Написание ISR

Написанный вами ISR должен соответствовать прототипу, ожидаемому от alt_ic_isr_register(). Прототип для вашей функции ISR должен соответствовать следующему прототипу:

```
void (*alt_isr_func) (void* isr_context)
```

Определение параметра контекста такое же, как и для функции alt_ic_isr_register().

С точки зрения системы обработки исключений HAL, главнейшей функцией ISR является сброс состояний прерываний ассоциированной периферии. Процедура сброса состояния аппаратного прерывания определена для периферии.

За подробной информацией обратитесь к соответствующей главе в "Руководстве пользователя по IP встроенной периферии".

Когда ISR закончит обработку аппаратного прерывания, используйте **ret** для возврата. Направитель (funnel) прерывания HAL выдаёт **eret** после восстановления контекста приложения.

Использование направителей прерываний

HAL создаёт таблицу векторов для каждого EIC, подключенного к процессору Nios II. В таблице векторов, HAL вставляет переход (ветвление) к корректному направителю для каждого устройства, управляемого прерываниями, поддерживаемого BSP, в зависимости от характеристики драйвера устройства и настроек приоритета прерываний. Направитель может совместно использоваться несколькими аппаратными прерываниями, если их драйверы имеют совместимые характеристики.

Код направителя принимает контроль от основного вектора исключений или прерываний, в зависимости от реализации контроллера прерываний. Направитель выполняет следующие задачи: переключение указателя стека, сохранение регистров и вызов процедуры контекстного переключения RTOS, передачу контроля обработчику. Когда происходит возврат обработчика, код направителя выполняет следующие задачи: вызов процедуры диспетчера процесса RTOS и восстановление регистров, передача контроля соответствующей приоритетной задаче.

HAL содержит следующие направители прерываний:

- Набор теневых регистров, приоритет запрещён – Аппаратное прерывание назначается набору теневых регистров, с запрещённым приоритетом внутри набора регистров. Этот направитель (funnel) не может сохранить контекст регистра. Устройство гарантирует только, что только одна ISR может быть одновременно запущена с набором теневых регистров.
- Набор теневых регистров, приоритет разрешён – Аппаратное прерывание назначается набору теневых регистров. Прерывание имеет приоритет по отношению к другим прерываниям, использующим тот же набор регистров. Этот направитель (funnel) может сохранить контекст регистра, так что обработчики, назначенные тому же набору регистров не смогут повредить другой контекст.
- Немаскируемое прерывание – Немаскируемое аппаратное прерывание, назначенное набору теневых регистров, с запрещённым приоритетом прерываний внутри набора регистров. Этот направитель (funnel) не может сохранить контекст регистра. Устройство гарантирует только, что только одна ISR может быть одновременно запущена с набором теневых регистров.

Код направителя HAL вызывается из таблицы векторов.

Запуск в ограниченной среде

ISR запускаются в ограниченной среде. Большое количество вызовов HAL API не доступны из ISR. Например, доступ к файловой системе HAL не возможен. Как правило, когда вы пишете собственную ISR, никогда не включайте функцию вызова, которая может быть заблокирована по любой причине (например ожидание аппаратного прерывания).

В главе "Справка по HAL API" в настольной книге программиста Nios II указаны API функции, не доступные ISR.

Будьте осторожны с вызовом функций стандартной библиотеки ANSI C внутри ISR. Избегайте использования I/O API стандартной библиотеки Си, поскольку вызов этих функций может спровоцировать зависание системы, т.е., система может перманентно заблокироваться в ISR.

В особенности не вызывайте printf() внутри ISR, если вы не уверены, что stdout размещён в драйвере устройства без прерываний. Иначе, printf() может повесить систему, ожидая аппаратного прерывания, которое никогда не появится, поскольку прерывания запрещены.

Управление приоритетом

Расширенные API прерываний HAL поддерживают приоритет прерываний. Когда приоритет разрешён, прерывания более высокого уровня могут получить контроль, даже если ISR уже запущена. Драйвер устройства должен быть написан определённым образом, чтобы корректно функционировать с приоритетом.

Когда драйвер устройства поддерживает приоритет, он публикует это свойство в настройках драйвера isr_preemption_supported. Когда собирается BSP, SBT проверяет каждый драйвер устройства на поддержку приоритета. Если все драйверы в BSP поддерживают приоритет, то он разрешается.

Устаревшие драйверы устройств не могут публиковать свойство isr_preemption_supported. Поэтому SBT считает, что они не поддерживают приоритет. Если ваш собственный драйвер устройства поддерживает приоритет, и вы хотите разрешить приоритет в BSP, вы должны обновить драйвер до использования расширенного API прерываний.

Чтобы разрешить расширенные API прерываний, убедитесь, чтоб все драйверы системы обновлены до использования расширенного API прерываний.

За подробной информацией о настройке драйвера isr_preemption_supported, обратитесь к команде set_sw_property в секции "Tcl команды" в главе "Справка по инструменту создания программ под Nios II" в настольной книге программиста Nios II.

Операционная система также может публиковать свойство isr_preemption_supported. Расширенные API прерываний HAL поддерживают автоматический приоритет. Автоматический приоритет подразумевает, что маскируемые исключения остаются разрешёнными, когда процессор принимает аппаратное прерывание. Это означает, что ваша ISR может быть заменена ISR с более высоким приоритетом, без необходимости исполнения инструкции **eret**. Автоматический приоритет может быть только тогда, когда приоритетные аппаратные прерывания используют другой набор регистров, чем прерывания получающие приоритет. Автоматический приоритет доступен только тогда, когда вы разрешаете его в настройках BSP.

Регистрирование в ISR с расширенным API прерываний

Перед тем, как программа сможет использовать ISR, вы должны зарегистрировать его, вызвав alt_ic_isr_register(). Прототип alt_ic_isr_register():

```
int alt_ic_isr_register(alt_u32 ic_id,
                        alt_u32 irq,
                        alt_isr_func isr,
                        void *isr_context,
                        void* flags)
```

Функция имеет следующие параметры:

- ic_id – идентификатор контроллера прерываний (ID), определённый в **system.h**. В EIC с последовательной цепью опроса, ic_id идентифицирует EIC в цепи опроса. В IIC ic_id ничего не значит.
- irq – номер аппаратного прерывания, определённый в **system.h**.
  - Для IIC, irq – это номер IRQ. Приоритет прерываний обратно пропорционален номеру IRQ. Поэтому IRQ0 означает наивысший приоритет, а IRQ31 – наинизший.
  - Для IEC, irq – это ID порта прерываний.
- isr_context – указатель на структуру данных, ассоциированную с элементом драйвера устройства. isr_context считается входным аргументом функции isr. Он используется для передачи информации заданного контекста в ISR, и может указывать на любую информацию заданного контекста ISR. Значение контекста непрозрачно для HAL; оно всецело предназначено для заданной пользователем ISR.
- isr – это указатель на функцию ISR, которая вызывает irq, соответствующее номеру IRQ. Прототип функции ISR следующий:

```
void (void* isr_context);
```

Входной аргумент для этой функции - isr_context.

- flags – зарезервирован.

HAL регистрирует ISR одним из следующих методов:

- Для IIC, сохраняя указатель функции isr в таблице соответствия.
- Для EIC, конфигурирую таблицу вектора соответствующим кодом направителя (funnel), как это описано в секции "Использование направителей прерываний" на стр. 8-12.

Код возврата функции alt_ic_isr_register() нуль, если функция удачно выполнена, и не нуль, если не удачно.

Если HAL удачно регистрирует вашу ISR, ассоциированные аппаратные прерывания Nios II (как определено в irq) разрешаются при возврате из функции alt_ic_isr_register().

Определённая аппаратная инициализация может также потребоваться.

Когда происходят определённые прерывания, код HAL следит корректной диспетчеризацией регистрированной ISR.

За подробной информацией об определённой для вашей периферии аппаратной инициализации, обратитесь к соответствующему параграфу в "Руководстве пользователя по IP встроенной периферии". За подробной информацией о функции alt_ic_isr_register(), обратитесь к главе "Справка по HAL API" в настольной книге программиста Nios II.

HAL устаревших API прерываний предлагает другие функции для регистрирования аппаратных прерываний. Для всех новых и обновлённых драйверов, Altera рекомендует использовать расширенное API, описанное в этой секции. Устаревшая функция API, alt_irq_register(), описана в главе "Справка по HAL API" в настольной книге программиста Nios II.

Разрешение и запрещение прерываний

Расширенное HAL API прерывания предоставляет функции alt_ic_irq_disable(), alt_ic_irq_enable(), alt_ic_irq_enabled(), alt_irq_disable_all(), alt_irq_enable_all() и alt_irq_enabled(), чтобы позволить программе запретить прерывания для определённых секций кода, а потом снова разрешить их. Функции alt_ic_irq_disable() и alt_ic_irq_enable() позволяют вам запрещать и разрешать отдельные прерывания. Функция alt_irq_disable_all() запрещает все прерывания и возвращает значение контекста. Чтобы заново разрешить аппаратные прерывания, вы вызываете alt_irq_enable_all() и передаёте параметр в контексте. В этом случае, прерывания восстанавливаются в состояние перед вызовом alt_irq_disable_all(). Функция alt_irq_enabled() возвращает ненулевое значение, если разрешены маскируемые исключения. Функция alt_ic_irq_enabled() определяет, разрешено ли заданное прерывание.

Запрещайте аппаратные прерывания на максимально короткое время. Максимальная задержка прерываний увеличивается с увеличением времени запрета прерываний. За дополнительной информацией о запрещённых прерываниях, обратитесь к секции "Разрешённое удержание прерываний" на стр. 8-20.

За подробной информацией об этих функциях обратитесь к главе "Справка по HAL API" в настольной книге программиста Nios II.

Устаревшее HAL API прерываний предлагает другие функции для разрешения и запрещения отдельных прерываний. Для всех новых и обновлённых драйверов, Altera рекомендует использовать расширенное API, описанное в этой секции. Устаревшие API функции - alt_irq_disable() и alt_irq_enable() – описаны в главе "Справка по HAL API" в настольной книге программиста Nios II.

Конфигурирование внешнего контроллера прерываний

Драйвер внешнего контроллера прерываний (EIC) предлагает специальные настройки драйвера, которые создаются на стадии генерирования вашего BSP. Эти настройки кастомизируют драйвер в соответствии с конфигурацией EIC, найденного в системе Nios II. Количество и тип настроек зависят от реализации EIC, также как и количество и конфигурация EIC в аппаратной части системы. SBT создаёт BSP со значениями по умолчанию, выбранными для достижения максимальных характеристик системы. Вы можете оптимизировать эти настройки на стадии создания BSP. За подробной информацией об управлении настройками драйвера EIC, обратитесь к документации на заданный EIC.

Драйвер EIC может предложить специальные функции для управления некоторыми специфическими средствами EIC. Например, можно изменить уровень приоритета во время прогона программы.

За примерами обратитесь к главе "Ядро векторного контроллера прерываний" в руководстве пользователя по встроенной IP периферии.

Пример на Си

В примере 8-1 показана ISR, которая обслуживает аппаратные прерывания от кнопок компонента параллельных I/O (PIO). Этот пример основан на системе Nios II с 4-битной PIO периферией, подключенной к кнопкам. IRQ генерируются по каждому нажатию кнопки. Код ISR читает регистр фронта захвата PIO периферии и сохраняет значение в глобальной переменной. Адрес глобальной переменной передаётся ISR в виде контекстного указателя.

Example 8–1. An ISR to Service a Button PIO Interrupt

```
#include "system.h"
#include "altera_avalon_pio_regs.h"
#include "alt_types.h"

#ifdef ALT_ENHANCED_INTERRUPT_API_PRESENT
static void handle_button_interrupts(void* context)
#else
static void handle_button_interrupts(void* context, alt_u32 id)
#endif
{
  /* Cast context to edge_capture's type. It is important that this
     be declared volatile to avoid unwanted compiler optimization. */
  volatile int* edge_capture_ptr = (volatile int*) context;

  /*
   * Read the edge capture register on the button PIO.
   * Store value.
   */
  *edge_capture_ptr =
    IORD_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE);

  /* Write to the edge capture register to reset it. */
  IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE, 0);

  /* Read the PIO to delay ISR exit. This is done to prevent a
     spurious interrupt in systems with high processor -> pio
     latency and fast interrupts. */
  IORD_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE);
}
```

В примере 8-2 показан пример кода основной программы, которая регистрирует ISR с помощью HAL.

Основываясь на этом коде, становится возможным следующий процесс исполнения:

1. Кнопка нажата, генерируется IRQ.
2. IRQ получает контроль.
   - Для IIC, основной направитель (funnel) HAL получает контроль от процессора и координирует ISR handle_button_interrupts().
   - Для EIC, процессор переходит на адрес в таблице векторов, который передаёт контроль ISR handle_button_interrupts().
3. Функция handle_button_interrupts() обслуживает аппаратное прерывание и возвращается.
4. Продолжается обычное исполнение программы с обновлённым значением edge_capture.

Example 8–2. Registering the Button PIO ISR with the HAL

```
#include "sys/alt_irq.h"
#include "system.h"

...
/* Declare a global variable to hold the edge capture value. */
volatile int edge_capture;
...

/* Initialize the button_pio. */
static void init_button_pio()
{
    /* Recast the edge_capture pointer to match the
        alt_irq_register() function prototype. */
    void* edge_capture_ptr = (void*) &edge_capture;

    /* Enable all 4 button interrupts. */
    IOWR_ALTERA_AVALON_PIO_IRQ_MASK(BUTTON_PIO_BASE, 0xf);

    /* Reset the edge capture register. */
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE, 0x0);

    /* Register the ISR. */
#ifdef ALT_ENHANCED_INTERRUPT_API_PRESENT
    alt_ic_isr_register(BUTTON_PIO_IRQ_INTERRUPT_CONTROLLER_ID,
                        BUTTON_PIO_IRQ,
                        handle_button_interrupts,
                        edge_capture_ptr, 0x0);
#else
    alt_irq_register( BUTTON_PIO_IRQ,
                        edge_capture_ptr,
                        handle_button_interrupts );
#endif
}
```

Дополнительные примеры программ, в которых показана реализация ISR, например шаблон проекта count_binary, инсталлированы вместе с Nios II EDS.

Усовершенствование расширенных HAL API прерываний

Если у вас есть собственные драйверы устройств, Altera рекомендует вам усовершенствовать его для использования расширенных HAL API прерываний. Расширенное API имеет совместимость с IIC, поскольку поддерживает внешние контроллеры прерываний. Устаревшие HAL API прерываний не рекомендуются и будут исключены в следующих релизах Nios II EDS.

Если вы планируете использовать EIC, вы должны усовершенствовать собственный драйвер до расширенного HAL API прерываний.

Усовершенствовать драйвер просто – это потребует некоторые незначительные изменения в вызове некоторых функций.

В табл. 8-2 показаны устаревшие API функции, которые нужно заменить соответствующими расширенными функциями API.

За подробной информацией об этих функциях обратитесь к главе "Справка по HAL API" в настольной книге программиста Nios II.

Table 8–2. HAL Interrupt API Functions to Upgrade

Legacy API Function	Enhanced API Function
`alt_irq_register()`	`alt_ic_isr_register()`
`alt_irq_disable()`	`alt_ic_irq_disable()`
`alt_irq_enable()`	`alt_ic_irq_enable()`

Если ваш обновлённый драйвер должен работать в BSP с устаревшими драйверами, напишите его с поддержкой обеих API, как описано в секции "Поддержка нескольких API прерываний" на стр. 8-11.