**ESP32: оцифровка звука**

Оригинал статьи:

[ESP32: оцифровка звука | arm | programming (microsin.net)](http://microsin.net/programming/arm/working-with-esp32-audio-sampling.html)

Это руководство показывает практическое применение аналого-цифрового преобразователя (**ADC**) микроконтроллера **ESP32** для оцифровки звука с помощью прерывания таймера.

**ESP32 Audio: таймеры и прерывания**

У ESP32 есть 4 аппаратных таймера общего назначения, распределенных по двум группам [2]. Все эти таймеры одинаковые, и оборудованы 16-разрядными прескалерами тактовой частоты и 64-разрядными счетчиками. Программированием прескалера можно поделить внутреннюю системную частоту 80 МГц для тактов таймера. Минимальное значение прескалера 2, это означает, что прерывания могут срабатывать с максимальной частотой 40 МГц. Это значит, что на самой высокой частоте тактирования таймера код обработчика должен выполняться не более чем за 6 тактов ядра процессора (при тактах ядра 240 МГц и частоте тактирования таймера 40 МГц, 240/40 = 6). У таймеров есть несколько связанных с ними свойств:

divider — значение коэффициента деления прескалера.  
counter\_en — определяет, включен ли соответствующий 64-битный счетчик таймера (обычно true при использовании таймера).  
counter\_dir — направление счета - инкремент или декремент счетчика с каждым тактом.  
alarm\_en — разрешено ли событие "alarm", т. е. действие по событию счетчика.  
auto\_reload — должен ли счетчик сбрасываться при событии alarm.

Некоторые важные режимы таймера:

**Таймер запрещен**. Счетчик таймера не работает.

**Таймер разрешен, но функция alarm запрещена**. Счетчик таймера работает, и увеличивает или уменьшает с каждым тактом свое значение, но больше ничего не происходит.

**Таймер разрешен, и также разрешена генерация событий alarm**. То же самое, что и в предыдущем режиме, однако при достижении счетчика определенного запрограммированного значения счетчик сбрасывается и/или генерируется прерывание.

Счетчики таймеров могут произвольно считываться любым кодом, но в большинстве случаев нам нужно будет это делать периодически, чтобы в строгие интервалы времени выполнять повторяющиеся действия. Обычно это делается для оцифровки сигналов с целью их дальнейшей программной обработки. Для этого на событие alarm таймера настраивается обработчик прерывания (Interrupt Service Routine, **ISR**).

Функция ISR должна завершаться как можно быстрее, до наступления момента следующего прерывания. В случае высокой частоты прерываний это накладывает жесткие ограничения на то, насколько сложным может быть этот ISR. Обычно ISR должен выполнять как можно меньше действий, чтобы успеть завершиться до наступления своего следующего запуска, и максимально разгрузить процессор для выполнения основной программы. Чаще всего ISR просто заполняет буфер данными, считанными с ADC, и после его заполнения взводит какой-нибудь флаг, оповещая фоновую обработку (задачу RTOS) о готовности данных.

В проектах среды разработки ESP-IDF в коде ISR может использоваться функция FreeRTOS vTaskNotifyGiveFromISR(), чтобы оповестить задачу о необходимости обработки буфера. Это может выглядеть примерно так:

portMUX\_TYPE DRAM\_ATTR timerMux = portMUX\_INITIALIZER\_UNLOCKED;

TaskHandle\_t complexHandlerTask;

hw\_timer\_t \* adcTimer = NULL; *// наш таймер*

*// Задача FreeRTOS, которая должна делать сложную обработку:*

void complexHandlerTask (void \*param)

{

**while** (true)

{

*// Задача блокируется на вызове ulTaskNotifyTake с таймаутом*

*// в 1 секунду, если ISR не разблокирует её вызовом*

*// функции vTaskNotifyGiveFromISR:*

uint32\_t tcount = ulTaskNotifyTake(pdFALSE, pdMS\_TO\_TICKS(1000));

**if** (check\_for\_work)

{

*// Здесь могут выполняться интенсивные вычисления, например*

*// БПФ на данными буфера:*

...

}

}

}

*// Обработчик прерывания таймера:*

void IRAM\_ATTR onTimer()

{

*// Мьютекс защищает обработчик от повторного входа (это не должно*

*// никогда произойти, но эта защита предоставлена на всякий случай):*

portENTER\_CRITICAL\_ISR(&timerMux);

*// Здесь выполняются какие-либо действия, например считывание ножки.*

**if** (some\_condition)

{

*// Оповещение задачи complexHandlerTask, что буфер заполнен.*

BaseType\_t xHigherPriorityTaskWoken = pdFALSE;

vTaskNotifyGiveFromISR(complexHandlerTask, &xHigherPriorityTaskWoken);

**if** (xHigherPriorityTaskWoken)

{

portYIELD\_FROM\_ISR();

}

}

portEXIT\_CRITICAL\_ISR(&timerMux);

}

void setup()

{

xTaskCreate(complexHandler,

"Handler Task",

8192,

NULL,

1,

&complexHandlerTask);

*// Для простоты настроим частоту тактирования таймера равной*

*// 80 МГц / 80 = 1 МГц:*

adcTimer = timerBegin(3, 80, true);

*// Подключение к таймеру функции ISR:*

timerAttachInterrupt(adcTimer, &onTimer, true);

*// Прерывания будут срабатывать, когда счетчик == 45,*

*// т. е. 22222 раз в секунду:*

timerAlarmWrite(adcTimer, 45, true);

timerAlarmEnable(adcTimer);

}

[**Кэши CPU и Гарвардская архитектура**]

Важно отметить, значение атрибута IRAM\_ATTR в определении ISR onTimer(). Причина в том, что ядра CPU могут выполнять инструкции (и получать доступ к данным) только из встроенной памяти RAM, но не из хранилища flash, где обычно хранятся программа и данные. Чтобы обойти это ограничение, часть из общего объема 520 KB RAM выделяется как IRAM, кэш 128 KB, чтобы прозрачно загружать код из хранилища flash. ESP32 использует для этого отдельные шины для кода и данных (Гарвардская архитектура), так что они работают раздельно, и это распространяется на свойства памяти: IRAM это специальная область, и к ней возможен доступ только в пределах 32-битного адреса.

На самом деле память ESP32 очень неоднородна. Различные её регионы выделены для разных целей: максимальная непрерывная область составляет около 160 KB, а вся "нормальная" область, доступная программам пользователя, составляет только около 316 KB.

Загрузка данных из хранилища flash может потребовать доступа по шине SPI, так что любой код, который требует быстрого запуска и выполнения, должен позаботиться о том, чтобы поместиться в кэш IRAM, и часто должен быть намного меньше (менее 100 KB), поскольку часть IRAM используется RTOS. Следует отметить, что система сгенерирует исключение, если при возникновении прерывания код ISR не был загружен в кэш. Загружать что-то в кэш при возникновении прерывания было бы очень медленно, и реализовать слишком сложно. Спецификатор IRAM\_ATTR обработчика onTimer() говорит компилятору и линкеру, что этот код особенный, и должен быть статически быть размещен в IRAM, чтобы никогда не выгружаться оттуда.

Важный момент: атрибут IRAM\_ATTR применяется только к функции, которая с ним определена, на затрагивая любые функции и подпрограммы, которые вызываются из IRAM\_ATTR-функции.

[**Оцифровка звука из прерывания таймера ESP32**]

Стандартная техника оцифровки звукового сигнала основана на прерывании таймера, которое заполняет некий буфер аудиоданными, и затем оповещает какую-либо задачу, что данные в буфере готовы для обработки.

ESP-IDF предоставляет функцию драйвера adc1\_get\_raw(), которая считывает данные на определенном канале ADC первого периферийного устройства ADC (второе из них используется для WiFi). Однако использование этой функции в ISR таймера приведет к нестабильному поведению программы, поскольку это сложная функция, вызывающая нетривиальное количество других функций IDF (в частности тех, что имеют дело с блокировками), и ни функция adc1\_get\_raw(), ни вызываемые ей функции не помечены атрибутом IRAM\_ATTR. ISR рухнет, как только начнет выполняться большой объем кода, что приведет к обновлению кэша программного кода с выгрузкой из кэша функций ADC. Вытеснять их из кэша IRAM могут другие функции, такие как стек WiFi, TCP/IP-HTTP, или библиотека файловой системы SPIFFS, или что-нибудь еще.

Замечание: некоторые функции IDF специально созданы (и помечены атрибутом IRAM\_ATTR) таким образом, чтобы они могли вызываться из кода ISR. Примером такой функции может служить вышеупомянутая vTaskNotifyGiveFromISR().

В IDF чаще всего решают проблему кода ISR так, чтобы переложить сложные вычисления (например, сжатие данных или перекодировка звука) на задачи фоновой обработки. К сожалению это бывает очень неэффективно. Как код ISR, так и оповещаемая им задача связаны с взаимодействием с операционной системой и тысячами выполняемых инструкций. Такой подход, хотя и являющийся теоретически верным, может снизить производительность CPU настолько, что у него может не хватить вычислительной мощности для других задач.

[**Погружение в исходный код IDF**]

Считывание данных с ADC обычно простая задача, и наша стратегия заключается в том, чтобы подсмотреть, как это делает IDF, и скопипастить часть её кода, чтобы не допустить вызовов предоставленного API. Функция adc1\_get\_raw() реализована в файле components\driver\adc.c среды разработки ESP-IDF, и из того, что она делает, нам нужен только код, который реально считывает выборку из ADC, что делается вызовом adc\_hal\_convert(). К счастью, функция adc\_hal\_convert() простая, она просто работает с ADC путем манипуляции с его регистрами периферийного устройства через глобальную структуру, которая называется SENS.

Можно адаптировать код IDF в нашей программе (имитируя поведение adc1\_get\_raw()). Это будет выглядеть примерно так:

int IRAM\_ATTR local\_adc1\_read(int channel)

{

uint16\_t adc\_value;

*// Будет выбран только один канал:*

SENS.sar\_meas\_start1.sar1\_en\_pad = (1 << channel);

**while** (SENS.sar\_slave\_addr1.meas\_status != 0);

SENS.sar\_meas\_start1.meas1\_start\_sar = 0;

SENS.sar\_meas\_start1.meas1\_start\_sar = 1;

**while** (SENS.sar\_meas\_start1.meas1\_done\_sar == 0);

adc\_value = SENS.sar\_meas\_start1.meas1\_data\_sar;

**return** adc\_value;

}

Чтобы переменная SENS стала доступной, необходимо подключить заголовки:

#include < soc/sens\_reg.h>

#include < soc/sens\_struct.h>

И наконец, поскольку adc1\_get\_raw() выполняет некоторые шаги по конфигурированию перед тем, как сделать чтение ADC, это должно быть сделано отдельно, например перед запуском таймера.

Недостаток такого подхода в том, что мы теряем гибкости взаимодействия с другими функциями IDF. Как только другое периферийное устройство, драйвер, или случайная часть кода сбросит конфигурацию ADC, наша функция перестанет правильно работать. На конфигурацию ADC не повлияют как минимум WiFi, PWM, I2C и SPI.

Использованием нашей функции local\_adc1\_read код ISR таймера будет выглядеть следующим образом:

#define ADC\_SAMPLES\_COUNT 1000

int16\_t abuf[ADC\_SAMPLES\_COUNT];

int16\_t abufPos = 0;

void IRAM\_ATTR onTimer()

{

portENTER\_CRITICAL\_ISR(&timerMux);

abuf[abufPos++] = local\_adc1\_read(ADC1\_CHANNEL\_0);

**if** (abufPos >= ADC\_SAMPLES\_COUNT)

{

abufPos = 0;

*// Оповещение задачи adcTaskHandle о том, что буфер заполнен.*

BaseType\_t xHigherPriorityTaskWoken = pdFALSE;

vTaskNotifyGiveFromISR(adcTaskHandle, &xHigherPriorityTaskWoken);

**if** (xHigherPriorityTaskWoken)

{

portYIELD\_FROM\_ISR();

}

}

portEXIT\_CRITICAL\_ISR(&timerMux);

}

Здесь adcTaskHandle это задача FreeRTOS, которая должна быть реализована для обработки буфера, она соответствует принципу обработки, показанному ранее в complexHandler. Здесь может быть сделана копия аудиобуфера, чтобы потом эти данные можно было обработать. Например, может быть запущен алгоритм БПФ, или данные звука могут быть сжаты для передачи через WiFi.

Парадоксально, что использование Arduino API вместо ESP-IDF API (например, analogRead() вместо adc1\_get\_raw()) работает, потому что функции Arduino помечены IRAM\_ATTR. Однако они намного медленнее, чем функции ESP-IDF, потому что предоставляют дополнительный уровень абстракции. Если говорить о производительности, то наша кустарная функция чтения ADC работает примерно в 2 раза быстрее, чем функция ESP-IDF.

[**За и против использования RTOS в проектах ESP32**]

То, что мы тут проделали - заново реализовали API операционной системы, чтобы обойти некоторые проблемы, которых при отсутствии операционной системы не было бы в принципе - хорошая иллюстрация плюсов и минусов использования в первую очередь RTOS.

Маленькие микроконтроллеры программируются напрямую, иногда даже на ассемблере, и у разработчиков имеется полное управление над каждым аспектом выполнения программы - над каждой инструкцией CPU и над каждым состоянием всех периферийных устройств чипа. Конечно это реально становится утомительным при разрастании программы, когда нужно подключать в приложении все большего количества оборудования и программных стеков (USB, Wi-Fi, HTTP, файловые системы и т. д.). Сложные микроконтроллеры, такие как ESP32, с большим набором периферийных устройств и функций, с двумя ядрами CPU и сложной, неоднородной компоновкой памяти, программировать "с нуля" становится практически невозможным, если не использовать FreeRTOS и сторонние библиотеки.

Хотя каждая операционная система накладывает определенные ограничения и требования к коду (и это плата за предоставляемые сервисы), её достоинства обычно перевешивают: разработка приложения ускоряется и упрощается. Однако мы можем работать с аппаратными ресурсами напрямую, и иногда должны это делать при проектировании встраиваемых систем.