов Unicode и прямоугольники в растровом изображении шрифта. Этот формат шрифта, который объединяет растровое изображение с файлом карты, называется BMFont, что означает «растровый шрифт». Существует несколько вариантов BMFont; Moddable SDK использует двоичный формат BMFont. На рис. 8-11 показан пример того, как выглядит шрифт Open Sans размером 16 пунктов в формате BMFont.

Рис.8-11. Символьные изображения шрифта в формате BMFont

Обратите внимание, что символы расположены не в том же порядке, как в стандарте Unicode или ASCII. Например, буквы A, B и C не появляются последовательно. Вместо этого символы располагаются по высоте, чтобы сделать растровое изображение как можно меньше за счет минимизации количества неиспользуемого пробела.

Инструменты, которые можно использовать для создания этих растровых файлов, не входят в состав Moddable SDK. Glyph Designer от 71 Squared работает хорошо. Moddable SDK включает в себя набор готовых шрифтов в формате BMFont, так что вы можете начать разработку без каких-либо дополнительных вложений в инструменты.

Формат BMFont имеет два файла для каждого шрифта: файл изображения, обычно в формате PNG, и файл карты шрифтов с расширением .fnt. Эти два файла должны иметь одинаковое имя с разными расширениями, как в OpenSans-Regular-16.png и OpenSans-Regular-16.fnt. Чтобы добавить их в свой проект, включите имя в манифест проекта, как показано в листинге 8-3.

Листинг 8-3.

```
"resources": {
    "*-mask": [
        "./assets/OpenSans-Regular-16"
    ]
}
```

Обратите внимание, что раздел `*-mask` используется для сжатых масок в градациях серого. Шрифты, включенные таким образом, также сжимаются; однако сжимается не все изображение, а каждый символ сжимается отдельно. Это позволяет распаковывать каждый символ напрямую, избегая накладных расходов, которые в противном случае потребовались бы для пропуска пикселей выше и слева от каждого глифа.

Сжатые глифы объединяются с данными из файла `.fnt` в единый ресурс. В результате получаются компактные файлы шрифтов, которые сохраняют отличное качество и могут быть эффективно воспроизведены. Предыдущий пример шрифта Open Sans размером 16 пунктов использует всего 6228 байт памяти как для сжатых символов, так и для информации о метриках шрифта, необходимой для макета и рендеринга. Кроме того, поскольку шрифты хранятся в том же формате сжатия, что и маски оттенков серого, они также могут отображаться в любом цвете.

Формат BMFont не требует, чтобы шрифты были в градациях серого. Этот формат популярен среди геймдизайнеров, потому что он позволяет им включать в свои игры креативные красочные шрифты. Полноцветные шрифты поддерживаются Pico и Piu. Они обычно не используются на микроконтроллерах, поскольку требуют значительно большего объема памяти. Если вы хотите попробовать их, Moddable SDK содержит примеры, которые помогут вам начать работу.

Использование масштабируемых шрифтов

Формат BMFont удобен и эффективен, но он устраняет одно из ключевых преимуществ шрифтов TrueType: возможность масштабировать шрифты до любого размера. Если в вашем проекте используется один и тот же шрифт в трех разных размерах, вам необходимо включить в

ГЛАВУ 8 ОСНОВЫ ГРАФИКИ три разные его версии, по одной для каждого размера пункта. Можно использовать масштабируемые шрифты непосредственно на некоторых более мощных микроконтроллерах, включая ESP32. Высококачественная реализация масштабируемых шрифтов TrueType, оптимизированная для микроконтроллеров, доступна в качестве коммерческого продукта от Monotype Imaging, ведущего поставщика шрифтов и технологии шрифтов. Средство визуализации масштабируемых шрифтов Monotype Spark было интегрировано сModdable SDK, поэтому его можно использовать как с Pico, так и с Pi. Для получения дополнительной информации свяжитесь с Moddable или Monotype.

Авторские права на шрифт

Для коммерческих продуктов вам необходимо убедиться, что у вас есть права на использование любого шрифта, который вы включаете в свои продукты. Так же, как книги и компьютерное программное обеспечение, шрифты могут быть защищены авторским правом их создателями. К счастью, есть много отличных шрифтов, доступных в свободном доступе или по лицензии FOSS (бесплатное программное обеспечение с открытым исходным кодом). Шрифт Open Sans, созданный Google для Android, является одним из таких шрифтов, который хорошо работает в пользовательских интерфейсах продуктов IoT.

Цветные изображения

Хотя маски оттенков серого являются мощным инструментом для создания пользовательских интерфейсов, бывают случаи, когда вам нужны полноцветные изображения. Pico использует несжатые растровые изображения для поддержки цветных изображений. Эти растровые изображения обеспечивают отличное качество и производительность; однако они могут быть довольно большими и поэтому обычно редко используются в интерфейсах для микроконтроллеров.

Вы можете использовать стандартные файлы JPEG и PNG для цветных изображений. Как и в случае с масками оттенков серого, mcconfig преобразует их во время сборки в формат, оптимальный для цели сборки. Чтобы включить цветные изображения в свой проект, добавьте их в раздел *-color в разделе ресурсов вашего манифеста (см. листинг 8-4). Обратите внимание, что расширение файла .jpg или .png опущено.

Листинг 8-4.

```
"resources": {
  "*-color": [
    "./quack"
  ]
}
```

Полноцветные изображения полностью непрозрачны; у них нет смешанных или прозрачных областей. На рис. 8.12 показано изображение quack JPEG из предыдущего фрагмента манифеста, визуализированное в простом пользовательском интерфейсе.

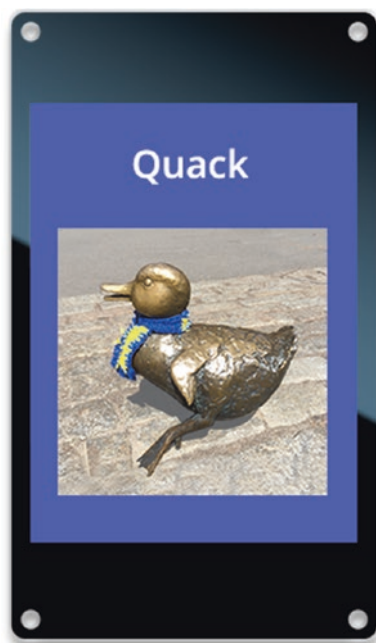


Рис. 8-12. Рендеринг полноцветного изображения

Форма представляет собой прямоугольник, потому что все пиксели изображения отрисовываются. Изображения, хранящиеся в файле PNG, могут содержать дополнительный альфа-канал. Альфа-канал подобен полутонной маске: он указывает, какие пиксели изображения следует отрисовывать, какие следует пропускать, а какие следует смешивать с фоном. Альфа-каналы обычно создаются при редактировании изображения в таких инструментах, как Adobe Photoshop. Poco поддерживает рендеринг альфа-каналов; вы указываете, что хотите сохранить альфа-канал, помещая изображение в раздел *-alpha раздела ресурсов вашего манифеста (см. листинг 8-5).

Листинг 8-5.

```
"resources": {
  "*-alpha": [
    "./quack-with-alpha"
  ]
}
```

Рисунок 8-13 показывает результат. Изображение утки такое же, как на рис. 8-12; однако для маскировки фона был добавлен альфа-канал. Следовательно, при рендеринге изображения рисуется только утка.



Рис 8-13. Отрисовка полноцветного изображения с альфа-каналом

Когда вы включаете в проект изображения с альфа-каналом, инструменты сборки создают два отдельных изображения: полноцветное изображение, как если бы вы поместили изображение в раздел *-color, и несжатый альфа-канал в виде 4-битной маски оттенков серого. Ресурс цвета называется quack-color.bmp, а ресурс сжатой маски — quack-alpha.bm4. На рис. 8-14 показан альфа-канал, используемый для маскирования изображения утки.



Рис. 8-14. Альфа-канал, используемый на рисунке 8-13.

Когда Поко и Пиу визуализируют изображение, они используют как цветное изображение, так и маску. Вы узнаете, как это сделать, в следующих двух главах.

Ротация (вращение) дисплея

Каждый дисплей имеет исходную ориентацию, то есть один край является «верхним». Эта ориентация определяется положением первого пикселя, отрисовываемого аппаратным обеспечением, и направлением, в котором отрисовка продолжается оттуда. Исходная ориентация определяется аппаратным обеспечением и не может быть изменена. Тем не менее, часто желательно рассматривать другой край экрана как верх, чтобы эффективно поворачивать изображение на дисплее. Это характерно для мобильных устройств, которые автоматически поворачивают изображение на дисплее в соответствии с ориентацией, в которой пользователь держит устройство. Эта возможность также присутствует в большинстве ЖК-телевизоров, так что пользователь может установить дисплей так, как это наиболее удобно, а затем вручную отрегулировать ориентацию для отображения изображения «правой стороной вверх».

Хотя многие продукты IoT не позволяют пользователю изменять ориентацию ни путем настройки, ни путем поворота устройства, в этих продуктах все же может потребоваться поворот дисплея, например, когда дизайн продукта требует альбомной ориентации, но исходная ориентация дисплея это портретный режим, или когда дисплей установлен в изделии вверх ногами для экономии места (что может показаться необычным, но такое действительно бывает). Кроме того, иногда разработчик аппаратного обеспечения по ошибке устанавливал дисплей вверх ногами,

и для экономии времени команду разработчиков программного обеспечения просят компенсировать это. По этим и другим причинам для многих продуктов IoT необходима возможность визуализации пользовательского интерфейса с ориентациями 0, 90, 180 и 270 градусов.

Как описано в следующих разделах, существует два разных метода поворота дисплея: программный подход, который работает со всеми дисплеями, и аппаратный подход, который работает с некоторыми дисплеями.

Программа ротации (вращения)

Самый распространенный метод поворота пользовательского интерфейса — отрисовка всего интерфейса в буфере внеэкранной памяти, как если бы дисплей находился в повернутой ориентации. Затем, когда пиксели передаются на дисплей, они преобразуются в соответствии с аппаратной ориентацией. Этот подход неприемлем для недорогих микроконтроллеров, потому что не хватает памяти для хранения полного кадра во внеэкранном буфере.

Росо использует совершенно другой подход: он поворачивает все ресурсы изображения в желаемую ориентацию во время сборки, чтобы их не нужно было поворачивать во время выполнения. Этот поворот выполняется одновременно с любыми необходимыми преобразованиями формата пикселей. Затем, когда приложение на встроенном устройстве выполняет вызовы рисования, Росо нужно только повернуть координаты чертежа в целевую ориентацию. После выполнения этих двух шагов Росо рендерится как обычно, и результат отображается на дисплее повернутым. Этот подход почти не требует измеримых накладных расходов во время выполнения — дополнительная память не используется, и для выполнения преобразования координат запускается лишь незначительное количество дополнительного кода — так что это почти бесплатная функция. Поскольку программное вращение полностью реализовано в средстве визуализации Росо, оно работает со всеми дисплеями.

При использовании программной ротации вы можете изменить ориентацию с помощью параметра командной строки `-r` на `mcconfig`. Поддерживаемые значения поворота: 0, 90, 180 и 270.

```
> mcconfig -d -m -p esp/moddable_one -r 90
```

Как и в конфигурации формата пикселей, вы также можете указать программу вращения в манифесте вашего проекта:

```
"config": {
  "rotation": 90
}
```

Есть одно заметное ограничение на ротацию программы: ротация фиксируется во время сборки и, следовательно, не может быть изменена во время выполнения. Следовательно, этот метод полезен в ситуациях, когда пользовательский интерфейс продукта IoT должен иметь ориентацию, отличную от исходной ориентации дисплея, но не тогда, когда он должен реагировать на действие пользователя, например поворот экрана. Аппаратная ротация, если она доступна, преодолевает это ограничение.

Аппаратная ротация

Аппаратная ротация использует функции контроллера дисплея для поворота изображения, когда дисплей получает пиксели от микроконтроллера. Для использования аппаратной ротации требуется, чтобы и контроллер дисплея, и драйвер дисплея поддерживали эту возможность, которая полностью поддерживается драйвером ILI9341 для MIPI-совместимых контроллеров дисплея.

Аппаратная ротация полностью работает во время выполнения, поэтому в командной строке сборки или манифесте проекта ничего не нужно определять. На самом деле важно, чтобы вы не использовали в своем проекте ротацию как аппаратного, так и программного обеспечения; они не предназначены для совместной работы, поэтому результаты могут быть непредсказуемыми.

При использовании аппаратной ротации вы задаете ротацию во время выполнения, а не настраиваете ее во время сборки. Вы используете глобальную переменную `screen` для связи с драйвером дисплея. Для дисплеев, поддерживающих аппаратную ротацию, глобальный экран имеет свойство поворота; вы можете проверить, поддерживается ли аппаратное вращение, увидев, определено ли это свойство.

```
if (screen.rotation === undefined)
    trace("no hardware rotation\n");
else
    trace("hardware rotation!\n");
```

Чтобы изменить вращение, установите свойство вращения:

```
screen.rotation = 270;
```

Ваш код может читать `screen.rotation` для получения текущего поворота:

```
trace(`Rotation is now ${screen.rotation}\n`);
```

Когда аппаратное вращение изменяется, дисплей не меняется. Все содержимое дисплея должно быть перерисовано до того, как пользователь увидит измененную ориентацию. Если вы обновите только часть экрана после изменения поворота, пользователь увидит часть экрана, нарисованную в исходной ориентации, и другие части с новой ориентацией.

Хост для цели M5Stack включает поддержку автоматического поворота пользовательского интерфейса проектов с использованием `Piu` в соответствии с ориентацией оборудования. Это приводит к тому же поведению, что и мобильный телефон с дисплеем, который подстраивается под то, как пользователь держит устройство. Эта функция возможна благодаря тому, что M5Stack включает в себя встроенный датчик акселерометра, который обеспечивает текущую ориентацию устройства. Для проектов M5Stack, которые не хотят использовать эту функцию, вы можете отключить ее в манифесте проекта.

```
"config": {
    "autorotate": false
}
```

Росо или Piu?

В этой главе вы узнали о механизме рендеринга графики Росо и о структуре пользовательского интерфейса Piu, которые можно использовать для создания пользовательского интерфейса продуктов IoT, работающих на недорогих микроконтроллерах

(включая ESP32 и ESP8266). POCO и Piu имеют схожие графические возможности, потому что Piu использует POCO для рендеринга. Когда вы начинаете создавать свои собственные проекты, вы должны решить, использовать ли API POCO, API Piu или, возможно, оба. В этом разделе объясняются некоторые различия, чтобы помочь вам сделать выбор.

POCO и Piu по своей сути являются разными API:

- POCO — это графический API. Вы делаете вызовы функций, которые в конечном итоге вызывают отрисовку частей экрана.
- Piu — это объектно-ориентированный API для создания пользовательского интерфейса. С помощью Piu вы создаете объекты пользовательского интерфейса, такие как текстовые метки, кнопки и изображения. Добавление этих объектов в приложение вызывает отрисовку частей экрана; вы не выполняете вызовы функций рисования самостоятельно.

Piu позаботится о многих деталях за вас, поэтому вы, скорее всего, будете писать меньше кода; например, он вызывает POCO для рендеринга объектов вашего пользовательского интерфейса, когда это необходимо. Поскольку вы сообщаете Piu обо всех активных объектах пользовательского интерфейса на текущем экране, Piu может свести к минимуму количество рисунков, необходимых при перемещении, изменении, отображении или скрытии элемента. Например, с помощью Piu вы меняете цвет элемента пользовательского интерфейса с помощью маски всего одной строкой кода; Piu определяет, какие пиксели на экране необходимо обновить, и автоматически рисует измененный элемент вместе с любыми объектами, которые его пересекают. В отличие от этого, POCO ничего не знает о пользовательском интерфейсе вашего приложения, поэтому вы должны написать код для обновления экрана и минимизации областей обновления. Код для этого часто начинается с простого, но его становится все труднее поддерживать по мере усложнения пользовательского интерфейса.

Piu использует память для отслеживания активных объектов пользовательского интерфейса и, следовательно, использует больше памяти, чем один только POCO. Конечно, если вы не используете Piu, ваш код должен сам отслеживать активные объекты пользовательского интерфейса, что также требует памяти.

Piu имеет встроенную поддержку реагирования на сенсорные события. На самом деле Piu автоматически поддерживает мультитач. (Дисплеи в Moddable One и Moddable Two поддерживают две точки касания.) Будучи графическим движком, Pico ориентирован на рисование и не поддерживает сенсорный ввод, поэтому ваше приложение должно напрямую взаимодействовать с драйвером сенсорного ввода; хотя это не так уж сложно сделать, вам нужно написать и поддерживать больше кода.

Возможно, самым большим преимуществом использования Piu является то, что в качестве фреймворка он обеспечивает базовую структуру вашего проекта. Следующие предопределенные объекты придадут вашему проекту четко определенную, хорошо продуманную организацию, которая поддерживается очень эффективной реализацией самого Piu:

- Объект Application поддерживает глобальное состояние и существует на протяжении всего времени существования приложения.
- Объекты Texture и Skin организуют ваши графические активы.
- Объекты Style управляют начертанием, размером и стилем шрифта с помощью каскадных таблиц стилей, таких как CSS в Интернете.
- Объекты Container и Content определяют элементы пользовательского интерфейса.
- Объекты Behavior группируют обработчики событий для определенной цели, например, для обеспечения поведения сенсорной кнопки.
- Каждый объект Transitions реализует уникальный переход либо всего экрана, либо его частей.

Когда вы используете Pico, вам приходится самостоятельно разрабатывать и реализовывать структуру приложения. Если пользовательский интерфейс вашего проекта чем-то похож на мобильное приложение, настольное приложение или веб-страницу, возможно, стоит использовать Piu, потому что он разработан и оптимизирован для этого. Если вам нравится писать фреймворки пользовательского интерфейса или если ваш пользовательский опыт совсем другой — например, игра — тогда использование Pico напрямую, вероятно, будет правильным выбором.

Некоторые проекты имеют стандартный стиль пользовательского интерфейса, но должны обеспечивать специализированную визуализацию части экрана. Одним из примеров этого является продукт IoT, показывающий график данных датчиков в реальном времени; кнопки и метки на экране хорошо подходят для Piu, но график будет наиболее эффективно отображаться с помощью Roco. Решение для такого проекта состоит в том, чтобы использовать Piu для экрана и для рисования графика встроить объект Piu Port, который позволяет вам выполнять команды рисования, аналогичные Roco, в макете Piu.

Заключение

В следующей главе обсуждается Roco и ее графическая среда Commodity, а в следующей главе обсуждается Piu. Читая эти две главы, подумайте о потребностях вашего собственного проекта и о том, что лучше подходит: высокоуровневый API пользовательского интерфейса Piu или низкоуровневый API рендеринга графики Roco. Работать с Roco и Piu совершенно по-разному, поэтому стоит поэкспериментировать с ними обоими, чтобы понять, что лучше всего подходит для ваших нужд.

ГЛАВА 9

Рисование графики с помощью POCO

Визуализатор POCO лежит в основе всей графики и кода пользовательского интерфейса в этой книге. Как вы узнали из главы 8, дизайн и реализация POCO оптимизированы для предоставления высококачественной высокопроизводительной графики на недорогих микроконтроллерах, используемых во многих продуктах IoT. В этой главе представлены все основные возможности POCO API с помощью ряда примеров. Название POCO — это термин из классической музыки, означающий «немного», отражающий компактный размер и возможности механизма рендеринга.

POCO является частью Commodity, графической библиотеки, которая предоставляет растровые изображения, создание экземпляров графических активов из ресурсов, внеэкранные графические буферы, драйверы дисплея и многое другое. Некоторые из примеров в этой главе используют эти функции Commodity. Название Commodity, также термин из классической музыки, означает «неторопливый», отражая простоту работы с графической библиотекой. Установка POCO-хоста

Установка POCO-хоста

Вы можете запустить все примеры из этой главы, следуя схеме, описанной в главе 1: установите хост на свое устройство с помощью `mcconfig`, затем установите примеры приложений с помощью `mcrun`.

Все примеры Poco требуют использования экрана, поэтому для вашей командной строки `mcconfig` необходимо указать платформу с драйвером экрана для вашей платы разработки. Примеры предназначены для работы на экранах с разрешением 240 x 320. Следующие командные строки предназначены для Moddable One, Moddable Two и M5Stack FIRE:

```
> mcconfig -d -m -p esp/moddable_one
> mcconfig -d -m -p esp32/moddable_two
> mcconfig -d -m -p esp32/m5stack_fire
```

Если вы подключаете экран к макетной плате с помощью макетной платы и перемычек, следуйте инструкциям в главе 1. Проводка, предоставленная там для ESP32, работает с целью `esp32/moddable_zero`; аналогично для ESP8266 и цели `esp/moddable_zero`.

Если на вашем устройстве нет экрана, вы можете запустить примеры из этой главы на симуляторе рабочего стола, предоставляемом Moddable SDK. Следующие командные строки предназначены для macOS, Windows и Linux:

```
> mcconfig -d -m -p mac
> mcconfig -d -m -p win
> mcconfig -d -m -p lin
```

Перейдите в этот каталог из командной строки и установите его с помощью `mcconfig`.

Если вы используете симулятор рабочего стола, убедитесь, что вы изменили размеры экрана на 240 x 320, прежде чем устанавливать примеры. Для этого выберите 240 x 320 в меню «Размер» на панели инструментов приложения.

Подготовка к рисованию

Чтобы использовать средство визуализации Poco, вам необходимо импортировать класс Poco из модуля `commodetto/Poco`:

```
import Poco from "commodetto/Poco";
```

Росо — универсальный рендерер. Пиксели, которые он визуализирует, могут быть отправлены на экран, в буфер памяти, в файл или в сеть. Росо не знает, как отправлять пиксели ни в одно из этих направлений; вместо этого он выводит пиксели в экземпляр класса `PixelsOut`, и каждый подкласс `PixelsOut` знает, как отправлять пиксели в определенное место назначения. Например, драйвер дисплея является подклассом `PixelsOut`, который знает, как отправлять пиксели на экран. `BufferOut`, еще один подкласс `PixelsOut`, отправляет пиксели в буфер памяти (как вы увидите в разделе «Эффективная визуализация градиентов» этой главы).

Когда вы создаете экземпляр Росо, вы предоставляете экземпляр класса `PixelsOut`, который Росо может вызывать с визуализированными пикселями. Хост для этой главы автоматически создает экземпляр `PixelsOut` для драйвера дисплея вашей макетной платы и сохраняет его в глобальной переменной `screen`. Чтобы работать с экраном, вы просто передаете его конструктору Росо.

```
let poco = new Poco(screen);
```

Формат пикселей и размеры дисплея драйвера дисплея настраиваются в манифесте хоста. Экземпляр экрана имеет свойства ширины и высоты, но они не включают эффекты программного поворота. Вместо этого при работе с Росо используйте свойства ширины и высоты экземпляра Росо, чтобы получить границы дисплея с любыми примененными настройками поворота (аппаратными или программными).

```
trace(`Display width is ${poco.width} pixels.`);
trace(`Display height is ${poco.height} pixels.`);
```

Как отмечалось в главе 8, Росо является визуализатором с фиксированным режимом, что означает, что вместо немедленного выполнения команд рисования он создает список операций рисования для одновременного рендеринга. Для этого списка отображения требуется память. Список отображения по умолчанию составляет 1024 байта. Если ваш рисунок выходит за пределы списка отображения, вам необходимо увеличить его. Если ваш проект не использует все ресурсы списка отображения по умолчанию, вы можете уменьшить его, чтобы освободить память для других целей. В следующем примере список отображения настраивается на 4 КБ:

ГЛАВА 9 DRAWING GRAPHICS WITH POCO

```
let poco = new Poco(screen, {displayListLength: 4096});
```

Вы можете отслеживать, какая часть списка отображения используется вашим проектом, наблюдая за строкой «Использованный список отображения Poco» панели инструментов в *xsbug* (см. рис. 9-1).

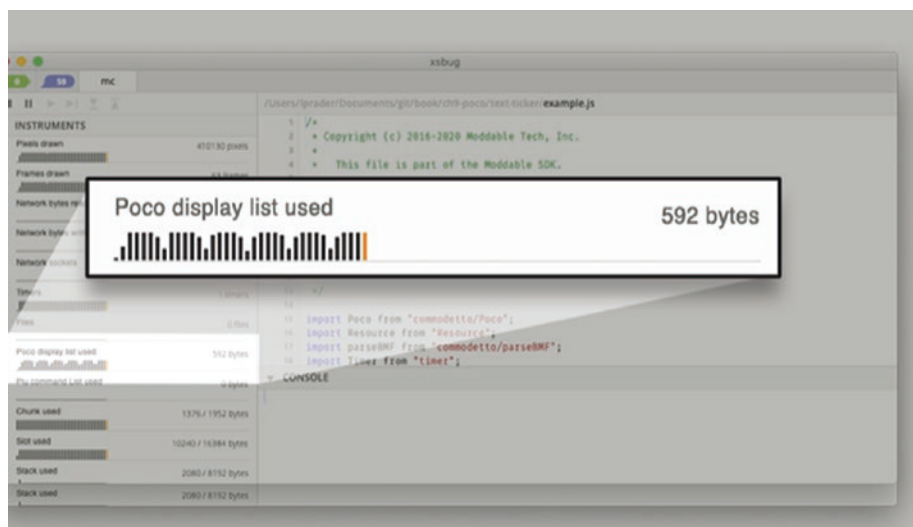


Рис. 9-1. Мониторинг использования списка отображения на панели инструментов *xsbug*

Росо также выделяет память для рендеринга. Буфер рендеринга по умолчанию состоит из двух аппаратных строк развертки. Ширина одной строки аппаратного сканирования доступна из `screen.width`. Если в вашем продукте очень мало памяти, вы можете сократить ее до одной строки сканирования, но не меньше.

```
let poco = new Poco(screen, {pixels: screen.width});
```

Росо может выполнять рендеринг быстрее, когда он рендерит несколько строк развертки одновременно. Следующий код увеличивает буфер рендеринга до восьми полных строк развертки, при этом размер списка отображения устанавливается на 2 КБ.

```
let poco = new Poco(screen,
    {displayListLength: 2048, pixels: screen.width * 8});
```

В качестве оптимизации Росо разделяет память, выделенную для списка отображения и буфера рендеринга. Если список отображения для рендеримого кадра не полностью заполнен, Росо включает эти неиспользуемые байты в буфер рендеринга, что часто позволяет немного ускорить рендеринг.

Росо предоставляет три основные операции рисования: рисование прямоугольников, растровых изображений и текста. Как упоминалось в главе 8, это может показаться не таким уж большим, но вы можете комбинировать эти элементы, чтобы создать богатый пользовательский опыт. В следующих разделах они подробно описаны.

Рисование прямоугольников

Рисование прямоугольников — это простейшая из трех основных операций рисования, предоставляемых Росо. Представляя эту первую операцию рисования, в этом разделе также представлены некоторые основы рисования с помощью Росо.

Заполнение экрана

Пример `$EXAMPLES/ch9-poco/rectangle` просто заполняет весь экран сплошным цветом. Код показан в листинге 9-1.

Листинг 9-1.

```
let poco = new Poco(screen);
let white = poco.makeColor(255, 255, 255);
poco.begin();
    poco.fillRectangle(white, 0, 0, poco.width, poco.height);
poco.end();
```

Первая строка вызывает конструктор Poco для создания экземпляра Poco. Экземпляр доставляет визуализированные пиксели на экран. Этот шаг является общим для всех примеров в этой главе, поэтому он будет опущен в остальных примерах, показанных здесь.

Давайте по очереди рассмотрим каждый из методов, вызываемых в этом примере:

1. Три аргумента poco.makeColor получают компоненты красного, зеленого и синего цветов, каждый из которых находится в диапазоне от 0 (нет) до 255 (полный). Здесь указанный цвет — белый, поэтому красный, зеленый и синий компоненты равны 255. Метод makeColor объединяет эти три значения в одно значение, оптимальное для рендеринга в месте назначения (экран в этом примере). Poco использует различные алгоритмы для создания цветового значения из цветовых компонентов в зависимости от назначения. Следовательно, вам следует передавать значение, возвращаемое makeColor, только тому же экземпляру Poco, который его создал.
2. Вызов poco.begin сообщает Poco, что вы заново начинаете рендеринг нового кадра. Все операции рисования, выполняемые после этого, добавляются в список отображения фрейма.
3. Вызов poco.fillRectangle добавляет в список отображения команду для рисования полноэкранного белого прямоугольника. Первым аргументом является цвет, за которым следуют координаты x и y , а затем ширина и высота. Координатная плоскость помещает $(0, 0)$ в верхний левый угол экрана с увеличением высоты и ширины вниз и вправо.

4. Вызов `poco.end` сообщает Росо, что вы завершили выполнение операций рисования для этого кадра. Затем Росо визуализирует пиксели и отправляет их на экран; это может занять некоторое время, в зависимости от размера дисплея, скорости микроконтроллера и сложности рендеринга кадра. На Moddable One или Moddable Two он заканчивается быстро.

Важно! Росо не заполняет фон цветом автоматически, поскольку это снижает производительность рендеринга. Это означает, что ваш код должен отрисовывать каждый пиксель в кадре. Если вы не укажете цвет для пикселя, Росо выводит неопределенный цвет. Убедитесь, что ваш код заполняет фон цветом, как показано в этом примере, или убедитесь, что комбинация вызовов рисования, которую вы делаете, покрывает каждый пиксель.

Обновление части экрана

Когда вы вызываете метод `begin`, у вас есть возможность указать область экрана для обновления. Возможно, вы помните, что обновление небольших частей экрана — это один из методов достижения более высокой частоты кадров.

В следующем примере квадрат 20 x 20 пикселей заливается красным; остальные пиксели на дисплее не изменились. Если вы добавите этот код к предыдущему примеру прямоугольника, экран будет белым, за исключением маленького красного квадрата в верхнем левом углу.

```
let red = poco.makeColor(255, 0, 0);
poco.begin(0, 0, 20, 20);
  poco.fillRectangle(red, 0, 0, 20, 20);
poco.end();
```

Здесь вызов `begin` определяет область рисования, называемую областью обновления, которая представляет собой только квадрат 20 x 20 в верхнем левом углу экрана. Рисуются только пиксели в области обновления, поэтому белые пиксели за пределами области обновления остаются неизменными. Когда вы вызываете `begin` без аргументов, как в примере с прямоугольником, областью обновления является весь экран. В этом примере вызов `fillRectangle` использует те же координаты и размеры, что и вызов `begin`, заполняя всю область обновления красными пикселями.

Как отмечалось ранее, код между `begin` и `end` должен выполнять вызовы отрисовки, охватывающие каждый пиксель, чтобы получить правильный результат, но что произойдет, если этот код отрисует область, указанную в вызове `begin`? Рассмотрим следующий пример, в котором вызывается `fillRectangle` с параметрами, определяющими полноэкранный режим:

```
let red = poco.makeColor(255, 0, 0);
poco.begin(0, 0, 20, 20);
    poco.fillRectangle(red, 0, 0, poco.width, poco.height);
poco.end();
```

Этот пример дает точно такой же результат, как и предыдущий пример. Вместо того, чтобы отвечать на запрос `fillRectangle` о рисовании на весь экран, Poco ограничивает вывод `fillRectangle` областью обновления, указанной в вызове для начала. Этот подход удобен для многих ситуаций рендеринга, особенно для анимации, поскольку он позволяет вам ограничить область обновления, не изменяя код, чтобы ограничить его отрисовку областью обновления.

Рисование случайных прямоугольников

Классическая демонстрация компьютерной графики заключается в непрерывной визуализации случайно окрашенных прямоугольников случайных размеров и в случайных местах. к текущему рисуемому прямоугольнику. Если вы запустите этот пример, вы увидите анимированную версию экрана, показанную на рис. 9-2.

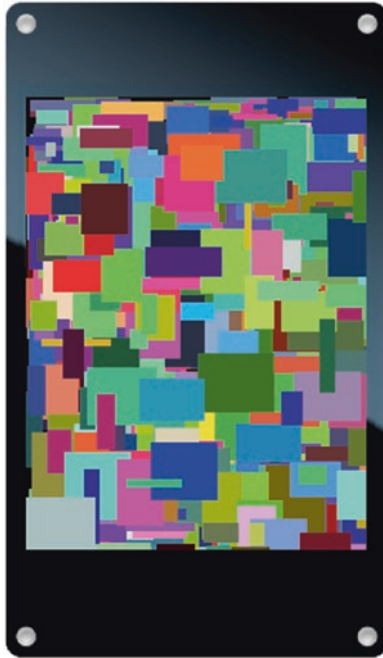


Рис. 9-2. Рендеринг из анимации случайных прямоугольников

Первый шаг — создать экземпляр Poco и очистить экран:

```
let black = poco.makeColor(0, 0, 0);  
poco.begin();  
    poco.fillRectangle(black, 0, 0, poco.width, poco.height);  
poco.end();
```

Затем повторяющийся таймер (листинг 9-2) должен работать со скоростью примерно 60 кадров в секунду. Когда таймер срабатывает, генерируются случайные координаты и размеры прямоугольника вместе со случайным цветом. Метод `begin` ограничивает рисунок областью прямоугольника.

Листинг 9-2.

```

Timer.repeat(function() {
    let x = Math.random() * poco.width;
    let y = Math.random() * poco.height;
    let width = (Math.random() * 50) + 5;
    let height = (Math.random() * 50) + 5;
    let color = poco.makeColor(255 * Math.random(),
                               255 * Math.random(), 255 * Math.random());
    poco.begin(x, y, width, height);
        poco.fillRectangle(color, 0, 0, poco.width,
                           poco.height);
    poco.end();
}, 16);

```

Все случайные значения являются числами с плавающей запятой, потому что вызов `Math.random` возвращает число от 0 до 1. Все функции Poco ожидают целочисленные значения для координат, поэтому `makeColor` и начинают автоматически округлять предоставленные числа с плавающей запятой до ближайшего целого числа. В главе 11 вы узнаете, как добавить свою собственную функцию случайных целых чисел, чтобы повысить производительность за счет устранения этих операций с плавающей запятой.

Рисование смешанных прямоугольников

Все прямоугольники, нарисованные до этой точки, были сплошными: пиксели полностью непрозрачны и полностью закрывают пиксели за ними. Смешанный прямоугольник сочетает в себе один цвет с пикселями за ним, что создает эффект, подобный взгляду через пару затемненных очков. Смешанные прямоугольники используются в пользовательских интерфейсах для создания многослойного эффекта и отрисовки теней.

Чтобы нарисовать смешанный прямоугольник, используйте метод `blendRectangle`. Параметры аналогичны параметрам `fillRectangle` с добавлением уровня наложения в качестве второго параметра. Уровень

наложения — это число от 0 до 255, где 0 означает полностью прозрачный (полностью невидимый), а 255 — полностью непрозрачный. Следующая строка смешивается по всему экрану с красным цветом с уровнем смешивания 128 (50%). Как и все другие операции рисования, это должно происходить между вызовами `begin` и `end`.

```
poco.blendRectangle(red, 128, 0, 0, poco.width, poco.height);
```

Если вы передаете `blendRectangle` уровень смешивания 0, он полностью игнорирует операцию рисования, даже не добавляя запись в список отображения. Если вы передаете уровень наложения 255, `blendRectangle` ведет себя точно так же, как `fillRectangle`.

Чтобы изучить, как выглядят смешанные прямоугольники и их производительность рендеринга, в примере `$EXAMPLES/ch9-poco/blended-rectangle` показана анимация смешанных прямоугольников. На рис. 9-3 показаны изображения смешанного прямоугольника в нескольких положениях на экране.



Рис. 9-3. Визуализация из анимации смешанного прямоугольника

Фон анимации состоит из четырех цветных полос — белой, красной, зеленой и синей. Бары рисуются вспомогательной функцией `drawBars`, показанной в листинге 9-3.

Листинг 9-3.

```
function drawBars(poco) {
    let w = poco.width;
    let h = poco.height / 4;
    poco.fillRect(poco.makeColor(255, 255, 255),
                  0, 0, w, h);
    poco.fillRect(poco.makeColor(255, 0, 0),
                  0, h, w, h);
    poco.fillRect(poco.makeColor(0, 255, 0),
                  0, h * 2, w, h);
    poco.fillRect(poco.makeColor(0, 0, 255),
                  0, h * 3, w, h);
}
```

Когда пример запускается, он покрывает весь экран, рисуя цветные полосы. Обратите внимание, что drawBars не начинается с единственного вызова fillRectangle для заполнения всего экрана сплошным цветом, а рисует четыре отдельные полосы, которые объединяются, чтобы покрыть всю область экрана.

```
poco.begin();
    drawBars(poco);
poco.end();
```

Затем определяются переменные для управления анимацией смешанного черного прямоугольника, который опускается из верхней центральной части экрана в нижнюю (см. листинг 9-4).

Листинг 9-4.

```
let boxSize = 30;
let boxBlend = 64;
let boxStep = 2;
let boxColor = poco.makeColor(0, 0, 0);
let x = (poco.width - boxSize) / 2, y = 0;
```

Размер блока в пикселях определяется `boxSize`. Уровень смеси составляет 64 (25%). В каждом кадре анимации блок шагает на два пикселя, как определено `boxStep`. Переменная `boxColor` определяет прямоугольник, который будет отображаться черным цветом. Наконец, начальные координаты верхнего левого угла прямоугольника задаются в переменных `x` и `y`.

Движение прямоугольника анимировано с помощью повторяющегося таймера, показанного в листинге 9-5. Вызов `begin` задает область рисования, которая включает в себя как текущую, так и предыдущую позиции блока, гарантируя, что предыдущая позиция будет полностью стерта, а новая позиция будет полностью нарисована за одну операцию. Вызов `drawBars` указывает координаты, которые заполняют экран, но они ограничены областью обновления, переданной для начала. В конце функции обратного вызова таймера координата `y` увеличивается на `boxStep`. Как только поле соскальзывает с нижней части экрана, координата `y` сбрасывается до 0, чтобы продолжить анимацию с верхней части экрана.

Листинг 9-5.

```
Timer.repeat(function() {
    poco.begin(x, y - boxStep, boxSize, boxSize + boxStep * 2);
    drawBars(poco);
    poco.blendRectangle(boxColor, boxBlend, x, y, boxSize,
                        boxSize);
    poco.end();

    y += boxStep;
    if (y >= poco.height)
        y = 0;
}, 16);
```

Эта анимация работает со скоростью 60 кадров в секунду как на ESP32, так и на ESP8266. Это связано с тем, что код оптимизирует область рисования, так что микроконтроллер отправляет на дисплей только около 60 000 пикселей в секунду, или менее одного полного кадра. Рендеринг и передача на экран этих пикселей распределены по 60 кадрам.

Это уменьшает количество визуализируемых и передаваемых пикселей на 98,6% по сравнению с полнокадровым рендерингом. Поэкспериментируйте, изменяя переменные, управляющие анимацией, чтобы увидеть эффекты изменения размера блока, уровня наложения и цвета блока.

При запуске примера вы можете заметить небольшой артефакт коробки внизу экрана, когда коробка возвращается наверх. Можно изменить код, чтобы устранить артефакт, но это сделает код более сложным. Это одна из деталей, о которых Пиу автоматически заботится, как вы увидите в главе 10.

Рисование растровых изображений

Рисование растровых изображений — это вторая основная операция рисования, предоставляемая Росо. Он используется как для растровых изображений маски, так и для растровых изображений. Поскольку существует так много различных видов растровых изображений и так много применений растровых изображений при построении пользовательского интерфейса, существует несколько различных функций для рисования растровых изображений. Этот раздел знакомит вас с некоторыми наиболее часто используемыми функциями.

Маски рисования

Как вы узнали из главы 8, маски — это наиболее распространенный тип растрового изображения, используемый при построении пользовательских интерфейсов с микроконтроллерами. Для этого есть много причин: они обеспечивают превосходное качество, поскольку поддерживают сглаживание, могут отображаться в разных цветах, быстро отображаются и могут быть сжаты для минимизации требований к памяти.

Маски хранятся в ресурсах. Вы выбираете изображения масок для использования в своем проекте, включая их в манифест вашего проекта, как показано в листинге 9-6 (и как вы узнали из раздела «Добавление масок в ваш проект» в главе 8).

Листинг 9-6.

```
"resources": {
    "*-mask": [
        "./assets/mask"
    ]
}
```

Чтобы использовать растровое изображение маски, вы должны сначала получить доступ к ресурсу, в котором оно хранится. Ресурс — это просто данные; растровый объект Poco необходим для визуализации маски с использованием Poco API. Commodity предоставляет функции для создания объектов Poco из данных ресурса.

Чтобы создать экземпляр растрового объекта из сжатой маски, используйте функцию `parseRLE` Commodity. («RLE» расшифровывается как «run-length encoding», алгоритм, используемый для сжатия маски.) Следующий код извлекает ресурс и использует `parseRLE` для создания растрового объекта:

```
import parseRLE from "commodity/parseRLE";

let mask = parseRLE(new Resource("mask-alpha.bm4"));
```

В этом небольшом примере есть несколько важных деталей, которые нужно понять:

- Как вы видели в главе 5, конструктор ресурсов ссылается на данные ресурсов во флэш-памяти, а не загружает их в ОЗУ. Функция `parseRLE` также обращается к имеющимся данным, а не копирует данные из флэш-памяти в ОЗУ; однако `parseRLE` выделяет небольшой объем оперативной памяти для растрового объекта Poco, который ссылается на эти данные.
- Обратите внимание, что ресурс загружается по пути `mask-alpha.bm4`, а не `mask.png`. Помните, что инструменты, запускаемые во время сборки, преобразуют файлы PNG в формат, оптимизированный для микроконтроллера, и эти инструменты помещают оптимизированные данные изображения в файл типа `bm4`.

Поскольку изображение используется в качестве альфа-канала, к имени добавляется -альфа. Следовательно, код, работающий на микроконтроллере, должен загружать данные с именем, отличным от исходного. (Piu автоматически использует правильное имя и расширение для вас).

Когда у вас есть растровый объект для маски, вы рисуете маску, вызывая метод `drawGray`:

```
poco.drawGray(mask, red, 10, 20);
```

Первый аргумент — маска, второй — применяемый цвет, а последние два аргумента — координаты *x* и *y*. Обратите внимание, что вы не указываете размеры; Poco всегда отображает растровые изображения в исходном размере без применения масштабирования. Это сделано потому, что высококачественное масштабирование потребует больше процессорного времени и увеличит объем кода рендеринга в Poco.

Объект растрового изображения маски, возвращаемый `parseRLE`, имеет свойства ширины и высоты, которые задают размеры растрового изображения в пикселях. Они могут быть полезны в вашем чертеже, позволяя ему автоматически адаптироваться при изменении размеров графических ресурсов. Например, следующий код рисует синий прямоугольник в области за маской, поэтому любые пиксели, которые не отрисовываются маской, являются синими, а любые пиксели в маске с прозрачностью сливаются с синим фоном. Размер прямоугольника синего фона всегда точно соответствует размеру маски.

```
poco.fillRectangle(blue, 10, 20, mask.width, mask.height);
poco.drawGray(mask, red, 10, 20);
```

Использование несжатой маски

Как вы знаете из главы 8, рисование только подмножества сжатой маски имеет некоторую неэффективность, потому что декомпрессор должен пропускать части изображения выше и левее того, что вы хотите нарисовать. Вместо этого вы можете использовать несжатую маску. Для этого поместите изображение маски в раздел *-alpha (а не в раздел *-mask) ресурсов вашего манифеста, чтобы он хранился в несжатом виде. Затем вместо использования parseRLE для его загрузки используйте parseBMP с расширением ресурса .bmp.

```
import parseBMP from "commodetto/parseBMP";

let mask = parseBMP(new Resource("mask-alpha.bmp"));
```

При переключении между сжатыми и несжатыми масками не забудьте сделать следующее:

- Поместите ресурс в соответствующий раздел: *-alpha для несжатого и *-mask для сжатого.
- Используйте соответствующую функцию загрузки для создания экземпляра растрового изображения: parseBMP для несжатого и parseRLE для сжатого.
- Используйте соответствующее расширение в имени ресурса: .bmp для несжатого и .bm4 для сжатого.

Получив растровое изображение, вы используете drawGray для рендеринга масок независимо от того, сжаты они или несжаты.

Рисование части маски

Изображение на рис. 9-4 (которое вы впервые увидели в главе 8) представляет собой одно несжатое изображение маски, содержащее значки, отображающие несколько различных состояний Wi-Fi.



Рис. 9-4. Полоса значков Wi-Fi

Очевидное использование этого изображения — нарисовать значок, отражающий текущий статус Wi-Fi. Ваше приложение будет отображать только один значок за раз, отражая текущий статус. Как обсуждалось в предыдущем разделе, из соображений эффективности изображение, объединяющее различные состояния, не должно сжиматься.

Чтобы нарисовать только часть растрового изображения, вы указываете исходный прямоугольник, область растрового изображения для использования. В примере \$EXAMPLES/ch9-poco/wifi-icons координаты *x* и *y*, ширина и высота исходного прямоугольника передаются в `drawGray` в качестве необязательных аргументов после координат рисования. Каждая отдельная иконка состояния представляет собой квадрат размером 27 пикселей. Следующий код из примера `wifi-icons` рисует четыре значка состояния, как показано на рис. 9-5:

```
poco.drawGray(mask, black, 10, 20, 0, 0, 27, 27); // верхний левый
poco.drawGray(mask, black, 37, 20, 0, 27, 27, 27); // Нижний левый
poco.drawGray(mask, black, 10, 47, 112, 0, 27, 27); // вверху справа
poco.drawGray(mask, black, 37, 47, 112, 27, 27, 27); // внизу справа
```

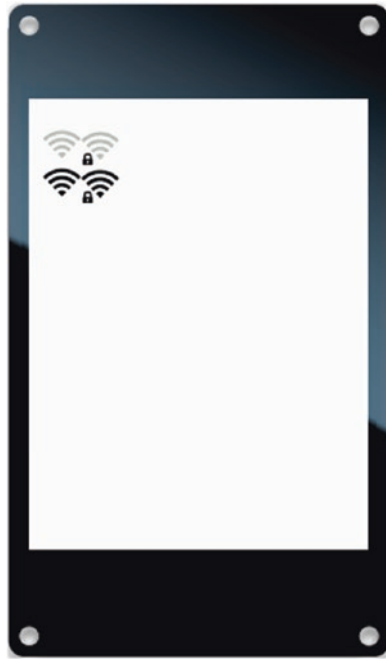


Рис. 9-5. Значки, созданные из полосы значков Wi-Fi

Появление и исчезновение маски

Появление или исчезновение изображения — распространенный переход в пользовательском интерфейсе. Метод `drawGray` имеет возможность смешивать маску с фоновыми пикселями. Это та же идея, что и у смешанных прямоугольников, но использование маски позволяет вам смешать любую форму. В примере `$EXAMPLES/ch9-poco/fade-mask` появляется и исчезает значок громкости, как показано на рис. 9-6.

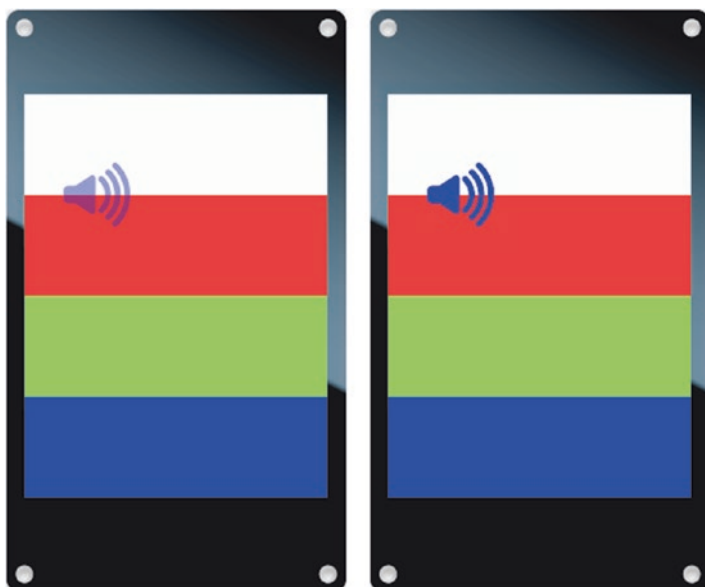


Рис. 9-6. Визуализация из анимации фейд-маски

Уровень наложения указывается в необязательном девятом аргументе `todrawGray`. Как и в `blendRectangle`, уровень смешивания представляет собой число от 0 до 255, где 0 означает полностью прозрачный, а 255 — полностью непрозрачный.

В листинге 9-7 показан код из примера маски исчезновения, который изменяет прозрачность ресурса маски на непрозрачную. Те же функции `drawBars`, что и в примере со смешанным прямоугольником (листинг 9-3), рисуют маску поверх фона. Листинг 9-7.

```
let mask = parseRLE(new Resource("mask-alpha.bm4"));
let maskBlend = 0;
let blendStep = 4;
let maskColor = poco.makeColor(0, 0, 255);
```

```

Timer.repeat(function() {
    let y = (poco.height / 4) - (mask.height / 2);
    poco.begin(30, y, mask.width, mask.height);
        drawBars(poco);
        poco.drawGray(mask, maskColor, 30, y,
            0, 0, mask.width, mask.height, maskBlend);
    poco.end();

    maskBlend += blendStep;
    if (maskBlend > 255)
        maskBlend = 0;
}, 16);

```

Обратите внимание, что для использования уровня наложения необходимо также указать исходный прямоугольник даже при рисовании всей маски. Размеры прямоугольника растрового изображения — в этом примере `mask.width` и `mask.height` — используются для исходного прямоугольника; это гарантирует, что код не нужно менять при изменении размеров актива.

Рисование цветных изображений

Вы добавляете цветные изображения в свой проект, используя файлы JPEG и PNG. Инструменты сборки преобразуют их в несжатые растровые изображения для рендеринга на устройстве, потому что обычно нецелесообразно использовать форматы сжатия JPEG и PNG на микроконтроллере для создания высокопроизводительного пользовательского интерфейса. Растровое изображение хранится в файле BMP (с расширением `.bmp`) и может быть довольно большим, поскольку оно не сжато. Например, квадратное изображение размером 40 пикселей для дисплея, использующего 16-битные пиксели, занимает 3200 байт памяти.

Вы создаете растровое изображение Poco для BMP-изображения с помощью функции `parseBMP`, как вы видели ранее, и рисуете его с помощью метода `drawBitmap`, передавая в качестве аргументов координаты `x` и `y`, где рисовать изображение.

```

let image = parseBMP(new Resource("quack-color.bmp"));
poco.drawBitmap(image, 30, 40);

```

Как и в случае с `drawGray`, вы можете дополнительно нарисовать только часть изображения, указав исходный прямоугольник. В следующем примере рисуется только верхний левый квадрант изображения:

```
poco.drawBitmap(image, 30, 40, 0, 0,
                image.width / 2, image.height / 2);
```

Рисование изображений JPEG

Из-за требований к памяти и ЦП сжатые изображения JPEG не являются хорошим универсальным способом хранения изображений на микроконтроллерах; однако они полезны, когда вам нужно хранить большое количество изображений в относительно небольшом пространстве, например, слайд-шоу или набор изображений для использования в пользовательском интерфейсе. *Commodetto* включает декомпрессор JPEG, который можно использовать вместе с *Росо* для рисования изображений JPEG в ваших проектах. В этом разделе описаны два различных способа сделать это.

Хранение данных JPEG в ресурсах

Как вы знаете, инструменты сборки автоматически конвертируют изображения в ваш манифест в файлы BMP. Если вы хотите сохранить файл JPEG в исходном сжатом формате, поместите изображение JPEG в раздел данных манифеста, а не в раздел ресурсов (см. листинг 9-8). Содержимое раздела данных всегда копируется без преобразования.

Листинг 9-8.

```
"data": {
  "*": [
    "./piano"
  ]
}
```


Подходы к рисованию изображения в формате JPEG, представленные в следующем разделе, несовместимы с программным поворотом изображения. Это связано с тем, что ротация программного обеспечения зависит от ротации образа во время сборки, и здесь манифест сообщает инструментам сборки не преобразовывать образы. Эти методы рисования изображений JPEG работают только тогда, когда вы используете аппаратное вращение или когда программное вращение составляет 0 градусов.

Рисование изображения JPEG из памяти

На компьютерах и телефонах изображения JPEG обычно распаковываются один раз в растровое изображение за пределами экрана; затем, когда требуется изображение JPEG, рисуется это растровое изображение. Такой подход обеспечивает превосходную производительность рендеринга, поскольку сложная операция распаковки изображения JPEG происходит только один раз. Однако для хранения распакованного изображения JPEG требуется много памяти. Следовательно, этот подход обычно подходит для микроконтроллеров только для относительно небольших изображений.

В следующем примере функция `loadJPEG` используется для распаковки ресурса, содержащего данные JPEG, в растровое изображение Poco. Как только изображение находится в растровом формате, вы используете `drawBitmap` для его рендеринга, как описано ранее.

```
import loadJPEG from "commodetto/loadJPEG";

let piano = loadJPEG(new Resource("piano.jpg"));
poco.drawBitmap(piano, 0, 0);
```

Вызов `loadJPEG` занимает некоторое время, потому что распаковка изображений JPEG является относительно сложной операцией для микроконтроллера. Время зависит от размера образа, уровня сжатия и производительности микроконтроллера.

Рисование изображения JPEG во время распаковки

Если у вас недостаточно памяти для хранения в памяти полностью распакованного изображения JPEG, вы все равно можете отобразить изображение, отображая его блоками по мере распаковки. В примере \$EXAMPLES/ch9-poco/draw-jpeg показано, как распаковать полноэкранное (240 x 320) изображение JPEG непосредственно на экран. Когда вы запустите пример, вы увидите экран, показанный на рис. 9-7.

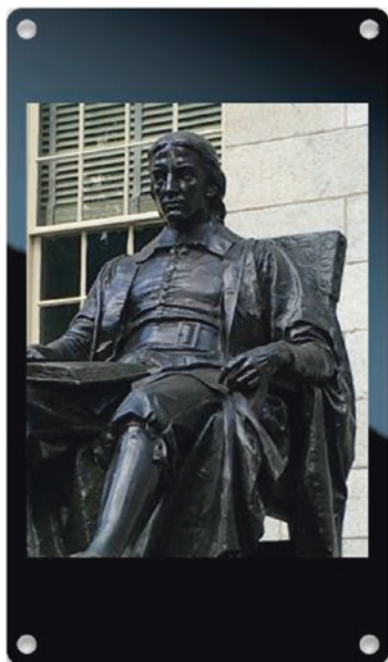


Рис. 9-7. Изображение JPEG из примера draw-jpeg

Сначала вы используете класс JPEG для создания растрового изображения Poco для изображения JPEG:

```
import JPEG from "commodetto/readJPEG";  
let jpeg = new JPEG(new Resource("harvard.jpg"));
```

Декомпрессор JPEG всегда декодирует один блок за раз. Размер блока варьируется в зависимости от того, как сжато изображение JPEG, и составляет от 8 x 8 до 16 x 16 пикселей. Поскольку блоки распакованы, ваш код может рисовать их прямо на экране.

В листинге 9-9 показан код из примера draw-jpeg, который распаковывает изображение JPEG на экран. Метод чтения распаковывает один блок изображения и возвращает его в виде растрового изображения Poco. Объект растрового изображения включает свойства x и y, которые предоставляют координаты блока в изображении JPEG, а также свойства ширины и высоты, которые предоставляют размеры блока. Свойство ready класса JPEG возвращает значение true, пока есть дополнительные блоки для отображения, и значение false после декодирования всех блоков.

Листинг 9-9.

```
while (jpeg.ready) {
    let block = jpeg.read();
    poco.begin(block.x, block.y, block.width, block.height);
    poco.drawBitmap(block, block.x, block.y);
    poco.end();
}
```

Заполнение цветными изображениями

Заполнение области экрана текстурой может создать более интересный пользовательский интерфейс, чем сплошной цвет. В примере \$EXAMPLES/ch9-poco/pattern-fill показано, как разместить изображение земли так, чтобы оно покрывало часть экрана, как показано на рис. 9-8.

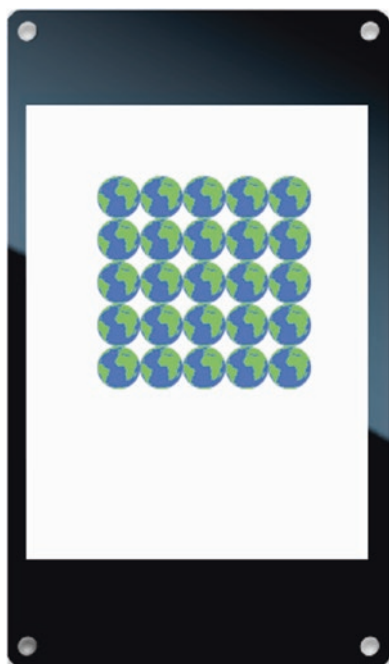


Рис. 9-8. Повторяющаяся текстура земли из примера заполнения узором

Использование большого изображения текстурированного узора занимает больше памяти, чем нужно. Хорошей альтернативой является использование небольшого узора, который можно выложить плиткой. Ваш код может просто рисовать маленькое изображение несколько раз; однако для выполнения всех этих вызовов `drawBitmap` требуется время, и это может привести к переполнению списка отображения Poco. Лучшим вариантом является использование метода `fillPattern` Poco, который замостил прямоугольную область растровым изображением Poco. Например, вот как заполнить весь экран растровым изображением, хранящимся в переменной с именем `tile`:

```
poco.fillPattern(tile, 0, 0, poco.width, poco.height);
```

Аргументы после растрового изображения — это координаты `x` и `y`, ширина и высота прямоугольника, который необходимо заполнить. Метод `fillPattern` также поддерживает необязательный исходный прямоугольник,

который позволяет использовать для плитки только часть растрового изображения. Например (как показано на рис. 9-9), изображение из примера с заливкой узором сочетает в себе 11 различных версий одной и той же текстуры, каждая из которых находится на разных этапах анимации.



Рис. 9-9. Изображение из *pattern-fill*example

В примере с заполнением узором исходный прямоугольник используется для заполнения области экрана анимированным узором. В листинге 9-10 показан код, создающий анимацию. Таймер используется для последовательного перемещения по восьми различным изображениям в объединенном изображении. Переменная фазы отслеживает, какой из восьми шагов анимированного шаблона нужно рисовать.

Листинг 9-10.

```
let tile = parseBMP(new Resource("tiles-color.bmp"));
let size = 30;
let x = 40, y = 50;
let phase = 0;
Timer.repeat(function() {
    poco.begin(x, y, size * 5, size * 5);
    poco.fillPattern(tile, x, y, size * 5, size * 5,
        phase * size, 0, size, size);
    poco.end();

    phase = (phase + 1) % 8;
}, 66);
```

Рисование замаскированных цветных изображений

Рисование цветного изображения через маску (альфа-канал) — распространенный метод в мобильных приложениях и на веб-страницах. Как вы видели в главе 8, он позволяет рисовать полноцветные изображения любой формы, а не только прямоугольники.

Используя метод `drawMasked` Poco, вы можете рисовать несжатое цветное изображение через несжатую маску в градациях серого.

Вызов `drawMasked` принимает множество аргументов, все из которых, кроме одного, являются обязательными. Эти параметры по порядку:

- `image` – Цветное растровое изображение.
- `x, y` – Координаты для рисования.
- `sx, sy, sw, sh` – Исходный прямоугольник для использования из цветного растрового изображения.
- `mask` – растровое изображение маски (несжатая 4-битная шкала серого; сжатые маски не поддерживаются).
- `mask_sx, mask_sy` – координаты левого верхнего угла исходного прямоугольника для использования из растрового изображения маски. (Ширина и высота такие же, как у исходного прямоугольника цветного растрового изображения.)
- `blend` — (необязательно) уровень смешивания от 0 до 255; по умолчанию 255 (полностью непрозрачный).

Чтобы попытаться нарисовать цветное изображение через маску, вам понадобится изображение и маска. В примере `$EXAMPLES/ch9-poco/masked-image` маска круга на рис. 9-10 используется для создания эффекта прожектора с изображением поезда на рис. 9-11.



Рис.9-10. Маска круга из примера маскированного изображения



Рис. 9-11. Обучить изображение из примера замаскированного изображения

Маска и цветное изображение загружаются с помощью `parseBMP`, потому что они оба несжатые:

```
let image = parseBMP(new Resource("train-color.bmp"));
let mask = parseBMP(new Resource("mask_circle.bmp"));
```

Как показано в следующем коде, место рисования устанавливается в координаты (30, 30) в переменных `x` и `y`. Переменная `sx` — это левая сторона исходного прямоугольника; он инициализируется с правой стороны изображения, поэтому рендеринг поезда начинается с передней части поезда. Для переменной `step` установлено значение 2, чтобы продвигать поезд на два пикселя в каждом кадре.

```
let x = 30, y = 30;
let sx = image.width - mask.width;
let step = 2;
```

В листинге 9-11 показан код, выполняющий анимацию. Таймер используется для движения поезда через равные промежутки времени. Расположение рисунка всегда одно и то же, поезд движется сквозь маску. Поезд движется, регулируя `sx`, левый край исходного прямоугольника изображения.

Листинг 9-11.

```
Timer.repeat(function() {
    poco.begin(x, y, mask.width, mask.height);
        poco.fillRectangle(gray, x, y, mask.width, mask.height);
        poco.drawMasked(image, x, y,
            sx, 0, mask.width, mask.height, mask, 0, 0);
    poco.end();

    sx -= step;
    if (sx <= 0)
        sx = image.width - mask.width;
}, 16);
```

На рис. 9-12 показан результат рисования части поезда через маску. Обратите внимание, что края маски сливаются с серым фоном.



Рис. 9-12. Поезд в маске с уровнем смешивания по умолчанию (255)

Необязательный аргумент `blend` для `drawMasked` изменяет относительную непрозрачность каждого пикселя. На рис. 9-13 показано то же изображение поезда, визуализированное с уровнем смешивания 128 (около 50 %). Обратите внимание, что все пиксели, а не только края, сливаются с фоном.



Рис. 9-13. Поезд в маске с уровнем смешивания 128

Рисование текста

Третьей и последней из основных операций рисования, поддерживаемых Росо, является рисование текста. Чтобы нарисовать текст, вам сначала нужен шрифт. Шрифты хранятся в виде растровых изображений и обычно сжимаются.

В ваших приложениях шрифты загружаются из ресурса с помощью функции `parseBMF`. Для сжатых шрифтов используется расширение `.bf4`. В этой главе ресурс шрифта идентифицируется с именем, состоящим из частей, разделенных дефисом, в соответствии с соглашением, которое обычно используется в приложениях, созданных с помощью Piu (как описано далее в главе 10).

```
import parseBMF from "commodetto/parseBMF";

let regular16 = parseBMF(new Resource("OpenSans-Regular-16.bf4"));
let bold28 = parseBMF(new Resource("OpenSans-Semibold-28.bf4"));
```

Росо не накладывает ограничений на количество шрифтов, которые может содержать ваш проект. Конечно, доступное пространство флэш-памяти на вашем целевом микроконтроллере ограничивает количество и размер шрифтов в вашем проекте.

Символы шрифта представляют собой маски оттенков серого, поэтому их можно рисовать любым цветом. Метод `drawText` требует в качестве аргументов текстовую строку, шрифт, цвет и координаты рисунка. Координаты определяют расположение верхнего левого угла первого нарисованного символа. Следующая строка рисует строку `Hello` 16-точечным шрифтом `Open Sans` обычного размера черным шрифтом, начиная с верхнего левого угла экрана:

```
poco.drawText("Hello", regular16, black, 0, 0);
```

Рисование тени текста

Вы можете добиться эффекта тени, нарисовав текст дважды, каждый раз с разными координатами — сначала как тень, а затем как основной текст. Пример `$EXAMPLES/ch9-poco/text-shadow` начинается с рисования текста тенью цветом вниз и справа от того места, где будет располагаться основной текст, а затем накладывается на него той же строкой основного цвета, нарисованной в основных координатах. В результате получится текст, показанный на рис. 9-14.

```
let text = "Drop Shadow";
poco.drawText(text, bold28, lightGray, 0 + 2, 100 + 2);
poco.drawText(text, bold28, blue, 0, 100);
```

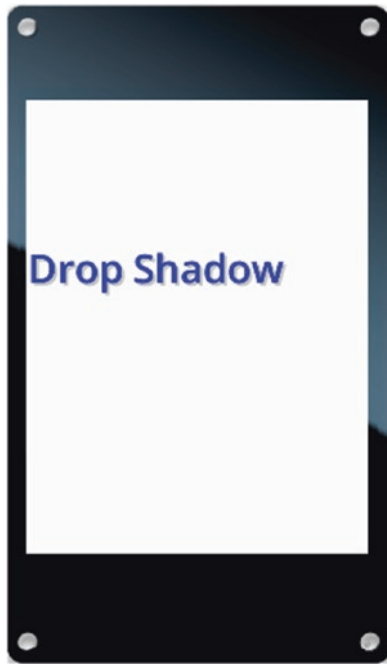


Рис. 9-14. Пример текста, нарисованного с помощью text-shadow.

Измерение текста

Высота нарисованного текста совпадает с высотой шрифта, которая содержится в свойстве `height` объекта шрифта. Ширина нарисованного текста определяется с помощью метода `getTextWidth`. Следующий код заполняет область за текстом зеленым цветом перед отрисовкой текста:

```
let text = "Hello";  
let width = poco.getTextWidth(text, regular16);  
poco.fillRectangle(green, 0, 0, width, regular16.height);  
poco.drawText(text, regular16, black, 0, 0);
```

Примечание Шрифт передается в функцию GETTExTWIdTH, поскольку он содержит размеры для каждого символа. Будьте осторожны, чтобы не измерять одним шрифтом, а рисовать другим; их измерения, вероятно, отличаются, поэтому вы можете получить неожиданные результаты.

Усечение текста

В ситуациях, когда текст, который вы хотите нарисовать, шире доступного для него пространства, обычное решение — нарисовать многоточие (...) в том месте, где текст обрезается. Метод `drawText` делает это автоматически, когда вы сообщаете ему ширину, доступную для рисования.

В следующем примере предложение рисуется в одной строке, обрезая его до ширины экрана. Результат показан на рис. 9-15.

```
let text = "JavaScript is one of the world's most widely used  
            programming languages.";
poco.drawText(text, regular16, black, 0, 0, poco.width);
poco.drawText(text, bold28, black, 0, 40, poco.width);
```



Рис. 9-15. Усеченный текст в двух разных шрифтах

Обтекание текста

В некоторых ситуациях вам может понадобиться нарисовать текст на нескольких строках дисплея. В общем случае поддержки письменных языков со всего мира такой перенос слов является сложной задачей. В примере \$EXAMPLES/ch9-poco/text-wrap представлен базовый подход, достаточный для обычных ситуаций, когда вы работаете с языками, написанными латинскими буквами.

В примере используется метод `split` объектов `String` для создания массива, содержащего слова строки:

```
let text = "JavaScript is one of the world's most widely used
           programming languages.";
text = text.split(" ");
```

Затем он последовательно перебирает все слова, как показано в листинге 9.12. Если в строке достаточно места для размещения текущего слова или если слово шире всей строки, текст рисуется; в противном случае width сбрасывается до полной ширины строки, а y увеличивается на высоту шрифта, чтобы рисование возобновлялось со следующей строки вниз.

Листинг 9-12.

```
let width = poco.width;
let y = 0;
let font = regular16;
let spaceWidth = poco.getTextWidth(" ", font);
while (text.length) {
    let wordWidth = poco.getTextWidth(text[0], font);
    if ((wordWidth < width) || (width === poco.width)) {
        poco.drawText(text[0], font, black, poco.width - width, y);
        text.shift();
    }
    width -= wordWidth + spaceWidth;
    if (width <= 0) {
        width = poco.width;
        y += font.height;
    }
}
```

На рис. 9.16 показан результат выполнения этого примера со шрифтом, установленным на обычный16 и полужирный28 соответственно.

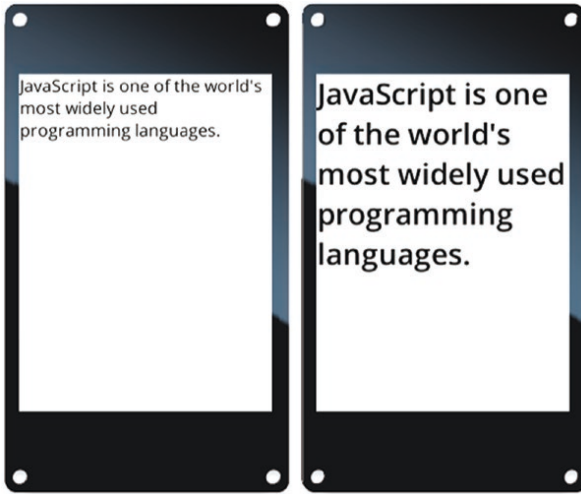


Рис. 9-16. Пример переноса текста с размером шрифта 16 (слева) и 28 (справа)

Дополнительные техники рисования

Росо и Commodetto предоставляют множество инструментов для упрощения и оптимизации рисования под конкретные нужды. В следующих разделах представлены три из них: использование отсечения для ограничения текста рамками, использование исходной точки для простого повторного использования кода рисования и рисование вне экрана для эффективной визуализации градиентов.

Ограничение текста блоком

Как вы знаете, Росо не рисует за пределами области обновления, определенной при вызове метода `begin` Росо; он обрезает эту область, устанавливая начальную область отсечения такой же, как область обновления. Ваш код также может настроить область отсечения во время рисования. Область отсечения всегда ограничена областью обновления, определенной параметром `begin`; вы можете уменьшить область отсечения, но вы всегда можете расширить ее за пределы исходной области рисования.

Одно из мест, где полезно отсечение, — это бегущая строка — текстовое сообщение с прокруткой, которое помещается в часть экрана. Текст никогда не должен выводиться за пределы бегущей строки, а должен быть доведен до его краев. Пример \$EXAMPLES/ch9-poco/text-ticker демонстрирует, как это сделать; на рис. 9.17 показана визуализация примера.

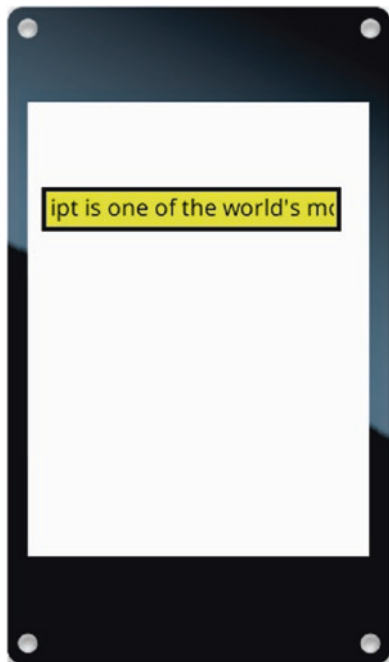


Рис. 9-17. Бегущая лента, нарисованная на примере текстовой бегущей строки

В листинге 9-13 показаны некоторые переменные, используемые в коде отрисовки. Снаружи есть черная рамка, размер которой в пикселях хранится в переменной `frame`. Внутри рамки есть небольшое поле, где текст не может быть прорисован; его размер в пикселях хранится в переменной `margin`. Ширина области, зарезервированной для текста тикера, хранится в `tickerWidth`. Из этих значений рассчитываются общая ширина и высота.

Листинг 9-13.

```

let frame = 3;
let margin = 2;
let x = 10, y = 60;
let tickerWidth = 200;
let width = tickerWidth + frame * 2 + margin * 2;
let height = regular16.height + frame * 2 + margin * 2;

```

Текст измеряется один раз перед началом рисования, чтобы избежать избыточных вычислений во время рендеринга. Результат сохраняется в `textWidth`.

```

let text = "JavaScript is one of the world's most widely used
           programming languages.";
let textWidth = poco.getTextWidth(text, regular16);

```

Переменная `dx` хранит текущее горизонтальное смещение текста от левого края текстовой области бегущей строки. Текст начинается сразу за правым краем и прокручивается оттуда.

```

let dx = tickerWidth;

```

Тикер состоит из двух частей. Сначала рисуется черная рамка и желтый фон бегущей строки:

```

poco.fillRect(black, x, y, width, height);
poco.fillRect(yellow, x + frame, y + frame, tickerWidth + margin
* 2, regular16.height + margin * 2);

```

Далее рисуется текст (листинг 9-14). В примере сначала используется метод `clip` для изменения области отсечения. Он вызывает `clip` с координатами `x` и `y`, шириной и высотой прямоугольника отсечения и помещает текущую область отсечения в стек, а затем пересекает ее с запрошенным клипом. Вызов клипа без аргументов извлекает стек клипов и восстанавливает предыдущий клип. Этот подход упрощает вложение изменений области отсечения.

Листинг 9-14.

```

poco.clip(x + frame + margin, y + frame + margin, tickerWidth,
         regular16.height);
poco.drawText(text, regular16, black, x + frame + margin + dx,
             y + frame);
poco.clip();

```

Наконец, горизонтальное смещение бегущей строки увеличивается, чтобы подготовиться к следующему кадру анимации. Когда текст полностью прокручивается от левого края, он сбрасывается, чтобы снова прокручиваться от правого края.

```

dx -= 2;
if (dx < -textWidth)
    dx = tickerWidth;

```

Простое повторное использование кода рисования

Исходной точкой для рисования (0, 0) является верхний левый угол экрана после вызова метода начала Poco, и исходная точка оставалась там во всех примерах до сих пор. Вы можете использовать метод происхождения для смещения начала координат. Это упрощает написание функции для рисования элемента пользовательского интерфейса в разных местах экрана. В примере \$EXAMPLES/ch9-poco/origin метод origin используется для рисования одинаковых желтых прямоугольников с черными рамками в разных местах, как показано на рис. 9-18.

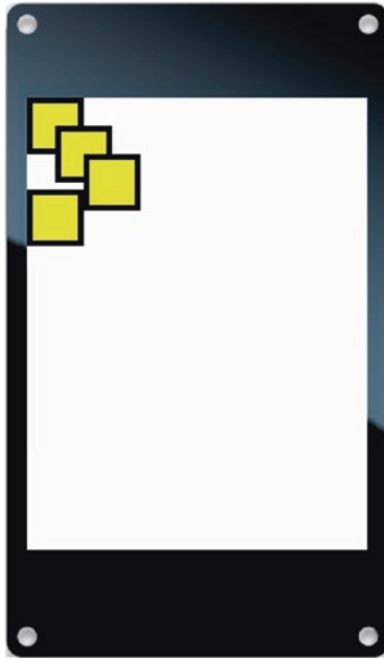


Рис. 9-18. Прямоугольники, нарисованные исходным примером

Следующая функция из исходного примера рисует желтый прямоугольник с черной рамкой:

```
function drawFrame() {  
    poco.fillRectangle(black, 0, 0, 20, 20);  
    poco.fillRectangle(yellow, 2, 2, 16, 16);  
}
```

В этой функции рисование выполняется в начале координат. Перемещение исходной точки перед вызовом `drawFrame` приводит к тому, что рисунок появляется в другом месте на экране. В листинге 9-15 показан код из примера исходной точки, который вызывает метод исходной точки для смещения исходной точки перед каждым вызовом `drawFrame`. В результате вы получите четыре прямоугольника, которые вы видели на рис. 9.19.

Листинг 9-15.

```
drawFrame();
poco.origin(20, 20);
drawFrame();
poco.origin(20, 20);
drawFrame();
poco.origin();
poco.origin();
poco.origin(0, 65);
drawFrame();
poco.origin();
```

Начало начинается с (0, 0). Первый вызов `poco.origin(20, 20)` перемещает источник в (20, 20). Поскольку значения являются относительными, второй вызов `poco.origin(20, 20)` перемещает начало координат в (40, 40).

Метод `origin` сохраняет текущий источник в стеке. Вызов источника без аргументов извлекает стек источника и восстанавливает предыдущий источник. Как и в случае с методом клипа, этот подход упрощает вложенные изменения исходной точки. В этом примере вызов `poco.origin(0, 65)` происходит после того, как все элементы в стеке удалены, поэтому источник снова находится в (0, 0). После вызова источник находится в (0, 65).

Хотя последний вызов источника может показаться ненужным, поскольку дальнейшая прорисовка не выполняется, Poco считает ошибкой, если вы не можете полностью очистить источник или стек клипов перед вызовом метода `end`. Если возникает эта несбалансированная ситуация, метод конца сообщает об ошибке.

Эффективная визуализация градиентов

Ваши проекты не ограничиваются растровыми изображениями, созданными во время сборки; вы также можете создавать растровые изображения во время работы вашего проекта. Вы уже видели один такой пример: функция `loadJPEG` создает растровое изображение в памяти из

сжатых данных JPEG. Поскольку эти растровые изображения должны храниться в оперативной памяти, они ограничены объемом доступной памяти. Вы можете создать растровое изображение во время выполнения с помощью класса `BufferOut`, который также создает виртуальный экран для растрового изображения. Это позволяет вам рисовать на закадровом растровом изображении с помощью `Poco` так же, как вы рисуете на физическом экране.

```
import BufferOut from "commodetto/BufferOut";
```

Пример `$EXAMPLES/ch9-poco/offscreen` создает закадровое растровое изображение, рисует простой градиент к растровому изображению, а затем анимирует растровое изображение на экране. При создании внеэкранного растрового изображения вы указываете его ширину и высоту, а также формат пикселей для нового растрового изображения. Здесь для формата пикселей установлено значение `poco.pixelsOut.pixelFormat`, чтобы закадровое растровое изображение и экран имели одинаковый формат пикселей.

```
let offscreen = new BufferOut({width: 64, height: 64,
pixelFormat: poco.pixelsOut.pixelFormat});
```

Это закадровое растровое изображение представляет собой квадрат размером 64 x 64 пикселя. Чтобы отрисовать его, вы создаете еще один экземпляр `Poco`, привязанный к экрану, а не к экрану, как первый экземпляр.

```
let pocoOff = new Poco(offscreen);
```

Затем в примере используется `pocoOff` для рисования растрового изображения точно так же, как если бы оно рисовалось на экране. В листинге 9-16 показан код, используемый для рисования градиента серого, показанного на рис. 9-19.

Листинг 9-16.

```
pocoOff.begin();
  for (let i = 64; i >= 1; i--) {
    let gray = (i * 4) - 1;
    let color = pocoOff.makeColor(gray, gray, gray);
    pocoOff.fillRect(color, 0, 0, i, i);
  }pocoOff.end();
```

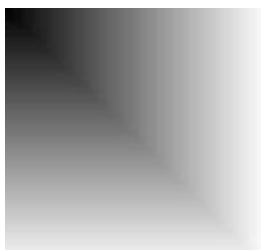


Рис. 9-19. Серый градиент, нарисованный закадровым примером

Растровое изображение, прикрепленное к `offscreen`, доступно из его свойства `bitmap`. Следующая строка рисует закадровое растровое изображение на экране:

```
poco.drawBitmap(offscreen.bitmap, 0, 0);
```

Для рендеринга содержимого этого закадрового растрового изображения требуется нарисовать 64 различных прямоугольника, каждый из которых немного отличается по размеру и цвету. Рисование этих прямоугольников снова и снова в анимации было бы слишком сложным вычислением для микроконтроллера. К счастью, рисовать закадровое растровое изображение намного проще.

Закадровый пример продолжает анимировать 19 копий закадрового растрового изображения, перемещая их влево и вправо с разной скоростью. В листинге 9-17 показан код анимации, а на рис. 9-20 показана визуализация анимации.

Листинг 9-17.

```
let step = 1;
let direction = +1;
Timer.repeat(function() {
  poco.begin(0, 0, 240, 240);
  poco.fillRectangle(white, 0, 0, poco.width, poco.height);
  for (let i = 0; i < 19; i += 1)
    poco.drawBitmap(offscreen.bitmap, i * step, i * 10);
});
```

```

    step += direction;
    if (step > 40) {
        step = 40;
        direction = -1;
    }
    else if (step < 1) {
        step = 0;
        direction = +1;
    }
    poco.end();
}, 33);

```

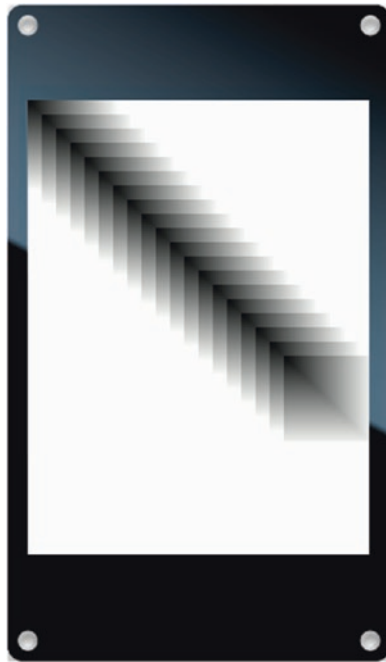


Рис. 9-20. Рендеринг закадровой анимации

Сенсорный ввод

Если вы используете Poco для рисования пользовательского интерфейса вашего продукта и хотите использовать возможности сенсорного ввода, вам необходимо реализовать поддержку сенсорного ввода путем считывания непосредственно из драйвера сенсорного ввода. Когда вы используете `Piц`, сенсорный ввод позаботится о вас автоматически. К счастью, чтение сенсорного ввода не представляет собой особой сложности.

Доступ к сенсорному драйверу

Наиболее распространенным емкостным сенсорным вводом является FocalTech FT6206. Эта часть используется в платах Moddable One и Moddable Two. Вы импортируете сенсорный драйвер в свой проект и создаете экземпляр следующим образом:

```
import FT6206 from "ft6206";
```

```
let touch = new FT6206;
```

В старых резистивных сенсорных экранах обычно используется сенсорный контроллер XPT2046.

```
import XPT2046 from "xpt2046";
```

```
let touch = new XPT2046;
```

Оба сенсорных драйвера реализуют один и тот же API, поэтому после создания экземпляра драйвера ваш код для чтения будет одинаковым для обоих.

Чтение сенсорного ввода

Чтобы получить точки касания от сенсорного драйвера, вы вызываете метод чтения. Вы передаете массив точек касания вызову чтения, и драйвер обновляет точки. Обычно вы выделяете точки касания один раз, после создания экземпляра драйвера касания, чтобы свести к минимуму работу, выполняемую диспетчером памяти и сборщиком мусора. В следующей строке выделяется массив с одной точкой касания. Массив присваивается свойству `points` экземпляра `touch inputdriver`.


```
touch.points = [{}];
```

Чтобы получить текущие точки касания, вызовите `read` с массивом точек:

```
touch.read(touch.points);
```

Драйвер устанавливает свойство состояния для каждой точки касания. Значения свойства состояния следующие:

- **0** – без касаний
- **1** – начать сенсорный ввод (палец вниз)
- **2** – продолжить сенсорный ввод (палец все еще внизу)
- **3** – коснитесь конца ввода (палец поднят)

Для всех значений состояния, кроме 0, свойства `x` и `y` точки касания указывают текущее местоположение касания. Код в листинге 9-18, взятый из `$EXAMPLES/ch9-poco/touch`, отбирает драйвер сенсорного экрана 30 раз в секунду, выводя текущее состояние на консоль отладки.

Листинг 9-18.

```
Timer.repeat(function() {
    let points = touch.points;
    let point = points[0];
    touch.read(points);
    switch (point.state) {
        case 0:
            trace("no touch\n");
            break;
        case 1:
            trace(`touch begin @ ${point.x}, ${point.y}\n`);
            break;
```

```

        case 2:
            trace(`touch continue @ ${point.x}, ${point.y}\n`);
            break;
        case 3:
            trace(`touch end @ ${point.x}, ${point.y}\n`);
            break;
    }
}, 33));

```

Некоторые версии FT6206 не могут надежно генерировать состояние `touchend`. Когда вы запустите пример, вы увидите поведение вашего компонента. Если состояние окончания касания не сгенерировано, можно определить, что последовательность касаний завершилась, когда точка касания переходит в состояние 0 (отсутствие касания).

Использование мультитач

Причина, по которой метод чтения принимает массив точек, а не одну точку, заключается в том, что он может поддерживать мультитач. Емкостные сенсорные датчики FT6206 поддерживают две точки одновременного касания, если они не расположены слишком близко друг к другу. Чтобы использовать мультитач, нужно просто передать массив с двумя точками.

```
touch.points = [{}, {}]; touch.read(touch.points);
```

Применение вращения

Драйвер сенсорного экрана всегда предоставляет точки, к которым не применяется ни аппаратное, ни программное вращение. Если вы используете вращение, вам нужно применить его к точкам касания. Как и следовало ожидать, `Più` позаботится о повороте точек касания за вас.

Вы можете использовать код из листинга 9-19 для преобразования координат для поворотов на 90, 180 и 270 градусов.

Листинг 9-19.

```
if (90 === rotation) {
    const x = point.x;
    point.x = point.y;
    point.y = screen.height - x;
}
else if (180 === rotation) {
    point.x = screen.width - point.x;
    point.y = screen.height - point.y;
}
else if (270 === rotation) {
    const x = point.x;
    point.x = screen.width - point.y;
    point.y = x;
}
```

Заключение

Средство визуализации Росо предоставляет все основные инструменты, необходимые для создания пользовательского интерфейса продукта IoT. Вы можете рисовать прямоугольники, растровые изображения и текст с множеством различных параметров. Возможности рендеринга включают сглаженный текст, маски оттенков серого, нарисованные любым цветом, и рендеринг цветных изображений с помощью масок альфа-канала. Вы можете оптимизировать производительность рендеринга, используя отсечение, чтобы ограничить область экрана, которую вы обновляете.

Росо дает вам большой контроль, но эта сила приносит с собой некоторые неудобства. Вы должны загрузить ресурсы и вызвать соответствующие функции для их разбора, вы должны рассчитать область экрана для обновления и вы должны позаботиться о некоторых деталях поворота. В следующей главе представлена структура пользовательского интерфейса `Piu`, которая позаботится обо многих из этих деталей за вас.

ГЛАВА 10

Создание пользовательских интерфейсов с Piu

Piu — это объектно-ориентированная структура пользовательского интерфейса, упрощающая процесс создания сложных пользовательских интерфейсов. Он использует средство визуализации POCO для рисования. В этой главе представлен обзор того, как работает Piu, и представлены некоторые из его основных возможностей на ряде примеров. Имя Piu означает «больше» в классической музыке и отражает удивительно богатый набор возможностей, которые Piu использует в POCO.

Имейте в виду, что изучение новой структуры пользовательского интерфейса может быть сложной задачей. Каждый фреймворк по-своему решает проблему создания пользовательского интерфейса и имеет собственный набор API-интерфейсов для решения этой проблемы. Чтобы полностью понять тонкости Piu, недостаточно просто следовать примерам из этой главы. Цель этой главы — научить вас наиболее важным и часто используемым функциям Piu, показать простые примеры того, где они используются, и объяснить их достаточно, чтобы вы могли использовать их в своих собственных пользовательских интерфейсах для ваших собственных продуктов.

Некоторые части Piu покажутся вам знакомыми, если вы уже знакомы с каскадными таблицами стилей, или CSS, языком для определения стилей — например, текста — который чаще всего используется при разработке веб-страниц, написанных в HTML. Сходство между Piu и CSS не случайно; Piu включает в себя множество соглашений CSS, чтобы обеспечить согласованность для разработчиков, работающих как над веб-продуктами, так и над продуктами IoT.

Ключевые идеи

Прежде чем погрузиться в код, важно понять несколько ключевых концепций, лежащих в основе Piu. Если вы новичок в работе с объектно-ориентированными средами пользовательского интерфейса, информация в этом разделе особенно важна, поскольку она поможет вам настроиться на работу с Piu. Если вы уже умеете работать с объектно-ориентированными платформами, этот раздел по-прежнему важен, поскольку в нем представлена информация, относящаяся к Piu.

Все является объектом

Самая важная концепция, которую нужно понять, заключается в том, что каждый элемент пользовательского интерфейса в приложении Piu имеет соответствующий объект JavaScript. Объекты JavaScript являются экземплярами классов, которые предоставляет Piu. Piu отличается от других функций Moddable SDK, представленных в этой книге, тем, что вам не нужно импортировать большинство классов Piu. Вместо этого Piu хранит конструкторы часто используемых классов в глобальных переменных, что упрощает их использование из любого модуля вашего приложения.

Каждое приложение Piu начинается с одного и того же объекта: экземпляра класса приложения Piu. Хост для этой главы создает экземпляр, поэтому ни в одном из примеров в этой главе его создавать не нужно. В листинге 10-1 показано, как хост создает экземпляр приложения, вызывая конструктор приложения.

Листинг 10-1.

```
new Application(null, {
    displayListLength: 8192,
    commandListLength: 4096,
    skin: new Skin({fill: "white"}),
    Behavior: AppBehavior
});
```

Пока не беспокойтесь о деталях различных свойств. Обратите внимание, что здесь используется свойство `displayLength` из `Poco`, так как `Piu` использует `Poco` для рисования.

Как часть конструктора приложения, `Piu` сохраняет экземпляр в глобальной переменной приложения. В примерах доступ к экземпляру приложения осуществляется через глобальное приложение.

Объект приложения является корнем приложения `Piu`. Думайте об этом как о контейнере, который содержит все графические элементы, появляющиеся на экране. Графические элементы, добавленные в контейнер, называются объектами содержимого. Чтобы отобразить объект содержимого на экране, вы создаете его экземпляр и добавляете его в объект приложения. Например, чтобы отобразить строку текста, вы создаете экземпляр класса `Label` `Piu`, своего рода объект содержимого, и добавляете его в объект приложения.

Примечание В этой главе к классам обращаются по их именам, написанным с заглавной буквы, например, «класс `LABEL`», а к экземплярам классов — по имени класса, написанному без заглавных букв, например, к «объекту метки» (или просто «метке»).

Вы можете создавать объекты содержимого `Piu`, не добавляя их к объекту приложения, но они не будут отображаться, пока не будут добавлены. Когда вы используете `Piu`, вы не вызываете функции рисования самостоятельно. Объекты контента умеют рисовать сами себя; они вызывают для вас функции рисования по мере необходимости для обновления экрана.

Конечно, вы также можете удалить объекты контента с экрана. Как вы уже догадались, вы делаете это, удаляя их из объекта приложения.

Каждый элемент пользовательского интерфейса является объектом контента

Как вы теперь знаете, каждый элемент пользовательского интерфейса в приложении Piu связан с объектом содержимого. В частности, каждый элемент пользовательского интерфейса связан с экземпляром класса, который наследуется от класса Content. Существует много таких классов, включая класс Label, упомянутый ранее. В этой главе вы узнаете о различных типах объектов контента.

Примечание В этой главе «content object - объект содержимого» относится к экземпляру класса Content, тогда как общий термин объекта содержимого относится к экземпляру любого класса, наследуемого от класса Content.

Когда вы создаете объект контента, вы указываете его свойства в словаре JavaScript. В случае объекта метки свойства включают строку метки и стиль текста. Словарь передается конструктору класса.

```
let sampleLabel = new Label(null, {  
  style: textStyle,  
  string: "Hello"  
});
```

Свойства объекта содержимого можно изменить в любое время. Вы изменяете свойство, устанавливая его значение в экземпляре, обычно используя то же имя свойства, которое вы использовали для инициализации свойства при вызове конструктора.

```
sampleLabel.style = OpenSansBold12;  
sampleLabel.string = "Goodbye";
```


Когда вы изменяете свойства объекта контента, который вы добавили в объект приложения, экран обновляется автоматически. `Piu` вызывает обновление, делая недействительными соответствующие части дисплея, а объект содержимого вызывает необходимые функции рисования для обновления экрана.

Не все объекты `Piu` являются объектами содержимого

В дополнение к объектам контента в `Piu` есть несколько других типов объектов, наиболее распространенные из которых представлены в этом разделе. Все они используются для изменения объектов контента — их внешнего вида, поведения или анимации. Ни один из этих объектов не наследуется от класса `Content`. Классы, которые их определяют, представлены здесь и описаны более подробно позже в этой главе.

Определение внешнего вида

Классы `Skin`, `Texture` и `Style` изменяют внешний вид объектов содержимого: объекты кожи и текстуры используются объектами содержимого для заполнения области цветом и изображениями, а объекты стиля определяют внешний вид текста, включая его шрифт и цвет. В предыдущем разделе, например, экземпляр `sampleLabel` был создан со словарем, содержащим свойство стиля, для которого задан объект стиля с именем `textStyle`. Объект стиля не связан ни с одним объектом содержимого; скорее, он может быть применен к одному или нескольким объектам этикетки и другим объектам контента.

Аналогично, объекты оформления связаны с объектами содержимого через свойство кожи объектов содержимого, и, подобно объектам стиля, они могут совместно использоваться многими объектами содержимого. С другой стороны, класс `Texture` не используется непосредственно объектами содержимого; Объекты текстуры связаны с объектами кожи через свойство текстуры объектов кожи, и они могут совместно использоваться многими объектами кожи.

Как и в случае с объектами содержимого, вы указываете свойства объектов скина, текстуры и стиля с помощью словаря, передаваемого их конструктору. В отличие от объектов содержимого, свойства объектов обложки, текстуры и стиля изменить нельзя. Это означает, например, что для изменения шрифта, используемого меткой, вы изменяете свойство стиля объекта метки, а не свойство шрифта объекта стиля.

Управление поведением

Поведения выполняют действия в ответ на события, такие как нажатия на экран, изменение значений датчиков или истечение таймера. Поведение объектов контента определяется подклассами класса Behavior. Поведения являются частью того, как Piu реализует стиль программирования, управляемый событиями. Если вы новичок в программировании, управляемом событиями, не волнуйтесь; в этой главе подробно объясняется, как объекты поведения и события работают в Piu.

Объекту контента должно быть назначено поведение, чтобы он мог реагировать на события. Объект контента не обязательно должен иметь назначенное поведение, но без него он не будет реагировать на события. Обычно объект контента имеет собственный экземпляр подкласса Behavior, хотя несколько объектов контента могут совместно использовать один экземпляр поведения.

Анимация

Для анимации объектов содержимого используются классы Timeline и Transition. Вы можете анимировать объекты контента, изменяя их свойства, заставляя их двигаться, менять цвет, проявляться или исчезать и т. д., а также вы можете заменять один объект контента другим, например, для перемещения между экранами.

Объекты содержимого не имеют временной шкалы или свойств перехода; вместо этого временная шкала и объекты перехода ссылаются на объекты содержимого, которые они анимируют.

Установка хоста Piu

Вы можете запустить все примеры из этой главы, следуя схеме, описанной в главе 1: установите хост на свое устройство с помощью `mcconfig`, затем установите примеры приложений с помощью `mcrun`.

Все примеры Piu требуют использования экрана, поэтому для вашей командной строки `mcconfig` необходимо указать платформу с драйвером экрана для вашей платы разработки. Примеры предназначены для работы на экранах с разрешением 320 x 240. Следующие командные строки предназначены для Moddable One, Moddable Two и M5Stack FIRE:

```
> mcconfig -d -m -p esp/moddable_one
> mcconfig -d -m -p esp32/moddable_two
> mcconfig -d -m -p esp32/m5stack_fire
```

Если вы подключаете экран к макетной плате с помощью макетной платы и перемычек, следуйте инструкциям в главе 1. Проводка, предоставленная там для ESP32, работает с целью `esp32/moddable_zero`; аналогично для ESP8266 и цели `esp/moddable_zero`.

Драйвер экрана для Moddable Zero, Moddable One, Moddable Two и M5Stack FIRE поддерживает ротацию оборудования. Хост настраивает поворот экрана таким образом, чтобы отображались пиксели в альбомной (320 x 240) ориентации, а не в книжной (240 x 320) ориентации по умолчанию.

Если на вашем устройстве нет экрана, вы можете запустить примеры из этой главы на симуляторе рабочего стола, предоставляемом Moddable SDK. Следующие командные строки предназначены для macOS, Windows и Linux:

```
> mcconfig -d -m -p mac
> mcconfig -d -m -p win
> mcconfig -d -m -p lin
```

Хост для этой главы находится в каталоге `$EXAMPLES/ch10-piu/host`. Перейдите в этот каталог из командной строки и установите его с помощью `mcconfig`.

Если вы используете симулятор рабочего стола, убедитесь, что вы изменили размеры экрана на 320 x 240, прежде чем устанавливать примеры. Для этого выберите 320 x 240 в меню «Размер» на панели инструментов приложения.

“Hello, World” с Pui

Когда вы запускаете пример `$EXAMPLES/ch10-pui/helloworld`, вы видите экран, показанный на рис. 10-1.

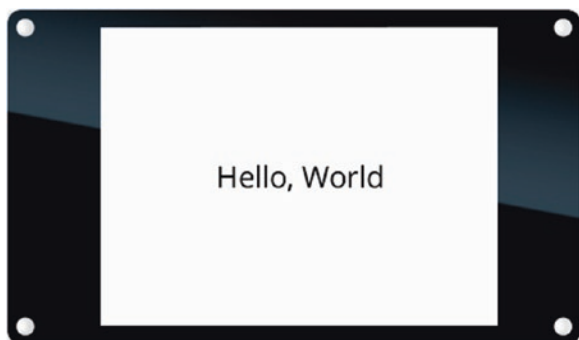


Рис.10-1. Пример helloworld

Это не самый захватывающий пользовательский интерфейс, но это хорошая отправная точка для демонстрации основ создания объектов Pui для создания простого экрана. Отображаемый текст определяется в объекте метки, а первая строка кода в примере создает стиль. — экземпляр класса `Style` — для определения внешнего вида текста.

```
const textStyle = new Style({  
    font: "24px Open Sans"  
});
```

Подробная информация о свойстве шрифта описана в следующем разделе. Пока просто обратите внимание, что свойства объекта стиля определены в словаре, переданном конструктору класса `Style`. Как вы узнали ранее, это соглашение применяется как к объектам содержимого Pui, так и к объектам, определяющим их внешний вид.

Каждый объект Piu требует указания определенных свойств, в то время как другие свойства являются необязательными. Например, конструктору Style требуется свойство шрифта, но свойства, связанные с цветом, выравниванием по горизонтали и вертикали и высотой строки, являются необязательными.

Затем в примере helloworld создается объект содержимого: объект метки с именем sampleLabel (см. листинг 10-2). Объект метки отображает текст в одной строке с одним стилем.

Листинг 10-2.

```
const sampleLabel = new Label(null, {  
  style: textStyle,  
  string: "Hello, World",  
  top: 0, bottom: 0, left: 0, right: 0  
});
```

Свойство string определяет текст, отображаемый на метке, а свойство style определяет стиль текста (textStyle, созданный ранее). Свойства top, bottom, left и right определяют положение метки, указывая поля между объектом метки и его контейнером, объектом приложения; установка всех этих значений на 0 заставляет объект метки заполнять весь экран. По умолчанию текст центрируется по горизонтали и вертикали, поэтому текст отображается в центре экрана.

Примечание Имейте в виду, что верх, низ, лево и право не являются абсолютными координатами, а указывают поля от соответствующего края родительского контейнера.

Как вы узнали ранее, простое создание объекта контента не приводит к его появлению на экране. Вы должны добавить объекты содержимого к объекту приложения, чтобы Piu мог их отрисовать. Это делается путем вызова `application.add`.

```
application.add(sampleLabel);
```

В этом примере `textStyle` присоединен только к одному объекту содержимого, но помните, что один объект стиля может быть присоединен более чем к одному объекту содержимого. Вы можете добавить вторую метку, которая использует тот же стиль, но с другим текстом, отображаемым в другом месте. Например, добавление к примеру кода из листинга 10-3 приведет к отображению текстовой строки `Second` в правом нижнем углу экрана.

Листинг 10-3.

```
application.add(new Label(null, {
    style: textStyle,
    string: "Second string",
    bottom: 0, right: 0
}));
```

Обратите внимание, что свойства `top` и `left` для этой метки не указаны. Если вы укажете нижнее свойство метки, но не верхнее, или наоборот, высота метки будет равна высоте текста в стиле, заданном ее свойством стиля. Точно так же, если вы укажете только одно из левого и правого, ширина метки равна ширине текста в его стиле.

Шрифты

Свойство `font` объекта стиля указывает шрифт, используемый для рисования текста, когда стиль применяется к объектам содержимого. Шрифты обычно представляют собой сжатые растровые изображения, хранящиеся в ресурсе, как описано в главе 8. В Piu нет встроенных шрифтов; вместо этого хост и приложение могут включать шрифты в свои манифесты. Хост для этой главы предоставляет два шрифта, которые используются во всех примерах, рисующих текст.

В листинге 10-4 показан фрагмент манифеста, включающий шрифты.

Листинг 10-4.

```
"resources": {
  "*-alpha": [
    "./OpenSans-Regular-24",
    "./OpenSans-Semibold-16"
  ]
},
```

Имена шрифтов

Piu использует свойство шрифта стиля, чтобы найти ресурс шрифта для использования. В этом разделе представлено соглашение об именовании шрифтов, а в следующем разделе объясняется, как эти имена сопоставляются с ресурсом шрифта.

В примере с helloworld имя шрифта задается строкой «24px Open Sans». Этот формат Piu для имени шрифта является подмножеством формата имени шрифта CSS. Название шрифта Piu состоит из пяти частей в следующем порядке:

1. **Style** (Стиль) — (необязательно) стиль шрифта, указанный как курсив или опущенный, если он обычный.
2. **Weight** (Вес) — (необязательно) толщина или толщина шрифта. Вы можете использовать те же ключевые слова и числовые значения, что и в CSS (например, *light*, *bold* (светлый, полужирный) или 800). Каждое ключевое слово имеет эквивалентное числовое значение; например, светлый вес эквивалентен 300, а полужирный — 700. В таблице 10-1 перечислены ключевые слова веса и их эквивалентные числовые значения. Значение по умолчанию *normal* (400) используется, если эта часть имени шрифта опущена.

3. **Stretch** (Растяжение) — (необязательно) расстояние между символами, указанное как сжатое или опущенное, если оно нормальное.
4. **Size** (Размер) — высота шрифта в пикселях. Высота простирается от нижней части выносного элемента до верхней части типичной прописной буквы, как показано на рис. 10-2. Вы можете использовать ключевые слова абсолютного размера из CSS (например, `x-small` или `medium`) или указать размер в пикселях (например, 24 пикселя). Обратите внимание, что фактическая высота зависит от семейства шрифтов. В Табл. 10-2 перечислены ключевые слова размера и соответствующие им размеры в пикселях.
5. **Family** (Семейство) — название семейства шрифтов (например, Times New Roman или Open Sans).



Рис. 10-2. Размер шрифта

Таб. 10-1. Вес ключевых слов

Ключевое слово	Эквивалентный размер
ultralight	100
thin	200
light	300
normal	400
medium	500
semibold	600
bold	700
heavy	800
black	900

Таб. 10-2. Размер ключевых слов

Ключевое слово	Эквивалентный размер
xx-small	9px
x-small	10px
small	13px
medium	16px
large	18px
x-large	24px
xx-large	32px

В Таб. 10-3 перечислены и объяснены примеры имен шрифтов, которые могут быть указаны в свойстве шрифта стиля текста.

Таб. 10-3. Примеры названий шрифтов

Название шрифта Piu	Объяснение
24px Open Sans	Семейство шрифтов — OPEN SANS, размер — 24 пикселя. Растяжка, вес или стиль не указаны, поэтому все они в норме.
bold 16px Fira Sans	Семейство шрифтов — FIRA SANS, размер — 16 пикселей, жирность — полужирный (эквивалент 700). Растяжка или стиль не указаны, поэтому они оба нормальные.
italic boldmedium Open Sans	Семейство шрифтов OPEN SANS и средний размер, или 16 пикселей. Начертание жирное (эквивалентно 700), курсив. Растяжка не указана, так что это нормально.
italic bold condensed small Open Sans	Семейство шрифтов — OPEN SANS, размер — 13 пикселей. Растяжка сокращена, вес выделен жирным шрифтом (эквивалентен 700), а начертание курсивное.

Шрифтовые ресурсы

Название шрифта 24px Open Sans относится к шрифту, хранящемуся в ресурсе с именем OpenSans-Regular-24.fnt. Хотя имя шрифта и имя ресурса явно похожи, они не идентичны. Piu получает от имени шрифта данные ресурса шрифта, применяя набор правил для создания имени ресурса из имени шрифта. Вам необходимо понимать эти правила, чтобы сопоставлять имена шрифтов, которые вы указываете в своем коде, с ресурсами шрифтов, которые вы включаете в манифест вашего проекта.

В следующем списке показаны по порядку части имени ресурса (за исключением расширения .fnt) и объясняется, как Piu генерирует их из имени шрифта.

Примечание Ключевые слова здесь (такие как LIGHT и REGULAR) чувствительны к регистру, поэтому их использование заглавных букв имеет большое значение.

1. **Family** (Семейство) — название семейства шрифтов без пробелов. Например, Open Sans становится OpenSans.
2. **Hyphen** (Дефис) (-) — дефис, отделяющий название семейства шрифтов от следующего за ним.
3. **Stretch** — опускается, если растяжка шрифта нормальная; в противном случае — Condensed.
4. **Weight** (Вес) — опускается, если вес шрифта нормальный; в противном случае вес шрифта — например, Light,Bold (Светлый, Полужирный) или числовое значение, такое как 200.
5. **Style** (Стиль) — опускается, если стиль шрифта обычный; в противном случае курсив.
6. **Regular** (Обычный) — если растяжка, вес и стиль являются нормальными, имя ресурса включает ключевое слово «keywordRegular» вместо всех трех.
7. **Hyphen** (Дефис) (-) — дефис, отделяющий растяжку, толщину и стиль (или Regular - обычный) от следующего за ним размера.
8. **Size** (Размер) — высота в пикселях в виде числа, например, 16 или 24.

Таб. 10-4 приведены примеры имен шрифтов Piu и имен ресурсов, на которые они сопоставляются.

Таб. 10-4. Примеры имен шрифтов, сопоставленных с именами ресурсов

Название шрифта Piu	Имя ресурса	примечание
24px Open Sans	OpenSans-Regular-24.fnt	Пробелы удалены из имени семейства шрифтов. Поскольку растяжка, вес и стиль являются нормальными, вместо этих трех частей в имени ресурса используется REGULAR.
bold 16px Fira Sans	FiraSans-Bold-16.fnt	Размер шрифта перемещается в конец, а жирный шрифт в имени ресурса пишется с заглавной буквы.
italic bold 16px Open Sans	OpenSans-BoldItalic-16.fnt	Несмотря на то, что имя шрифта ставит курсив перед полужирным, имя ресурса указывает полужирный курсив, поскольку вес всегда предшествует стилю. Также обратите внимание, что между жирным шрифтом и курсивом нет пробела или дефиса.

При создании собственных файлов растровых шрифтов назовите файлы в соответствии с соглашениями об именовании ресурсов Piu. Это гарантирует, что когда ваш код укажет имя шрифта, Piu найдет соответствующие данные шрифта в ваших ресурсах.

Дополнительные примечания о шрифтах

Соглашения об именах шрифтов, которые Piu заимствовал из CSS, разработаны так, чтобы быть удобными для разработчиков и в то же время достаточно выразительными для создания сложных пользовательских интерфейсов. Они также обеспечивают согласованность для веб-разработчиков. Однако, хотя CSS обладает мощными возможностями, некоторые разработчики находят его скорее запутанным, чем полезным.

Если вы предпочитаете, вы можете просто использовать имя ресурса шрифта в качестве имени шрифта. Например, `textStyle` в примере с `helloworld` можно определить следующим образом:

```
const textStyle = new Style({
  font: "OpenSans-Regular-24"
});
```

Помните, что для вашего проекта доступны только те шрифты, которые вы включили в свой манифест или предоставили хост. Во многих случаях это всего лишь несколько шрифтов. Если вы укажете неустановленный шрифт, `Piu` не сможет его отобразить. Это отличается от настольных и веб-сред разработки, где всегда есть резервный шрифт.

Поскольку каждый ресурс шрифта соответствует только одному семейству, растяжке, весу, стилю и размеру, вам нужен отдельный ресурс для каждого варианта. Если вы создаете текстовый стиль со свойством шрифта `24px Open Sans`, у вас должен быть ресурс шрифта с именем `OpenSans-Regular-24.fnt`. Даже если у вас есть соответствующий ресурс шрифта, такой как `OpenSans-Regular-12.fnt`, `Piu` не может изменить его размер, чтобы он соответствовал размеру 24 пикселя, указанному в вашем текстовом стиле. Это также отличается от настольных и веб-сред разработки, где распространены шрифты с изменяемым размером.

Добавление цвета

Пример `$EXAMPLES/ch10-piu/helloworld-color` добавляет цвета в пример `helloworld`, чтобы сделать его более интересным. Он имеет всего несколько отличий от `helloworld`.

Во-первых, объект стиля в `helloworld-color` указывает свойство цвета, которое заставляет метку рисовать строку желтым цветом:

```
const textStyle = new Style({
  font: "24px Open Sans",
  color: "yellow"
});
```

В примере также создается объект темы оформления с именем `labelBackground`. Скины управляют отрисовкой фона объектов контента. Объект скина здесь указывает свойство заливки в шестнадцатеричном представлении, как цвет `#1932ab`, оттенок синего.

```
const labelBackground = new Skin({
  fill: "#1932ab"
});
```

Объект `sampleLabel` (листинг 10-5) добавляет свойство `skin` для задания фона, в результате чего фон метки заполняется оттенком синего, указанным в `labelBackground`.

Листинг 10-5.

```
const sampleLabel = new Label(null, {
  left: 0, right: 0, top: 0, bottom: 0,
  style: textStyle,
  string: "Hello, World",
  skin: labelBackground
});
```

Когда вы запускаете пример `helloworld-color` на своем устройстве, вы видите тот же текст и макет, что и для `helloworld`, но с желтым текстом на синем фоне вместо черного текста на белом фоне.

Когда свойство скина не указано, как в примере с `helloworld`, метка ничего не рисует в качестве фона, в результате чего текст появляется перед тем, что находится за ним. Фон белый, потому что в отсутствие скина текст появляется перед самим объектом приложения (созданным хостом, как показано в разделе «Все является объектом»); поскольку хост устанавливает свойство скина приложения в белый, это цвет фона для всего экрана.

Указание цвета

Свойство `color` в объекте стиля примера `helloworld-color` установлено на имя цвета, а свойство `fill` в объекте скина обозначает цвет в шестнадцатеричном представлении. Вы можете указать цвет для этих двух свойств любым способом, как описано в этом разделе.

Свойство цвета в объекте стиля примера установлено в строку «желтый». Piu поддерживает 18 наименований цветов: черный, серебристый, серый, белый, темно-бордовый, красный, фиолетовый, фуксия, зеленый, салатовый, оливковый, желтый, темно-синий, синий, бирюзовый, цвет морской волны, оранжевый и прозрачный. Цвета и их значения RGB взяты из спецификации CSS Level 2.

Свойство заливки в объекте скина в примере имеет значение «`#1932ab`», оттенок синего, указанный в шестнадцатеричной системе счисления. Как показано в листинге 10-6, Piu поддерживает указание цветов в виде строк в любой из четырех шестнадцатеричных нотаций: «`#RGB`», «`#RGBA`», «`#RRGGBB`» и «`#RRGGBBAA`». В этих обозначениях `A` означает «альфа-канал» и представляет уровень прозрачности цвета: альфа-значение `0xFF` означает полностью непрозрачный, `0` означает полностью прозрачный, а промежуточные значения выполняют смешивание. (Альфа-значение совпадает с уровнем смешивания, используемым в некоторых функциях рендеринга POCO, таких как `blendRectangle` и `drawGray`.)

Листинг 10-6.

```
const redSkin = new Skin({
    fill: "#f00"
});
const blendedRedSkin = new Skin({
    fill: "#f008"
});
const greenSkin = new Skin({
    fill: "#00ff00"
});
```

```
const blendedGreenSkin = new Skin({
    fill: "#00ff0080"
});
```

Все эти формы шестнадцатеричной записи цвета также используются в CSS.

Изменение цвета в зависимости от состояния

Ранее в этой главе вы узнали, что свойства объектов обложки и стиля нельзя изменить. Так, например, вы не можете изменить цвет объекта контента, изменив свойство цвета его обложки и объектов стиля; вместо этого вы создаете другую обложку или объект стиля, чтобы изменить цвет. Однако есть и другой подход к изменению цвета, более распространенный и удобный.

Часто причина, по которой вы хотите изменить цвет элемента пользовательского интерфейса, заключается в том, чтобы указать его текущее состояние. Кнопка, например, может иметь три состояния: отключена, включена, но не нажата, включена и нажата. Или метка, отображающая показания датчика, может иметь состояния, когда они находятся в пределах 5 % от целевого значения, в пределах 15 % от целевого значения и более чем на 15 % от целевого значения. Для поддержки таких ситуаций у каждого объекта содержимого есть свойство состояния — число, указывающее его текущее состояние. Piu использует свойство состояния вместе со свойствами объектов стиля для изменения внешнего вида элемента пользовательского интерфейса.

Статическое состояние — это всего лишь число; состояние, которому соответствует номер, зависит от вас, как от разработчика. Вы также должны определить, как изменится пользовательский интерфейс, когда объект содержимого изменит состояние. Например, вы можете сделать кнопку светло-серой, когда она отключена, зеленой, когда она включена, но не нажимается, и темно-зеленой, когда она включена и на нее нажимают.

Простой способ изменить цвет объектов содержимого — использовать их свойство состояния в качестве индекса свойств в объектах темы оформления и стиля. Вы делаете это, устанавливая свойства заливки или цвета объектов обложки или стиля в массив из двух, трех или четырех цветов, а не в строку, представляющую один цвет. Например:

```
const blackAndWhiteStyle = new Style({
  color: ["black", "white"]
});
```

Следуя этому примеру, в листинге 10-7 создается метка с черным текстом, поскольку состояние равно 0, а цвет по индексу 0 — «черный».

Листинг 10-7.

```
const sampleLabel = new Label(null, {
  top: 0, bottom: 0, left: 0, right: 0,
  style: blackAndWhiteStyle,
  state: 0,
  string: "Hello, World"
});
```

Когда вы изменяете свойство состояния, элемент пользовательского интерфейса перерисовывается соответствующим цветом из его стиля. Изменение состояния на 1 здесь приводит к перерисовке метки с белым текстом.

```
sampleLabel.state = 1;
```

Вы также можете использовать нецелые значения для состояния, вызывая смешение цветов из окружающих состояний. Например, вы можете сделать текст серым в этом примере следующим образом:

```
sampleLabel.state = 0.5;
```

Возможность указывать дробные значения для состояния может показаться странной с концептуальной точки зрения; что означает, например, что кнопка находится на полпути между отключенным и включенным состояниями? Однако есть несколько интересных применений, например, когда вы анимируете между двумя состояниями: вы можете создать стиль с двумя цветами и медленно изменять цвет метки от первого до второго, изменяя ее состояние с 0 на 1 с небольшими приращениями.

Реакция на события поведением

После того, как у вас есть несколько объектов содержимого на экране, следующим шагом будет предоставление им возможности выполнять действия в ответ на события. Вы делаете это с объектами поведения.

Объект поведения представляет собой набор методов. Вы прикрепляете объект поведения к объекту содержимого, устанавливая его свойство поведения. Когда объект содержимого получает событие, он ищет в своем объекте поведения метод, соответствующий этому событию; если он находит метод с тем же именем, что и у события, он вызывает этот метод для обработки события.

Piu определяет набор событий, которые он запускает по мере необходимости. Например, он запускает событие `onTouchBegan`, когда палец помещается на объект содержимого, и событие `onTouchEnded`, когда палец удаляется. Класс `TraceBehavior` в листинге 10-8 содержит методы, которые реагируют на события Piu `onTouchBegan` и `onTouchEnded` путем трассировки до консоли отладки.

Листинг 10-8.

```
class TraceBehavior extends Behavior {
    onTouchBegan(label) {
        trace("touch began\n");
    }
    onTouchEnded(label) {
        trace("touch ended\n");
    }
}
```

События, определенные и запускаемые Piu, называются событиями низкого уровня. Вы также можете определить свои собственные события, используя любое имя; они называются событиями высокого уровня. Например, вы можете создать событие `onSensorValueChanged`, которое ваше приложение инициирует при изменении значения датчика. В оставшейся части этого раздела представлены некоторые часто используемые низкоуровневые события; позже в этой главе вы узнаете, как определять и запускать свои собственные высокоуровневые события.

“Hello, World” с поведением

Пример `$EXAMPLES/ch10-piu/helloworld-behavior` добавляет поведение к примеру `helloworld`, чтобы строка «Hello, World» отображалась по одному символу за раз при касании экрана. Это простое поведение показывает, как Piu доставляет события объектам контента.

Объект `sampleLabel` в примере поведения `helloworld` (листинг 10-9) аналогичен объекту из `helloworld`. Однако есть три важных отличия:

- Свойство `string` объекта `sampleLabel` начинается с пустой строки, поэтому в ответ на касание его можно заполнить по одному символу за раз.
- Активному свойству присвоено значение `true`. Это свойство указывает, должен ли объект содержимого реагировать на события касания. Если установлено значение `true`, Piu инициирует события, связанные с касанием, такие как `onTouchBegan`. Значение по умолчанию — `false`, поэтому вы должны явно установить для `active` значение `true`, чтобы сделать содержимое доступным для касания.
- Свойство `Behavior` указано в словаре, переданном конструктору. Он задает поведение `sampleLabel` как класса `LabelBehavior`.

Листинг 10-9.

```
const sampleLabel = new Label(null, {
  top: 0, bottom: 0, left: 0, right: 0,
  style: textStyle,
  string: "",
  active: true,
  Behavior: LabelBehavior
});
sampleLabel.message = "Hello, World";
```

LabelBehavior — это класс, расширяющий встроенный класс Behavior:

```
class LabelBehavior extends Behavior {
  ...
}
```

При создании sampleLabel Piu также создает экземпляр LabelBehavior и присваивает его свойству поведения sampleLabel. Обратите внимание, что свойство Behavior пишется с заглавной буквы в словаре, передаваемом конструктору Label, тогда как свойство поведения созданного экземпляра пишется с маленькой буквы; это потому, что имена свойств следуют тому же соглашению о написании заглавных букв, что и значения, которые они принимают: класс передается конструктору в свойстве Behavior (а имена классов в JavaScript по соглашению пишутся в верхнем регистре), тогда как свойство поведения sampleLabel содержит экземпляр класса (а имена экземпляров в JavaScript по соглашению пишутся строчными буквами).

LabelBehavior имеет только один метод onTouchBegan, показанный в листинге 10-10. Аргументом этого метода является сам объект метки. Первым аргументом всех методов обработчиков событий, вызываемых в поведении, является объект содержимого, к которому они присоединены. При вызове этот метод добавляет следующий символ из строки «Hello, World» в объект метки до тех пор, пока не будут добавлены все символы. Затем он устанавливает для свойства active объекта метки значение false, чтобы запретить ему получать дальнейшие события касания.

Листинг 10-10.

```
onTouchBegan(label) {
    const message = label.message;
    label.string = message.substring(0, label.string.length + 1);
    if (label.string === message)
        label.active = false;
}
```

Это все, что нужно для реализации базового сенсорного поведения. Когда вы запускаете пример и нажимаете на объект метки (который занимает весь экран), Piu запускает событие `onTouchBegan` объекта. Затем объект `label` проверяет свое поведение, чтобы узнать, есть ли у него метод `onTouchBegan`; это так, поэтому он вызывает этот метод, передавая ссылку на себя в качестве первого аргумента.

Многие низкоуровневые события имеют дополнительные аргументы, которые могут быть полезны в ваших проектах. Например, событие `onTouchBegan` также передает эти четыре аргумента:

- `id` — идентификатор точки касания, используемой для поддержки мультитач. В этом примере поддерживается только одна точка касания, поэтому `id` всегда равен 0. Значение `id` — это число, поступающее от сенсорного контроллера, позволяющее различать разные точки касания на экране.
- `x` и `y` — глобальные координаты события, то есть точки касания, в пикселях.
- `ticks` — глобальное время события в миллисекундах. Это значение не является временем суток и не связано с UTC; оно используется только для определения времени, прошедшего между двумя событиями.

Когда вы работаете с событием в первый раз, хороший способ хорошо его понять — добавить в поведение метод для трассировки аргументов, которые оно получает, в консоль отладки. Например, чтобы узнать подробности того, как и когда вызывается `onTouchBegan`, измените пример `helloworld-behavior` на функцию, показанную в листинге 10.11.

Листинг 10-11.

```
onTouchBegan(label, id, x, y, ticks) {
    trace(`id: ${id}\n`);
    trace(`{x, y}: ${x}, ${y}\n`);
    trace(`ticks: ${ticks}\n`);
}
```

События `onTimeChanged` и `onDisplaying`

В этом разделе представлены эти часто используемые низкоуровневые события:

- Событие `onTimeChanged` дает вам доступ к часам, встроенным в каждый объект содержимого `Piu`.
- Событие `onDisplaying` дает вашему поведению возможность настроить себя до того, как объект контента появится на экране.

Эти события представлены в примере `$EXAMPLES/ch10-piu/helloworld-ticking`, который похож на пример `helloworld-behavior` тем, что добавляет на экран по одному символу «Hello, World» за раз; однако вместо того, чтобы добавлять символы при нажатии на экран, они добавляются через равные промежутки времени. Обратите внимание на следующее в этом примере:

- Объект `sampleLabel` идентичен объекту в `helloworld-behavior`, за исключением того, что для его свойства `active` не задано значение `true`, поскольку он не реагирует на события касания.

- Класс `LabelBehavior` включает методы `onDisplaying` и `onTimeChanged` (листинг 10-12) вместо метода `onTouchBegan`. Их первый аргумент — это ссылка на объект метки, связанный с поведением, как и со всеми событиями, определенными `Piu`.

Листинг 10-12.

```
class LabelBehavior extends Behavior {
    onDisplaying(label) {
        ...
    }
    onTimeChanged(label) {
        ...
    }
}
```

Событие `onDisplaying` запускается после того, как объект контента добавлен в объект приложения, но до того, как он станет видимым для пользователя. Это полезно для инициализации состояния объекта, особенно объектов содержимого, которые могут быть скрыты, а затем показаны несколько раз. Одним из распространенных применений события `onDisplaying` является запуск таймера, который используется для анимации внешнего вида объекта содержимого.

Поскольку анимация является такой распространенной частью современных пользовательских интерфейсов, `Piu` снабжает каждый объект контента встроенными часами. Часы «тикают» с интервалом, заданным свойством `interval` объекта содержимого. И свойство `interval`, и часы выражают время в миллисекундах. Каждый раз, когда часы тикают, генерируется событие `onTimeChanged`. Часы не всегда идут и изначально остановлены; вы используете методы запуска и остановки объекта контента, чтобы контролировать, когда работают его часы.

В этом примере метод поведения `onDisplaying` (листинг 10-13) начинается со сброса свойства `index`, в котором хранится количество символов строки, находящихся в объекте метки в любой момент времени. Код задает для свойства `interval` значение 250 миллисекунд, чтобы запросить создание события `onTimeChanged` каждую четверть секунды. Наконец, метод запускает отсчет часов метки, вызывая свой метод запуска.

Листинг 10-13.

```
onDisplaying(label) {  
    this.index = 0;  
    label.interval = 250;  
    label.start();  
}
```

Метод поведения `onTimeChanged` (листинг 10-14) добавляет один новый символ в строковое свойство объекта метки с каждым интервалом. Он использует метод подстроки, который возвращает часть строки. Аргументы `substring` определяют индексы первого символа, который нужно включить, и первого символа, который нужно исключить, соответственно. После отображения всей строки `onTimeChanged` вызывает метод остановки метки, чтобы часы не тикали, и `onTimeChanged` больше не срабатывал.

Листинг 10-14.

```
onTimeChanged(label) {  
    const message = label.message;  
    this.index += 1;  
    if (this.index > message.length)  
        label.stop();  
    else  
        label.string = message.substring(0, this.index);  
}
```


В последующих примерах показано, как использовать часы объекта содержимого для управления анимацией.

Добавление изображений

Изображения являются фундаментальной частью создания пользовательского интерфейса. Так же, как Piu использует объекты оболочки для заполнения области экрана сплошным цветом, он также использует их для заполнения области экрана изображением, позволяя любому объекту содержимого рисовать изображения. Текстура в объекте скина определяет используемое изображение.

Чтобы показать, как визуализировать изображение, пример \$EXAMPLES/ch10-piu/js-icon рисует логотип JavaScript. В примере показан экран, показанный на рис. 10-3.



Рис. 10-3. Пример js-icon

Объект скина используется для создания значка. Первым шагом является создание ссылки на файл изображения для использования путем создания экземпляра объекта текстуры. Свойство path словаря, переданного конструктору Texture, — это имя ресурса, содержащего изображение.

```
const jsLogoTexture = new Texture({
  path: "js.png"
});
```

Обратите внимание, что имя ресурса имеет расширение .png вместо .bmpas, которое вы видели для Росо в главе 9. Хотя изображение PNG все еще преобразуется в другой формат для рендеринга на микроконтроллере, Piu знает об этом преобразовании и автоматически изменяет .png. расширение на правильное расширение для устройства.

В примере с helloworld-color вы использовали объект кожи со свойством fill для создания однотонного фона. В этом примере вместо fill вы используете свойство текстуры вместе со свойствами высоты и ширины, чтобы создать скин jsLogoSkin, который заполняет объекты содержимого с помощью jsLogoTexture. Свойства высоты и ширины установлены в соответствии с размерами файла изображения js.png, 100 x 100 пикселей.

```
const jsLogoSkin = new Skin({
  texture: jsLogoTexture,
  height: 100, width: 100
});
```

Последний шаг — создать объект контента, который ссылается на jsLogoSkin:

```
const jsLogo = new Content(null, {
  skin: jsLogoSkin
});
```

Поскольку обложка определена как 100 x 100 пикселей, объект содержимого jsLogo по умолчанию имеет такие же размеры.

Рисование части изображения

Возможно, вы задавались вопросом, почему ранее вам приходилось указывать свойства высоты и ширины в конструкторе кожи. Почему скин не использовал по умолчанию все изображение целиком? Причина в такой особенности объекта кожи, которая позволяет рисовать только часть текстуры. Чтобы указать область текстуры для рисования, вы используете свойства x, y, height и width для определения исходного прямоугольника в пикселях. Свойства x и y по умолчанию равны 0, но свойства высоты и ширины являются обязательными.

Код в листинге 10-15 является альтернативой `jsLogoSkin` в примере `js-icon`. Здесь скин определен для рисования квадрата 70 x 70 пикселей из нижнего правого угла изображения. Результат показан на рисунке 10-4.

Листинг 10-15.

```
const jsLogoSkin = new Skin({
  texture: jsLogoTexture,
  x: 24, y: 30,
  height: 70, width: 70
});
```

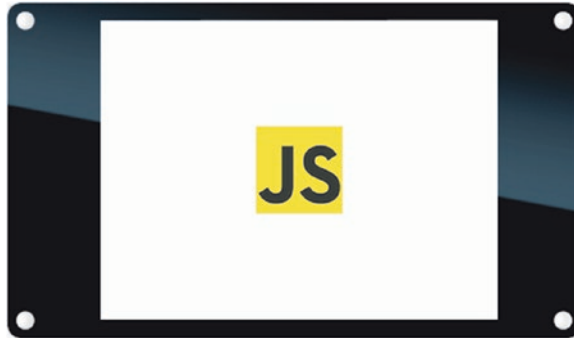


Рис. 10-4. Пример `js`-иконки с обрезанным `jsLogoSkin`

Рисовать часть одного значка обычно бывает нечасто. В конце концов, если вы хотите нарисовать только правый нижний угол, вы можете также обрезать файл изображения и сэкономить место для хранения. Однако часто бывает удобно хранить несколько значков в одном изображении, и в этом случае очень полезна возможность рисовать часть изображения. В следующем разделе рассматривается пример.

Рисование нескольких значков из одного изображения

Вспомните значки на рис. 10.5, которые вы впервые увидели в главе 8. Эти значки показывают несколько различных состояний соединения Wi-Fi и объединены в одно изображение.



Рис. 10-5. Иконки Wi-Fi

Значки организованы в единую сетку, в которой столбцы и строки следующие:

- Каждый столбец представляет собой различное состояние значка Wi-Fi, представляющее уровни мощности сигнала от слабого до сильного.
- Каждая строка представляет собой отдельный вариант значка Wi-Fi. Верхний ряд — это вариант открытой точки доступа Wi-Fi, а нижний ряд — вариант защищенной точки доступа Wi-Fi.

Точно так же, как `Piu` использует свойство состояния объекта контента, чтобы определить, какой цвет рисовать из стиля, он может использовать свойства состояния и варианта объекта контента, чтобы определить, какой значок рисовать из текстуры, содержащей сетку значков. Для этого скин, содержащий текстуру здесь, должен указать ширину каждого столбца и высоту каждой строки, используя свойства состояний и вариантов соответственно в словаре, используемом для создания скина (см. листинг 10-16).

Листинг 10-16.

```
const wifiTexture = new Texture({
    path: "wifi-strip.png"
});
```

```
const wifiSkin = new Skin({
    texture: wifiTexture,
    width: 28, height: 28,
    states: 28,
    variants: 28
});
```

Изображение в этом примере содержит квадратные значки 28 пикселей, поэтому свойства состояний и вариантов имеют значение 28. Кроме того, свойства высоты и ширины имеют значение 28, так что размер кожи точно соответствует размеру одного значка.

Пример `$EXAMPLES/ch10-piu/wifi-status` рисует один значок из этого изображения за раз, меняя значок раз в секунду. Он начинается со значка в верхнем левом углу (состояние и вариант равны 0), как показано в листинге 10.17.

Листинг 10-17.

```
const wifiIcon = new Content(null, {
    skin: wifiSkin,
    state: 0,
    variant: 0,
    Behavior: WifiIconBehavior
});
```

Свойства состояния и варианта объекта содержимого могут быть обновлены в любое время. В этом примере они изменяются для перемещения по полосе значков по одному значку за раз, слева направо, сначала по верхнему ряду, а затем по нижнему ряду; затем он возвращается к верхнему ряду и повторяет это бесконечно. Как показано в листинге 10.18, обработчики событий `onDisplaying` и `onTimeChanged` в поведении `wifiIcon` используют встроенные часы объекта содержимого для управления анимацией (как вы видели в примере с `helloworld-ticking`): поведение изменяет свойство `variant` на каждом такте, перемещение по ряду иконок; когда достигается последний значок в строке, он изменяет свойство состояния, чтобы переключиться на другую строку.

Листинг 10-18.

```

class WifiIconBehavior extends Behavior {
    onDisplaying(content) {
        content.interval = 1000;
        content.start();
    }
    onTimeChanged(content) {
        let variant = content.variant + 1;
        if (variant > 4) {
            variant = 0;
            content.state = content.state ? 0 : 1;
        }
        content.variant = variant;
    }
}

```

Использование масок

Сжатые маски оттенков серого более эффективны для хранения изображений в градациях серого, чем полноцветные растровые изображения, и (как вы узнали из главы 8) маски можно рисовать любым цветом. Многие значки, нарисованные в пользовательских интерфейсах, имеют только один цвет и, следовательно, могут быть сохранены как маска. Объект текстуры может относиться к ресурсу изображения маски, а также к ресурсу цветового растрового изображения, позволяя вашему пользовательскому интерфейсу включать и то, и другое.

Добавление изображения маски в ваше приложение очень похоже на добавление полноцветного растрового изображения. Пример `$EXAMPLES/ch10-piu/mask-icon` показывает значок, сохраненный как маска. При касании значка он меняет цвет.

Текстура и свойства кожи в примере (листинг 10-19) должны выглядеть знакомыми. Ключевое отличие состоит в том, что `maskSettingsSkin` определяет свойство цвета с двумя цветами: «оранжевый», когда свойство

состояния объекта содержимого имеет значение 0, и «желтый», когда оно равно 1. (Обратите внимание, что существует два разных способа указать цвет объекта: когда вы используете скин для рисования текстуры маски, вы указываете свойство цвета; чтобы создать однотонный фон, вы указываете свойство заливки.)

Листинг 10-19.

```
const maskSettingsTexture = new Texture({
  path: "settings-mask.png"
});

const maskSettingsSkin = new Skin({
  texture: maskSettingsTexture,
  width: 80, height: 80,
  color: ["orange", "yellow"]
});
```

Как обычно, вам нужно создать объект содержимого, который ссылается на скин. В листинге 10-20 показан объект, созданный в этом примере: объект содержимого, который также имеет поведение и активное свойство (установленное в значение true, чтобы объект мог получать события касания).

Листинг 10-20.

```
const maskSettingsIcon = new Content(null, {
  skin: maskSettingsSkin,
  state: 0,
  active: true,
  Behavior: SettingsIconBehavior
});
```

Когда значок рисуется впервые, маска рисуется оранжевым цветом, потому что значение состояния 0 означает, что используется цвет с индексом 0 массива в свойстве color.

Как показано в листинге 10-21, этот пример обеспечивает обратную связь при касании с помощью событий `onTouchBegan` и `onTouchEnded`, запускаемых в начале и в конце касания:

- Когда `maskSettingsIcon` получает событие `onTouchBegan`, его поведение устанавливает свое состояние в 1, заставляя его перерисовывать цвет с индексом 1 его свойства `color` — в данном случае желтый.
- Когда `maskSettingsIcon` получает событие `onTouchEnded`, его поведение меняет свое состояние на 0, и значок снова становится оранжевым.

Листинг 10-21.

```
class SettingsIconBehavior extends Behavior {
    onTouchBegan(content) {
        content.state = 1;
    }
    onTouchEnded(content) {
        content.state = 0;
    }
}
```

Мозаика изображений

Вы можете рисовать повторяющиеся узоры, накладывая текстуру кожи на мозаику. Это еще один способ уменьшить пространство для хранения, поскольку вы можете использовать файлы изображений, которые представляют собой одну плитку фона, а не полный размер экрана.

Мозаика одного изображения

Пример `$EXAMPLES/ch10-piu/tiled-background` использует изображение на рис. 10-6 для создания мозаичного фона, показанного на рис. 10-7.

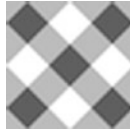


Рис.10-6. Изображение из примера плиточного фона

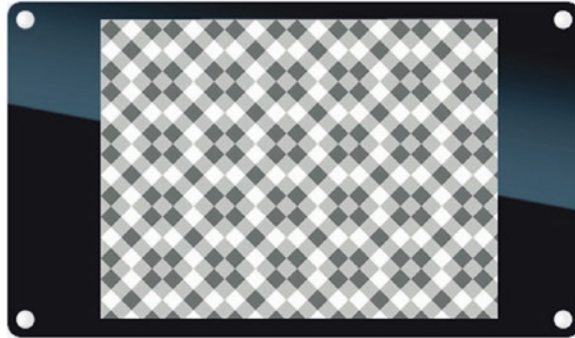


Рис. 10-7. Пример *tiled-background*

Как и скин значка, мозаичный скин использует объект текстуры и определяет свойства высоты и ширины, чтобы указать область текстуры для рисования (в данном случае всю). Как показано в листинге 10.22, вы также включаете свойство плитки — объект со свойствами слева, справа, сверху и снизу, указывающими разные части текстуры для мозаичного покрытия; здесь все они равны 0, потому что в этом примере все изображение используется как повторяющаяся плитка. (В следующем разделе, посвященном рисованию изображений с 9 фрагментами, объясняется, как использовать эти четыре свойства с другими значениями.)

Листинг 10-22.

```
const tileTexture = new Texture({
    path: "tile.png"
});
```

```
const tileSkin = new Skin({
    texture: tileTexture,
    height: 50, width: 50,
    tiles: {
        left: 0, right: 0, top: 0, bottom: 0
    }
});
```

Когда вы присоединяете tileSkin к объекту полноэкранного контента, он отрисовывается, как показано на рис. 10-7:

```
const background = new Content(null, {
    left: 0, right: 0, top: 0, bottom: 0,
    skin: tileSkin
});
```

Рисование изображений с 9 патчами с помощью плиток

Рисование изображений с 9 патчами с помощью плиток. Изображение с 9 участками используется для эффективного рисования прямоугольных фигур, таких как прямоугольник со скругленными углами, разных размеров. Термин «9-патч» происходит от мобильной ОС Android, хотя эта концепция широко используется в других местах; это относится к тому, как ресурс изображения делится на девять частей, как вы скоро увидите. Множество интересных эффектов можно создать с помощью изображений с 9 патчами. Piu воплощает эту концепцию за счет использования мозаичного скина.

Вспомним ранее, что свойства объекта плитки указывают разные участки текстуры для плитки. Более конкретно, эти свойства определяют части изображения с 9 фрагментами, указывая количество пикселей от краев изображения, как показано на рис. 10-8 для объекта плитки, все свойства которого задают 14. нарисуйте девять частей и присвойте номер каждой из них. Все изображение представляет собой квадрат размером 56 пикселей.

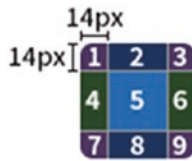


Рис. 10-8. Скругленный прямоугольник с девятью очерченными частями

Мозаичный скин для этого изображения будет определен, как в листинге 10-23.

Листинг 10-23.

```
const tileSkin = new Skin({
    texture: tileTexture,
    height: 56, width: 56,
    tiles: {
        left: 14, right: 14, top: 14, bottom: 14
    }
});
```

Когда этот скин применяется к объекту контента, Piu рисует девять частей изображения, используя следующие правила:

- Зоны 1, 3, 7 и 9 рисуются по одному разу в соответствующем углу объекта содержимого.
- Зоны 2 и 8 повторяются горизонтально вверху и внизу объекта содержимого соответственно.
- Зоны 4 и 6 повторяются по вертикали вдоль левой и правой сторон объекта содержимого соответственно.
- Зона 5 повторяется по вертикали и горизонтали, чтобы заполнить пространство в середине объекта содержимого, не закрытое другими плитками.

На рис. 10-9 показано, как этот мозаичный скин визуализируется объектами содержимого со следующими размерами (слева направо): 28 x 28, 56 x 56, 110 x 100 и 70 x 165. Обратите внимание, что девять частей изображения только повторяются, и их размер никогда не изменяется.

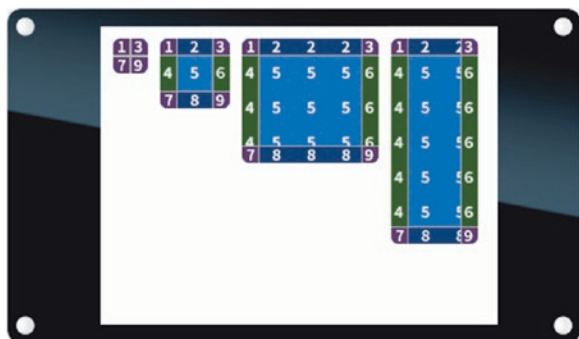


Рис. 10-9. *tileSkin отображается в разных размерах*

В примере \$EXAMPLES/ch10-piu/rounded-buttons используется простой сплошной прямоугольник со скругленными углами для создания кнопок разных размеров (рис. 10.10).

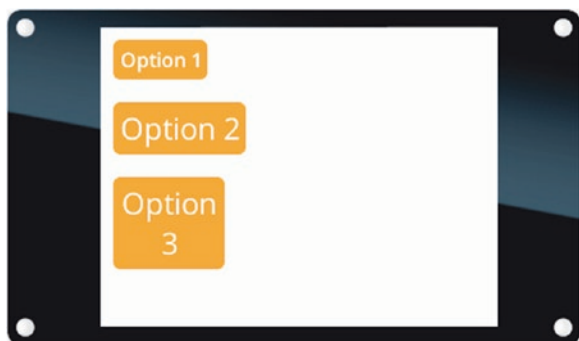


Рис.10-10. *Пример rounded-buttons*

Обложка в этом примере определена, как показано в листинге 10-24. Он похож на предыдущий пример кожи, но ресурс изображения меньше, а свойствам левого, правого, верхнего и нижнего объекта плитки присвоено значение 5. Он также определяет свойство цвета; тайловые кожи могут использовать маски.

Листинг 10-24.

```
const roundedTexture = new Texture({
  path: "button.png"
});

const roundedSkin = new Skin({
  texture: roundedTexture,
  width: 30, height: 30,
  color: ["#ff9900", "#ffd699"],
  tiles: {
    top: 5, bottom: 5, left: 5, right: 5
  }
});
```

Три кнопки в этом примере — это метка и текстовые объекты (листинг 10-25). Они имеют разную высоту и ширину, но `roundedSkin` разбивает свою текстуру, как описано выше, чтобы соответствовать всем размерам.

Листинг 10-25.

```
const button1 = new Label(null, {
  top: 10, left: 10,
  skin: roundedSkin,
  style: smallTextStyle,
  string: "Option 1",
  active: true,
  Behavior: ButtonBehavior
});
```

```

const button2 = new Label(null, {
    top: 60, left: 10,
    skin: roundedSkin,
    style: textStyle,
    string: "Option 2",
    active: true,
    Behavior: ButtonBehavior
});

const button3 = new Text(null, {
    top: 120, left: 10, width: 90,
    skin: roundedSkin,
    style: textStyle,
    string: "Option 3",
    active: true,
    Behavior: ButtonBehavior
});

```

Вспомните, что объект метки отображает текст в одной строке; в этом примере используется текстовый объект для третьей кнопки, чтобы проиллюстрировать, что текстовые объекты, в отличие от объектов меток, могут отображать текст на нескольких строках.

Поведение `ButtonBehavior` в этом примере идентично `SettingsIconBehavior` в примере с значком маски, а методы `onTouchBegan` и `onTouchEnded` обеспечивают обратную связь при нажатии кнопок.

Создание составных элементов пользовательского интерфейса

Пользовательские интерфейсы реальных продуктов состоят из более сложных элементов, чем просто строка текста или один значок в центре экрана. Первые примеры в этой главе используют простую структуру для введения фундаментальных концепций Пиу; Теперь вы готовы объединить эти элементы для создания более сложных интерфейсов.

Добавление объектов содержимого к объекту приложения создает древовидную структуру данных, называемую иерархией включения. Простые примеры до сих пор создавали двухуровневую иерархию сдерживания, с объектом приложения в корне и объектами содержимого в качестве листьев, но в иерархии может быть много уровней.

Иерархия вложений организует объекты содержимого в пользовательском интерфейсе, помещая их в группы, называемые контейнерами. Контейнеры реализуются классом `Container`, ключевым встроенным классом `Piu`. Сам объект приложения является контейнером, поэтому он может содержать другие объекты содержимого. Иерархия сдерживания делает больше, чем группирует объекты содержимого вместе; это также влияет на то, как объекты рисуются и как они получают события.

Идея вмещающей иерархии должна быть вам знакома, если вы когда-либо создавали пользовательский интерфейс с помощью HTML или других объектно-ориентированных фреймворков пользовательского интерфейса. Если нет, пример в следующем разделе поможет вам начать работу, проведя вас через этапы построения иерархии включения.

Создание заголовка

Как и предыдущие примеры в этой главе, пример `$EXAMPLES/ch10-piu/header` добавляет на экран текст и значок. Но вместо того, чтобы рассматривать их как отдельные элементы, как в этих примерах, они объединяются в один составной элемент пользовательского интерфейса, заголовок которого показан на рис. 10.11.

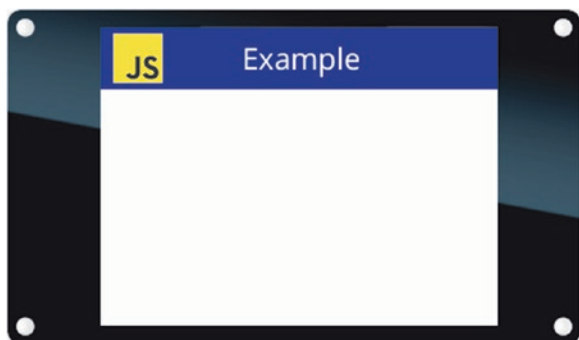


Рис. 10-11. Пример *header*

Объекты `jsLogo` и `headerText` в примере заголовка (листинг 10-26) аналогичны объектам содержимого и метки в предыдущих примерах.

Листинг 10-26.

```
const jsLogo = new Content(null, {
  left: 10,
  skin: jsLogoSkin
});

const headerText = new Label(null, {
  style: textStyle,
  string: "Example"
});
```

Объект заголовка (листинг 10-27) является экземпляром класса `Container`. Класс `Container` наследуется от класса `Content` и расширяет его, позволяя хранить другие объекты содержимого.

Листинг 10-27.

```
const header = new Container(null, {
  top: 0, height: 50, left: 0, right: 0,
  skin: headerSkin,
```



```

    contents: [
        jsLogo,
        headerText
    ]
});

```

Объект заголовка содержит объекты `jsLogo` и `headerText`, которые помещаются в массив свойств содержимого. Свойство `skin` придает объекту заголовка синий фон (поскольку свойство `headerSkin` имеет свойство заполнения `"#1932ab"`).

Поскольку объекты `jsLogo` и `headerText` содержатся в объекте заголовка, когда объект заголовка добавляется к объекту приложения, все элементы — синий фон, значок и текст — появляются на экране:

```
application.add(header);
```

Точно так же удаление объекта заголовка приводит к исчезновению всех элементов, а перемещение объекта заголовка по экрану приводит к одновременному перемещению всех содержащихся в нем элементов.

Когда объект контента добавляется в контейнер, объект контента считается дочерним объектом или просто дочерним элементом контейнера; соответственно, говорят, что контейнер является родительским объектом объекта содержимого или просто родительским. В этом примере заголовок является родительским контейнером, а текст и значок — дочерними объектами заголовка.

Свойство контейнера объекта можно использовать для доступа к его родительскому объекту-контейнеру, а свойство длины — для определения количества дочерних объектов в контейнере. Если у объекта нет родительского контейнера, его свойство контейнера равно `null`. Если дочерних объектов нет, свойство `length` равно 0. Несколько различных методов доступа к объектам в контейнере описаны в разделе «Доступ к объектам содержимого в контейнере» далее в этой главе.

Относительные и абсолютные координаты

Как вы узнали, свойства `left`, `right`, `top` и `bottom`, передаваемые в конструктор объекта содержимого, определяют положение объекта содержимого путем указания полей между объектом и его контейнером. Поскольку эти свойства выражают расположение точек относительно родительского контейнера, они называются относительными координатами. Например, когда вы передаете `left` со значением 10, это не обязательно означает, что объект содержимого будет находиться в 10 пикселях от левой стороны экрана при его рисовании; это означает, что содержимое будет находиться на расстоянии 10 пикселей от левой стороны любого контейнера, в который оно помещено.

Координаты объекта содержимого после его рисования на экране называются абсолютными координатами, которые выражают расположение точек как расстояние от краев экрана. Когда контейнер представляет собой весь экран, что обычно имеет место для объекта приложения, относительные и абсолютные координаты дочерних объектов контейнера совпадают.

Когда контейнер перемещается, `Piu` корректирует абсолютные координаты всех дочерних объектов содержимого контейнера. Это значительно упрощает анимацию составных элементов пользовательского интерфейса, таких как заголовок в примере с заголовком, поскольку ваш код должен перемещать только контейнер составного элемента, а не каждый отдельный элемент содержимого.

Добавление и удаление содержимого контейнера

Содержимое контейнера не фиксируется. Точно так же, как вы можете добавлять и удалять объекты из объекта приложения, вы можете добавлять и удалять объекты из объекта-контейнера в любое время. Класс `Container` и все классы, которые унаследованы от него, имеют методы добавления и удаления, которые вы используете для изменения их массива содержимого. Класс `Application` — это один общий класс, который наследуется от класса `Container`.

Вы можете вызвать метод добавления контейнера в любое время, независимо от того, является ли контейнер частью иерархии включения. Например, вместо того, чтобы передавать массив

содержимого в конструктор при создании объекта заголовка (как в листинге 10-27 ранее), вы можете добавить каждый объект содержимого в заголовок после создания экземпляров всех объектов, но до добавления заголовка в приложение. объект (см. листинг 10.28).

Листинг 10-28.

```
const header = new Container(null, {
    top: 0, height: 50, left: 0, right: 0,
    skin: headerSkin
});

header.add(jsLogo);
header.add(headerText);
application.add(header);
```

В любом случае результат один и тот же: jsLogo и headerText содержатся в заголовке, а заголовок содержится в объекте приложения. Это создает трехуровневую иерархию сдерживания с объектом приложения в корне, заголовком в качестве ветви и jsLogo и headerText в качестве листьев.

Вот как вы можете использовать метод удаления, чтобы удалить jsLogo из дочернего списка заголовка:

```
header.remove(jsLogo);
```

Пустой метод удаляет все дочерние элементы из контейнера. Это полезно, когда вам нужно перестроить содержимое составного узла, например, при переходе на другой экран (как вы увидите позже, в разделе «Логика приложения»).

```
header.empty();
```

Один контейнер для каждого объекта контента

Объект содержимого может быть потомком только одного контейнера в любой момент времени. Вы можете добавлять и удалять объект из его контейнера столько раз, сколько хотите, и вы можете перемещать объект в новый контейнер, удаляя его из текущего контейнера и добавляя в новый; однако вы не можете добавить один и тот же объект содержимого в несколько контейнеров одновременно. Если вы попытаетесь добавить объект содержимого, уже находящийся в одном контейнере, в другой контейнер, Piu выдаст ошибку.

Это может показаться странным. Вы можете подумать, что добавление одного и того же объекта в несколько контейнеров приведет к созданию идентичных объектов, которые помещаются в разные контейнеры, но это не так. Каждый графический элемент, отображаемый на экране, связан с одним объектом содержимого.

Если это все еще кажется странным, метафора реального мира может помочь вам понять это. Представьте, что у вас есть две коробки и один физический объект — например, ручка. Вы можете положить ручку в любую коробку, но не в обе коробки одновременно. То же правило применяется к объектам контента и контейнерам в Piu.

Конечно, вы всегда можете создать одинаковое содержимое и поместить его в разные контейнеры. Далее в этой главе вы узнаете о простом способе создания похожего или идентичного содержимого с помощью шаблонов.

Создание адаптивных макетов

На экране, показанном на рис. 10-12, отображается панель навигации, состоящая из трех кнопок, каждая из которых имеет значок и текст, определяющие ее назначение. Если бы их попросили описать положение этих кнопок, большинство людей сказали бы что-то вроде: «В середине экрана есть ряд равномерно расположенных кнопок». Мало кто скажет: «Есть одна кнопка на 20 пикселей слева и 74 пикселя сверху экрана, одна на 120 слева и 74 сверху, а еще одна на 220 слева и 74 сверху». Другими словами, люди, скорее всего, описывают правило расположения, а не координаты каждой кнопки.

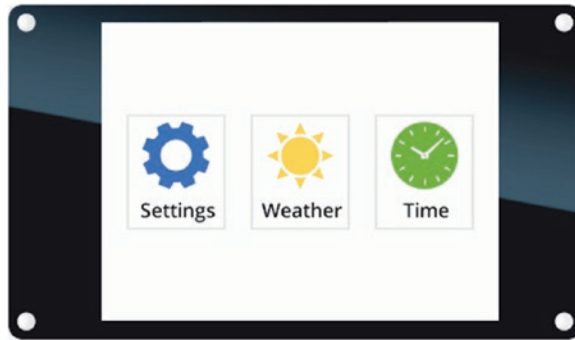


Рис. 10-12. Ряд кнопок в центральной панели навигации

Правило компоновки — это краткий способ описания того, как упорядочивать объекты содержимого в контейнере. Правило компоновки может не зависеть от текущего размера контейнера, подстраиваясь под любой текущий размер. Например, макет, показанный здесь, может равномерно распределять кнопки независимо от ширины контейнера (экрана) 320 или 480 пикселей. Правило макета, которое разумно адаптируется к изменениям размера родительского контейнера, называется *адаптивным макетом*.

Если у вас есть опыт работы в веб-дизайне или написании мобильных приложений, вы, вероятно, знакомы с концепцией адаптивного макета. Хорошие веб-страницы должны хорошо отображаться независимо от размера окна браузера или экрана; другими словами, они реагируют на различия в размерах. Многие мобильные приложения поворачиваются в зависимости от ориентации экрана; то есть они реагируют на изменение ориентации.

Piu также имеет функции, которые позволяют создавать адаптивные макеты. Эти функции часто бывают полезны, как показано в нескольких следующих примерах, даже если размер экрана во всех моделях вашего продукта одинаков.

Макеты строк и столбцов

В примере \$EXAMPLES/ch10-piu/nav-bar отображается панель навигации, показанная на рис. 10-12. Объект столбца объединяет значок и метку для каждого из трех составных элементов кнопки. В листинге 10-29 показан код самой левой кнопки. Код для двух других кнопок следует тому же шаблону, но с другим оформлением и меткой для каждой. (Для упрощения примера поведение каждой кнопки опущено.)

Листинг 10-29.

```
const settingsButton = new Column(null, {
  skin: outlineSkin, width: 80,
  contents: [
    Content(null, {
      top: 5,
      skin: settingsSkin
    }),
    Label(null, {
      top: 0,
      style: textStyle,
      string: "Settings"
    })
  ]
});
```

Класс Column расширяет класс Container правилом компоновки для размещения его содержимого в вертикальном столбце. В этом примере объект содержимого имеет верхнее поле 5, а объект метки имеет верхнее поле 0. Если вы поместите их в контейнер, они будут перекрываться; однако, поскольку они находятся в объекте-столбце, верхнее поле объекта содержимого относится к объекту-столбцу, а верхнее поле объекта метки — к нижнему краю объекта содержимого. Если вы добавите другой объект, его верхнее поле будет относительно нижнего края объекта этикетки и т. д.

Объекты-столбцы для всех трех кнопок размещены в объекте-строке, как показано в листинге 10-30. Класс Row является еще одним подклассом класса Container.

Листинг 10-30.

```
const navBar = new Row(null, {
  left: 0, right: 0,
  contents: [
    Content(null, {left: 0, right: 0}),
    settingsButton,
    Content(null, {left: 0, right: 0}),
    weatherButton,
    Content(null, {left: 0, right: 0}),
    timeButton,
    Content(null, {left: 0, right: 0})
  ]
});
```

Объект-строка упорядочивает свое содержимое горизонтально и, как и объект-столбец, относительно друг друга. Левое поле первого элемента в массиве содержимого объекта строки относится к левому краю строки, левое поле второго элемента — к первому элементу и так далее.

Вам, вероятно, интересно, почему в листинге 10.30 есть объекты содержимого. Вот несколько важных вещей, которые следует отметить о них:

- Они не имеют кожи и поэтому прозрачны. Они представляют собой пустое пространство вокруг кнопок в ряду.
- Ширина для них не указывается; вместо этого объект строки вычисляет количество пустого пространства вокруг кнопок.

- Их левое и правое поля (как и у трех кнопок) равны 0; в противном случае поля будут участвовать в вычислениях, которые делает объект строки, что обычно не то, что вам нужно.

Чтобы лучше понять это, давайте сначала посмотрим, каким был бы результат, если бы объекты контента были удалены из строки, оставив только три кнопки. Поскольку все кнопки имеют заданную ширину 80, но не имеют левого или правого поля, размещение их в строке `navBar` по отдельности приводит к их объединению в 240 пикселей в левой части экрана, как показано на рис. 10-13. .

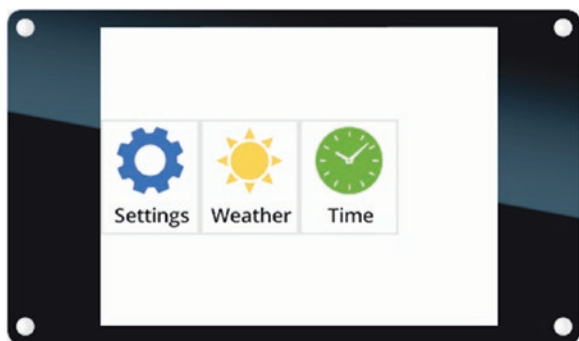


Рис. 10-13. *navBar без объектов содержимого*

Если вы затем зададите каждой кнопке левое поле размером 20, вы получите желаемую компоновку на экране 320 x 240, как показано ранее на рис. 10-12. Но теперь представьте, что используется экран другого размера, скажем, 480 x 320; На рис. 10-14 показан результат в этом случае.

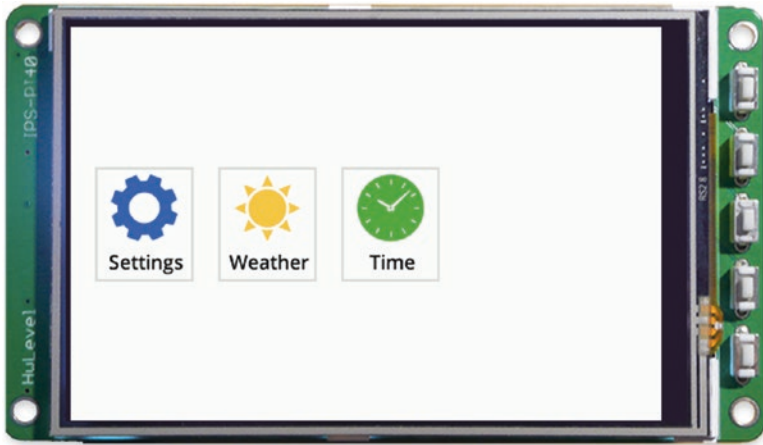


Рис. 10-14. *navBar без объектов содержимого, но с полями, увеличенный экран*

Объекты контента — это то, что делает макет адаптируемым к разным размерам экрана. Поскольку объекты содержимого не имеют ширины, объект строки вычисляет, насколько широкими их сделать, чтобы добиться желаемого макета: он вычисляет ширину, занимаемую тремя кнопками — в данном случае 240 — и остальными кнопками. пиксели, доступные в строке, равномерно распределяются между остальным содержимым (в результате остается одинаковое пространство перед первой кнопкой, между кнопками и после последней). На экране 320 x 240 на рис. 10.12 получается $(320 - 240)/4$, или 20 пикселей на объект контента; на экране 480 x 320 (рис. 10-15) оно составляет 60 пикселей каждый.

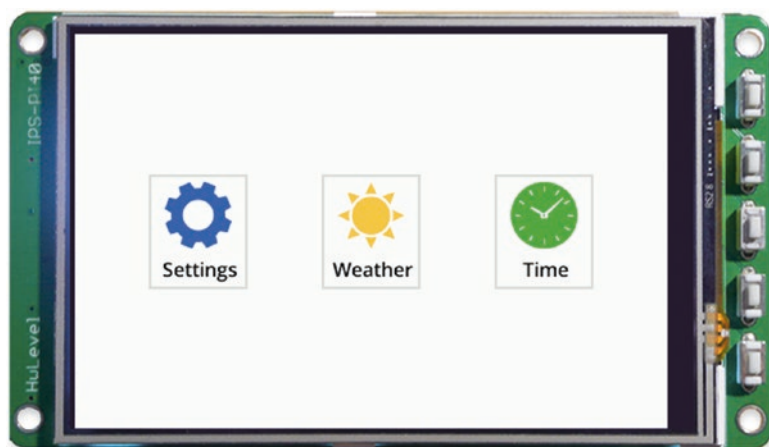


Рис. 10-15. *navBar на большом экране, с правильным центрированием*

Если вы хотите, чтобы кнопки в этом примере находились ровно в 20 пикселях друг от друга, но по-прежнему располагались по центру экрана, вы можете указать ширину 20 для двух средних объектов содержимого. Затем объект строки будет вычислять только количество места, которое нужно поместить перед первой кнопкой и после последней.

Если вы уверены, что размер экрана не изменится и не повернется, добавлять прозрачные объекты контента не нужно; вы можете просто определить левое и правое поля для размещения элементов по желанию. Тем не менее, полезно знать, разрабатываете ли вы дизайн для нескольких размеров экрана.

Прокрутка содержимого

Когда у вас больше контента, чем вы можете разместить на экране одновременно, одним из распространенных решений является использование прокрутки для перемещения по контенту. Пример `$EXAMPLES/ch10-piu/scrolling-text` использует прокрутку для отображения содержимого, которое слишком велико для экрана 320 x 240. На рис. 10-16 показан экран в том виде, в котором он появляется изначально.

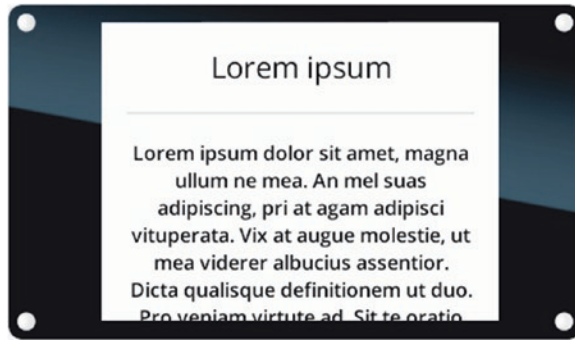


Рис. 10-16. Пример *scrolling-text*

В этом примере прокручивается заголовок, серая полоса и образец текста, определяемый меткой, содержимым и текстовыми объектами соответственно. Эти объекты находятся в контейнере столбцов, располагая их вертикально. Столбец является первым элементом массива содержимого объекта прокрутки, как показано в листинге 10.31.

Листинг 10-31.

```
const sampleVerticalScroller = new Scroller(null, {
  left: 0, right: 0, top: 0, bottom: 0,
  contents: [
    Column(null, {
      left: 0, right: 0, top: 0,
      contents: [
        sampleHeader,
        grayBar,
        sampleText
      ]
    })
  ],
  active: true,
  Behavior: VerticalScrollerBehavior
});
```

Класс `Scroller` расширяет класс `Container` правилом макета, которое прокручивает первый элемент в его массиве содержимого, оставляя другое содержимое (в данном примере отсутствующее) для следования поведению макета контейнера по умолчанию. Класс `Scroller` может прокручивать по горизонтали, вертикали или по обоим направлениям; этот пример прокручивается вертикально. То, как прокручивается объект прокрутки, определяется его поведением.

Поведение `VerticalScrollerBehavior` в этом примере (листинг 10-32) использует сенсорный ввод для управления прокруткой. Когда вы касаетесь экрана и проводите пальцем вверх или вниз, скроллер перемещает содержимое вверх или вниз. Событие `onTouchMoved` — это низкоуровневое событие, которое запускается при перемещении пальца по экрану. Объект содержимого может получать множество событий `onTouchMoved` после события `onTouchBegan` (и до события `onTouchEnded`, если таковое имеется).

Листинг 10-32.

```
class VerticalScrollerBehavior extends Behavior {
    onTouchBegan(scroller, id, x, y, ticks) {
        this.initialScrollY = scroller.scroll.y;
        this.initialY = y;
        scroller.captureTouch(id, x, y, ticks);
    }
    onTouchMoved(scroller, id, x, y, ticks) {
        const dy = y - this.initialY;
        scroller.scrollTo(0, this.initialScrollY - dy);
    }
}
```

Обратите внимание на следующее в этом коде:

- Метод `onTouchBegan` вызывает метод `captureTouch` объекта скроллера, который не позволяет другим объектам содержимого инициировать события касания, связанные с касанием. Здесь в этом нет необходимости, потому что нет других активных объектов содержимого для получения событий касания, но это включено, потому что это делает поведение более пригодным для повторного использования.

- Метод `onTouchMoved` вызывает метод `scrollTo` объекта прокрутки для прокрутки содержимого по вертикали на основе движений пальцев. Лучше использовать `scrollTo`, чем менять координаты контента; `scrollTo` предотвращает перемещение контента за пределы экрана, поэтому вам не нужно писать дополнительный код, чтобы избежать этого.
- Метод `onTouchEnded` отсутствует, поскольку в конце касания не обеспечивается обратная связь.

Шаблоны для объектов содержимого

Пользовательские интерфейсы часто используют один и тот же элемент, иногда с небольшими вариациями, во многих местах. Например, каждый экран приложения может использовать заголовок с одним и тем же значком, но с другим текстом, или каждая кнопка на панели навигации может иметь разные значок и текст, как в примере с навигационной панелью. Для создания каждой из трех кнопок в примере с панелью навигации использовался практически один и тот же код. Шаблоны Piu — более лаконичный и эффективный способ добиться того же результата. Шаблон — это класс, который вы создаете с помощью метода шаблона объекта содержимого. Возможность создавать подобные классы во время выполнения — мощная функция JavaScript, на которой построен Piu.

Создание класса шаблона кнопки

Напомним, что в примере с панелью навигации создается ряд кнопок с использованием трех объектов столбца, каждый из которых содержит объект содержимого (для значка) и объект этикетки. Эти объекты столбца отличаются только свойством `skin` объекта содержимого и строковым свойством объекта метки. Кнопки размещаются в объекте строки вместе с невидимыми объектами содержимого, чтобы макет реагировал на различные размеры экрана.

Написание почти одинакового кода для создания каждой из трех кнопок может показаться неразумным, но представьте, что вы хотите создать десять кнопок: у вас будет более сотни строк кода, которые выглядят одинаково. И если вы затем решите сделать каждую из этих кнопок на несколько пикселей шире, было бы утомительно и подвержено ошибкам изменять каждое свойство ширины по отдельности.

Пример \$EXAMPLES/ch10-piu/nav-bar-template создает класс Button для кнопок панели навигации. Для этого вызывается метод статического шаблона класса Column, как показано в листинге 10.33.

Листинг 10-33.

```
const Button = Column.template($ => ({
  skin: outlineSkin,
  width: 80,
  contents: [
    Content(null, {
      top: 5,
      skin: $.skin
    }),
    Label(null, {
      top: 0,
      style: textStyle,
      string: $.string
    })
  ]
}));
```

Вызванный здесь метод шаблона создает и возвращает конструктор для класса Button. Этот новый класс расширяет Column, поскольку метод шаблона является частью класса Column. Все объекты содержимого Piu имеют метод статического шаблона.

Несмотря на то, что класс Button не создается с использованием ключевого слова class, вы все равно создаете экземпляры с использованием нового ключевого слова, как в новой кнопке. Прежде чем приступить к использованию класса Button, давайте рассмотрим реализацию класса.

Единственным аргументом `Column.template` является функция, возвращающая объект. Синтаксис немного необычен в том смысле, что тело стрелочной функции представляет собой значение, а не серию операторов; это значение становится возвращаемым значением функции. Чтобы проиллюстрировать это на простом примере, следующий код определяет стрелочную функцию с именем `test`:

```
let test = () => ({one: 1});
test();    // возврат {one: 1}
```

Когда вызывается стрелочная функция, она возвращает объект. Теперь рассмотрим этот пример, определяющий версию `test`, принимающую один аргумент:

```
let test = $ => ({one: $});
test(1); // возврат {one: 1}
```

Аргумент стрелочной функции присваивается переменной с именем `$`. Хотя `$` — необычное имя переменной, оно допустимо в JavaScript, и переменная `$` ведет себя так же, как и любая другая. (Обратите внимание, что это не связано с `$`, используемым в литералах шаблонов для подстановки строк, как описано в главе 2).

Аналогично, в реализации класса `Button`, показанной в листинге 10.33, аргументом `Column.template` является анонимная стрелочная функция, которая возвращает объект, для которого некоторые значения свойств берутся из переданной переменной `$`. Когда вы вызываете конструктор `Button`, как показано для `settingsButton` в следующем коде, вы передаете словарь, содержащий свойства для замены в шаблоне, где используется переменная `$`, здесь заменяя `$.skin` и `$.string`; конструктор вызывает стрелочную функцию, указанную в его реализации, передавая показанный здесь словарь в качестве аргумента `$`:

```
const settingsButton = new Button({
  skin: settingsSkin,
  string: "Settings"
});
```

При использовании шаблона `Button` каждая дополнительная кнопка создается кратким вызовом конструктора `Button`, как показано в листинге 10-34. (Остальной код, создающий панель навигации, такой же, как и в примере с панелью навигации.)

Листинг 10-34.

```
const weatherButton = new Button({
    skin: sunSkin,
    string: "Weather"
});

const timeButton = new Button({
    skin: clockSkin,
    string: "Time"
});
```

Как видите, определение класса шаблона имеет следующие преимущества:

- Это значительно улучшает читаемость кода за счет устранения избыточного кода для определения каждой кнопки (что также экономит флэш-память).
- Это упрощает поддержку вашего кода. Чтобы изменить общее свойство, такое как ширина каждого столбца, все, что вам нужно сделать, это изменить это свойство шаблона.

Аргументы конструктора содержимого

Вы, наверное, заметили, что вызовы конструктора `Button` в примере с `thenav-bar-template` выглядят иначе, чем конструкторы объектов содержимого в более ранних примерах: вызовы конструктора `Button` передают словарь в качестве первого аргумента, а не `null`, и опускают второй аргумент, который, если присутствует, является словарем, который настраивает объект. В этом разделе более подробно рассматриваются эти два аргумента, которые принимает каждый конструктор содержимого `Piu`, и объясняются эти различия.

Аргумент создания экземпляра данных

Первый аргумент, который принимает конструктор содержимого, называется данными создания экземпляра. Эта концепция наиболее актуальна при работе с шаблонами. Например, созданный ранее класс шаблона `Button` использует данные, переданные в качестве первого аргумента, для создания словаря, из которого создается экземпляр класса (другими словами, словарь обычно передается в качестве второго аргумента).

Данные создания экземпляра могут быть любым значением или объектом JavaScript. Класс, который вы создаете, определяет, какие данные являются допустимыми. Например, шаблон класса `Button`, как определено в листинге 10-33, предполагает, что данные создания экземпляра будут объектом со свойствами оболочки и строки. Альтернативная реализация класса `Button` показана в листинге 10-35.

Листинг 10-35.

```
const Button = Column.template($ => ({
  skin: outlineSkin,
  width: 80,
  contents: [
    Label(null, {
      top: 0,
      style: textStyle,
      string: $
    })
  ]
}));
```

Этот класс `Button` не имеет значка и ожидает, что в качестве аргумента `$` будет передана строка, как в этих примерах:

```
const weatherButton = new Button("Weather");
const timeButton = new Button("Time");
```

Данные создания экземпляра имеют еще одно интересное свойство: они передаются в метод `onCreate` поведения созданного экземпляра. Например, в листинге 10-36 показан другой способ реализации класса `Button` из листинга 10-35.

Листинг 10-36.

```
const Button = Column.template($ => ({
  skin: outlineSkin,
  width: 80,
  contents: [
    Label(null, {
      top: 0,
      style: textStyle
    })
  ],
  Behavior: class extends Behavior {
    onCreate(column, $) {
      column.first.string = $;
    }
  }
}));
```

Функциональность создания экземпляров данных не ограничивается шаблонами; ее могут использовать любые конструкторы объектов контента. Например, в листинге 10-37 создается метка со строкой «Hello, World».

Листинг 10-37.

```

const sampleLabel = new Label("Hello, World", {
    top: 0, bottom: 0, left: 0, right: 0,
    style: textStyle,
    Behavior: class extends Behavior {
        onCreate(label, data) {
            label.string = data;
        }
    }
});

```

Далее в этой главе вы познакомитесь с расширенным использованием аргумента создания экземпляра данных: определите привязки к содержимому.

Аргумент словаря содержимого

Второй аргумент конструктора содержимого — это словарь, определяющий свойства созданного экземпляра. Свойства, которые вы включаете в этот словарь содержимого, связаны со встроенными свойствами класса содержимого, экземпляр которого вы создаете, например, с обложкой или шириной экземпляра. За исключением приведенных ранее примеров шаблона кнопки, все примеры в этой главе определяют аргумент словаря содержимого; однако это необязательно и по умолчанию не определено.

В листинге 10-37 продемонстрировано использование как аргумента создания экземпляра данных, так и аргумента словаря содержимого для создания объекта метки. В примере \$EXAMPLES/ch10-piu/colored-squares показано, как использовать оба этих аргумента при вызове конструктора шаблона для создания цветных квадратов, показанных на рис. 10.17.

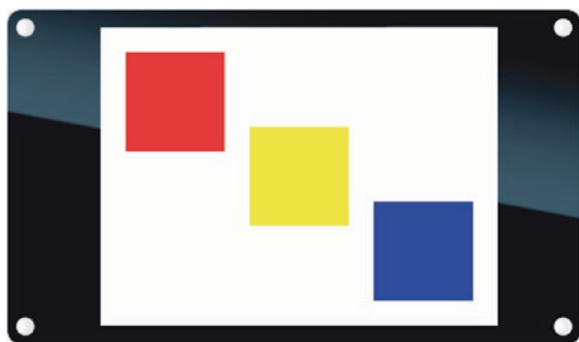


Рис. 10-17. Пример *colored-squares*

В листинге 10-38 показан код для создания шаблона и построения квадратов.

Листинг 10-38.

```
const Square = Content.template($ => ({
  width: 80, height: 80,
  skin: new Skin({fill: $})
}));

const redSquare = new Square("red", {left: 20, top: 20});
const yellowSquare = new Square("yellow");
const blueSquare = new Square("blue", {right: 20, bottom: 20});
```

В этом примере данные создания экземпляра представляют собой строку, определяющую цвет заливки квадрата. Положение красного и синего квадратов определяется вторым аргументом словаря, в то время как желтый квадрат пропускает второй аргумент и, следовательно, по умолчанию центрируется в своем родительском контейнере.

Доступ к объектам содержимого в контейнере

В примерах, которые вы видели до сих пор, вы обращались к объектам контента через локальные переменные, но вы не видели, как получить доступ к объектам контента, когда у вас нет ссылки на них в локальной переменной. Во многих ситуациях вам может потребоваться доступ к объектам непосредственно из иерархии вложений, например, при работе с составными объектами, которые вы создали с помощью шаблона.

Вы уже узнали, что объект-контейнер содержит список дочерних объектов, количество которых можно узнать из свойства длины контейнера. В следующих разделах представлены несколько методов доступа к объектам контента в иерархии включения.

Использование первого, последнего, следующего и предыдущего

Вы можете использовать первое свойство контейнера для получения его первого дочернего элемента, а последнее свойство — для получения последнего дочернего элемента. Если у контейнера нет дочерних объектов, `first` и `last` равны нулю.

У каждого объекта содержимого есть свойство `next`, которое можно использовать для доступа к следующему объекту содержимого в его контейнере, или `null`, если его нет. Аналогично, предыдущее свойство возвращает предыдущий объект содержимого (или `null`).

Использование этих свойств — это простой способ доступа к содержимому контейнера. Они хорошо работают в некоторых ситуациях, но не во всех. Например, код для доступа к четвертому дочернему элементу контейнера с именем `myContainer` с использованием `first` и `next` трудно читать и утомительно писать.

```
let button = myContainer.first.next.next.next;
```

В следующем разделе представлено лучшее решение для таких ситуаций.

Доступ к дочерним элементам по индексу и имени

Метод `content` обеспечивает доступ к дочерним объектам контейнера по индексу. Значения индекса начинаются с 0, поэтому вы можете получить доступ к третьему дочернему элементу в контейнере с именем `myContainer` следующим образом:

```
myContainer.content(2);
```

Подобно первым, последним, следующим и предыдущим свойствам, этот метод доступа к дочерним объектам прост, но требует изменения кода при изменении порядка содержимого в контейнере. Кроме того, вы можете использовать метод `content` для доступа к дочерним объектам по их имени. Вы определяете свойство имени для объекта содержимого в словаре, переданном в конструктор.

```
let myContent = new Content(null, {  
  name: "foo"  
});
```

Если `myContent` является дочерним элементом `myContainer`, вы можете получить к нему доступ следующим образом:

```
let foo = myContainer.content("foo");
```

Этот метод хорошо работает для многих иерархий включения, но обратите внимание, что объект содержимого должен быть прямым потомком контейнера, чтобы он работал. Вы не можете использовать метод `content` для доступа к внукам, правнукам и т. д. контейнера.

Доступ к контенту с якорями

Якорь — это ссылка на объект контента, сохраненный как свойство в данных создания экземпляра объекта контента. Якоря — лучший метод доступа к содержимому в сложных интерфейсах с множеством уровней в их иерархии; однако их труднее всего понять. Попытка объяснить якоря концептуально часто скорее сбивает с толку, чем помогает, поэтому давайте сразу рассмотрим их на примере.

Пример \$EXAMPLES/ch10-piu/anchors демонстрирует базовое использование привязок для создания анимированного пользовательского интерфейса. Когда вы нажимаете кнопку «Start - Пуск», фон и цветной квадрат мигают двумя разными цветами. На рис. 10-18 показаны два состояния, между которыми экран переключается при нажатии кнопки «Start - Пуск».



Рис. 10-18. Пример *anchors*

Этот интерфейс состоит из трех объектов контента:

- Кнопка «Start» (экземпляр класса `StartButton`)
- Цветной квадрат (экземпляр класса `AnimatedSquare`)
- Фоновый объект (экземпляр класса `MainContainer`), который заполняет фон цветом и, как показано в листинге 10-39, содержит кнопку «Start» и цветной квадрат.

Листинг 10-39.

```
const MainContainer = Container.template($ => ({
  ...
  contents: [
    new StartButton($),
    new AnimatedSquare($),
  ],
  ...
}));
```

Обратите внимание, что всем трем объектам передаются одни и те же данные создания экземпляра через переменную \$. В этом примере данные создания экземпляра начинаются как пустой словарь.

```
let instantiatingData = {};
application.add(new MainContainer(instantiatingData));
```

Поведение цветных квадратных и фоновых объектов меняет цвет заливки дважды в секунду, когда работают их внутренние часы, то есть когда вызывается метод start каждого объекта и объект начинает получать события onTimeChanged. Кнопка «Start» отвечает за вызов метода запуска этих объектов при нажатии; цветной квадрат и фоновый объект создают якоря, чтобы кнопка «Start» могла ссылаться на них для этого.

Чтобы создать привязку для объекта содержимого, вы указываете свойство привязки в словаре, переданном его конструктору. Шаблон MainContainer задает для свойства привязки строку «BACKGROUND», как показано в листинге 10-40.

Листинг 10-40.

```
const MainContainer = Container.template($ => ({
  ...
  anchor: "BACKGROUND",
  ...
}));
```

Аналогично, шаблон AnimatedSquare задает для свойства привязки строку «SQUARE - КВАДРАТ» (листинг 10.41).

Листинг 10-41.

```
const AnimatedSquare = Content.template($ => ({
  ...
  anchor: "SQUARE",
  ...
})));
```

Когда создается экземпляр объекта контента со свойством привязки, `Piu` присваивает экземпляру свойству с именем привязки в данных создания экземпляра. Вспомните, что `InstantiatingData` начинался как пустой словарь; если вы используете якоря, данные создания экземпляра должны быть словарем, чтобы к нему можно было добавить якоря. После создания экземпляров цветного квадрата и фоновых объектов `instanceiatingData` выглядит следующим образом:

```
{
  BACKGROUND: <reference to the background object>,
  SQUARE: <reference to the colored square object>
}
```

Свойства `BACKGROUND` и `SQUARE` в `instanceiatingData` являются привязками к фону и цветным квадратным объектам. Все, у кого есть доступ к `instanceiatingData`, могут использовать эти якоря для ссылки на эти объекты. В этом примере кнопка «Start» использует якоря для запуска анимации фона и квадрата. Весь код, использующий якоря и запускающий анимацию, содержится в поведении шаблона `StartButton`.

Как вы знаете, данные создания экземпляра, переданные конструктору объекта контента, передаются в метод `onCreate` поведения созданного контента. `StartButtonBehavior` сохраняет ссылку на данные создания экземпляра в свойстве данных, чтобы ее можно было использовать в других методах.

```
class StartButtonBehavior extends Behavior {
    onCreate(label, data) {
        this.data = data;
    }
}
```

Затем `StartButtonBehavior` использует свое свойство данных в своем методе `onTouchEnded` (листинг 10-42) для доступа к привязкам к фону и цветному квадрату, чтобы вызвать их методы запуска, что, в свою очередь, вызывает запуск анимации.

Листинг 10-42.

```
onTouchEnded(label) {
    ...
    this.data.SQUARE.start();
    this.data.BACKGROUND.start();
}
```

Обратите внимание, что при использовании привязок уровень объектов содержимого в иерархии вложений не имеет значения. В этом примере кнопка «Start» и цветной квадрат являются дочерними объектами фоновой объекта, но вы можете изменить иерархию вложений — например, вы можете сделать цветной квадрат дочерним элементом объекта приложения — без необходимости изменять реализацию `StartButtonBehavior` на запуск анимации. Эта гибкость делает якоря очень полезными при создании иерархий сдерживания, которые могут меняться.

Определение и запуск собственных событий

Вы видели несколько примеров с объектами поведения, которые реагируют на низкоуровневые события, определенные и запускаемые `Piu`. Вашим приложениям могут потребоваться другие события, события высокого уровня, не определенные `Piu`; например, продукт с подключенным датчиком может инициировать событие

`onSensorValueChanged`, когда датчик обнаруживает изменение, чтобы приложение могло обновить дисплей или сообщить об изменении сетевой службе. Чтобы обрабатывать события высокого уровня, вы добавляете методы к своему поведению точно так же, как вы делаете это для событий низкого уровня.

Часто несколько объектов контента должны реагировать на одно событие. Например, при изменении значения датчика может потребоваться обновление нескольких элементов пользовательского интерфейса. Ваш обработчик событий для одного объекта может распространять события — полученные им события или события, которые он создает, — на другие объекты по всей иерархии включения. `Piu` предоставляет методы делегата, распределения и всплывающих окон для распространения событий.

В этом разделе показано, как определять и запускать собственные события. Он также представляет методы для распространения событий на один или несколько объектов содержимого в иерархии включения.

Запуск событий в объекте содержимого

Пример `$EXAMPLES/ch10-piu/counter` хранит счетчик в объекте метки и позволяет другому объекту, в данном случае кнопке, увеличивать счетчик с помощью высокоуровневого события увеличения. На рис. 10-19 показаны шаги в примере: счетчик начинается с 0, пользователь касается кнопки и, наконец, счетчик увеличивается до 1, когда прикосновение заканчивается.



Рис. 10-19. Пример counter

Как показано в листинге 10-43, счетчик представляет собой объект метки с поведением, называемым `CounterBehavior`.

Листинг 10-43.

```
const counter = new Label(null, {
    top: 70, height: 30, left: 0, right: 0,
    style: textStyle,
    string: "0",
    Behavior: CounterBehavior
});
```

Счетчик хранится в свойстве `count` поведения метки (листинг 10-44) и инициализируется в 0 обработчиком события `onDisplaying` из `CounterBehavior`. Поведение также реализует обработчик события приращения, который увеличивает счетчик объекта метки и обновляет его строковое свойство новым значением.

Листинг 10-44.

```
class CounterBehavior extends Behavior {
    onDisplaying(label) {
        this.count = 0;
    }
    increment(label) {
        label.string = ++this.count;
    }
}
```

Объект `incrementButton` (листинг 10-45) также является объектом метки с поведением, называемым `IncrementButtonBehavior`.

Листинг 10-45.

```
const incrementButton = new Label(null, {
    top: 120, height: 40, left: 140, width: 40,
    style: textStyle,
```

```

    string: "+",
    skin: buttonSkin,
    active: true,
    Behavior: IncrementButtonBehavior
  });

```

Когда кнопка нажата, `IncrementButtonBehavior` (листинг 10-46) обеспечивает обратную связь, изменяя свойство состояния кнопки в методах `onTouchBegan` и `onTouchEnded`. Метод `onTouchEnded` также делегирует событие приращения объекту счетчика. Метод делегата объекта содержимого немедленно запускает событие, указанное в первом аргументе метода. Здесь событие приращения запускается для объекта счетчика.

Листинг 10-46.

```

class IncrementButtonBehavior extends Behavior {
  onTouchBegan(label) {
    label.state = 1;
  }
  onTouchEnded(label) {
    label.state = 0;
    counter.delegate("increment");
  }
}

```

Вы можете передать дополнительные аргументы обработчику событий, передав их методу делегата после имени события; например, событие `onSensorValueChanged` может получить новое показание датчика как часть события. Чтобы изменить пример счетчика для увеличения на любое число, вы можете изменить метод увеличения в листинге 10-44, чтобы он принимал дополнительный аргумент, определяющий величину увеличения, как показано в листинге 10-47.

Листинг 10-47.

```
class CounterBehavior extends Behavior {
    ...
    increment(label, delta) {
        this.count += delta;
        label.string = this.count;
    }
}
```

Затем вы должны передать число методу делегата в методе `onTouchEnded`. Например:

```
counter.delegate("increment", 1); // increments by 1
counter.delegate("increment", 5); // increments by 5
```

Распространение событий внутри контейнера

Пример `$EXAMPLES/ch10-piu/color-scheme` предоставляет кнопки для изменения внешнего вида приложения между **Light** (светлым) и **Dark** (темным) режимами. Когда пользователь нажимает кнопку «Light» или «Dark», кнопка вызывает событие для всех объектов внутри контейнера приложения. Объекты отвечают, обновляя свои цвета до запрошенного режима. На рис. 10-20 показан интерфейс, запускающийся в режиме **Light**, кнопка «Dark» при нажатии и интерфейс в режиме **Dark**. Текст, отображаемый над кнопками, указывает на текущий режим.



Рис. 10-20. Пример *color-scheme*

Кнопки **Light** и **Dark** вызывают событие с именем `onModeChanged`. Каждая кнопка является экземпляром `ModeButton`, шаблона, основанного на `Label`, как показано в листинге 10-48.

Листинг 10-48.

```
const ModeButton = Label.template($ => ({
  top: 110, height: 40, width: 120,
  skin: buttonSkin,
  active: true,
  Behavior: ModeButtonBehavior
}));
```

`ModeButtonBehavior` (листинг 10-49) обеспечивает обратную связь при нажатии кнопки путем изменения свойства состояния кнопки в методах `onTouchBegan` и `onTouchEnded`. Метод `onTouchEnded` также распространяет событие `onModeChanged` по всему контейнеру приложения, вызывая метод распределения объекта приложения. Метод распределения инициирует событие для каждого объекта содержимого в контейнере. В своем вызове `application.distribute ModeButtonBehavior` передает имя кнопки, в данном примере «Light» или «Dark», в качестве аргумента для указания режима, на который следует перейти.

Листинг 10-49.

```
class ModeButtonBehavior extends Behavior {
  onTouchBegan(label) {
    label.state = 1;
  }
  onTouchEnded(label) {
    label.state = 0;
    application.distribute("onModeChanged", label.string);
  }
}
```

Все объекты-контейнеры имеют метод распределения, который иницирует указанное событие в контейнере и во всех объектах содержимого ниже по иерархии включения. Распространение события завершается, когда событие доставляется всем объектам в контейнере или когда один из обработчиков событий возвращает значение `true`, чтобы указать, что событие было полностью обработано. Вы можете думать о методе распределения как о способе передачи события содержимому контейнера. В этом примере было бы легко напрямую вызвать делегат для нескольких объектов содержимого с обработчиком `onModeChanged` в их поведении; однако по мере того, как ваше приложение становится более сложным, проще использовать метод распределения для автоматического обхода всего в контейнере.

Теперь, когда вы знаете, как распределять события триггеров по содержимому контейнеров, давайте посмотрим, как объекты содержимого реагируют на эти события. Государственная собственность играет ключевую роль. Контейнер `LightDarkScreen`, который содержит кнопки и строку текста, имеет скин, который становится белым, когда его свойство состояния равно 0, и черным, когда его свойство состояния равно 1.

```
const backgroundSkin = new Skin({
    fill: ["white", "black"]
});
```

Строка текста — это объект метки со стилем, который будет делать обратное: текст будет черным, если его свойство состояния равно 0, и белым, если его свойство состояния будет равно 1. (См. листинг 10.50.)

Листинг 10-50.

```
const textStyle = new Style({
    font: "24px Open Sans",
    color: ["black", "white"],
    top: 10, bottom: 10, left: 10, right: 10
});
```


Код `LightDarkScreen` показан в листинге 10-51.

Листинг 10-51.

```
const LightDarkScreen = new Container(null, {
  top: 0, bottom: 0, left: 0, right: 0,
  skin: backgroundSkin,
  style: textStyle,
  contents: [
    Label(null, {
      top: 50, height: 30, left: 0, right: 0,
      string: "Light",
      Behavior: TextBehavior
    }),
    ModeButton(null, {
      left: 30,
      string: "Dark"
    }),
    ModeButton(null, {
      right: 30,
      string: "Light"
    })
  ],
  Behavior: LightDarkScreenBehavior
});
```

Как `LightDarkScreen`, так и содержащийся в нем объект `label` имеют поведение, которое меняет свое свойство состояния при получении события `onModeChanged`. Метка изменяет свое строковое свойство, чтобы отразить, какая кнопка была нажата. В листинге 10-52 показано это поведение.

Листинг 10-52.

```

class LightDarkScreenBehavior extends Behavior {
    onModeChanged(container, mode) {
        container.state = (mode === "Dark")? 1 : 0;
    }
}

class TextBehavior extends Behavior {
    onModeChanged(label, mode) {
        label.state = (mode === "Dark")? 1 : 0;
        label.string = mode;
    }
}

```

Всплывающие события вверх по иерархии сдерживания

Пример \$EXAMPLES/ch10-piu/background-color предоставляет кнопки для изменения цвета фона экрана. Когда пользователь нажимает кнопки, они запускают событие вверх по иерархии сдерживания. Родительский контейнер кнопок занимает весь экран и обновляет свойство кожи в ответ на событие. На рис. 10.21 показан фон в исходном белом состоянии, желтая кнопка при нажатии, а затем фон после того, как он стал желтым.



Рис. 10-21. Пример *background-color*

Как показано в листинге 10-53, кнопки создаются с помощью шаблона, который создает объект метки с поведением, называемым `ColorButtonBehavior`.

Листинг 10-53.

```
const ColorButton = Label.template($ => ({
  height: 40, left: 10, right: 10,
  skin: buttonSkin,
  active: true,
  Behavior: ColorButtonBehavior
}));
```

ColorButtonBehavior (листинг 10-54) обеспечивает обратную связь при нажатии кнопки путем изменения свойства состояния кнопки в методах `onTouchBegan` и `onTouchEnded`. Метод `onTouchEnded` также поднимает событие `onColorSelected` вверх по иерархии включения, вызывая метод пузырька и передавая ему строковое свойство кнопки — «Желтый», «Красный» или «Синий» — в качестве аргумента обработчику событий.

Листинг 10-54.

```
class ColorButtonBehavior extends Behavior {
  onTouchBegan(label) {
    label.state = 1;
  }
  onTouchEnded(label) {
    label.state = 0;
    label.bubble("onColorSelected", label.string);
  }
}
```

Все объекты содержимого имеют метод пузырька, который заставляет их, их родительский контейнер и все объекты-контейнеры выше в иерархии включения инициировать указанное событие. Распространение события заканчивается, когда событие доставлено всем объектам, вплоть до объекта приложения, или когда один из обработчиков событий возвращает значение `true`, чтобы указать, что событие было полностью обработано. Как и в случае методов делегата и распределения, событие указывается по имени и передается в качестве первого аргумента методу пузырька.

Теперь, когда вы знаете, как использовать метод пузырьков для запуска событий, давайте посмотрим, как организована иерархия включения в примере, прежде чем изучать подробности того, как событие `onColorSelected` распространяется через эту конкретную иерархию включения.

Кнопки содержатся в объекте строки. Эта строка является частью объекта-контейнера с именем `colorScreen`, который добавляется к объекту приложения. Как показано в листинге 10.55, у строки нет связанного с ней поведения, но `colorScreen` ссылается на поведение с именем `ColorScreenBehavior`.

Листинг 10-55.

```
const colorScreen = new Container(null, {
  top: 0, bottom: 0, left: 0, right: 0,
  skin: whiteSkin,
  style: textStyle,
  contents: [
    Row(null, {
      height: 50, width: 320,
      contents: [
        new ColorButton(null, {string: "Red"}),
        new ColorButton(null, {string: "Yellow"}),
        new ColorButton(null, {string: "Blue"})
      ]
    })
  ],
  Behavior: ColorScreenBehavior
});

application.add(colorScreen);
```

ColorScreenBehavior меняет цвет фона при получении события onColorSelected; как показано в листинге 10-56, новый цвет передается в качестве аргумента. Первая буква каждой строки кнопки — прописная («Красный»), но все цвета CSS — строчные, поэтому обработчик событий использует toLowerCase для преобразования строки во все буквы нижнего регистра.

Листинг 10-56.

```
class ColorScreenBehavior extends Behavior {
    onColorSelected(container, color) {
        container.skin = new Skin({
            fill: color.toLowerCase()
        });
    }
}
```

Вот что происходит при нажатии одной из кнопок:

2. Событие onColorSelected сначала запускается на самой кнопке. Поведение кнопки не имеет соответствующего метода onColorSelected, поэтому событие всплывает в своем родительском контейнере. Родительским контейнером кнопки является объект строки. Этот объект не имеет поведения и, следовательно, метод onColorSelected, поэтому событие перемещается в родительский контейнер строки.
4. Родительским контейнером строки является colorScreencontainer. Поведение этого контейнера имеет метод onColorSelected, поэтому метод вызывается, когда поведение запускает событие onColorSelected. Затем событие переходит к родительскому контейнеру этого контейнера.

4. Родительским контейнером контейнера `colorScreen` является объект приложения. Этот объект имеет метод `noonColorSelected` и является корнем иерархии вложений, поэтому обход завершен.

Как и в других примерах распространения событий, было бы легко просто делегировать событие всему содержимому, в поведении которого есть соответствующий метод `onColorSelected`. Но приложения со многими уровнями в своей иерархии включения могут использовать метод пузырьков объектов содержимого, чтобы упростить код, распространяющий событие, и свести к минимуму изменения кода, необходимые при изменении иерархии включения.

Анимация

Включение анимации в пользовательские интерфейсы может значительно улучшить взаимодействие с пользователем. Анимации используются для осмысленных функциональных целей, например, для обеспечения обратной связи, когда пользователь нажимает кнопку. Они также используются в эстетических целях, чтобы придать продукту особый вид — например, для создания анимированного перехода при переходе между экранами.

Уравнения смягчения

Анимации, которые линейно изменяют свойства объектов контента, часто выглядят неестественно. Уравнения плавности — распространенный инструмент для реализации анимации, которая кажется более естественной, или для добавления визуального стиля.

`Piu` расширяет объект JavaScript `Math` известными уравнениями плавности Роберта Пеннера. Названия этих функций в `Piu` говорят сами за себя — например, `bounceEaseInOut` создает эффект подпрыгивания в начале и в конце анимации. Подробная информация об уравнениях Пеннера доступна на сайте robertpenner.com/easing/.

Все реализации `Piu` этих уравнений замедления принимают один аргумент, число в диапазоне $[0, 1]$, и возвращают число в диапазоне $[0, 1]$ с примененной функцией замедления. Уравнения широко используются во всех типах анимации. Входное значение — это часть завершенной анимации; функция смягчения настраивает дробь на другое значение, которое затем используется для вычисления состояния значений в анимации. Вы увидите примеры этого в следующих разделах.

Некоторые уравнения замедления создают тонкий эффект, чтобы сделать анимацию более естественной. Например, функции плавности четырехугольника — `Math.quadEaseIn`, `Math.quadEaseOut` и `Math.quadEaseInOut` — слегка изменяют скорость на протяжении всей анимации, чтобы сделать начало и/или конец анимации менее резкими. Другие создают смелый эффект. Например, функции смягчения отскока — `Math.bounceEaseIn`, `Math.bounceEaseOut` и `Math.bounceEaseInOut` — заставляют объекты подпрыгивать в начале и/или в конце анимации.

Конечно, вы не ограничены функциями плавности, включенными по умолчанию; вы можете легко добавить свои собственные уравнения замедления в соответствии с потребностями вашего продукта. Детали создания собственных уравнений плавности выходят за рамки этой книги, но в Интернете есть много информации, если вы решите, что это необходимо для вашего продукта.

Анимация объектов контента

Пример с галочкой `helloworld` показал, как использовать встроенные часы объекта содержимого для создания простой анимации. Создание более сложных анимаций, особенно тех, которые независимо перемещают несколько элементов интерфейса одновременно, затруднено.

В примере `$EXAMPLES/ch10-piu/timeline` показано, как создать последовательность анимации, включающую несколько объектов на экране. Анимация в этом примере проста, но понимание кода даст вам основу для создания собственных гораздо более сложных анимаций. На рис. 10-22 показан пользовательский интерфейс в несколько моментов анимации.

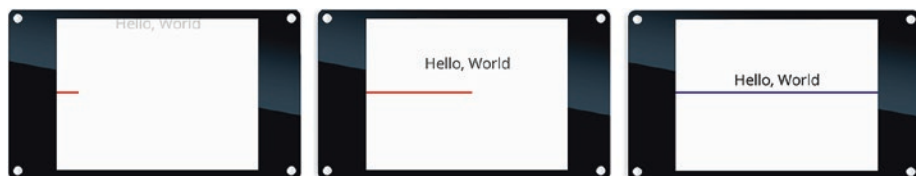


Рис. 10-22. Пример *timeline*

Интерфейс в примере состоит из объекта-контейнера с именем `animatedContainer` (листинг 10-57), который содержит объект метки и объект содержимого.

Листинг 10-57.

```
const animatedContainer = new Container(null, {
  top: 0, bottom: 0, left: 0, right: 0,
  skin: whiteSkin,
  contents: [
    new Label(null, {
      style: textStyle,
      top: 80, left: 0, right: 0,
      string: "Hello, World"
    }),
    new Content(null, {
      top: 115, height: 3, left: 0, width: 320,
      skin: colorfulSkin
    })
  ],
  Behavior: TimelineBehavior
});
```

Анимация управляется `TimelineBehavior`, поведением `animatedContainer`. `TimelineBehavior` создает экземпляры объекта временной шкалы в своем обработчике событий `onDisplaying`. `Piu` предоставляет класс `Timeline` для упрощения и структурирования кода для реализации анимации.

Этот класс можно использовать как для анимации элементов внутри одного экрана, так и для анимации переходов между экранами. Использование класса `Timeline`, как правило, является лучшим способом организации и реализации анимации нескольких объектов содержимого; например, он легко справляется с ситуацией, когда время начала анимации каждого объекта содержимого сдвинуто. API для класса `Piu Timeline` основан на API для `TimelineLite` от `GreenSock`, популярной библиотеки JavaScript, используемой для анимации веб-страниц.

Обработчик события `onDisplaying` также инициализирует свойство `reverse`, которое используется, чтобы анимация временной шкалы выполнялась как вперед, так и назад. В листинге 10-58 показан соответствующий код.

Листинг 10-58.

```
class TimelineBehavior extends Behavior {
  onDisplaying(container) {
    let timeline = this.timeline = new Timeline();
    this.reverse = false;
    ...
  }
}
```

Объект временной шкалы состоит из набора твинов, каждый из которых описывает, как одно или несколько свойств одного объекта содержимого изменяются от начального значения до конечного значения. Анимации добавляются на временную шкалу с помощью методов `from` и `to`, которые определяют анимацию на основе следующих аргументов:

1. `target` – объект контента для анимации
2. `properties` – словарь, ключи которого являются свойствами целевого объекта для анимации
3. `duration` – продолжительность анимации движения в миллисекундах.
4. `easing` – (необязательно) функция плавности для анимации движения.
5. `delay` – (необязательно) количество миллисекунд, через которое эта анимация должна начаться после завершения предыдущей анимации на временной шкале; по умолчанию 0

Анимация движения, добавленная методом `from` временной шкалы, называемая `from-tween`, упрощает свойства целевого объекта от значений, указанных в объекте свойств, до исходных значений целевого объекта в течение миллисекунд. Метод `onDisplaying` в листинге 10-58 продолжается добавлением следующего элемента `from-tween`. В этом примере объект метки перемещается из положения `y` за пределами верхней части экрана в исходное положение в 80 пикселях от верхней части экрана. В то же время его состояние анимируется из состояния 1 в состояние 0, заставляя его переходить от белого к черному. Обратите внимание, что доступ к метке здесь осуществляется как `container.first`, потому что это первый объект содержимого, добавленный в контейнер. Анимация движения имеет продолжительность 750 миллисекунд и использует функцию плавности `quadEaseOut`.

```
timeline.from(container.first, {
    y: -container.first.height,
    state: 1
}, 750, Math.quadEaseOut, 0);
```

Как показано в следующем коде, второй вызов метода временной шкалы `from` затем добавляет анимацию движения для перемещения цветной полосы из положения `x` от левого края экрана в исходное положение 0 пикселей от левого края. Каждый вызов `from` расширяет временную шкалу на длительность анимации, и, если не используется аргумент задержки, анимация движения, добавляемая следующим вызовом `from`, начинается в конце временной шкалы. Чтобы две анимации запускались одновременно, в этом примере для свойства задержки задается значение `-750` миллисекунд, что приводит к ее запуску одновременно с первой анимацией. Эта анимация не меняет продолжительность временной шкалы, потому что она заканчивается в то же время, что и первая анимация.

```
timeline.from(container.last, {
    x: -320
}, 750, Math.linearEase, -750);
```

Анимация движения, добавленная методом `to` временной шкалы, называемая `to-tween`, упрощает свойства целевого объекта от его текущих значений до целевых значений, указанных в объекте свойств, в течение миллисекунд.

Метод `onDisplaying` продолжается добавлением анимации движения, как показано ниже. В этом примере цветная полоса переходит из текущего состояния 0 в состояние 1. Свойство задержки здесь не указано, поэтому по умолчанию оно равно 0, что приводит к тому, что эта анимация начинается сразу после завершения предыдущей.

```
timeline.to(container.last, {
    state: 1
}, 750, Math.linearEase, 0);
```

После добавления всех твинов временная шкала готова к использованию, как показано в следующем коде в оставшихся вызовах метода `onDisplaying`. Временная шкала имеет текущее время между 0 и длительностью временной шкалы, которое указывает на ход анимации и может быть установлено с помощью его метода `seekTo`. Подобно свойству `duration` (и часам объекта контента), `seekTo` выражает время в миллисекундах. В этом примере временная шкала перематывается к началу с помощью `seekTo`, чтобы установить текущее время временной шкалы на 0. Затем он использует часы объекта содержимого — в данном случае часы контейнера — для управления анимацией: после установки продолжительности контейнера в соответствии с продолжительностью временной шкалы, он перематывает часы контейнера и начинает их отсчет.

```
timeline.seekTo(0);
container.duration = timeline.duration;
container.time = 0;
container.start();
```

`TimelineBehavior` включает два дополнительных обработчика событий, `onTimeChanged` и `onFinished` (листинг 10.59):

- Когда часы тикают, `onTimeChanged` вызывается через равные промежутки времени. Поскольку продолжительность временной шкалы равна продолжительности часов контейнера, `onTimeChanged` использует `seekTo` для синхронизации временной шкалы со свойством `time` часов контейнера.

- Когда часы контейнера достигают своей длительности, запускается событие `onFinished`. Это также означает, что последовательность анимации завершена. В этом примере временная шкала движется в обратном направлении после того, как достигает конца, и бесконечно за циклируется назад и вперед.

Листинг 10-59.

```
onTimeChanged(container) {
    let time = container.time;
    if (this.reverse)
        time = container.duration - time;
    this.timeline.seekTo(time);
}
onFinished(container) {
    this.reverse = !this.reverse;
    this.timeline.seekTo(0);
    container.time = 0;
    container.start();
}
```

Анимация переходов

Класс `Piu Transition` предоставляет еще один метод реализации анимации. Чаще всего он используется для замены одного объекта содержимого на другой в иерархии включения, например для перемещения между экранами. В этом разделе основное внимание уделяется встроенным переходам стирания и гребенки, которые являются подклассами класса `Transition`. В отличие от анимации на временной шкале, которая изменяет свойства объектов содержимого, переходы стирания и гребенки являются чисто графическими операциями, воздействующими на пиксели дисплея. Поскольку они оптимизированы для минимизации количества пикселей, отрисовываемых в каждом кадре, эти переходы обеспечивают высокую

частоту кадров даже на микроконтроллере ESP8266. Вы также можете создавать свои собственные переходы, создавая подкласс класса `Transition`, но это выходит за рамки данной книги.

Вы импортируете классы переходов вытеснения и гребня из модулей:

```
import WipeTransition from "piu/WipeTransition";
import CombTransition from "piu/CombTransition";
```

Переход с вытеснением показывает новый экран, начиная с края или угла экрана. Конструктор этого перехода имеет следующие аргументы для управления очисткой:

1. Продолжительность в миллисекундах
2. Уравнение смягчения
3. Горизонтальное направление, например «center - центр», «left» или «right».
4. Вертикальное направление, например «middle - середина», «top - верх» или «bottom - низ».

Горизонтальное и вертикальное направления определяют, где начинается переход. Например, если они центральные и верхние, переход начинается с верхнего края; если они правые и нижние, переход начинается с правого нижнего угла.

```
const wipeFromCenter = new WipeTransition(250,
    Math.quadEaseOut, "center", "top");
const wipeFromTopRight = new WipeTransition(250,
    Math.quadEaseOut, "right", "bottom");
```

Гребенчатый переход показывает новый экран через ряд чередующихся полос, которые выходят либо из верхнего и нижнего краев экрана, либо из левого и правого краев экрана. Конструктор гребенчатого перехода имеет следующие аргументы:

1. Продолжительность в миллисекундах
2. Уравнение смягчения

3. Направление, как "горизонтальное" или "вертикальное"
4. Количество баров

Если направление установлено как горизонтальное, полосы выходят из левого и правого краев; если он установлен на вертикаль, полосы выходят из верхнего и нижнего краев.

```
const horizontalComb = new CombTransition(250,
                                          Math.quadEaseOut, "horizontal", 4);
const verticalComb = new CombTransition(250,
                                       Math.quadEaseOut, "vertical", 8);
```

Когда у вас есть экземпляр перехода, вы вызываете метод `run` родительского контейнера объекта, из которого выполняется переход, передавая в качестве аргументов переход, объект содержимого для перехода и объект содержимого для перехода. Переход выполняется асинхронно и поэтому не блокирует выполнение вашего кода. Когда переход завершается, объект контента, от которого следует перейти, заменяется в иерархии включения объектом контента, к которому осуществляется переход. Например, код в листинге 10-60 выполняет переход `wipeFromTopRightTransition` для замены `firstScreen` на `nextScreen`.

Листинг 10-60.

```
const firstScreen = new Content(...);
const nextScreen = new Content(...);
const sampleContainer = new Container(null, {
    ...
    contents: [
        firstScreen
    ]
});

sampleContainer.run(wipeFromTopRightTransition, firstScreen,
                  nextScreen);
```

Пример `$EXAMPLES/ch10-piu/transitions` показывает несколько вариантов переходов вытеснения и гребенки. Он переключается между двумя экранами через равные промежутки времени.

Рисование графика в реальном времени

Иногда есть элементы вашего пользовательского интерфейса, которые более удобно или эффективно визуализировать с помощью функций рисования, таких как предоставляемые `Poco`, вместо создания и обновления объектов, как это делает `Piu`. Например, представьте, что вы хотите создать гистограмму, подобную той, что показана на Рисунок 10-23, который обновляется в реальном времени на основе показаний датчика.

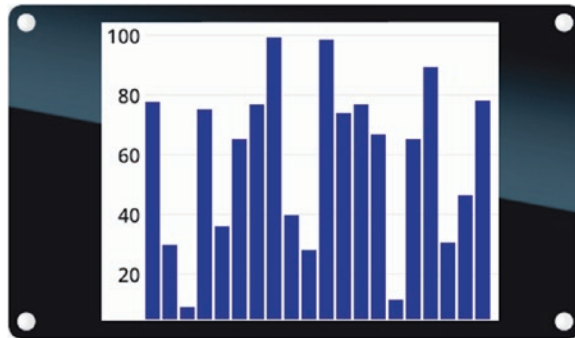


Рис. 10-23. *Bar graph that updates in real time*

Вы можете использовать объекты содержимого `Piu`, но это не самая эффективная реализация. У вас будет много объектов контента, которые нужно отслеживать и обновлять — по крайней мере, один объект контента для каждой полосы графика и фона графика, а также один или несколько объектов меток для меток на оси Y. Каждый объект занимает некоторое количество ОЗУ, поэтому использование ОЗУ быстро увеличится.

К счастью, вам не нужно выбирать между подходами `Piu` и `Poco`; вы можете комбинировать их, используя класс `Piu Port`. Порт — это объект содержимого, который позволяет вам выполнять команды рисования, аналогичные `Poco`, в макете `Piu`, что делает его идеальным для

элементов пользовательского интерфейса, таких как графики, для которых в противном случае потребовалось бы множество объектов содержимого.

Пример \$EXAMPLES/ch10-piu/graph использует один объект порта для эффективного отображения гистограммы в реальном времени, показанной на рисунке 10-23:

```
const graph = new Port(null, {
  top: 0, bottom: 0, left: 0, right: 0,
  Behavior: GraphBehavior
});
```

Поведение этого порта, GraphBehavior (листинг 10-61), поддерживает список выборочных значений для построения графика в массиве, хранящемся в свойстве values. Каждые 100 миллисекунд обработчик события onTimeChanged удаляет первое значение в списке и заменяет его случайным числом от 0 до 100. Эти случайные числа являются смоделированными показаниями датчика. После создания нового значения onTimeChanged вызывает метод недействительности порта, который сообщает Piu, что порт необходимо перерисовать.

Листинг 10-61.

```
class GraphBehavior extends Behavior {
  onDisplaying(port) {
    this.values = new Array(20);
    this.values.fill(0);
    port.interval = 100;
    port.start();
  }
  onTimeChanged(port) {
    this.values.shift();
    this.values.push(Math.random() * 100);
    port.invalidate();
  }
}
```


Вызов недействительности приводит к тому, что объект порта инициирует событие `onDraw` для самого себя. Обратите внимание, что `onDraw` вызывается не из вызова метода недействительности, а через некоторое время после него. Как показано в листинге 10-62, в этом случае обработчик события `onDraw` заполняет фон белым цветом, рисует метки оси Y и соответствующие серые линии, а затем рисует синюю полосу для каждого случайно сгенерированного значения.

Листинг 10-62.

```
onDraw(port, x, y, width, height) {
    port.fillColor(WHITE, x, y, width, height);

    for (let i = 100, yOffset = 0; yOffset < height;
        yOffset += height / 5, i -= 20) {
        port.drawString(i, textStyle, "black",
            30 - textStyle.measure(i).width,
            yOffset);
        port.fillColor(GRAY, 35, yOffset + 10, width, 1);
    }

    let xOffset = 35;
    const values = this.values;
    for (let i = 0; i < values.length; i++) {
        let value = values[i];
        let barHeight = (value / 100) * (height - 10);
        port.fillColor(BLUE, xOffset, height - barHeight,
            12, barHeight);
        xOffset += 14;
    }
}
```

В этом примере используются два метода рисования, предоставляемые объектом содержимого порта:

- Он вызывает функцию `drawString` для рисования строки текста так, как это сделал бы объект метки, с заданными стилем и цветом. Метод измерения объекта `textStyle` вызывается для вычисления ширины строк метки, чтобы они располагались точно.
- Он вызывает `fillColor` для рисования прямоугольника указанным цветом.

Объект порта имеет несколько других методов рисования, в том числе `drawTexture` для рисования изображения, заданного текстурой, и `drawSkin` для рисования прямоугольника с оболочкой, как это сделал бы любой объект содержимого. Для получения подробной информации обо всех командах рисования, доступных для переноса объектов, см. документацию Piu в Moddable SDK.

Добавление экранной клавиатуры

Во многих продуктах IoT бывают ситуации, требующие от пользователя ввода текста, например, для ввода пароля Wi-Fi при настройке продукта. Сегодня эта операция обычно выполняется в сопутствующем приложении на мобильном телефоне, требующем от пользователя установки нового мобильного приложения и выполнения сложного, подверженного ошибкам процесса настройки Wi-Fi. В продуктах IoT с сенсорным экраном пользователь может настраивать Wi-Fi и вводить текст для других целей непосредственно на продукте. Чтобы включить это, вам просто нужна экранная клавиатура.

Проблема заключается в том, что точный ввод текста легче, когда клавиатура больше, но большие сенсорные экраны стоят дороже. Чтобы решить эту проблему, Moddable SDK включает модуль, предоставляющий расширяющуюся экранную клавиатуру, позволяющую точно вводить текст на небольших сенсорных экранах. Ввод символа на этой клавиатуре — это двухэтапный процесс:

сначала вы нажимаете рядом с символом, который хотите ввести (либо на этом символе, либо рядом с ним); клавиатура расширяется вокруг того места, где вы нажали, и затем вы нажимаете нужный символ. Вы нажимаете ОК, когда закончите вводить текст.

Вы можете попробовать это, запустив пример \$EXAMPLES/ch10-piu/keyboard. Когда пример запущен, вы видите клавиатуру в нераскрытом состоянии (рис. 10-24) с мигающим курсором в текстовом поле над клавиатурой.



Рис. 10-24. Нерасширенная клавиатура

На рис. 10-25 левое изображение показывает, как клавиатура расширяется после нажатия на букву *a* или рядом с ней, а правое изображение показывает, как она расширяется после нажатия на букву *g* или рядом с ней.



Рис. 10-25. Клавиатура расширена вокруг буквы *a* (слева) и *g* (справа)

Затем вы нажимаете на нужный символ, который появляется перед мигающим курсором в текстовом поле, и клавиатура возвращается в нераскрытое состояние. (Обратите внимание, что в развернутом состоянии кнопка «ОК» меняется на отображение значка клавиатуры; вы должны нажать ее, если не хотите вводить символ, а вместо этого хотите вернуться к нераскрытой клавиатуре и кнопке «ОК».)

Доступны два варианта расширяемой клавиатуры:

`VerticalExpandingKeyboard` для экранов шириной 240 пикселей и `HorizontalExpandingKeyboard` для экранов шириной 320 пикселей. В примере с клавиатурой используется горизонтальный вариант, поэтому он импортирует объекты `HorizontalExpandingKeyboard` и `KeyboardField` из модулей клавиатуры.

```
import {HorizontalExpandingKeyboard} from "keyboard";
import {KeyboardField} from "common/keyboard";
```

Эти модули являются частью Moddable SDK, поэтому вы можете ознакомиться с их исходным кодом и полной документацией. Теперь, когда вы прочитали эту главу, все в модулях будет вам знакомо; все элементы клавиатуры созданы с использованием уже знакомых вам классов `Pic`, включая `Port`, `Timeline` и `Behavior`. В этом разделе не описывается реализация модулей клавиатуры, а основное внимание уделяется тому, как использовать модули для включения клавиатуры в ваши проекты.

Шаблон `KeyboardContainer` (листинг 10-63) — хорошее место для начала изучения этого примера. Первый элемент в его содержимом — это экземпляр `KeyboardField`, класса объектов содержимого, импортированного из модуля `common/keyboard`. В этом поле будет размещаться текст, который вы вводите. Класс `KeyboardField` имеет поведение, которое реагирует на ввод текста и мигает курсором. Второй элемент — это контейнер для клавиатуры, хотя он начинается как пустой контейнер. Обратите внимание, что оба этих объекта содержимого имеют свойство привязки, поэтому привязки к ним создаются в данных инстанцирования.

Листинг 10-63.

```

const KeyboardContainer = Column.template($ => ({
  left: 0, right: 0, top: 0, bottom: 0,
  contents: [
    KeyboardField($, {
      anchor: "FIELD",
      left: 32, right: 0, top: 0, bottom: 0,
      style: fieldStyle
    }),
    Container($, {
      anchor: "KEYBOARD",
      left: 0, right: 0, bottom: 0, height: 164
    })
  ],
  active: true,
  Behavior: KeyboardContainerBehavior
}));

```

В KeyboardContainerBehavior (листинг 10-64) методы, связанные с событиями onDisplaying и onTouchEnded (с которыми вы уже знакомы), делают одно и то же: они вызывают метод addKeyboard.

Листинг 10-64.

```

class KeyboardContainerBehavior extends Behavior {
  ...
  onDisplaying(column) {
    this.addKeyboard();
  }
}

```

```

onTouchEnded(column) {
    this.addKeyboard();
}
...
}

```

Метод `addKeyboard` (листинг 10-65) проверяет, содержит ли уже объект-контейнер, на который ссылается `data.KEYBOARD`, клавиатуру. Если это не так, метод добавляет экземпляр `HorizontalExpandingKeyboard` в пустой объект-контейнер на основе трех переданных аргументов:

- Стил — это стиль символов на клавишах клавиатуры.
- Целью является объект, который должен получать события при нажатии клавиши, в данном случае это объект `KeyboardField`, на который ссылается `data.FIELD`.
- Параметр `doTransition` указывает, должна ли клавиатура переходить внутрь. Если это значение равно `true`, клавиатура переходит по одной строке за раз; если `false`, то появляется все сразу.

Листинг 10-65.

```

addKeyboard() {
    if (1 !== this.data.KEYBOARD.length) {
        this.data.KEYBOARD.add(HorizontalExpandingKeyboard(
            this.data, {
                style: keyboardStyle,
                target: this.data.FIELD,
            }
        ));
    }
}

```

```
        doTransition: true
    }));
}}
```

Когда пользователь нажимает кнопку ОК, клавиатура отправляет событие `onKeyboardOK` в контейнер приложения с текстовой строкой, введенной пользователем. В этом примере `KeyboardContainerBehavior` реагирует на событие, отслеживая введенную строку и скрывая поле, в котором отображается строка и курсор.

```
onKeyboardOK(application, string) {    trace(`User entered: $
{string}\n`);    this.data.FIELD.visible = false;
}
```

Клавиатура появляется со скользящим переходом и выдвигается, чтобы исчезнуть, когда пользователь нажимает ОК. Когда любой из этих переходов завершается, клавиатура выдает событие `onKeyboardTransitionFinished` с параметром, указывающим, предназначен ли переход для появления или исчезновения клавиатуры. Ваш код может использовать эти события для выполнения таких действий, как отображение элементов пользовательского интерфейса, скрытых во время использования клавиатуры.

В этом примере метод `onKeyboardTransitionFinished` (листинг 10-66) реагирует на исчезновение клавиатуры, удаляя ее из иерархии включения, и метод реагирует на появление клавиатуры, делая текстовое поле над клавиатурой видимым.

Листинг 10-66.

```
onKeyboardTransitionFinished(application, out) {    if (out) {
    let keyboard = this.data.KEYBOARD;keyboard.remove
    (keyboard.first);
}
```

```

else
    this.data.FIELD.visible = true;
}

```

Обратите внимание, что клавиатуру не нужно удалять из иерархии сдерживания после перехода; вы можете продолжать перемещать один и тот же экземпляр в поле зрения и за его пределами. Однако во многих приложениях нажатие кнопки «ОК» вызывает переход на другой экран, поэтому лучше удалить клавиатуру из иерархии сдерживания, чтобы ее можно было очистить от мусора.

Организация кода пользовательского интерфейса с помощью модулей

Каждый пример в этой главе содержится в одном модуле и, следовательно, в одном файле исходного кода. По мере того как ваши приложения становятся все более сложными — с несколькими экранами, взаимодействием с облачными службами и другими устройствами и т. д. — вы, вероятно, захотите разделить свой код на несколько модулей. Разделение кода на модули имеет следующие преимущества:

- Повторное использование кода упрощается, поскольку код, не относящийся к одному продукту, можно хранить в отдельных файлах исходного кода. Клавиатурные модули являются примером этого.
- Редактировать и поддерживать код проще, если он организован в виде логических модулей.
- Легче распределять работу по команде.

Пример `$EXAMPLES/ch10-piu/multiple-screens`, обсуждаемый в этом разделе, показывает распространенный способ организации пользовательского интерфейса. Это простое приложение с двумя экранами: экраном-заставкой и начальным экраном, как показано на рис. 10-26. Приложение сначала отображает анимированный экран-заставку, а затем переходит на главный экран с кнопкой перезагрузки и меткой. При нажатии на кнопку перезагрузки возвращается экран-заставка.

Попутно в примере демонстрируются полезные приемы создания удобных в сопровождении, эффективных с точки зрения памяти приложений с несколькими модулями и экранами.



Рис. 10-26. Экран-заставка (слева) и главный экран (справа) из примера с несколькими экранами

Модули

Пример с несколькими экранами состоит из трех модулей:

- `example.js` – логика приложения для навигации между экранами
- `assets.js` – объекты текстур, скинов и стилей, используемые во всем приложении
- `screens.js` – шаблоны для двух экранов приложения

В этом примере модули `assets` и `screens` не особенно длинные, поэтому их разделение может показаться странным, так как модуль `assets` экспортирует объекты, которые использует только модуль `screens`. Тем не менее, это часто полезно в больших приложениях, потому что вам нужно изменить только один файл, чтобы изменить цвета и активы, используемые на всех экранах. Это также полезно, когда вы создаете линейку продуктов с похожим брендом; вы можете придать всем своим продуктам единообразный внешний вид, создав общий файл активов, который определяет общие текстуры, скины и стили, используемые вашими экранами.

Вы уже видели много примеров текстур, скинов и объектов стилей в этой главе, поэтому модуль ресурсов здесь подробно не описывается. В следующих разделах основное внимание уделяется примерам и модулям экранов, а также тому, как они взаимодействуют.

Логика приложения

Пример модуля содержит всю специфичную для приложения логику, которая в этом приложении представляет собой простую логику для перемещения между экранами. При запуске пример создает экземпляра шаблона `MainContainer` (листинг 10-67) и добавляет его к объекту приложения. Этот контейнер используется в примере для хранения экранов.

Листинг 10-67.

```
const MainContainer = Container.template($ => ({
  top: 0, bottom: 0, left: 0, right: 0,
  Behavior: MainContainerBehavior
}));

application.add(new MainContainer({}));
```

Экземпляр `MainContainer` изначально пуст. Его поведение добавляет и удаляет экраны, определенные в модуле `screens`. Как показано в листинге 10-68, поведение добавляет первый экран в обработчик события `onDisplaying`, вызывая его метод `switchScreen` с именем экрана «SPLASH».

Листинг 10-68.

```
class MainContainerBehavior extends Behavior {
  onCreate(container, data) {
    this.data = data;
  }
}
```

```

onDisplaying(container) {
    this.switchScreen(container, "SPLASH");
}
...
}

```

Следующий обработчик событий в поведении — это `switchScreen`, который приложение вызывает каждый раз, когда ему нужно переключиться на новый экран. Метод `switchScreen` запускает событие `doSwitchScreen` для перехода к новому экрану; однако вместо того, чтобы инициировать событие с помощью метода делегата, который вызвал бы его немедленно, он использует метод отсрочки, который откладывает доставку события до следующей итерации цикла обработки событий. Единственная разница между отложенным и делегатом заключается во времени доставки события.

```

switchScreen(container, nextScreenName) {
    container.defer("doSwitchScreen", nextScreenName);
}

```

Одна из причин, по которой вы должны отложить доставку события, — избежать переполнения стека. Стек микроконтроллера небольшой, и код для создания экрана часто занимает довольно много места в стеке. Если вы сразу переключаете экраны, часть стека уже используется вызовами, вызывающими обработчик событий вашего поведения. Откладывая доставку события, ваш обработчик событий работает с почти пустым стеком, тем самым снижая пиковое использование стека.

Еще одна причина отложить доставку события — уменьшить пиковое использование памяти при переключении экранов. Из-за того, как работает сборка мусора, если вы немедленно доставляете событие `doSwitchScreen`, сборщик мусора сохраняет в памяти как предыдущий, так и следующий экраны в течение короткого периода времени. Использование `defer` позволяет сначала освободить предыдущий экран перед созданием экземпляра следующего экрана. Это то, что делает метод `doSwitchScreen` (листинг 10-69) `MainContainer` следующим образом:

1. Он использует метод `empty` для очистки текущего экрана. Поскольку это делается из отложенного события, объекты, связанные с этим экраном, становятся доступными для сборки мусора.
2. Он вызывает `application.purge`, который освобождает кэши, созданные `Piu`, и запускает сборщик мусора, освобождая память, используемую объектами старого экрана.
3. Он создает и добавляет следующий экран.

Листинг 10-69.

```
doSwitchScreen(container, nextScreenName) {  
    container.empty();  
    application.purge();  
    switch (nextScreenName) {  
        case "SPLASH":  
            container.add(new SCREENS.SplashScreen(this.data));  
            break;  
        case "HOME":  
            container.add(new SCREENS.HomeScreen(this.data));  
            break;  
    }  
}
```

Этот процесс является хорошим способом управления использованием ОЗУ приложения, поскольку он помогает гарантировать, что в ОЗУ никогда не будет объектов на два экрана одновременно. Включение логики переключения экранов в поведение `MainContainer` также полезно, поскольку это избавляет вас от необходимости повторять ее в поведении каждого шаблона экрана; вместо этого каждый экран может просто делегировать событие `switchScreen`, когда пришло время перейти на новый экран.

Заставка

Как и многие мобильные и веб-приложения, этот пример отображает простой экран-заставку при запуске приложения. Как показано в листинге 10-70, логотип на этом экране создается путем наложения трех объектов содержимого, что позволяет анимировать каждую часть отдельно с помощью объекта временной шкалы. Заголовок на экране представляет собой простой объект метки.

Листинг 10-70.

```
const SplashScreen = Container.template($ => ({
  top: 0, bottom: 0, left: 0, right: 0,
  skin: ASSETS.backgroundSkin,
  contents: [
    Content($, {
      anchor: "LOG01",
      top: 30,
      skin: ASSETS.logoSkin1
    }),
    Content($, {
      anchor: "LOG02",
      top: 30,
      skin: ASSETS.logoSkin2
    }),
    Content($, {
      anchor: "LOG03",
      top: 30,
      skin: ASSETS.logoSkin3
    })
  ]
})
```

```

        Label($, {
            anchor: "TITLE",
            top: 155,
            style: ASSETS.bigTextStyle,
            string: "lorem ipsum"
        })
    ],
    Behavior: SplashScreenBehavior
}));

```

Как обычно, временная шкала определяется в поведении (листинг 10-71) и управляется внутренними часами объекта-контейнера.

Листинг 10-71.

```

class SplashScreenBehavior extends Behavior {
    ...
    onDisplaying(container) {
        let data = this.data;
        let timeline = this.timeline = new Timeline;
        ...
    }
    onTimeChanged(container) {
        this.timeline.seekTo(container.time);
    }
}

```

Когда анимация завершается, метод `onFinished` поведения (Листинг 10-72) делает следующее:

- Он удаляет привязки для всех объектов контента на экране. Обратите внимание, что при этом не удаляются сами объекты контента, а удаляются только ссылки на них в объекте данных. Важно удалить эти ссылки, потому что объект данных является общим для объекта `MainContainer` и передается на все экраны, которые он

создает; если ссылки не удалены, сборщик мусора не сможет освободить оперативную память, связанную с объектами содержимого, при вызове `application.purge` в методе `doSwitchScreen`.

- Затем он выводит событие `switchScreen`, которое в конечном итоге достигает объекта `MainContainer`. Он передает строку «HOME» в качестве второго аргумента, поэтому `MainContainer` загружает главный экран следующим.

Листинг 10-72.

```
onFinished(container) {
    let data = this.data;
    // Delete anchors
    delete data.LOG01;
    delete data.LOG02;
    delete data.LOG03;
    delete data.TITLE;
    // Transition to next screen
    container.bubble("switchScreen", "HOME");
}
```

Главный (домашний) экран

Главный экран (листинг 10-73) представляет собой строку, в центре которой находится кнопка перезапуска и метка. Кнопка перезагрузки и домашний экран имеют поведение `RestartButtonBehavior` и `HomeScreenBehavior` соответственно.

Листинг 10-73.

```

const HomeScreen = Row.template($ => ({
  top: 0, bottom: 0, left: 0, right: 0,
  skin: ASSETS.backgroundSkin,
  contents: [
    Content($, {
      left: 0, right: 0
    }),
    Container($, {
      anchor: "BUTTON",
      skin: ASSETS.buttonBackgroundSkin,
      contents: [
        Content($, {
          skin: ASSETS.restartArrowSkin
        })
      ],
      active: true,
      Behavior: RestartButtonBehavior
    }),
    Label($, {
      anchor: "TEXT",
      left: 10,
      style: ASSETS.bigTextStyle,
      string: "Restart",
      left: 0, right: 0
    })
  ],
  Behavior: HomeScreenBehavior
}));

```


Обработчик события `onDisplaying` класса `HomeScreenBehavior` анимирует кнопку перезапуска и метку, как показано в листинге 10.74.

Листинг 10-74.

```
class HomeScreenBehavior extends Behavior {
    onCreate(container, data) {
        this.data = data;
    }
    onDisplaying(container) {
        let data = this.data;
        let timeline = this.timeline = new Timeline();
        ...
        container.start();
    }
    ...
}
```

В отличие от экрана-заставки, домашний экран не переключает экраны автоматически после анимации. Вместо этого он ожидает получения события `animateOut`; метод `animateOut` его поведения (листинг 10-75) создает объект временной шкалы и устанавливает для свойства `transitioningOut` значение `true`.

Листинг 10-75.

```
animateOut(container) {
    let data = this.data;
    this.transitioningOut = true;
    let timeline = this.timeline = new Timeline();
    ...
    container.start();
}
```

Когда в конце анимации запускается событие `onFinished`, соответствующий обработчик событий (листинг 10-76) проверяет свойство `transitioningOut`, чтобы определить, какое действие предпринять:

- Если для перехода `Out` установлено значение `true`, привязки к кнопке и метке удаляются, а событие `switchScreen` передается в объект `Main Container`.
- Если `transitioningOut` имеет значение `false`, свойство временной шкалы удаляется, что делает объект временной шкалы пригодным для сборки мусора. Поскольку сборщик мусора запускается только тогда, когда ему нужно освободить оперативную память, и никакие другие объекты не создаются между переходами внутрь и наружу, сборщик мусора не запустится, поэтому удалять свойство временной шкалы здесь не нужно. Тем не менее, полезно иметь привычку удалять ссылки на объекты, которые больше не используются.

Листинг 10-76.

```
onFinished(container) {
  if (this.transitioningOut) {
    let data = this.data;
    // Delete anchors
    delete data.BUTTON;
    delete data.TEXT;
    // Transition to next screen
    container.bubble("switchScreen", "SPLASH");
  }
  else
    delete this.timeline;
}
```

Поведение кнопки перезагрузки (листинг 10-77) зависит только от одного события: `onTouchEnded`. Метод `onTouchEnded` поведения просто делегирует событие `onAnimateOut` контейнеру кнопки, который является экземпляром шаблона `HomeScreen`. Как вы только что видели, это запускает анимацию и, в конце концов, приводит к переходу обратно к экрану-заставке.

Листинг 10-77.

```
class RestartButtonBehavior extends Behavior {
    onTouchEnded(content) {
        content.container.delegate("animateOut");
    }
}
```

Добавление дополнительных экранов

Теперь, когда вы знаете, как переключаться между двумя экранами, добавить дополнительные экраны несложно. Вот шаги:

1. Определите шаблон для нового экрана.
2. Добавьте его в стандартный экспорт модуля экранов.
3. В модуле-примере добавьте `case` к оператору `switch` в методе `doSwitchScreen` класса `MainContainerBehavior`, чтобы создать экземпляр шаблона экрана и добавить его в `MainContainer`.
4. Иницилируйте событие `switchScreen` по мере необходимости в вашем коде, передав имя, которое вы использовали для нового экрана, в операторе `switch`.

Заключение

В этой главе вы изучили основы создания пользовательских интерфейсов с помощью Piu, в том числе то, как добавлять графику и текст, придавать им поведение, управляемое событиями, и создавать анимацию. Вы изучили несколько приемов экономии оперативной памяти, таких как повторное использование текстур и скинов и удаление ссылок на неиспользуемые объекты. Вы также изучили методы сохранения ПЗУ, в том числе с использованием шаблонов. С помощью информации из этой главы вы сможете создавать красивые современные пользовательские интерфейсы, используя недорогое оборудование.

В этой главе представлены ключевые функции Piu, которые используются для создания пользовательских интерфейсов для встраиваемых продуктов. Piu имеет множество других функций, которые могут оказаться полезными в ваших продуктах, например, поддержка эффективной локализации текстовых строк для продуктов, которые должны поддерживать несколько языков. Подробную документацию по всем функциям Piu и ссылки на примеры их использования см. в документации Piu в Moddable SDK.

ГЛАВА 11

Добавление собственного кода

Бывают случаи, когда JavaScript — не лучший язык для реализации частей вашего продукта IoT. К счастью, вам не нужно выбирать JavaScript или C (или C++) для создания вашего продукта: вы можете выбрать оба. XS in C — это низкоуровневый API C, предоставляемый движком XS JavaScript, чтобы вы могли интегрировать код C в свои проекты JavaScript (или код JavaScript в свои проекты C!).

Вот три распространенные причины использования нативного кода в вашем проекте:

- **Производительность.** Языки высокого уровня, включая JavaScript, не могут превзойти оптимизированный нативный код в высокопроизводительных задачах. Вы можете добавлять собственные оптимизированные нативные функции и вызывать их из кода JavaScript.
- **Доступ к аппаратным функциям.** Как язык программирования общего назначения, JavaScript не имеет встроенной поддержки уникальных функций вашего хост-оборудования. Вы можете реализовать свои собственные функции и классы для их настройки и использования.
- **Повторное использование существующего нативного кода.** У вас может быть большой объем существующего нативного кода, который хорошо работает для ваших продуктов, и вы бы предпочли не переписывать его на JavaScript. Вы можете использовать этот код в своих проектах JavaScript, используя XS в C для связи между ним и вашим кодом JavaScript.

XS в C позволяет вам работать с функциями JavaScript из C. Как вы знаете, в JavaScript есть возможности, которые C не поддерживает напрямую, например, динамические типы и объекты. Работать с этими функциями с помощью XS в C может быть неудобно, но становится просто, когда вы немного потренируетесь и изучите некоторые общие шаблоны. Эта глава знакомит с XS в C через серию примеров, демонстрирующих различные методы построения моста между JavaScript и кодом C.

Обратите внимание, что многие движки, реализующие язык программирования высокого уровня, предоставляют API для связи между этим языком и собственным кодом. Язык Java определяет для этой цели Java Native Interface (JNI), а механизм JavaScript V8 предоставляет C++ API.

Важно Информация, представленная в этой главе, является дополнительной темой. Предполагается, что вы умеете программировать на C и хорошо понимаете основные концепции JavaScript, обсуждаемые в этой книге.

Установка хоста

Для этой главы не нужно устанавливать хост, потому что весь машинный код должен быть частью самого хоста; поэтому вы создаете каждый пример в этой главе как автономный хост. Вместо использования `mcrun` для установки примеров вы используете `mcconfig`. Следующие командные строки предназначены для целей ESP32 и ESP8266 соответственно:

```
> mcconfig -d -m -p esp32
> mcconfig -d -m -p esp
```

В этих командных строках не указана отладочная плата (например, `esp32/moddable_two`), поскольку в примерах используются только общие функции микроконтроллера и они не зависят от особенностей конкретной платы.

Когда вы создаете примеры с помощью `mconfig`, создаются как код JavaScript, так и код C. Если при сборке возникает ошибка, об этом сообщается в командной строке.

Генерация случайных целых чисел

Первый пример интеграции собственного кода генерирует случайные целые числа. В главе 9 вы видели, что в примере со случайными прямоугольниками используются случайные числа, сгенерированные встроенной функцией JavaScript `Math.random`. Этот пример менее эффективен, чем мог бы быть, потому что `Math.random` возвращает значение с плавающей запятой, заставляя Росо преобразовывать несколько значений с плавающей запятой в целые числа для каждого прямоугольника. Операции с плавающей запятой, как правило, медленны на микроконтроллерах, и здесь от них нет никакой пользы. Функция `rand` стандартной библиотеки C генерирует случайные целые числа, а пример `$EXAMPLES/ch11-native/random-integer` начинается с использования `rand` для генерации случайных целых чисел для кода JavaScript.

Создание собственной функции

Первым шагом является создание функции JavaScript, которую код JavaScript может вызывать для вызова вашей функции C. Пример `random-integer` объявляет функцию `randomInt` в файле исходного кода `main.js`.

```
function randomInt() @ "xs_randomInt";
```

Этот синтаксис создает функцию JavaScript с именем `randomInt`, которая при вызове вызывает нативную функцию `xs_randomInt`, по сути, создавая мост между JavaScript и C. Использование `@` здесь не является стандартным синтаксисом JavaScript, а языковым расширением, предоставляемым XS для упрощения добавления собственного кода к вашим проектам. Следовательно, этот код вряд ли будет компилироваться или работать так же с другими движками JavaScript.

После создания функции вы можете вызвать ее, как и любую другую функцию JavaScript. Модуль `main.js` вызывает его 100 раз, отслеживая результат до консоли отладки.

```
for (let i = 0; i < 100; i++)
    trace(randomInt(), "\n");
```

Реализация нативной функции

Реализация `xs_randomInt` содержится в `main.c`. Когда вы создаете файл с расширением `.js`, `mcconfig` также создает файл с расширением `.c` с таким же именем. В листинге 11-1 показано все содержимое файла `main.c`.

Листинг 11-1.

```
#include "xsmc.h"

void xs_randomInt(xsMachine *the){
    xsmcSetInteger(xsResult, rand());}
```

Команда препроцессора `include` вводит заголовочный файл для `XSin C`. (Имя файла, `xsmc`, означает «XS Microcontroller»). Также имеется заголовочный файл `xs.h`, который используется некоторым кодом. Два заголовка обеспечивают эквивалентную функциональность, но функции в заголовочном файле `xsmc.h` более эффективны и поэтому предпочтительны для использования на микроконтроллерах.

Прототип собственной функции `xs_randomInt` используется для всех функций, которые реализуют собственные методы с использованием `XS в C`. Аргументы JavaScript не передаются в качестве аргументов функции `C`. Позже в этой главе вы увидите, как получить доступ к аргументам.

В этом примере необходимо вернуть значение — результат вызова `rand`. Результатом `rand` является целое число, поэтому в этом примере используется функция `xsmcSetInteger`, которая присваивает собственное 32-разрядное целочисленное значение значению JavaScript. Здесь значением JavaScript является `xsResult`, которое относится к возвращаемому значению функции в стеке JavaScript.

Использование аппаратного генератора случайных чисел

Вы видели, как просто объявить, вызвать и реализовать простую нативную функцию. Когда вы запускаете пример со случайными целыми числами, вы видите 100 случайных чисел от 0 до 2 147 483 647, отслеживаемых в консоли отладки. Но когда вы перезапустите микроконтроллер и запустите пример во второй раз, вы увидите точно такой же список чисел. Это не очень случайно. Почему это происходит?

Функция `rand` — это генератор псевдослучайных чисел. Это алгоритм генерации чисел, которые кажутся случайными; однако, когда вы перезапускаете микроконтроллер, вы также перезапускаете алгоритм генератора псевдослучайных чисел, заставляя его генерировать ту же самую последовательность чисел. Вы можете использовать функцию `srand`, чтобы алгоритм запускал другую последовательность, но вы должны указывать `srand` другую начальную точку при каждом перезапуске. Наиболее распространенным способом инициализации последовательности является использование текущего времени. К сожалению, многие микроконтроллеры, в том числе ESP32 и ESP8266, не знают время при запуске, поэтому эту технику нельзя применить.

К счастью, многие микроконтроллеры, в том числе ESP32 и ESP8266, имеют аппаратные средства для генерации случайных чисел, и эти значения более случайны, чем те, которые генерируются методом `rand`. В примере `$EXAMPLES/ch11-native/random-integer-esp` показано, как использовать аппаратный генератор случайных чисел.

Важно! Не все случайные числа гарантированно достаточно непредсказуемы для безопасного использования в решениях по обеспечению безопасности, таких как протокол TLS, защищающий сетевые подключения. (Случайные числа, которые имеют эту гарантию, называются криптографически безопасными.) Вы должны всегда проверять, что источник случайных чисел, который вы используете, соответствует требованиям безопасности вашего проекта. Сделать это непросто, но важно, поскольку слабый генератор случайных чисел — это уязвимость в общей безопасности вашего проекта.

В ESP32 для доступа к аппаратному генератору случайных чисел достаточно просто заменить вызов `rand` вызовом функции ESP-IDF `esp_random`. Степень случайности, которую обеспечивает `esp_random`, зависит от ряда факторов, в том числе от того, включено ли радио (Wi-Fi или Bluetooth).

```
xsmcSetInteger(xsResult, esp_random());
```

На ESP8266 есть недокументированный аппаратный генератор случайных чисел, который работает хорошо. Его следует использовать с осторожностью, так как его точные характеристики неизвестны. Для доступа к генератору случайных чисел, считывается напрямую его аппаратный регистр .

```
uint32_t random = *(volatile uint32_t *)0x3FF20E44;
```

В листинге 11-2 показана исправленная собственная реализация с использованием собственных генераторов случайных чисел. Поскольку доступ к генератору на ESP32 и ESP8266 осуществляется по-разному, код C использует условную компиляцию, чтобы выбрать правильную версию и сгенерировать ошибку, когда код компилируется для неподдерживаемой цели.

Листинг 11-2.

```

void xs_randomInt(xsMachine *the)
{
  #if ESP32
    xsmcSetInteger(xsResult, esp_random());
  #elif defined(__ets__)
    xsmcSetInteger(xsResult, (*(volatile uint32_t *)0x3FF20E44));
  #else
    #error Unsupported platform
  #endif
}

```

Есть две проблемы с использованием этой функции `randomInt`:

- Аппаратные генераторы случайных чисел ESP32 и ESP8266 возвращают 32-битные значения без знака. Для функции `xsmcSetInteger` требуется 32-битное значение со знаком. Следовательно, использование метода аппаратных случайных чисел изменяет результат функции JavaScript `randomInt`, возвращая диапазон значений от `-2 147 483 648` до `2 147 483 647`. Вспомните, что когда вы используете `rand`, все значения положительны. Вместо этого вы можете использовать `xsmcSetNumber` для возврата беззнакового 32-битного значения как число с плавающей запятой; однако это противоречит цели возврата случайного числа в виде целочисленного значения.
- Обычно вам нужно случайное число в определенном диапазоне, а для генерации значения в этом диапазоне требуется операция деления или деления по модулю. Операция деления обычно требует операции с плавающей запятой, так как результат может иметь дробную часть. Операция по модулю может использовать целочисленное деление, если оба операнда являются

целыми числами. Однако вместо того, чтобы требовать, чтобы вызывающая сторона `randomInt` эффективно ограничивала возвращаемое значение желаемым диапазоном, вы можете изменить собственную функцию, чтобы сделать это.

В следующем разделе рассматриваются эти вопросы.

Ограничение случайных чисел диапазоном

Пример `$EXAMPLES/ch11-native/random-integer-esp-range` ограничивает случайные числа диапазоном. Первый шаг — объявить функцию, которая принимает диапазон случайных значений. Функция `randomIntRange` принимает один аргумент, указывающий диапазон случайных значений, начиная с 0 и заканчивая максимальным значением.

```
function randomIntRange(max) @ "xs_randomIntRange";
```

Вызывающий код в `main.js` обновляется для передачи в диапазоне, который в этом примере равен 1000.

```
for (let i = 0; i < 100; i++)
  trace(randomIntRange(1000), "\n");
```

Собственная функция должна сначала получить диапазон, переданный в качестве первого аргумента. Доступ к аргументам осуществляется по индексу с помощью `xsArg`. Аргументы нумеруются, начиная с 0, поэтому доступ к первому аргументу осуществляется как `xsArg(0)`. Если вызывающая программа не передала никаких аргументов, `xsArg(0)` выдает исключение, поэтому в собственном коде обычно нет необходимости проверять количество переданных аргументов. (Если вашей функции необходимо знать количество аргументов, используйте целочисленное значение `xsmcArgc`.) Исключения, выдаваемые XS в C, являются обычными исключениями JavaScript, которые могут быть перехвачены знакомыми блоками `try` и `catch` в коде JavaScript.

Код C не может делать никаких предположений о типе аргумента, потому что JavaScript не применяет никаких правил относительно типов аргументов, передаваемых функции. XS в C предоставляет функции для преобразования значения JavaScript в определенный собственный тип.

В функции `xs_randomIntRange` (листинг 11-3) вызов `xsmcToInteger` просит XS преобразовать свойство JavaScript в 32-разрядное целое число со знаком. Если XS может выполнить преобразование, он возвращает результат; в противном случае генерируется исключение JavaScript. Например, передача строкового значения «100» или числового значения 100,1 выполняется успешно, поскольку JavaScript знает, как преобразовать их в целое число; однако при передаче пустого объекта `{}` возникает ошибка `IntegerDivideByZero`, а на ESP8266 — исключение `Illegal`

Листинг 11-3.

```
void xs_randomIntRange(xsMachine *the)
{
    int range = xsmcToInteger(xsArg(0));
    if (range < 2)
        xsRangeError("invalid range");
    ...
}
```

Затем реализация собственной функции проверяет запрошенный диапазон. Диапазон меньше двух значений не имеет смысла для целочисленных случайных чисел. Если диапазон недействителен, функция вызывает `xsRangeError`, чтобы выдать ошибку JavaScript `RangeError`. Предыдущий код C эквивалентен этим строкам JavaScript:

```
if (range < 2)
    throw new RangeError("invalid range");
```

Важно включить проверку ошибок в собственный код, который связывает ваш код JavaScript и код C. Программисты JavaScript ожидают, что язык будет безопасным — не должно быть никакого способа сломать или повредить устройство — и механизм JavaScript и среда выполнения делают все возможное для достижения этой цели. Ваш собственный код должен делать то же самое. Например, если код JavaScript передает 0 для диапазона, результат не определяется языком. Операция по модулю с 0 справа на ESP32 генерирует исключение `IntegerDivideByZero`,

а на ESP8266 — исключение `Illegal Instruction`, оба из которых сбрасывают микроконтроллер.

Оставшаяся реализация `xs_randomIntRange` (листинг 11-4) проста. Вместо прямого возврата 32-битного целого числа без знака оператор по модулю (%) ограничивает случайное значение указанным диапазоном.

Листинг 11-4.

```
#if ESP32
    xsmcSetInteger(xsResult, esp_random() % range);
#elif defined(__ets__)
    xsmcSetInteger(xsResult,
                   (*(volatile uint32_t *)0x3FF20E44) % range);
#else
    #error Unsupported platform
#endif
```

Сравнение подходов случайных чисел

Нативная функция `randomIntRange` — это всего несколько строк нативного кода, но эти несколько строк имеют много преимуществ для разработки IoT по сравнению со встроенной функцией `Math.random`:

- Возвращаемые значения являются целыми числами, а не числами с плавающей запятой, что обеспечивает более эффективное выполнение на микроконтроллерах.
- Возвращаемые значения эффективно ограничены запрошенным диапазоном.
- Числа являются более случайными, поскольку они используют аппаратный генератор случайных чисел.

Конечно, есть и недостатки:

- Возвращаемые значения являются целыми числами, а не числами с плавающей запятой, что обеспечивает более эффективное выполнение на микроконтроллерах.
- Возвращаемые значения эффективно ограничены запрошенным диапазоном.
- Числа являются более случайными, поскольку они используют аппаратный генератор случайных чисел.

Когда у вас есть возможность добавить в свой проект нативную функциональность, вы должны основывать свое решение на балансе преимуществ и недостатков.

Класс BitArray

Типизированные массивы JavaScript, такие как `Uint8Array` и `Uint32Array`, позволяют работать с массивами 8-, 16- и 32-разрядных целых чисел, используя минимум памяти. Класс `BitArray` реализует 1-битный массив, то есть массив, в котором хранятся только значения 0 и 1. Это полезно для эффективного хранения большого количества выборок, полученных с цифрового входа.

В этом разделе представлены две разновидности `BitArray`, каждая из которых использует один и тот же JavaScript API. Первый использует JavaScript `ArrayBuffer` для хранения битов, а второй использует собственную память, выделенную с помощью `calloc`-функции C.

Конструктор класса `BitArray` принимает единственный аргумент: количество битов, которое должен хранить массив. Класс предоставляет методы `get` и `set` для доступа к значениям в массиве. В листинге 11-5 показан тестовый код, использующий класс `BitArray`.

Листинг 11-5.

```
import BitArray from "bitarray";

let bits = new BitArray(128);
bits.set(2, 1);
bits.set(3, bits.get(3) ? 0 : 1);
```

Первый аргумент как `get`, так и `set` — это индекс бита в массиве, который нужно получить или установить. Индекс первого элемента массива равен 0. Последняя строка примера переключает значение бита с индексом 3.

Использование памяти, выделенной `ArrayBuffer`

Реализация `BitArray` в примере `$EXAMPLES/ch11-native/bitarray-arraybuffer` показана в листинге 11-6. Он начинается с объявления класса в JavaScript. Как и в предыдущих примерах со случайными целыми числами, для соединения функции JavaScript с собственной функцией C используется специальный синтаксис `XS @`. Обратите внимание, что конструктор реализован на JavaScript, а методы `get` и `set` реализованы на C. Нет требования, чтобы класс был полностью реализован на JavaScript или C; вы можете выбрать язык, который лучше всего подходит для каждого метода.

Листинг 11-6.

```
class BitArray {
  constructor(count) {
    this.buffer = new ArrayBuffer(Math.ceil(count / 8));
  }
  get(index) @ "xs_bitarray_get";
  set(index, value) @ "xs_bitarray_set";
}

export default BitArray;
```


Конструктор выделяет `ArrayBuffer` для хранения битовых значений. Поскольку память нового `ArrayBuffer` всегда инициализируется нулем, дальнейшая инициализация не требуется. Количество битов для хранения делится на 8, чтобы определить необходимое количество байтов, а затем округляется с помощью `Math.ceil`, чтобы гарантировать, что выделено достаточно байтов, когда количество битов не делится без остатка на 8. `ArrayBuffer` назначается для свойства буфера экземпляра `BitArray`. Собственные реализации `get` и `set` обращаются к памяти с помощью свойства буфера.

Функция получения

Собственная реализация функции `get`, `xs_bitarray_get`, начинается с получения индекса бита, первого аргумента функции. Он использует аргумент `index` для вычисления `byteIndex`, индекса байта, содержащего бит, и `bitIndex`, индекса бита в этом байте.

```
int index = xsmcToInteger(xsArg(0));
int byteIndex = index >> 3;
int bitIndex = index & 0x07;
```

Затем `xs_bitarray_get` получает указатель на память, выделенную `ArrayBuffer`, сохраненную в свойстве буфера. Для этого он сначала выделяет одну временную переменную JavaScript в стеке JavaScript, вызывая `xsmcVars` с аргументом 1, указывающим количество временных переменных.

```
xsmcVars(1);
```

Доступ к переменным, выделенным с помощью `xsmcVars`, осуществляется с помощью `xsVar`, который похож на `xsArg`, но обращается к локальным временным переменным вместо аргументов функции. Переменные автоматически освобождаются, когда нативная функция, которая их выделила, — в данном случае `xs_bitarray_get` — возвращается. Вы должны вызывать `xsmcVars` только один раз в функции, выделяя сразу все необходимые временные переменные.

Реализация `xs_bitarray_get` извлекает ссылку на свойство буфера своего экземпляра, вызывая `xsmcGet`. Значение свойства буфера помещается в `xsVar(0)`.

```
xsmcGet(xsVar(0), xsThis, xsID_buffer);
```

Второй аргумент, здесь `xsThis`, сообщает `xsmcGet`, из какого объекта вы хотите получить свойство. Третий аргумент, здесь `xsID_buffer`, указывает, что имя свойства, которое вы хотите получить, — `буфер`.

В предыдущих шагах используется много незнакомых вызовов из XS в C. То, что они делают, довольно просто в JavaScript и гораздо более многословно для выражения в C. JavaScript-эквивалент вызовов `xsmcVars` и `xsmcGet` выглядит следующим образом:

```
let var0;
var0 = this.buffer;
```

Свойство `buffer` не является указателем на буфер памяти, используемый экземпляром `ArrayBuffer`; это ссылка на экземпляр. Точно так же, как вы используете `xsmcToInteger` для преобразования значения JavaScript в целое число, вы используете `xsmcToArrayBuffer` для преобразования значения JavaScript в собственный указатель. Если значение JavaScript не является экземпляром `ArrayBuffer`, вызов `xsmcToArrayBuffer` создает исключение.

```
uint8_t *buffer = xsmcToArrayBuffer(xsVar(0));
```

Теперь, когда `xs_bitarray_get` имеет указатель буфера, он использует рассчитанные ранее значения `byteIndex` и `bitIndex`, чтобы считать бит и установить возвращаемое значение вызова функции JavaScript равным 0 или 1.

```
if (buffer[byteIndex] & (1 << bitIndex))    xsmcSetInteger
(xsResult, 1);
else
    xsmcSetInteger(xsResult, 0);
```

Функция set

Реализация функции set (листинг 11-7) в `xs_bitarray_set` очень похожа на реализацию get. Таким же образом определяются значения `byteIndex`, `bitIndex` и `buffer`. Единственное отличие состоит в том, что значение второго аргумента, доступ к которому осуществляется с помощью `xsArg(1)`, используется для определения, следует ли установить или сбросить указанный бит.

Листинг 11-7.

```
int value = xsmcToInteger(xsArg(1)); if (value)
    buffer[byteIndex] |= 1 << bitIndex; else
    buffer[byteIndex] &= ~(1 << bitIndex);
```

Уязвимость безопасности

Эта реализация `BitArray`, использующая память, выделенную `ArrayBuffer`, работает хорошо, но имеет критический недостаток, делающий ее непригодной для безопасного использования в реальных продуктах. Функции `get` и `set` не проверяют, находится ли аргумент индекса в пределах выделенной памяти. Это позволяет коду, использующему эту реализацию `BitArray`, считывать и записывать произвольную память на встроенных устройствах, что может вызвать сбой или использоваться в качестве основы для атаки на конфиденциальность. Есть несколько способов решить эту проблему; в следующем разделе обсуждается один из них.

Использование памяти, выделенной `calloc`

Реализация `BitArray` в примере `$EXAMPLES/ch11-native/bitarray-calloc` решает проблему безопасности, представленную в только что обсуждаемом примере с `bitarray-arraybuffer`. Он сохраняет количество битов, выделенных конструктором, а затем проверяет индекс, переданный вызовам `get` и `set`, по этому сохраненному значению.

Реализация BitArray в примере с bitarray-calloc использует calloc вместо ArrayBuffer для выделения памяти. Память, выделенная этими двумя подходами, поступает из двух разных пулов памяти: память, выделенная calloc, берется из кучи системной памяти, тогда как память, выделенная ArrayBuffer, находится внутри кучи памяти, управляемой XS. Некоторые хосты имеют больше свободного места в одном из этих пулов, чем в другом, что может повлиять на ваше решение о том, откуда выделять память. Для работы с памятью, выделенной calloc, требуется немного меньше кода, хотя эта разница может быть незначительной.

Пример bitarray-calloc иллюстрирует некоторые важные методы интеграции собственного кода в ваш проект. В дополнение к собственному конструктору этот класс BitArray также имеет собственный деструктор для выполнения очистки, когда экземпляр класса удаляется сборщиком мусора. В XS объект с собственным деструктором называется хост-объектом.

Декларация класса

В листинге 11-8 показано объявление класса. Эта реализация BitArray использует в основном нативные методы, в отличие от реализации из примера bitarray-arraybuffer. Обратите внимание, что имя собственной функции C, реализующей деструктор, `xs_bitarray_destructor`, следует за объявлением имени класса.

Листинг 11-8.

```
class BitArray @ "xs_bitarray_destructor" {
  constructor(count) @ "xs_bitarray_constructor";
  close() @ "xs_bitarray_close";

  get(index) @ "xs_bitarray_get";
  set(index, value) @ "xs_bitarray_set";
```

```

    get length() @ "xs_bitarray_get_length";
    set length(value) {
        throw new Error("read-only");
    }
}

```

Объявления методов `get` и `set` такие же, как и в предыдущем примере, хотя реализации несколько отличаются.

Родной конструктор, деструктор и функции `close` тесно связаны между собой. Следующие разделы рассматривают каждый по очереди.

Конструктор

Собственный конструктор в листинге 11.9 начинается так же, как реализация JavaScript, с вычисления количества байтов, необходимых для хранения запрошенного числа битов, и последующего выделения этих байтов. Конструктор выделяет дополнительное пространство размером с целое число для хранения счетчика битов. Если выделение не удастся, конструктор вызывает `xsUnknownError`, чтобы создать исключение. Использование `Unknown` в имени `xsUnknownError` означает, что это ошибка общего назначения, в которой используется класс JavaScript `Error`, а не конкретная ошибка, такая как `RangeError`.

Листинг 11-9.

```

void xs_bitarray_constructor(xsMachine *the)
{
    int bitCount = xsmcToInteger(xsArg(0));
    int byteCount = (bitCount + 7) / 8;
    uint8_t *bytes = calloc(byteCount + sizeof(int), 1);
    if (!bytes)
        xsUnknownError("no memory");

    *(int *)bytes = bitCount;
    xsmcSetHostData(xsThis, bytes);
}

```

Как только память выделена, количество запрошенных битов сохраняется в начале блока. Поскольку память выделяется с помощью `calloc`, все биты инициализируются 0.

Вызов `xsmcSetHostData` сохраняет ссылку на память, выделенную этому хост-объекту. Затем этот указатель становится доступным для всех собственных методов объекта через вызов `xsmcGetHostData`. У вас может возникнуть соблазн просто сохранить указатель байтов в глобальной переменной; однако этот подход терпит неудачу, когда существует более одного экземпляра объекта, поскольку два объекта не могут совместно использовать одну глобальную переменную `C`. Использование `xsmcSetHostData` для хранения указателя данных означает, что реализация `BitArray` поддерживает произвольное количество одновременных экземпляров.

Деструктор (Разрушитель)

Впервые в этой книге вы видите деструктор. Они распространены в C++ при работе с объектами, но не являются видимой частью языка JavaScript. Вместо этого JavaScript автоматически освобождает память, используемую объектами, когда они удаляются сборщиком мусора. Механизм JavaScript не знает, как освободить ресурсы, выделенные вашим хост-объектом, например память, выделенную с помощью `calloc`. Следовательно, вы должны реализовать деструктор.

Для `BitArray` деструктор (листинг 11-10) просто вызывает `free`, чтобы освободить память, выделенную `calloc`.

Листинг 11-10.

```
void xs_bitarray_destructor(void *data)
{
    if (data)
        free(data);
}
```

Вот некоторые детали, которые следует учитывать при реализации деструктора:

- Прототип функции деструктора отличается от обычных вызовов собственных методов. Вместо того, чтобы передавать ссылку на виртуальную машину XS в качестве аргумента, он имеет аргумент, который является указателем данных, тем же значением, которое вы передали `xsmcSetHostData`.
- Поскольку нет ссылки на виртуальную машину XS (нет аргумента), вы не можете делать вызовы XS в C. Например, вы не можете вызывать `xsmcGetHostData`, поэтому указатель данных всегда передается в функцию деструктора. Это также означает, что ваш деструктор не может создавать новые объекты, изменять значения свойств или выполнять вызовы функций для объекта. Эти ограничения необходимы, поскольку деструктор вызывается изнутри сборщика мусора, когда такие операции небезопасны.
- Значение данных может быть NULL. Это происходит, например, при сбое выделения памяти в конструкторе. Как вы увидите в следующем разделе, это также происходит после вызова метода `close`. Поэтому хорошей практикой является всегда проверять, что аргумент данных не равен NULL в вашем деструкторе, прежде чем использовать его, как это делается в этом примере.

Функция `close`

Главы 3 и 5 содержат примеры объектов JavaScript, у которых есть метод `close`. Этот метод освобождает любые собственные ресурсы — память, дескрипторы файлов, сетевые сокеты и т. д., — которыми владеет объект. Если объект не закрыт явно, эти ресурсы в конечном итоге освобождаются, когда сборщик мусора определяет, что объект больше не используется. Однако невозможно узнать, когда сборщик мусора примет это решение, а это значит, что до освобождения ресурсов может пройти очень много

времени. Вызов `close` решает эту проблему, предоставляя коду способ явного освобождения этих ресурсов.

Многие хост-объекты имеют реализацию `close`, подобную реализации для `BitArray` (листинг 11-11).

Листинг 11-11.

```
void xs_bitarray_close(xsMachine *the)
{
    uint8_t *buffer = xsmcGetHostData(xsThis);
    xs_bitarray_destructor(buffer);
    xsmcSetHostData(xsThis, NULL);
}
```

Вот что делают эти строки кода:

1. Вызов `xsmcGetHostData` извлекает указатель данных, который был выделен в конструкторе и связан с этим объектом вызовом `toxsmcSetHostData`.
2. Указатель данных передается деструктору, который выполняет работу по освобождению ресурсов.
3. Вызов `xsmcSetHostData` устанавливает для сохраненного указателя данных значение `NULL`. Это гарантирует, что при двойном вызове `close` указатель данных освобождается только один раз.

Функции `get` и `set`

Эта реализация `xs_bitarray_get` вычисляет значения битового и байтового индекса так же, как и в версии `get` для `ArrayBuffer`:ы

```
int index = xsmcToInteger(xsArg(0));
int byteIndex = index >> 3;
int bitIndex = index & 0x07;
```


Как показано в листинге 11-12, `xs_bitarray_get` использует `xsmcGetHostData` для извлечения буфера данных. Если буфер равен `NULL`, это указывает на то, что экземпляр уже был закрыт, и `get` выдает ошибку. Счетчик количества выделенных битов хранится в первом целом буфере; он извлекается в локальную переменную `bitCount`, а затем указатель буфера продвигается вперед, чтобы указывать на значения битового массива.

Листинг 11-12.

```
uint8_t *buffer = xsmcGetHostData(xsThis);int bitCount;
if (NULL == buffer)      xsUnknownError("closed");
bitCount = *(int *)buffer;buffer += sizeof(int);
```

Перед доступом к запрошенному биту реализация сначала проверяет, находится ли значение в диапазоне. Поскольку индекс представляет собой целое число со знаком, он проверяет, не больше ли он числа выделенных битов, и что индекс не является отрицательным.

```
if ((index >= bitCount) || (index < 0))      xsRangeError("
invalid bit index");
```

После завершения этой проверки чтение запрошенного бита и установка возвращаемого значения идентичны предыдущей версии:

```
if (buffer[byteIndex] & (1 << bitIndex))      xsmcSetInteger
(xsResult, 1);
else
    xsmcSetInteger(xsResult, 0);
```

Реализация `set` применяет те же изменения, что и для `get` в этом разделе, и поэтому здесь не повторяется.

Свойство длины

Классы типизированных массивов включают в свои экземпляры свойство длины, которое, как и в экземплярах `Array`, указывает количество элементов в массиве. Это значение полезно, когда вы перебираете массив. Поскольку эта реализация `BitArray` хранит количество выделенных битов, она также может предоставлять свойство длины.

Свойство `length` реализовано с помощью геттера и сеттера, двух специальных видов функций JavaScript, которые вызываются, когда код обращается к свойству. Использование геттера и сеттера для длины позволяет вам написать код, подобный следующему, для инициализации всех битов в 1:

```
let bits = new BitArray(55);
for (let i = 0; i < bits.length; i++)
    bits.set(i, 1);
```

Первым шагом в реализации свойства `length` является добавление методов получения и установки в класс `BitArray`. Здесь получателем является собственная функция `xs_bitarray_get_length`. Свойство `length` доступно только для чтения, поэтому реализация установщика вместо собственного кода представляет собой функцию JavaScript, которая всегда генерирует исключение. Обратите внимание, что хост-объект может иметь методы JavaScript.

```
get length() @ "xs_bitarray_get_length";
set length(value) {
    throw new Error("read-only");}
```

Реализация `xs_bitarray_get_length`, показанная в листинге 11.13, проста. Он использует `xsmcGetHostData` для получения указателя данных, созданного в конструкторе. Если экземпляр был закрыт, то есть если буфер имеет значение `NULL`, он выдает исключение; в противном случае он устанавливает возвращаемое значение в битовый счетчик, извлеченный из начала указателя данных.

Листинг 11-13.

```

void xs_bitarray_get_length(xsMachine *the)
{
    uint8_t *buffer = xsmcGetHostData(xsThis);
    if (NULL == buffer)
        xsUnknownError("closed");

    int bitCount = *(int *)buffer;
    xsmcSetInteger(xsResult, bitCount);
}

```

Преимущества этого подхода

Эта вторая реализация BitArray, использующая память, выделенную `calloc`, имеет много преимуществ по сравнению с первой версией:

- Он проверяет входные значения, устраняя возможность небрежного кода вызвать сбой, а вредоносного кода — нарушить конфиденциальность.
- Предоставляет свойство длины, что делает работу с ним более удобной.
- Он использует системную память для хранения битовых данных, уменьшая объем памяти, используемой в куче памяти, управляемой механизмом JavaScript.
- Он использует функцию данных хоста XS в C для отслеживания буфера памяти, требуя меньше кода и работая быстрее, чем при использовании свойства JavaScript.

Уведомления о сигналах Wi-Fi

Вы узнали, как реализовать класс для управления собственными ресурсами в качестве основного объекта. В следующем примере показано, как выполнять вызовы из кода C обратно в JavaScript и как настраивать хост-объект с помощью словаря. Оба эти метода используются многими объектами хоста в Moddable SDK.

Пример `$EXAMPLES/ch11-native/wifi-rssi-notify` реализует класс `WiFiRSSINotify`, который позволяет регистрировать обратные вызовы для вызова, когда уровень сигнала Wi-Fi превышает или ниже указанного порогового значения. Вы можете использовать это в своем продукте, чтобы дать пользователю указание о том, когда Wi-Fi, вероятно, будет работать хорошо, или ограничить объем сетевого трафика, который вы генерируете, когда сигнал слабый. Этот класс можно полностью реализовать на JavaScript с использованием `Timer` вместе с модулем `net`, представленным в разделе «Получение информации о сети» главы 3. Эта реализация с использованием собственного кода немного эффективнее и обеспечивает удобную отправную точку для демонстрации того, как настроить ваш хост-объект из словаря и как вызывать функции обратного вызова.

При запуске этого примера необходимо указать точку доступа Wi-Fi для подключения микроконтроллера. Это потому, что RSSI измеряет уровень сигнала между вашим микроконтроллером и точкой доступа, к которой он подключен; если соединения нет, то и измерять нечего. Вот типичная командная строка для сборки и запуска этого примера:

```
> mcconfig -d -m -p esp32 ssid="My Wi-Fi" password="secret"
```

Тестовый код

Класс `WiFiRSSINotify` следует общему шаблону наличия конструктора, который принимает объект словаря параметров конфигурации. В листинге 11.14 показан тестовый код в `main.js`, который создает экземпляр этого класса. Вам необходимо указать порог RSSI, ниже которого сигнал считается слабым и при котором сигнал считается сильным. Необязательное свойство опроса настраивает частоту проверки уровня сигнала; в этом примере установлено значение 1000 миллисекунд. Частота опроса по умолчанию составляет 5000 миллисекунд.

Листинг 11-14.

```
import WiFiRSSINotify from "wifirssinotify";

let notify = new WiFiRSSINotify({
  threshold: -66,
  poll: 1000
});
```

После создания экземпляра уведомления вы можете установить обратный вызов `onWeakSignal` и/или `onStrongSignal`, как показано в листинге 11-15. Обратный вызов `onWeakSignal` вызывается, когда RSSI достигает или падает ниже указанного порога, а `onStrongSignal` вызывается, когда RSSI превышает порог. Функции вызываются при пересечении порога, а не при каждом опросе RSSI. Текущее значение RSSI передается функциям обратного вызова.

Листинг 11-15.

```
notify.onWeakSignal = function(rssi) {
  trace(`Weak Wi-Fi signal. RSSI ${rssi}.\n`);
}
notify.onStrongSignal = function(rssi) {
  trace(`Strong Wi-Fi signal. RSSI ${rssi}.\n`);
}
```

Класс WiFiRSSINotify

Класс JavaScript для WiFiRSSINotify — это просто хост-объект с деструктором, конструктором и функцией закрытия, реализованными в собственном коде:

```
class WiFiRSSINotify @ "xs_wifirssinotify_destructor"
{
  constructor(options) @ "xs_wifirssinotify_constructor";
  close() @ "xs_wifirssinotify_close";
}
```

Функции по умолчанию для обратных вызовов onWeakSignal и onStrongSignal не являются частью класса. Перед вызовом обратного вызова WiFiRSSINotify подтверждает, что у экземпляра есть свойство с именем обратного вызова.

Собственная структура RSSINotifyRecord

Класс WiFiRSSINotify должен поддерживать состояние для выполнения своей работы. Это состояние сохраняется в структуре языка C с именем RSSINotifyRecord, показанной в листинге 11-16. Вы можете думать об этой структуре данных как о C-эквиваленте свойств экземпляра JavaScript.

Листинг 11-16.

```
struct RSSINotifyRecord {      int          threshold;      int
state;      modTimer      timer;      xsMachine      *the;
xsSlot      obj;
};
```

Прежде чем рассматривать код, использующий эту структуру данных, полезно рассмотреть, как используется каждое поле:

- `threshold` — порог RSSI, ниже которого сигнал считается слабым, а при котором сигнал считается сильным.
- `state` — экземпляр `WiFiRSSINotify` всегда находится в одном из трех состояний: `kRSSIUnknown` при создании, а затем либо `kRSSIWeak`, либо `kRSSIStrong`. Это состояние используется для устранения избыточных обратных вызовов, когда состояние не изменилось.
- `timer` – встроенный таймер, используемый для реализации опроса.
- `the` – ссылка на виртуальную машину XS, содержащую экземпляр `WiFiRSSINotify`. Он используется для вызова обратных вызовов из таймера.
- `obj` — ссылка на объект `WiFiRSSINotify`, который используется для вызова обратных вызовов от таймера. Тип этого поля, `xsSlot`, используется XS для хранения любого значения JavaScript. Уже известные вам функции `xsArg`, `xsVar` и `xsGet` возвращают значения типа `xsSlot`.

Дополнительные сведения об использовании этих полей приведены в следующих разделах.

Реализация также определяет `RSSINotify` как указатель на `RSSINotifyRecord` для удобства:

```
typedef struct RSSINotifyRecord *RSSINotify;
```

Конструктор

Конструктор `WiFiRSSINotify` начинает с выделения памяти для структуры `RSSINotifyRecord`. После полной инициализации этой структуры она присоединяется к объекту с помощью `xsmcSetHostData`. Как правило, структура данных не прикрепляется к объекту перед инициализацией, чтобы избежать частично инициализированной структуры в случае возникновения ошибки во время выполнения конструктора.

```

RSSINotify rn = calloc(sizeof(RSSINotifyRecord), 1);
if (!rn)
    xsUnknownError("no memory");

```

Далее конструктор инициализирует поля `state`, `the` и `obj`:

```

rn->state = kRSSIUnknown;
rn->obj = xsThis;
rn->the = the;

```

Конструктор выполняет несколько операций, которые могут завершиться ошибкой. Когда они терпят неудачу, они выдают ошибку, которая может быть перехвачена вызывающим кодом JavaScript. Поскольку первая операция, которую выполняет конструктор, — выделение памяти, ему необходимо освободить эту память в случае возникновения исключения. Если этого не происходит, память теряется, вызывая утечку памяти, которая в конечном итоге может привести к сбою системы. Чтобы предотвратить это, конструктор окружает эти операции с помощью `xsTry`, перехватывая любые исключения с помощью `xsCatch`. После перехвата исключения конструктор освобождает память, хранящуюся в `rn`, а затем использует `xsThrow` для повторного выдачи ошибки. В С такое использование `xsTry` и `xsCatch` имеет структуру, показанную в листинге 11-17.

Листинг 11-17.

```

xsTry {
    ...
}
xsCatch {
    free(rn);
    xsThrow(xsException);
}

```


Вспомните, что XS в C предоставляет способы доступа и реализации основных возможностей JavaScript в вашем коде C. Код C для `xsTry-xsCatch` аналогичен версии кода JavaScript, показанной в листинге 11-18.

Листинг 11-18.

```
try {
    ...}
catch(e) {    ...    throw e;
}
```

Блок `xsTry` начинается с объявления локальной переменной `poll` для хранения запрошенного интервала опроса из аргумента словаря и использования `xsmcVars` для резервирования места для временного значения в стеке JavaScript:

```
int poll;xsmcVars(1);
```

Как показано в листинге 11.19, конструктор затем вызывает `xsmcHas`, чтобы проверить, содержит ли аргумент словаря свойство `poll`. Если это так, свойство извлекается, преобразуется в целое число и присваивается локальной переменной `poll`; в противном случае используется значение по умолчанию 5000.

Листинг 11-19.

```
if (xsmcHas(xsArg(0), xsID_poll)) {    xsmcGet(xsVar(0), xsArg
(0), xsID_poll);    poll = xsmcToInteger(xsVar(0));
}else
    poll = 5000;
```

Функция `xsmcHas` аналогична оператору `in`, используемому в JavaScript. Предыдущий код примерно такой же, как код JavaScript в листинге 11.20.

Листинг 11-20.

```
let poll;
if ("poll" in options)
    poll = options.poll; else
    poll = 5000;
```

Затем конструктор снова вызывает `xsmcHas`, на этот раз для подтверждения наличия требуемого порогового свойства. Если нет, выдается ошибка; в противном случае свойство порога JavaScript извлекается, преобразуется в целое число и присваивается полю порога `rn`.

```
if (!xsmcHas(xsArg(0), xsID_threshold))      xsUnknownError("
threshold required");
xsmcGet(xsVar(0), xsArg(0), xsID_threshold); rn->threshold =
xsmcToInteger(xsVar(0));
```

Наконец, блок `xsTry` выделяет собственный таймер с помощью `modTimerAdd` из Moddable SDK. Здесь вы можете использовать другой механизм таймера, специфичный для вашего микроконтроллера. Этот код использует `modTimerAdd` для удобства, поскольку он доступен как для устройств ESP32, так и для устройств ESP8266. Если таймер не может быть выделен — например, из-за недостаточной памяти — конструктор выдает исключение.

```
rn->timer = modTimerAdd(1, poll, checkRSSI, &rn, sizeof(rn));
if(!rn->timer) xsUnknownError("no timer");
```

Вызов `modTimerAdd` создает таймер, который сначала срабатывает через 1 миллисекунду, а затем срабатывает с интервалом, указанным в опросе. Когда таймер срабатывает, он вызывает собственную функцию `checkRSSI`, передавая ей значение `rn`. В следующем разделе показано, как собственный обратный вызов извлекает это значение и вызывает обратные вызовы JavaScript.

Это конец блока `xsTry`. Даже в этом относительно простом объекте есть два исключения, которые генерирует сам конструктор. Кроме того, вызовы `xsmcToInteger` вызывают исключения при передаче значения, которое нельзя преобразовать в целое число. Эти многочисленные возможности для исключений делают важным для конструктора гарантировать, что никакая память или другие ресурсы не будут потеряны, если возникнет исключение. В этом часто помогает использование `xsTry` с `xsCatch`.

В конструкторе осталось еще два шага. Первый — сохранить указатель данных `rn` вместе с объектом:

```
xsmcSetHostData(xsThis, rn);
```

Во-вторых, гарантировать, что объект будет удален сборщиком мусора только после того, как код JavaScript вызовет метод `close` для объекта. Такое поведение характерно для хост-объектов JavaScript, поддерживающих обратные вызовы. Для этого конструктор вызывает функцию `xsRemember` с объектом, сохраненным в файле `RSSINotifyRecord`.

```
xsRemember(rn->obj);
```

Вы можете передать `xsRemember` только значение в хранилище, выделенное вашим кодом. Если вы вызываете `xsRemember` с такими значениями, как `xsThis`, `xsArg(0)`, `xsVar(1)` или другими значениями, предоставленными XS, он автоматически завершается ошибкой. Как и следовало ожидать, есть соответствующий вызов `xsForget`, который необходимо вызвать в процессе закрытия. Память, в которой хранится объект, здесь `rn->obj`, должна сохраняться до вызова `xsForget` и, следовательно, не должна быть локальной переменной в конструкторе.

Деструктор (Разрушитель)

Деструктор для WiFiRSSINotify (листинг 11-21) подобен другим деструкторам в этой главе, но с добавлением кода для освобождения таймера, выделенного в конструкторе. Чтобы получить доступ к таймеру в структуре RSSINotifyRecord, аргумент указателя данных приводится к указателю RSSINotify. Реализация конструктора гарантирует, что поле таймера никогда не будет NULL в деструкторе, если rn не равно NULL. Поэтому нет необходимости проверять, что rn->timer не равен NULL перед вызовом modTimerRemove.

Листинг 11-21.

```
void xs_wifirssinotify_destructor(void *data){
    RSSINotify rn = data;if (rn) {
        modTimerRemove(rn->timer);free(rn);
    }}
```

Функция close

Метод close WiFiRSSINotify (листинг 11.22) также следует знакомой схеме. Однако вдобавок он должен вызывать xsForget, чтобы сделать объект подходящим для сборки мусора, противодействуя вызову xsRemember в конструкторе. Поскольку вызов xsForget обращается к полю obj из rn, реализация close должна защищать от повторного вызова, проверяя, возвращает ли xsmcGetHostData значение, отличное от NULL.

Листинг 11-22.

```

void xs_wifirssinotify_close(xsMachine *the)
{
    RSSINotify rn = xsmcGetHostData(xsThis);
    if (rn) {
        xsForget(rn->obj);
        xs_wifirssinotify_destructor(rn);
        xsmcSetHostData(xsThis, NULL);
    }
}

```

Вызов `xsForget` не может быть сделан в деструкторе, потому что деструктор не может использовать XS в C, как объяснялось ранее.

Функция обратного вызова

Функция `checkRSSI`, показанная в листинге 11.23, лежит в основе класса `WiFiRSSINotify`. Он вызывается в интервале опроса, чтобы определить, когда значение RSSI пересекает указанное пороговое значение. Функция начинается с восстановления значения `rn`, указателя на структуру `RSSINotifyRecord`, выделенную в конструкторе. Поскольку обратный вызов `checkRSSI` вызывается не непосредственно XS, а `modTimer`, указатель не может быть получен с помощью `xsmcGetHostData`, как обычно, а вместо этого извлекается путем разыменования аргумента `refcon`.

Листинг 11-23.

```

void checkRSSI(modTimer timer, void *refcon, int refconSize)
{
    RSSINotify rn = *(RSSINotify *)refcon;
    ...
}

```

Следующим шагом является получение текущего значения RSSI, что делается по-разному на ESP32 и ESP8266. Листинг 11-24 содержит условные случаи `foreach` и ошибку для других целей.

Листинг 11-24.

```
int rssi = 0;

#ifdef ESP32
    wifi_ap_record_t config;

    if (ESP_OK == esp_wifi_sta_get_ap_info(&config))    rssi =
        config.rssi;
#elif defined(__ets__)
    rssi = wifi_station_get_rssi();#else
    #error Unsupported target#endif
```

Как показано в листинге 11-25, функция опроса использует текущее значение RSSI, чтобы решить, необходимо ли вызывать функцию обратного вызова `onStrongSignal` или `onWeakSignal` JavaScript. Он проверяет, находится ли текущее значение выше или ниже заданного порога, хранящегося в `rn->threshold`. Если значение RSSI находится на той же стороне порога, что и предыдущая проверка, `checkRSSI` возвращается немедленно; в противном случае он обновляет `rn->state` до нового состояния и присваивает идентификатор вызываемого обратного вызова, либо `xsID_onStrongSignal`, либо `xsID_onWeakSignal`, локальной переменной `callbackID`.

Листинг 11-25.

```
if (rssi > rn->threshold) {
    if (kRSSIStrong == rn->state)    return;
    rn->state = kRSSIStrong;
```

```

    callbackID = xsID_onStrongSignal;
}
else {
    if (kRSSIWeak == rn->state)
        return;
    rn->state = kRSSIWeak;
    callbackID = xsID_onWeakSignal;
}

```

Для вызова функции JavaScript из собственного кода требуется допустимый кадр стека JavaScript. Когда собственный метод вызывается из JavaScript, XS уже создал этот кадр стека. Функция `checkRSSI` вызывается не XS, а `modTimer`, и, следовательно, должна сама устанавливать кадр стека. Она делает это, вызывая `xsBeginHost` перед обратным вызовом. После этого он вызывает `xsEndHost`, чтобы удалить кадр стека, созданный `xsBeginHost`. Обе функции принимают ссылку на виртуальную машину JavaScript в качестве единственного аргумента. Между `xsBeginHost` и `xsEndHost` вы можете выполнять вызовы XS в C, как обычно.

Код в листинге 11-26 создает временную переменную JavaScript с помощью `xsmcVars(1)` и присваивает ей целочисленное значение `rssI` с помощью `xsmcSetInteger`. Затем он вызывает `xsmcHas`, чтобы подтвердить, что объект имеет функцию обратного вызова. Если это так, он использует `xsCall1` для вызова функции обратного вызова, передавая значение `RSSI`, хранящееся в `xsVar(0)`.

Листинг 11-26.

```

xsBeginHost(rn->the);
    xsmcVars(1);
    xsmcSetInteger(xsVar(0), rssI);
    if (xsmcHas(rn->obj, callbackID))
        xsCall1(rn->obj, callbackID, xsVar(0));
xsEndHost(rn->the);

```

Вы используете `xsCall1` для вызова функций с одним аргументом (и `xsCall0` для вызова функций без аргументов, `xsCall2` для функций с двумя аргументами и т. д., вплоть до `xsCall9`).

Дополнительные методы

Теперь вы знаете, как вызывать собственный код из кода JavaScript и код JavaScript из собственного кода, что дает вам возможность интегрировать собственный код и сценарии любым способом, наиболее подходящим для вашего проекта. В этом разделе кратко представлены несколько важных тем, которые могут оказаться полезными при интеграции нативного кода в ваши собственные продукты на базе JavaScript. Наряду с обсуждением различных методов, которые помогут вам построить мост между вашим собственным кодом и кодом JavaScript, он включает предупреждения о некоторых распространенных ошибках.

Отладка нативного кода

По мере разработки все более сложного машинного кода вам может потребоваться отладка этого кода. Хотя у вас может не быть встроенного отладчика, ваш код может взаимодействовать с `xsbug`.

Распространенным методом отладки является отправка диагностических выходных данных на консоль отладки. Во встроенном JavaScript для этого используется трассировка. Используя XS в C, вы можете сделать то же самое с `xsTrace`.

```
xsTrace("about to get RSSI\n");
```

Аргументом `xsTrace` является строка, что позволяет удобно отображать ход выполнения функции. Если вам нужно вывести более подробную информацию, используйте `xsLog`, который обеспечивает функциональность в стиле `printf`.

```
xsLog("RSSI is %d.\n", rssi);
```


И для `xsTrace`, и для `xsLog` требуется допустимый кадр стека XS; поэтому они должны вызываться либо из метода, вызываемого непосредственно XS, либо между парой `xsBeginHost`-`xsEndHost`. Например, чтобы вывести текущий уровень RSSI в консоль отладки из обратного вызова `checkRSSI`, вы используете этот код:

```
xsBeginHost(rn->the);
xsEndHost(rn->the);
xsLog("RSSI is %d.\n", rssi);
```

Может быть полезно активировать точку останова в `xsbug` из вашего собственного кода, чтобы увидеть кадры стека, ведущие к вызову вашей собственной функции, и переданные ей аргументы. Хотя вы не можете установить точку останова в собственном коде с помощью `xsbug`, вы можете активировать точку останова, вызвав `xsDebugger` в своем коде C.

```
xsDebugger();
```

Доступ к глобальным переменным

Ваш код может получать и устанавливать значения глобальных переменных напрямую. Все глобальные переменные являются частью глобального объекта, доступ к которому в JavaScript осуществляется с помощью `globalThis`. В XS в C глобальный объект доступен для вашего собственного кода как `xsGlobal`. Вы можете использовать `xsGlobal` в своем собственном коде, как и любой другой объект. Например, вы используете функции `xsmcSet*` для присвоения значений глобальной переменной, а следующие строки устанавливают статус глобальной переменной в 0x8012:

```
xsmcSetInteger(xsVar(0), 0x8012);
xsmcSet(xsGlobal, xsID_status, xsVar(0));
```

Вы получаете значение глобального с помощью `xsmcGet`:

```
xsmcGet(xsVar(0), xsGlobal, xsID_status);
int status =xsmcToInteger(xsVar(0));
```

Следующий код проверяет, существует ли глобальная переменная с именем `onRestart`. Если есть, он вызывает функцию, хранящуюся в глобальном `onRestart`.

```
if (xsmcHas(xsGlobal, xsID_onRestart))  
    xsCall0(xsGlobal, xsID_onRestart);
```

Получение возвращаемого значения функции

Когда вы используете семейство функций `xsCall*` для вызова функции JavaScript из C, вы можете получить доступ к возвращаемому значению, назначив результат значению JavaScript. Например, следующий код вызывает функцию для свойства обратного вызова `this` и отслеживает результат на консоли:

```
xsmcVars(1);  
xsVar(0) = xsCall0(xsThis, xsID_callback);  
xsTrace(xsVar(0));
```

Получение значений

В примерах этой главы `xsmcToInteger` используется для получения целочисленного значения из значения JavaScript. Существуют аналогичные функции для получения логического значения, числа с плавающей запятой, строки и буфера массива из значения JavaScript, как показано в листинге 11.27.

Листинг 11-27.

```
uint8_t boolean = xsmcToBoolean(xsArg(0));  
double number = xsmcToNumber(xsArg(1));  
const char *str = xsmcToString(xsArg(2));  
uint8_t *buffer = xsmcToArrayBuffer(xsArg(3));  
int bufferLength = xsmcGetArrayBufferLength(xsArg(3));
```

Все эти функции терпят неудачу, если значение JavaScript не может быть преобразовано в запрошенный тип. Например, `xsmcToArrayBuffer` завершается ошибкой, если значение является строкой.

Особая осторожность требуется при работе с указателями на строки и с указателями `ArrayBuffer`. Подробнее см. в разделе «Обеспечение корректности указателей буфера».

Установка значений

Вы уже видели, как использовать `xsmcSetInteger` для установки свойства JavaScript в целочисленное значение. Кроме того, существуют функции `xsmcSet*` для установки других базовых значений JavaScript, как показано в листинге 11.28.

Листинг 11-28.

```
xsmcSetNull(xsResult);
xsmcSetUndefined(xsVar(0));

xsmcSetBoolean(xsVar(2), value);
xsmcSetTrue(xsVar(3));
xsmcSetFalse(xsResult);

xsmcSetNumber(xsResult, 1.2);

xsmcSetString(xsResult, "off");

const char *string = "a dog!";
xsmcSetStringBuffer(xsResult, string + 2, 3); // "dog"
```

Вы также можете создавать объекты с помощью XS в C. Следующий код создает объект `ArrayBuffer` размером 16 байтов и устанавливает первый байт равным 1:

```
xsmcSetArrayBuffer(xsResult, NULL, 16);
uint8_t *buffer = xsmcToArrayBuffer(xsResult);
buffer[0] = 1;
```

В листинге 11.29 создается объект и добавляются к нему несколько свойств. Используя этот подход, ваш код может возвращать объекты точно так же, как это делает следующий метод класса File.

Листинг 11-29.

```
xsmcSetNewObject(xsResult);  
  
xsmcSetString(xsVar(0), "test.txt");  
xsmcSet(xsResult, xsID_name, xsVar(0));  
  
xsmcSetInteger(xsVar(0), 1024);  
xsmcSet(xsResult, xsID_length, xsVar(0));
```

Эквивалент этого кода в JavaScript выглядит следующим образом:

```
return {name: "test.txt", length: 1024};
```

В листинге 11.30 создается массив из восьми элементов и используется `xsmcSet`, чтобы установить для каждого элемента массива квадрат его индекса. Вы уже видели, как `xsmcSet` используется для установки значения свойства объекта; здесь он используется для установки значения элемента массива путем передачи индекса элемента вместо идентификатора символа в стиле `xsID_*`.

Листинг 11-30.

```
xsmcSetNewArray(xsResult, 8);  
  
for (i = 0; i < 8; i++) {  
    xsmcSetInteger(xsVar(0), i * i);  
    xsmcSet(xsResult, i, xsVar(0));  
}
```

Определение типа значения

Ваш собственный код иногда должен знать тип значения JavaScript. Например, некоторые функции меняют свое поведение в зависимости от того, является ли аргумент объектом или числом. Вы используете `xsmcTypeOf` для определения базового типа значения.

```
int typeOf = xsmcTypeOf(xsArg(1));
if (xsStringType == typeOf)
    ...;
```

`xsmcTypeOf` возвращает следующие типы: `xsUndefinedType`, `xsNullType`, `xsBooleanType`, `xsIntegerType`, `xsNumberType`, `xsStringType` и `xsReferenceType`. Большинство из них напрямую соответствуют типам JavaScript, с которыми вы уже знакомы. Обратите внимание, однако, что существуют типы как для целых чисел, так и для чисел (значения с плавающей запятой). В то время как сам JavaScript использует тип `Number` для обоих, XS сохраняет их как отдельные типы в целях оптимизации. Если ваш собственный код проверяет, относится ли значение JavaScript к типу `Number`, ему необходимо проверить как `xsIntegerType`, так и `xsNumberType`.

Тип `xsReferenceType` соответствует объекту JavaScript. Эта константа одного типа используется для всех объектов JavaScript. Вы используете функцию `xsmcIsInstanceOf`, чтобы определить, является ли объект экземпляром определенного класса. Тип `xsmcIsInstanceOf` похож на оператор `instanceof` в JavaScript. XS определяет значения для встроенных объектов, например `xsArrayPrototype`. Следующий код устанавливает для переменной `isArray` значение 1, если первый аргумент нативного метода является массивом, или 0, если это не так:

```
int typeOf = xsmcTypeOf(xsArg(0));
int isArray = (xsReferenceType == typeOf) &&
    xsmcIsInstanceOf(xsArg(0), xsArrayPrototype);
```

Функция `xsmcIsInstanceOf` возвращает значение `true`, если объект является подклассом указанного типа. Например, в разделе «Доступ к значениям представления данных» в главе 2 класс `Header` определяется как подкласс `DataView`.

Передача экземпляра `Header` следующему вызову возвращает `true`:

```
if (xsmcIsInstanceOf(xsArg(0), xsDataViewPrototype))
    ...;    // это представление данных
```

Другие полезные прототипы, определенные XS, которые можно использовать с `xsmcIsInstanceOf`, включают `xsFunctionPrototype`, `xsDatePrototype`, `xsErrorPrototype` и `xsTypedArrayPrototype`. Полный список см. в заголовочном файле `xs.h` в Moddable SDK.

Работа со строками

Строки обычно используются в JavaScript. Поскольку XS хранит их в кодировке UTF-8, со строками удобно работать в C. Вот несколько деталей, о которых следует помнить:

- Вам гарантируется, что строки, которые вы получаете от XS, являются действительными UTF-8. Вы должны убедиться, что все строки, которые вы передаете XS, также являются допустимыми UTF-8.
- XS обрабатывает нулевой символ (ASCII 0) как конец строки, поэтому не включайте в свои строки нулевые символы. (Поскольку язык C также использует нулевой символ для завершения строки, это должно быть знакомо.) Ваш код, вероятно, не создает намеренно недопустимые строки UTF-8 или не включает нулевые символы в строку, но они могут прокрасться, когда вы импортируете строки из файла или сетевого подключения; рекомендуется проверять эти строки перед передачей их в XS.
- В JavaScript строки доступны только для чтения. Не предусмотрено никаких функций для изменения содержимого строки. Вы можете нарушить это правило в своем нативном коде, но не делайте этого!

Это нарушило бы фундаментальное допущение, на которое полагаются программисты JavaScript. Кроме того, это может привести к сбою, поскольку некоторые строки хранятся во флэш-памяти, доступной только для чтения, и попытка записи в них приводит к перезагрузке микроконтроллера.

- Указатель строки, возвращенный из `xsmcToString`, может стать недействительным при выполнении других вызовов с помощью XS в C. Подробности объясняются в следующем разделе.

Обеспечение корректности указателей буфера

Когда вы вызываете `xsmcToString` или `xsmcToArrayBuffer`, они не возвращают копию данных; они возвращают указатель на структуру данных XS. Такое поведение важно для микроконтроллеров, где дополнительное время и память, необходимые для создания копии, неприемлемы. Указатель может стать недействительным, когда вы вызываете XS в C, что приводит к запуску сборщика мусора. Сборщик мусора не может освободить `ArrayBuffer` или строку, поскольку они уже используются. Однако сборщик мусора может переместить структуру данных, когда он сжимает кучу памяти, чтобы освободить больше места за счет объединения областей свободного пространства.

С некоторой осторожностью, как в следующих подходах, вы можете избежать любых проблем, когда сборщик мусора уплотняет кучу:

- Никогда не используйте указатель, возвращенный XS в C, после другого вызова XS в C. Это может показаться сложным, но все примеры, приведенные до сих пор в этой главе, делали именно это.
- Сделайте копию данных. Хотя этот подход не является оптимальным, иногда он необходим.

Две функции могут помочь при работе с указателями на строки и указателями `ArrayBuffer`. Функция `xsmcToStringBuffer`

аналогична `xsmcToString`, но вместо возврата указателя на строку она копирует строку в буфер. Если буфер слишком мал для хранения строки, выдается ошибка `RangeError`.

```
char str[40];
xsmcToStringBuffer(xsArg(0), str, sizeof(str));
```

Функция `xsmcGetArrayBufferData` копирует весь или часть `ArrayBuffer` в другой буфер. Второй аргумент — это смещение `ArrayBuffer` (в байтах), с которого следует начать копирование данных, третий аргумент — буфер назначения, а последний аргумент — размер буфера назначения в байтах. В этом примере пять байтов, начиная со смещения 10, копируются из `ArrayBuffer` в буфер локальной переменной.

```
uint8_t buffer[5];
xsmcGetArrayBufferData(xsResult, 10, buffer, sizeof(buffer));
```

Интеграция с C++

XS в C позволяет вам соединять не только код C и JavaScript, но и код C++ и JavaScript. Хотя и JavaScript, и C++ поддерживают объекты, детали того, как они реализуют объекты и их функции, сильно различаются. Поэтому обычно нереально пытаться создать прямое сопоставление между вашими классами C++ и вашими классами JavaScript. Вместо этого создавайте свои классы JavaScript так, чтобы они были понятны программистам JavaScript, а классы C++ — так, чтобы они были понятны программистам C++. Код моста, который вы пишете с помощью XS в C, может переводиться между ними.

Использование потоков

Две функции могут помочь при работе с указателями на строки и указателями `ArrayBuffer`. Функция `xsmcToStringBuffer` аналогична тому же потоку или задаче. Вы не должны вызывать XS в C из прерывания или потока, отличного от того, который создал виртуальную машину.

Методы, обеспечивающие многозадачное выполнение кода JavaScript, такие как класс `Web Workers`, построены вне языка JavaScript. `Moddable SDK` поддерживает подмножество класса `Web Workers` на ESP32, что позволяет нескольким виртуальным машинам JavaScript сосуществовать, каждая в своем собственном потоке. Каждая виртуальная машина является однопоточной, но несколько машин могут работать параллельно. Реализация `Web Workers` для ESP32 соответствует требованию однопоточности каждой отдельной виртуальной машины JavaScript.

Заключение

Возможность соединять JavaScript и нативный код с помощью `XS in C API` открывает множество новых возможностей для ваших проектов. Он позволяет оптимизировать использование памяти, повысить производительность, повторно использовать существующие библиотеки кода C и C++ и получить доступ к уникальным аппаратным возможностям. Однако использование `XS` в C значительно сложнее, чем работа в JavaScript, и, следовательно, более подвержено ошибкам. Как правило, использование как можно меньшего количества нативного кода минимизирует риски.

Чтобы помочь вам узнать больше о работе с `XS` в C, доступны эти два превосходных ресурса:

- Документация `XS in C` — это полный справочник по API. Это часть модифицируемого SDK.
- Все классы в `Moddable SDK`, которые имеют доступ к собственным возможностям, реализованы с использованием `XS` в C. Если вам интересно узнать, как они работают, вы можете прочитать исходный код и изучить его.

Глоссарий

absolute coordinates (в Piu)

Координаты объекта содержимого на экране, выраженные как расстояние от краев экрана. См. также **relative coordinates** - относительные координаты.

access point

В этой книге точка соединения между вашей сетью Wi-Fi и Интернетом; также называется базовой станцией (или маршрутизатором). Точка доступа создает локальную сеть, которая позволяет подключенным к ней устройствам взаимодействовать напрямую, без использования Интернета.

alpha channel

Указание степени прозрачности (или непрозрачности) сплошного цвета или пикселей в цветном изображении: какие пиксели следует рисовать, какие следует пропускать, а какие следует смешивать с фоном.

anchor (в Piu)

Ссылка на объект содержимого, сохраненный как свойство в данных создания экземпляра объекта содержимого.

ArrayBuffer

В **JavaScript** - блок памяти с фиксированным числом байтов, без указания типа, связанного с данными в нем. Память инициализируется **0**. Чтобы получить доступ к данным, вы заключаете **ArrayBuffer** в представление; см. также представление данных и типизированный массив.

arrow function

В современном *JavaScript* — компактный синтаксис для объявления функций (с использованием синтаксиса `=>`), которые при вызове имеют значение *this*, совпадающее со значением *this* функции, в которой определена стрелочная функция; формально называется лямбда-функцией.

asynchronous networking

See **non-blocking networking**.

bare module specifier

В *JavaScript* — спецификатор модуля, который не является путем. В этой книге используются только голые спецификаторы модулей, которые более распространены во встроенном *JavaScript*.

base station

Смотри **access point**.

behavior (в Piu)

Набор методов, определяющих действия, которые должен выполнять объект содержимого в ответ на события; в частности, экземпляр подкласса класса *Behavior*, который назначен объекту содержимого `t`.

BLE (Bluetooth Low Energy)

Протокол беспроводной связи с низким энергопотреблением, который широко используется между двумя устройствами, находящимися в непосредственной близости друг от друга.

block (в флеш-памяти)

Организационная единица для флэш-памяти. Размер блока зависит от используемого компонента флэш-памяти; общее значение равно 4096 байт.

blocking networking - блокировка сети

Сеть, в которой устройство не будет реагировать на ввод данных пользователем во время работы сети, если только не используется более сложный и ресурсоемкий метод, такой как потоки.

BMFont

Формат шрифта, объединяющий растровое изображение с файлом карты. Существует несколько вариантов BMFont; Moddable SDK использует двоичный формат BMFont.

bubble an event (в Piu)

Чтобы инициировать указанное событие для объекта содержимого, его родительского контейнера и всех объектов-контейнеров вверх по иерархии вложений, используйте метод пузыря объекта.

Central (в GAP)

Одна из двух основных ролей, определенных GAP. Центральный сканирует устройства, действующие как периферийные устройства, и инициирует запросы на установление нового соединения с периферийными устройствами.

characteristic (в GATT)

В иерархии GATT, определяющей формат данных, значение услуги GATT. См. также **profile** и **service**.

child object, child (в Piu)

В иерархии включения - объект содержимого, добавленный в контейнер, который считается его родительским объектом. Полный срок часто сокращается до ребенка - *child*.

chunked transfer encoding - кодирование передачи по частям

Функция протокола HTTP, которая часто используется для доставки больших ответов. Класс HTTP Request декодирует фрагменты перед вызовом функции обратного вызова, поэтому вашей функции обратного вызова не нужно анализировать заголовки фрагментов.

claiming (в mDNS)

Процесс, при котором устройство проверяет, не используется ли уже выбранное вами имя mDNS, поскольку mDNS требует, чтобы каждое устройство имело уникальное имя..

Client (в GATT) См. GATT Client.

clipping

Ограничение рисования подразделами дисплея. Он используется Pico для реализации рендеринга строки сканирования и Piu для частичного обновления кадров, а также доступен для использования в ваших приложениях — например, для рисования подмножества изображения. См. также область отсечения - **clipping area**.

clipping area - область отсечения

Область, которая ограничивает, где будет происходить рисование в области обновления; один прямоугольник в Pico, но, возможно, несколько прямоугольников в Piu. Рисуются только та часть каждой операции рисования, которая пересекает область отсечения.

closure

В JavaScript - привязка функции к группе переменных вне функции. Ссылки на внешние переменные сохраняются в течение всего времени существования замыкания.

Commodetto

Графическая библиотека, включающая Росо и добавляющая такие функции, как внеэкранные графические буферы, растровые изображения и создание экземпляров графических ресурсов из ресурсов.

constructor

Особый вид функции, которая при вызове с `new` создает экземпляр типа, указанного конструктором. Функция конструктора выполняется для инициализации экземпляра.

контейнер (в Piu)

Организующий элемент вмещающей иерархии — группа, в которую помещаются объекты контента внутри иерархии; в частности, экземпляр любого класса, наследуемого от класса `Container`, который наследуется от класса `Content` и расширяет его, позволяя содержать другие объекты содержимого.

containment hierarchy - иерархия сдерживания (in Piu)

Дерево объектов контента с объектом приложения в корне, которые составляют пользовательский интерфейс приложения `Piu`.

content object - объект содержимого (in Piu)

Объект JavaScript, связанный с графическим элементом пользовательского интерфейса. в частности, экземпляр любого класса, наследуемого от класса `Content`.

CSS (Cascading Style Sheets - каскадные таблицы стилей)

Язык для определения стилей (например, текста), чаще всего используемый на веб-страницах. `Piu` включает в себя множество соглашений CSS, чтобы обеспечить согласованность для разработчиков, работающих как над веб-продуктами, так и над продуктами IoT.

data view - просмотр данных

Представление, в которое вы можете обернуть `ArrayBuffer` для доступа к данным в нем. В отличие от типизированных массивов, в которых все значения имеют один и тот же тип, представления данных используются для чтения и записи целых чисел разного размера и значений с плавающей запятой в буфер.

defer an event - отложить событие (в Piu)

Чтобы инициировать указанное событие в объекте содержимого на следующей итерации цикла обработки событий, используя метод объекта `defer`.

delegate an event - делегировать событие (in Piu)

Чтобы немедленно инициировать указанное событие в объекте контента, используя метод делегата объекта.

distribute an event - распространять событие (in Piu)

Чтобы инициировать указанное событие в контейнере и всех объектах содержимого ниже по иерархии включения, используя метод распределения контейнера.

easing equations - уравнения ослабления

Уравнения, которые реализуют общие ускорения и замедления для анимированных изменений состояния; также известные как функции смягчения. Они часто используются, чтобы придать анимации более естественный вид, заставляя что-то двигаться медленнее в начале или ближе к концу.

event (в Piu)

Событие, такое как прикосновение к показателю значения или исчезновение изменения таймера, которое может быть достигнуто или достигнуто чрезмерным действием.

extended mode - расширенный режим (для датчика TMP102)

Режим, который увеличивает разрешение TMP102 с 12 бит по умолчанию до 13 бит, позволяя измерять температуры до 150°C. от подростков (в Пиу)

from-tween (in Piu)

Анимация движения, добавленная на временную шкалу методом временной шкалы from, который изменяет свойства целевого объекта со значений, указанных в объекте свойств, на исходные значения целевого объекта в течение миллисекунд.

GAP (Generic Access Profile - общий профиль доступа)

Уровень протокола BLE, который определяет, как устройства рекламируют себя, как они устанавливают соединения друг с другом и безопасность их соединения.

GATT (Generic Attribute Profile - общий профиль атрибутов)

Уровень протокола BLE, который определяет, как устройства BLE передают данные туда и обратно после того, как между ними установлено соединение — отношения клиент-сервер.

GATT Client

Устройство BLE, которое получает доступ к данным с удаленного сервера GATT, отправляя запросы на чтение/запись.

GATT Server

Устройство BLE, которое хранит данные локально, получает запросы на чтение/запись и уведомляет удаленный клиент GATT об изменении значений его характеристик.

GLOSSARY

high-level event - мероприятие высокого уровня (в Piu)

Событие, определяемое и запускаемое вашим приложением с использованием любого имени, например, событие `onSensorValueChanged`, которое должно запускаться при изменении значения датчика. См. также **low-level event** - низкоуровневое событие. .

host

Коллекция модулей JavaScript, переменных конфигурации и другого доступного программного обеспечения, которое составляет среду, в которой выполняется ваш код. У каждой главы в репозитории этой книги на GitHub есть собственный хост, который содержит программную среду, необходимую для запуска примеров из этой главы.

host object - хост-объект

В XS - объект с собственным деструктором, который выполняет очистку, когда экземпляр класса удаляется сборщиком мусора.

I²C

Последовательный протокол для подключения нескольких устройств к одной двухпроводной шине.

I²S

Протокол для подключения цифровых аудиоустройств; один из двух аппаратных протоколов, поддерживаемых классом `AudioOut`. Он передает немодифицированные аудиосэмплы по цифровому соединению от микроконтроллера к выделенному компоненту аудиовыхода, который выполняет цифро-аналоговое преобразование. См. также **PDM**.

immediate mode rendering - рендеринг в немедленном режиме

Метод рендеринга, используемый в большинстве графических библиотек, используемых для микроконтроллеров, который выполняет запрошенную операцию рисования, когда вы вызываете функцию рисования. См. также **retained mode rendering** рендеринг в сохраненном режиме. .

immutable

неизменный Характеристика объектов и значений JavaScript, означающая, что они доступны только для чтения, то есть их нельзя изменять. Строки JavaScript неизменяемы и поэтому не могут быть изменены на месте.

instantiating data - создание экземпляров данных (в Piu)

При вызове конструктора объекта содержащего значение или объект JavaScript передается в качестве первого аргумента и используется для создания экземпляра класса. Эти данные также передаются в метод onCreate поведения созданного экземпляра.

iterator protocol - протокол итератора

Протокол, определенный JavaScript и реализованный классом Iterator файлового модуля, который предоставляет стандартный способ реализации и использования итераторов. Например, он позволяет использовать циклы for-of. лямбда-функция См. arrow function - функцию стрелки.

lambda function See **arrow function**.**lexical this**

Особенность стрелочных функций, при которой значение this внутри стрелочной функции берется из объемлющей функции.

low-level event - событие низкого уровня (в Piu)

Событие, определенное и запускаемое Piu, например событие onTouchBegan, когда палец помещается на объект содержащего, и событие onTouchEnded, когда палец удаляется. См. также мероприятие высокого уровня - **high-level event**.

mDNS (многоадресный DNS)

Протокол, производный от DNS (системы доменных имен), позволяющий устройствам легко подключаться друг к другу в локальной сети. В то время как DNS представляет собой централизованную структуру, которая зависит от уполномоченных серверов для сопоставления имен с IP-адресами, mDNS является децентрализованным, и каждое отдельное устройство отвечает на запросы о сопоставлении своего имени с IP-адресом.

MQTT (Message Queuing Telemetry Transport)

Сетевой протокол публикации и подписки (предназначенный для использования облегченными клиентскими устройствами IoT), который упорядочивает сообщения по темам.

non-blocking networking

Характеристика сетевых API, означающая, например, что когда вы запрашиваете данные из сети с использованием протокола HTTP, ваш код продолжает выполняться, пока выполняется запрос; также называется асинхронной сетью. Вот как работает сеть, когда вы используете JavaScript в Интернете.

однократный режим (для датчика TMP102)

Режим для снятия только одного показания температуры с датчика TMP102. Самый энергоэффективный способ снятия нечастых показаний, он доступен только тогда, когда устройство находится в режиме отключения.

parent object, parent (в Piu)

В иерархии вложенности - контейнер, в который добавлен объект содержимого (называемый его дочерним объектом). Полный срок часто сокращается до родителя.

раздел (во флэш-памяти)

Организационная единица флэш-памяти, доступная вашему микроконтроллеру. Например, один раздел содержит код вашего проекта, другой — данные настроек, а третий — хранилище для файловой системы SPIFFS. Каждый раздел идентифицируется именем.

PDM ((модуляция плотности импульсов)

Быстрое переключение цифрового выходного контакта для создания уровней энергии, соответствующих желаемому выходному сигналу; один из двух аппаратных протоколов, поддерживаемых классом AudioOut. См. также I2S.

Peripheral (в GAP)

Одна из двух основных ролей, определенных GAP. Периферийные устройства сообщают о себе централям и принимают от них запросы на установление соединения.

Piu

В Moddable SDK — объектно-ориентированная структура пользовательского интерфейса, которая упрощает процесс создания сложных пользовательских взаимодействий и использует POCO для рисования.

POCO

В Moddable SDK — механизм рендеринга для встроенных систем, который можно использовать для рисования на дисплеях. См. **Piu**.

port (в Piu)

Объект содержимого, который позволяет вам выполнять команды рисования, аналогичные POCO, в макете aPiu; экземпляр класса Port.

private method

Метод, который можно вызвать только из реализации класса.

профиль (в GATT)

В GATT - верхний уровень иерархии, определяющий формат данных. Профиль определяет конкретное использование BLE для связи между несколькими устройствами, включая роли задействованных устройств и их общее поведение. См. также характеристики и услуги **characteristic** и **service**.

promise

Функция современного JavaScript, предоставляющая альтернативу обратным вызовам для упрощения асинхронного программирования. Обратные вызовы можно превратить в обещания с помощью небольших вспомогательных функций, чтобы приложения могли использовать асинхронные функции.

property

Характеристика объекта JavaScript, похожая на поле в С или С++, за исключением того, что вы можете добавлять свойства к объекту во время выполнения (без необходимости объявлять их заранее).

PWM (широтно-импульсная модуляция)

Тип цифрового сигнала, при котором цифровой вывод выводит прямоугольную волну с различной шириной высоких и низких значений. Усреднение этих высоких и низких импульсов во времени создает уровень мощности между высокими и низкими значениями, пропорциональный ширине импульса.

relative coordinates - относительные координаты (в Piu)

The coordinates of a content object relative to the object's parentКоординаты объекта содержимого относительно родительского контейнера объекта, выраженные в виде отступов от краев контейнера; свойства left, right, top и bottom задают поля от соответствующего края. См. также **absolute coordinates** - **абсолютные координаты**.

responsive layout - адаптивный макет (в Piu)

Правило макета, которое разумно адаптируется к изменениям размера родительского контейнера.

rest parameters - остальные параметры

Функция современного JavaScript, предоставляющая функции, аналогичные специальной переменной arguments, объединяющая несколько аргументов в массив, но всегда доступная и более гибкая.

рендеринг в сохраненном режиме (в Poco)

Техника рендеринга, которая не рисует сразу, а поддерживает список команд рисования, который выполняется только тогда, когда вы сообщаете Poco, что закончили рисовать. См. также **immediate mode rendering**.

роутер См. точку доступа - **access point**.

RSSI (received signal strength indication - индикатор уровня принимаемого сигнала)

Мера мощности сигнала, полученного от точки доступа Wi-Fi.

scanline rendering - рендеринг строки сканирования (в Poco)

Метод рендеринга, который делит кадр на горизонтальные полосы размером в одну строку пикселей и после рендеринга каждой полосы немедленно передает ее на дисплей.

Server (в GATT)

См. **GATT Server**.

обслуживание (в GATT)

В иерархии GATT, определяющей формат данных, набор характеристик, описывающих поведение части устройства BLE. См. также характеристику и профиль, режим отключения (для датчика TMP102)

режим отключения (для датчика TMP102)

Режим, который полностью отключает аппаратное обеспечение преобразования температуры в датчике TMP102, снижая потребление энергии до 0,5 мкА. Ваше приложение может перевести TMP102 в режим отключения в промежутке между показаниями.

skin (в Piu)

Объект, управляющий отрисовкой фона одного или нескольких объектов содержимого, заполняющий область цветом или изображениями. Экземпляр класса Skin, указанный в свойстве skin объекта содержимого. См. также **texture**.

sloppy mode

Режим JavaScript, в основном используемый для обратной совместимости с веб-сайтами. Небрежный режим включает в себя функции, которые могут привести к ошибкам или снизить производительность. См. также **strict mode** - строгий режим.

SMBus (System Management Bus - шина управления системой)

Подмножество I2C для устройств на основе регистров.

SNTP (Simple Network Time Protocol - простой протокол сетевого времени)

Облегченный способ для устройства IoT получать текущее время из сети.

source rectangle - исходный прямоугольник (в POCO)

При рисовании растрового изображения - используемая область растрового изображения, позволяющая указать, что должна быть нарисована только часть маски или изображения. разреженный массив

sparse array - разреженный массив

Массив JavaScript, в котором не все элементы имеют назначенное значение.

spread syntax - синтаксис распространения

Функция современного JavaScript, которая разделяет элементы массива или свойства объекта на отдельные аргументы.

SSID (service set identifier - идентификатор набора услуг)

Удобочитаемое имя сети Wi-Fi, предоставляемое базовой станцией Wi-Fi.

strict equality operator - оператор строгого равенства

Оператор JavaScript ===, который можно использовать вместо ==, чтобы избежать преобразования типов. Этот оператор никогда не выполняет преобразование типов; если его операнды разных типов, они всегда не равны.

strict inequality operator - оператор строгого неравенства

Оператор JavaScript !==, который можно использовать вместо !=, чтобы избежать преобразования типов. Этот оператор никогда не выполняет преобразование типов; если его операнды разных типов, они всегда не равны.

strict mode - строгий режим

Режим, представленный в JavaScript 5th Edition, который устраняет несколько запутанных и неэффективных функций. В этой книге используется исключительно строгий режим. См. также **sloppy mode**.

GLOSSARY

style (в Piu)

Объект, управляющий внешним видом текста, включая шрифт и цвет текста, в одном или нескольких объектах содержимого. Экземпляр класса Style, указанный в свойстве стиля объекта содержимого.

tag

Функция JavaScript, позволяющая функции изменять поведение литералов шаблонов по умолчанию. Например, вы можете использовать эту функцию для преобразования строкового представления UUID в двоичные данные.

template (в Piu)

Класс, созданный с использованием метода шаблона объекта контента, который позволяет устранить избыточность при создании нескольких похожих объектов.

literal шаблона

Способ разграничения строк JavaScript, использующий символ обратной кавычки (``). Строки, определенные таким образом, могут занимать несколько строк и включать замены строк.

текстура (в Piu)

Объект, предоставляющий изображение, отрисовываемое (полностью или частично) одним или несколькими скинами. Экземпляр класса Texture, указанный в свойстве текстуры скина.

TLS (Transport Layer Security - безопасность транспортного уровня)

Низкоуровневый инструмент для защиты связи, работающий с множеством различных протоколов, включая HTTP; более поздняя версия Secure Sockets Layer (SSL).

to-tween (in Piu)

Анимация движения, добавленная на временную шкалу с помощью метода временной шкалы `to`, который изменяет свойства целевого объекта с его текущих значений на целевые значения, указанные в объекте свойств, в течение миллисекунд.

topic (в MQTT)

В протоколе MQTT - организационная единица для сообщений. Сообщения к серверу MQTT и от него организованы по темам; конкретный сервер может поддерживать множество тем, но клиент получает сообщения только для тех тем, на которые он подписан.

tween (в Piu)

Описание того, что происходит с указанным объектом контента на временной шкале. Каждая анимация описывает, как одно или несколько свойств объекта изменяются от начального значения до конечного значения. См. также **from-tween** и **to-tween**.

typed array

Представление, в которое вы можете обернуть `ArrayBuffer` для доступа к данным в нем; коллекция классов (подклассы `TypedArray` для определенных типов), которые позволяют работать с массивами целых чисел и значений с плавающей запятой, хранящихся в `ArrayBuffer`. См. также **data view** - представление данных.

update area (в POCO)

Начальная область рисования, определяемая методом начала POCO. Его можно ограничить отсечением; см. также **clipping area** - область отсечения

WebSocket

Одноранговый протокол, при котором два устройства обмениваются данными через постоянное сетевое соединение, что обеспечивает эффективную передачу кратких сообщений. В отличие от HTTP, в котором только клиент может сделать запрос, а сервер всегда отвечает, WebSocket позволяет обоим устройствам отправлять и получать сообщения.

GLOSSARY

XS

Механизм JavaScript, оптимизированный для сред с ограниченными ресурсами, таких как микроконтроллеры. XS реализует полный язык JavaScript и поддерживает отладку на устройстве, в отличие от других механизмов для встроенного использования. Созданный Kinoma, XS поддерживается Moddable как ядро Moddable SDK.

XS in C

Низкоуровневый API-интерфейс C, предоставляемый движком XS JavaScript, чтобы вы могли интегрировать код C в свои проекты JavaScript (или код JavaScript в свои проекты C).