**Kendryte K210**

**SPI**

Для работы с любым модулем к210 необходим пакет sdk. Также можно подключить необходимые файлы для конкретного интерфейса. Для работы с SPI я подключила файлы из папки demo spi\_slave. В ней находятся файлы с функциями для работы как в режиме слейв, так и в режиме мастер. Режим задается дефайном в мейне. Также необходимо задать правильные номера ног для платы maix go в файле spi\_master.c.

Для работы с модулем RFID  нужно было написать свои функции для приема и передачи данных, т.к. стандартные не используют значение для регистра и задается 0 по умолчанию. Точно также нужно сделать для приема-передачи через дма. Эти функции написаны, но нужно написать свой вариант для задания регистра модуля.

С работой через прерывания возникли сложности, т.к. это плохо описано в документации и отсутствуют api функции для работы в режиме мастер.

Для работы прерываний необходимо вызвать функции, описанные в блоке PLIC документа Programming Guide, задать все необходимые значения и прописать функцию, которая будет вызываться из прерывания. В даташите говорится, что можно задать разные флаги для разных событий, но не указано, какие это флаги и как конкретно их нужно задать. Как мне кажется, маску нужно задать в регистре imr (SPI Interrupt Mask Register). Если записать там число 0х10 в функции инициализации spi\_init, то моя функция для прерывания вызывается после приема сообщения.

**I2C**

Поднять i2c, как он описан в Programming Guide в режиме мастер, инициализацией соответствующих ножек как функций i2c не получилось. Проверила пример из  Demo, где i2c работает инициализацией ног как gpio output, и далее вручную они поднимаются и опускаются по протоколу. Этот пример работает. Есть функции передачи и чтения просто значений, так и значений из заданного регистра слейв-устройства.

В режиме слейв нужно использовать те ножки, которые сконфигурированы для  i2c (через функцию fpioa) и функции из Demo i2c\_slave. Устройство должно работать следующим образом: При инициализации как слейв, задается прерывание и функция i2c\_slave\_irq, которая вызывается из прерывания. Эта функция читает статус регистр и, в зависимости от установленных флагов, должна вызывать соответствующие функции, но при попытках отправить сообщение и прочитать что-то мастером, устанавливается только флаг старт-сигнала. Последовательность от мастера приходит и устанавливаются биты ACK и NACK в конце последнего отправленного байта, как видно логическим анализатором.

I2C\_INTR\_STAT\_RX\_FULL

Насколько я поняла, функция приема должна вызываться при установке флага о заполнении буфера приема. Аргументом передается значение из регистра data\_cmd. Далее в этой функции проверяется переданное значение. Если это адрес регистра (возможно, номер слейв-устройства), то аргумент записывается как значение acces\_reg структуры \_slave\_info

{

    uint8\_t acces\_reg;

    uint8\_t reg\_data[SLAVE\_MAX\_ADDR];

} slave\_info\_t;

В противном случае, если адрес уже был получен, аргумент записывается в массив reg\_data и acces\_reg инкрементируется. Запись начинается с индекса массива, равному значению переданного аргумента в функцию и продолжается до индекса SLAVE\_MAX\_ADDR (равен 14).

Видимо, принятые значения потом можно прочитать из данной структуры.

I2C\_INTR\_STAT\_RD\_REQ

Когда установлен данный флаг, должна вызываться функция передачи i2c\_slave\_transmit(). Она читает данные из массива reg\_data и возвращает прочитанное значение, которое потом, в функции обработки прерывания записывается в регистр data\_cmd.

**UART**

Для работы с юартом на плате указаны ножки 4 и 5, и 6 и 7. Но в первом наборе не работает ножка 4 – RX.

Инициализируются ножки таким образом:

fpioa\_set\_function(6, FUNC\_UART1\_RX + UART\_NUM \* 2);

    fpioa\_set\_function(7, FUNC\_UART1\_TX + UART\_NUM \* 2);

инициализирующие функции:

uart\_init(UART\_NUM);

    uart\_configure(UART\_NUM, 115200, 8, UART\_STOP\_1, UART\_PARITY\_NONE);

Тестирование примера демо  **uart**. Отправка сообщения работает, прием – нет. Во время выполнения функции uart\_receive\_data не выполняется блок if.

int uart\_receive\_data(uart\_device\_number\_t channel, char \*buffer, size\_t buf\_len)

{

    size\_t i = 0;

    for(i = 0; i < buf\_len; i++)

    {

        if(uart[channel]->LSR & 1)

        {

                    buffer[i] = (char)(uart[channel]->RBR & 0xff);

        }

        else

        {

            break;

        }

    }

    return i;

}

С дма прием не работает также, отправка работает. Видимо, не установлен бит разрешения работы приема в соответствующем регистре. Для работы с дма необходимо вызвать функцию инициализации дма и вызвать функцию отправки с дма  соответствующими параметрами.

Тестирование примера  **uart\_interrupt**. Для работы через прерывания необходимо разрешить работу прерываний:

plic\_init();

    sysctl\_enable\_irq();

Далее вызывается функция задания триггера uart\_set\_send\_trigger(UART\_NUM, UART\_SEND\_FIFO\_0); есть 4 маски триггера, на как именно они отличаются друг от друга выяснить не удалось (не на длительность посылки), количество отправляемых байт также на результат работы функций не влияет. После этого вызываются функция uart\_irq\_register(UART\_NUM, UART\_SEND, on\_uart\_send, &v\_uart\_num, 2);, которая задает колбек после срабатывания триггера.

Также в этом файле протестировала работу функции uart\_send\_data\_dma\_irq. Для ее работы необходимо лишь вызвать функции инициализации прерываний и дма. В самой функции задается колбек после вызова прерывания.

Тестирование примера **uart\_dma\_irq**.  В этом примера вызывается функция uart\_handle\_data\_dma из файла uart.c, которая принимает в качестве аргументов структуры, с заданием необходимых параметров и функций: основной, например, функция приема или передачи данных, и функции, которая вызовется после окончания работы основной функции и вызова прерывания. В целом, похоже на работу из двух предыдущих примера, но теперь вся работа выполняется из одной функции.

Также имеется высокоскоростной юарт UARTHS. Для работы с ним необходимо подключить соответствующий хидер.

Прием сообщений начал работать после того, как была изменена частота работы в функции uart\_configure, а потом возвращены обратно прежние значения. Также добавлен вывод принтф для отладки. Проверена работа функций приема стандартного, через дма, с прерываниями, дма с прерываниями.

**GPIO**

Для работы с gpio необходимо инициализировать этот модуль и назначить ножке, которую хотим использовать значение gpio:

fpioa\_set\_function(24, FUNC\_GPIO3);

 gpio\_init();

здесь ножке 24 присваивается функция gpio3, и в дальнейшем, с этой ножкой нужно работать как с gpio3. Например, следующий код

gpio\_set\_drive\_mode(3, GPIO\_DM\_OUTPUT);

    gpio\_pin\_value\_t value = GPIO\_PV\_HIGH;

    gpio\_set\_pin(3, value);

конфигурирует как выход ножку 24 и устанавливает ее в 1. Всего ножек, которые могут быть сконфигурированы как gpio, на мк 8.

API  для работы с gpio невелик. Это функции установки режима, установки значения на ножке и чтение значения.

**GPIOHS**

Высокоскоростные gpio, которых на мк 32 штуки. Могут выступать как выходы\входы, так и использоваться с прерываниями.

Для работы с ними, нужно также как и с gpio, инициализировать ножки, установить им соответствующие функции и значения.

fpioa\_set\_function(25, FUNC\_GPIOHS3);

    gpiohs\_set\_drive\_mode(3, GPIO\_DM\_OUTPUT);

    gpio\_pin\_value\_t value = GPIO\_PV\_HIGH;

    gpiohs\_set\_pin(3, value);

    fpioa\_set\_function(26, FUNC\_GPIOHS2);

    gpiohs\_set\_drive\_mode(2, GPIO\_DM\_INPUT\_PULL\_UP);

Для работы с прерываниями необходимо инициализировать модуль:

plic\_init();

sysctl\_enable\_irq();

Далее устанавливается триггер для прерывания:

gpiohs\_set\_pin\_edge(2, GPIO\_PE\_BOTH);

и установить функцию-обработчик прерывания:

gpiohs\_irq\_register(GPIO\_KEY, 1, irq\_gpiohs2, &g\_count);

либо без аргументов:

 gpiohs\_set\_irq(GPIO\_KEY, 1, irq\_gpiohs2);

Теперь, если подключить на ножку 26 кнопку, то по фронту сигнала на этой ножке будет вызываться функция-обработчик.

**DMA**

Работа с ДМА во всех рассмотренных интерфейсах заключается в инициализации модуля дма и вызове функций:

sysctl\_dma\_select(…);

dmac\_set\_single\_mode(…);

dmac\_wait\_done(dma\_channel\_num);

В юарте также реализованы отправка и прием через дма с прерываниями. Для этого вызываются такие функции:

dmac\_irq\_register(…);

 sysctl\_dma\_select(…);

 dmac\_set\_single\_mode(…)

Попробовала работать в режиме mem-to-mem, данные записываются, но похоже, не совпадает скорость. Возможно, стоит попробовать написать функцию для передачи через прерывания.

**TIMER**

Для работы с таймерами необходимо лишь добавить хедер. Всего есть 3 таймера, с 4 каналами каждый. Тестовая программа драйвер работает полностью, проверяет работу функций

timer\_init(j); - инициализирует нужный таймер;

timer\_set\_interval(j, i, 500000000); - задает время срабатывания прерывания;

timer\_irq\_register(j, i, 1, 1, timer\_callback, &ctx\_table[j \* TIMER\_CHANNEL\_MAX + i]); - задает функцию, которая будет вызвана из прерывания. Третий аргумент задает режим вызова функции – если 1 – единичный режим, функция будет вызвана лишь 1 раз, 0 – повторяющийся режим, функция будет вызываться каждый раз, после истечения времени, указанного в интервале, пока таймер будет включен.

Включение и выключение таймера выполняется функцией

timer\_set\_enable(j, i, 1); - если последний аргумент 1 – включить таймер, 0 – выключить.

После выключения также необходимо вызвать функцию

timer\_irq\_unregister(j, i);

В этом примере j – номер таймера,  I – номер канала.

**PWM**

Для работы в режиме шим необходимо инициализировать и включить работу шима:

pwm\_init(TIMER\_PWM);

 pwm\_set\_enable(TIMER\_PWM, TIMER\_PWM\_CHN, 1);

Частота и период задается функцией

pwm\_set\_frequency(…);

PWM – это таймер, поэтому нельзя использовать таймер, который уже занят под другую задачу.

**RTC**

Кроме стандартных функций, описанных в Programming Guide:

rtc\_init();

 rtc\_timer\_set(…);

rtc\_timer\_get(…);

есть функции из файла rtc.h:

rtc\_alarm\_set(…); - задает срабатывание таймера по событию alarm.

rtc\_tick\_irq\_register(

        false,

        RTC\_INT\_SECOND,

        on\_timer\_interrupt,

        NULL,

        1

    ); - задает частоту срабатывания прерывания, повторяемость(да/нет), функцию, которая будет вызвана, ее аргумент и приоритет прерывания.

    rtc\_alarm\_irq\_register(

        false,

        (rtc\_mask\_t) {

            .second = 0, /\* Second mask \*/

            .minute = 0, /\* Minute mask \*/

            .hour = 1,   /\* Hour mask \*/

            .week = 0,   /\* Week mask \*/

            .day = 0,    /\* Day mask \*/

            .month = 0,  /\* Month mask \*/

            .year = 0,   /\* Year mask \*/

        },

        on\_alarm\_interrupt,

        NULL,

        1

    ); - похоже на вышеописанную функцию, но второй аргумент – маска – задает длительность срабатывания аларма, т.е. если прерывание ртц задано каждую секунду, то прерывания по аларму будет срабатывать каждую секунду (или каждую единицу времени, указанную функцией rtc\_timer\_set(…);)

 в течении часа, причем начинать срабатывать будет с часа, указанного функцией rtc\_alarm\_set(…); не смотря на минуты. Т.е. если ртц задано с 22.30, а аларм – с 22.31 – прерывание аларм не будет ждать минуту, оно будет срабатывать сразу, потому что указанный час уже пошел.

Можно изменить маску в процессе работы функцией

rtc\_alarm\_set\_mask(rtc\_mask\_t);

или прочитать текущую маску rtc\_mask\_t rtc\_alarm\_get\_mask(void);

Отключить/включить прерывание аларм -  rtc\_alarm\_set\_interrupt(int enable);

Проверить включено ли прерывание аларм -  rtc\_alarm\_get\_interrupt(void);

 Изменить режим срабатывания прерывания ртц - rtc\_tick\_set\_interrupt\_mode(rtc\_tick\_interrupt\_mode\_t mode);

Узнать текущий режим: rtc\_tick\_get\_interrupt\_mode(void);

Функции int rtc\_tick\_interrupt\_set(int enable); и int rtc\_tick\_interrupt\_get(void); из файла

rtc.h не реализованы в файле rtc.c, вместо них используются функции int rtc\_tick\_set\_interrupt(int enable) и int rtc\_tick\_get\_interrupt(void), для этого их сигнатуры необходимо добавить в хедер.