Разработка программного обеспечения для процессора Nios II

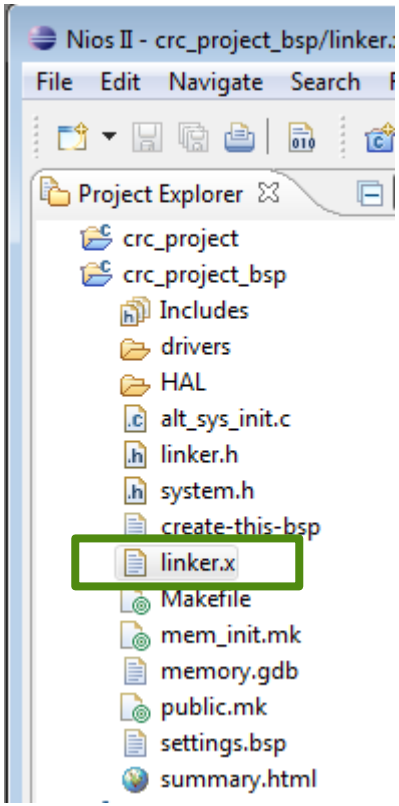
Разработка ПО

- Настройка компоновщика
- Прерывания
- Измерение быстродействия кода
- Разработка драйверов устройств

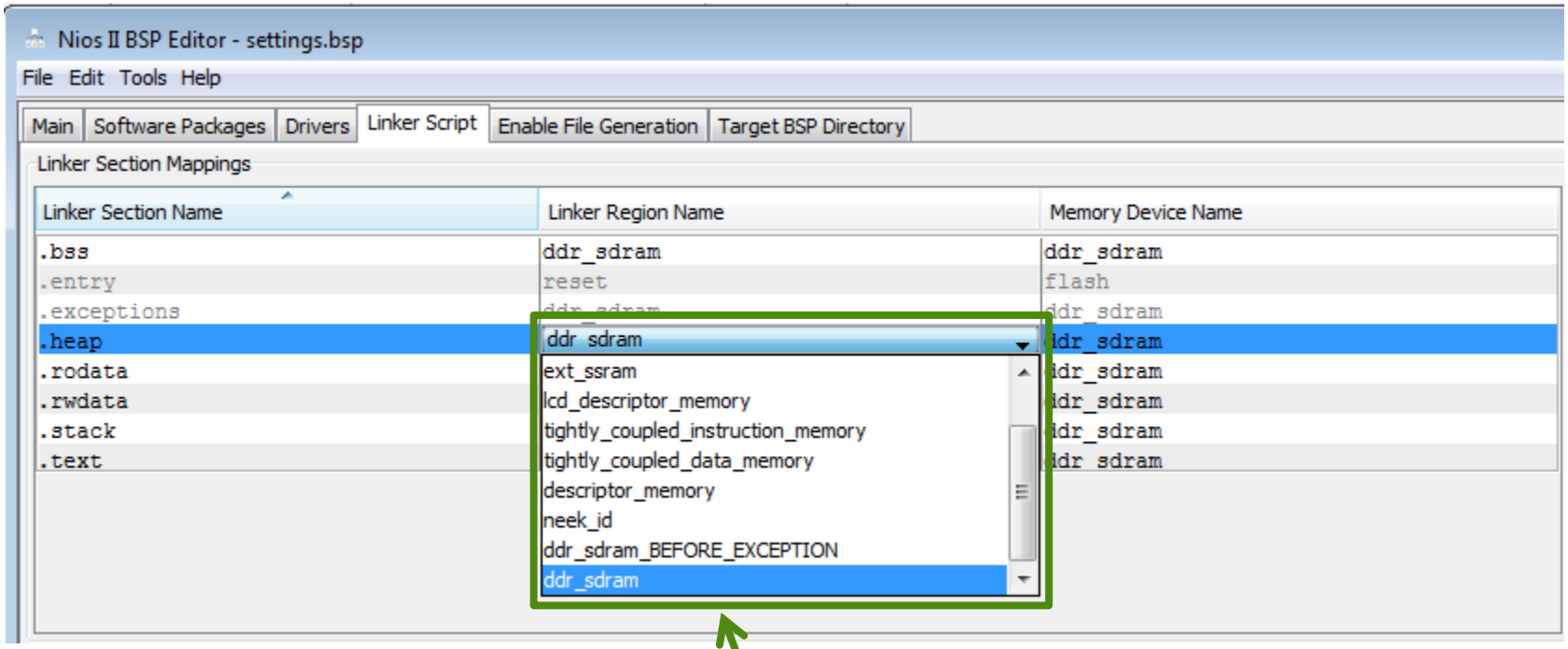
Настройка компоновщика

Скрипт компоновщика (Linker Script)

- Генерируется и управляется автоматически
  - Предназначен для систем на основе HAL
  - Находится в папке **BSP**
- Создает стандартные секции формата .elf для кода и данных
- Создает регионы для каждого физического устройства памяти в системе (TCM, Flash, SDRAM,…)
- Контролирует отображение секций кода и данных в регионы памяти



Выбор секций

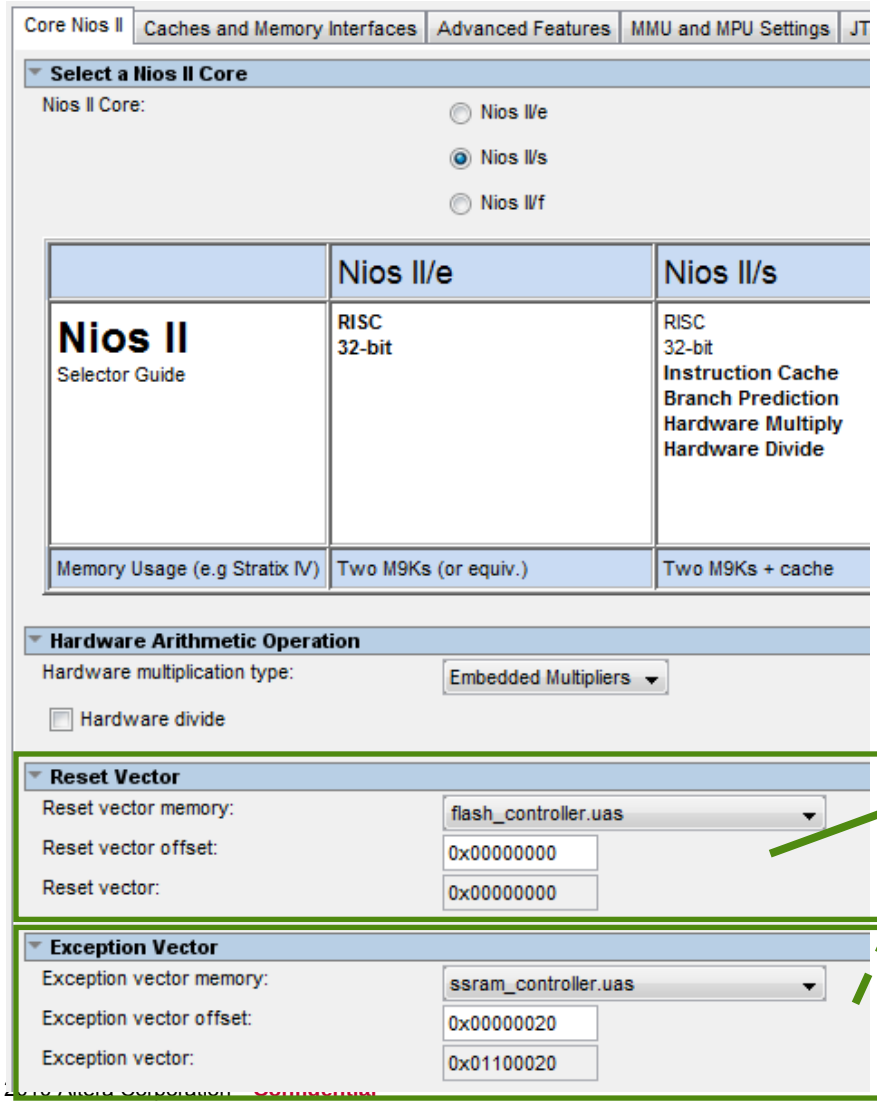


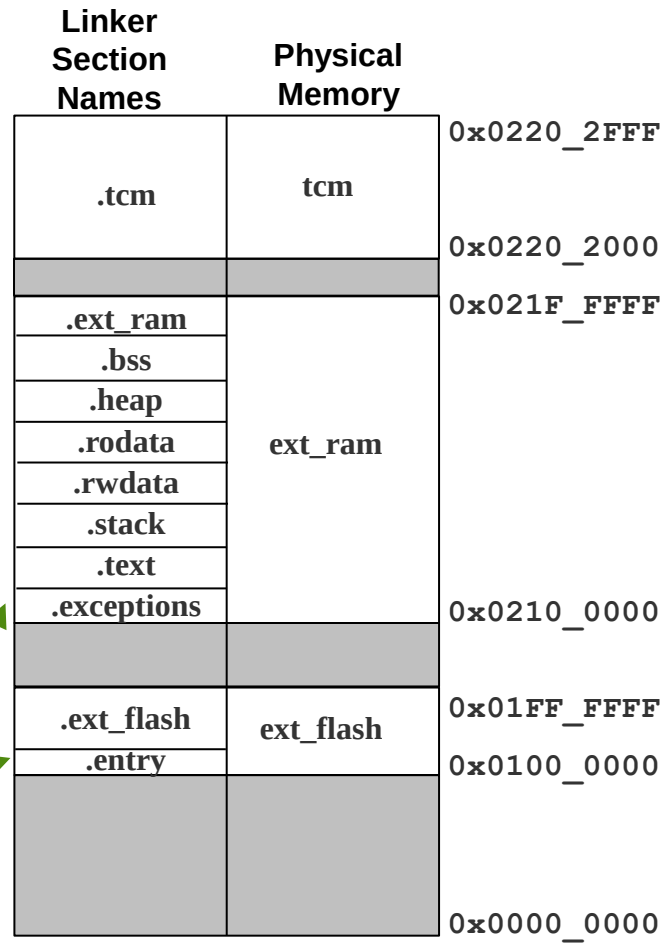
Настройка осуществляется на странице Linker Script в BSP Editor

Имена секций

Nios II Default Section Names	
Name	**Description**
.entry	Секция размером 32 байта по адресу вектора сброса reset
.exceptions	Секция по адресу обработчика исключений
.text	Остальной код
.rodata	Любые данные, доступные только на чтение
.rwdata	Инициализируемые данные, доступные для чтения и записи
.bss	Остальные данные, начальное значение – 0
.heap	Куча
.stack	Стек

Настройка векторов сброса и исключения





Пример Linker Script

```
SECTIONS
{
    .entry :
    {
        KEEP (*(.entry))
    } > reset

    .exceptions :
    {
```

Имя секции

Регион размещения секции

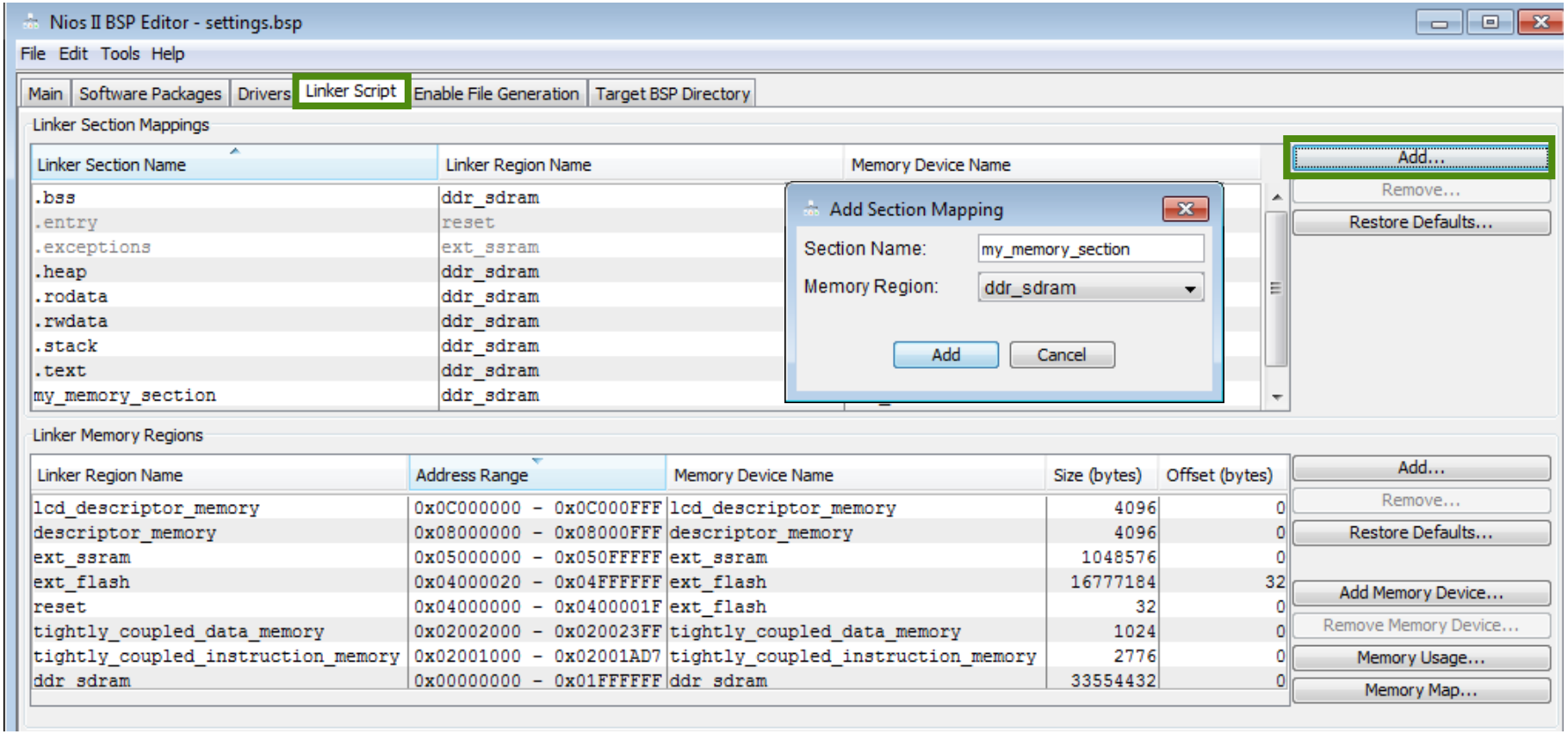
```
    } > ext_ram

.rodata :
    {
        PROVIDE (__ram_rodata_start = ABSOLUTE(.));
        . = ALIGN(4);
        *(.rodata .rodata.* .gnu.linkonce.r.*)
        *(.rodata1)
        . = ALIGN(4);
        PROVIDE (__ram_rodata_end = ABSOLUTE(.));
    } > ext_ram

    PROVIDE (__flash_rodata_start = LOADADDR(.rodata));
```

Размещение переменных в памяти

- Укажите имя секции для переменных и функций через атрибут `section` в коде C or C++
  - Для функций указывается для прототипа
- По умолчанию
  - Глобальные переменные - **.rwdata**
  - Функции - **.text**

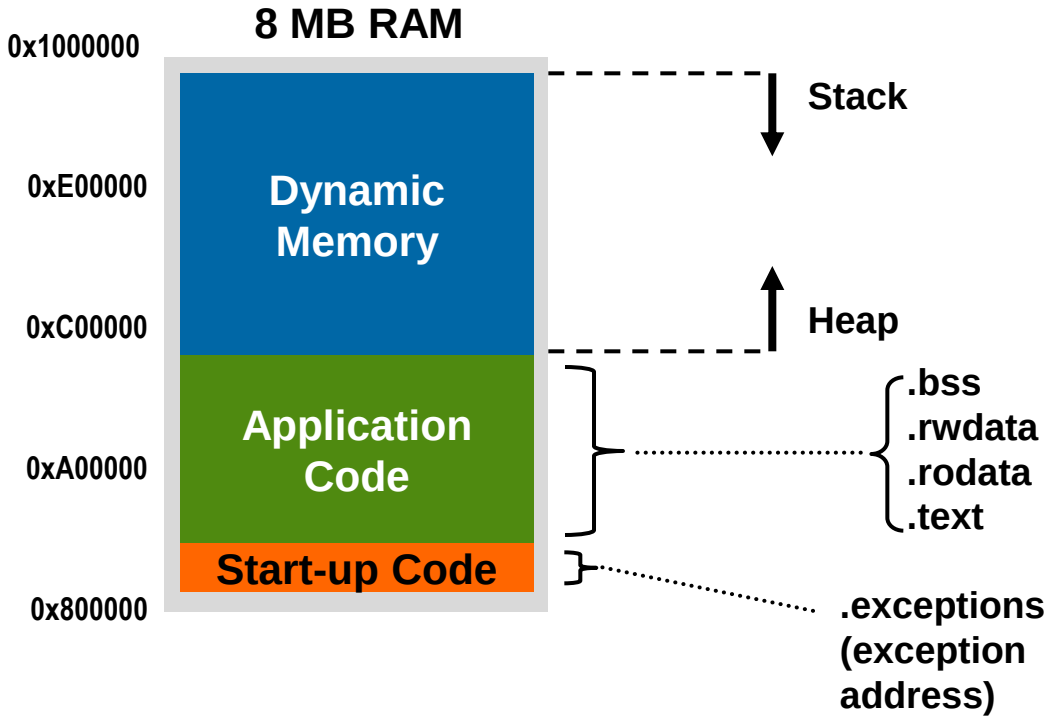
```
/* Force variable or function to reside in on_chip memory
Data using the "section" attribute should be initialized */

int  foo_var __attribute__ ((section (".on_chip_memory"))) = 0;
void bar_func(void* ptr) __attribute__ ((section (".ext_ram")));
```

Создание новых регионов и секций



Стек и куча

- Стек начинает расти с конца региона памяти и растет вниз (к младшим адресам)
- Куча растет вверх и размещается с первого свободного адреса в регионе памяти



Обработка прерываний

Обработка исключений в Nios II

- Все исключения обрабатываются одним обработчиком, размещенным по адресу “exception location”
  - Реализован в системной библиотеке, адрес определяется в **QSys** в мастере конфигурации процессора **Nios II**
- Поддерживаемые типы исключений
  - Программные исключения
    - Нереализованные или неверные команды и т.д.
    - Ловушки
    - Ошибки защиты памяти
  - Аппаратные прерывания
    - Поддерживается до 32 источников прерывания (без внешнего контроллера)
    - Поддерживаются прерывания по уровню

Контроллеры прерываний в Nios II

- Поддерживается два контроллера прерываний
  - Встроенный контроллер (**IIC**)
  - Внешний векторный контроллер (**EIC**)
- Внешний векторный контроллер позволяет также использовать теневые наборы регистроы для быстрого переключения контекста
- В лабораторных работах используется встроенный контроллер
- Программный интерфейс EIC поддерживается для обоих типов контроллеров
  - Рекомендуется к применению для обоих типов контроллеров
  - Тип контроллера распознается автоматически при создании BSP

Настройка приоритета EIC

- Программная настройка
  - Если в системе несколько EIC
  - **Настройка в BSP Editor → Drivers** (Больше номер – выше приоритет)
- Аппаратная настройка внутри EIC
  - Настраивается в QSys (Меньше номер – выше приоритет)

Функции работы с прерываниями

- **Библиотечные функции обработчиков прерываний**
  - alt_ic_isr_register()
    - Регистрирует обработчик прерывания.
  - alt_ic_irq_enable()
    - Разрешает прерывание.
  - alt_ic_irq_disable()
    - Запрещает прерывание.
  - alt_ic_irq_enabled()
    - Проверяет, разрешено ли прерывание.
  - alt_irq_disable_all()
    - Запрещает все аппаратные прерывания.
  - alt_irq_enable_all()
    - Разрешает все аппаратные прерывания.

Реализация обработчиков

Реализуйте обработчик

```
void sample_isr ( void* context, alt_u32
id);

context - указатель на данные, заданный
при регистрации обработчика
```

Зарегистрируйте обработчик alt_irq_register()

Прототип:

```
alt_ic_irq_register(alt_u32 ic_id, alt_u32
irq, alt_isr_func isr, void* isr_context,
void* flags);
```

Вызов:

```
alt_ic_irq_register
(PERIPH_IRQ_INTERRUPT_CONTROLLER_ID,
PERIPH_IRQ, sample_isr, &some_data,
0x0);
```

Пример 1. Обработчик прерывания

```
void my_isr( void* context, alt_u32 id)

{  /* Declare "context" as local pointer. This can be used to pass information
      into or out of this ISR. The "volatile" qualifier is used to prevent erroneous
      compiler optimization. */
     volatile int* my_var_ptr = (volatile int*) context;

   /* Either read value of my_var_ptr and act upon it, or set it's value by reading
      register in the hardware that caused the IRQ or both. */
      *my_var_ptr = IORD_ALTERA_AVALON_<DEVICE's_CAPTURE_REG>(<periph_base>);

                                  *From peripheral register map file
                                                          *From "system.h" file

   /* Reset hardware register/s.  */
     IOWR_ALTERA_AVALON_<DEVICE's_CAPTURE_REG>(<periph_base>, <value>);
     IOWR_ALTERA_AVALON_<DEVICE>_IRQ_MASK(<periph_base>, 0xf);

   /* NOTE:  It's common, in hardware,  to read a status register to reset an IRQ.
      This is what happens in the above instruction. */
   /* Also worth noting is the brevity of the ISR!
      It's good practice to service your interrupt as quickly as possible.
      Get the data you need, perform any *necessary* device control operation,
      clear interrupt, and move on!  */
}
```

Пример 1. Основная программа

```
/*
     NOTE:  The following steps usually occur in the main() function or
     -----   reside in a function called from main()
*/

int main (void)
{
   // Declare a variable, which is referenced (via a pointer) from within the ISR:
      volatile int my_var;

   // Initialize "my device" and register an ISR with the device:
      IOWR_ALTERA_AVALON_<DEVICE>_IRQ_MASK (<periph_base>, 0xf);
      alt_ic_irq_register((<PERIPHERAL>_IRQ_INTERRUPT_CONTROLLER_ID ,
                     <PERIPHERAL>_IRQ, my_isr, (void*) &my_var,
                     0x0);

   /*
      Once everything is initialized, the ISR will update the value of my_var any
      time a "my device" IRQ occurs.  How to act upon this change is an exercise
      left up to the user and is highly device dependent.
   */
                                                .
                                                .
                                                .
```

*From "system.h" file

Задержка обработки прерывания

- **Latency - задержка обработки прерывания**
  - Время (в тактах процессора), которое проходит от активизации источника прерывания до выполнения первой инструкции по адресу Exception Address
- **Response time - время отклика прерывания**
  - Время (в тактах процессора), которое проходит от активизации источника прерывания до выполнения первой инструкции обработчика прерывания
- **Recovery time - время выхода из обработчика прерывания**
  - (в тактах процессора), которое проходит от выполнения последней инструкции обработчика до продолжения выполнения прерванного кода

Table 8–1. Interrupt Performance Data *(1)*

Core	Latency	Response Time	Recovery Time
Nios II/f	10	105	62
Nios II/s	10	128	130
Nios II/e	15	485	222

Для кода во внутренней блочной памяти ПЛИС

Рекомендации по реализации

- Переносите сложную обработку из обработчика в приложение
- Не используйте стандартные библиотечные функции или функции операционной системы, которые могут блокироваться
  - Например printf()
- Избегайте запрещения прерываний на длительное время

- ***Литература:***

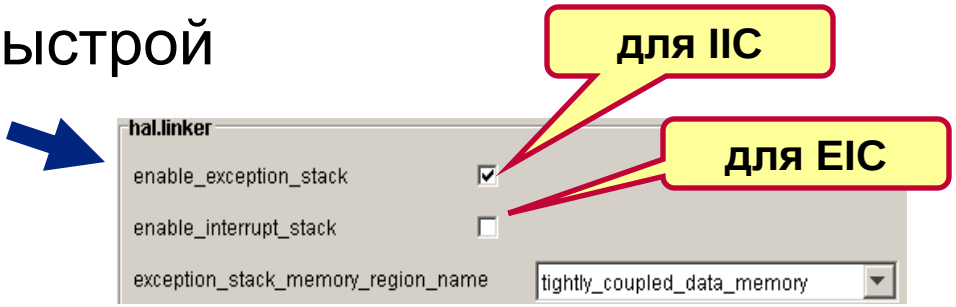
  (раздел Exception Handling в “Nios II Software Developer’s Handbook”)

Уменьшение времени отклика

- Задействуйте внешний контроллер прерываний
- Размещайте функции обработчиков в памяти на кристалле или модулях тесно связанной памяти

```
void my_isr __attribute__ ((section                    \
  (".tightly_coupled_instruction_memory ")));
```

- Размещайте стек программы или стек прерываний в быстрой памяти



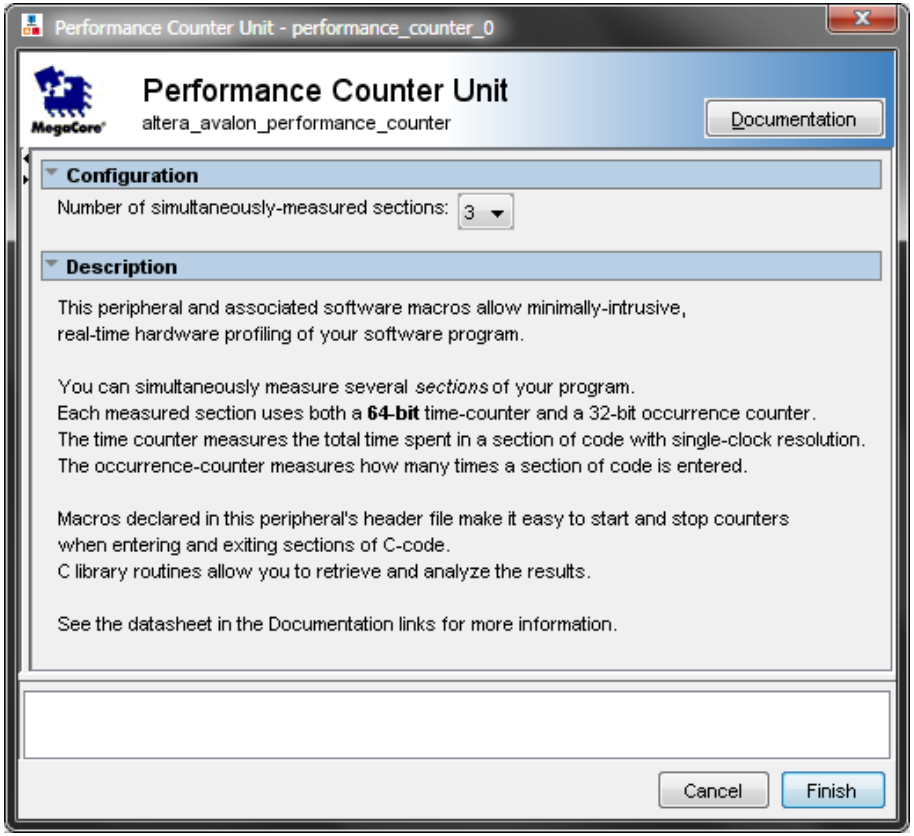
Измерение быстродействия кода

Измерение быстродействия кода

- Модуль Performance Counter
- Профайлер

Модуль Performance Counter

- Счетчик производительности
- Периферийный модуль QSys и набор макросов для использования ПО
- Модуль счетчика производительности может содержать несколько счетчиков секции **“section counters”**
  - По умолчанию 3
  - Могут использоваться параллельно для измерения быстродействия различных частей кода



Модуль Performance Counter

- Обеспечивает возможность профилирования быстродействия программного и аппаратного обеспечения системы на кристалле с минимальными накладными расходами
- Каждый счетчик секции “section counter” состоит из двух счетчиков:
  - 64-битный счетчик **Time** (считает такты ТИ системы на кристалле). Измеряет общее время пребывания в контролируемой части кода с точностью до одного периода тактового импульса.
  - 32-битный счетчик **Event** (считает количество обращений к счетчику). Измеряет количество обращений к контролируемой части кода.

Макросы Performance Counter

- Объявлены в
  - altera_avalon_performance_counter.h
- Обеспечивают возможность сброса, запуска и остановки счетчиков при входе в контролируемые секции кода и выходе их них
  - **PERF_RESET**(PERFORMANCE_COUNTER_BASE)
  - **PERF_START_MEASURING**(PERFOMANCE_COUNTER_BASE)
    - **PERF_BEGIN**(PERFORMANCE_COUNTER_BASE,SECTION_COUNTER_#)
    - **PERF_END**(PERFORMANCE_COUNTER_BASE,SECTION_COUNTER_#)
  - **PERF_STOP_MEASURING**(PERFOMANCE_COUNTER_BASE)
- Функция системной библиотеки обеспечивает считывание и форматированный вывод результатов измерений
  - **perf_print_formatted_report()**

Использование Performance Counter

- Счетчики должны быть сброшены перед использованием
- Запустите счетчик секции 0 для запуска всех счетчиков секций. Секция 0 используется для служебных целей – запуска/останова всех секций и подсчета общего времени исполнения и количества событий.
- Используйте остальные счетчики секций, начиная с 1 для измерения быстродействия
- Остановите счетчик секции 0 для остановки всех счетчиков
- Дополнительная информация:
  - Файл **system.h**
  - *Embedded IP user guide*

Пример использования

```
#include "altera_avalon_performance_counter.h"
#include "system.h"

int main() {

// Reset the counters before use:
  PERF_RESET (PERFORMANCE_COUNTER_BASE);
                :
                :
  PERF_START_MEASURING (PERFORMANCE_COUNTER_BASE);

  PERF_BEGIN (PERFORMANCE_COUNTER_BASE, 1);
                :
                :
    <code loop to be tested>
                :
                :
  PERF_END (PERFORMANCE_COUNTER_BASE, 1);

  PERF_STOP_MEASURING (PERFORMANCE_COUNTER_BASE);
```

Измеряем быстродействие

Отчет Performance Counter

```
perf_print_formatted_report ( PERFORMANCE_COUNTER_BASE,
                              ALT_CPU_FREQ, 3,
                              "DAC Wait",
                              "MLA loop",
                              "synth_frame"  );
```

Количество секций для вывода

Имена секций

Вывод:

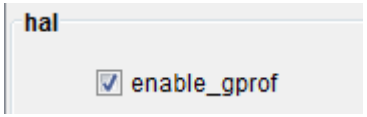
```
--Performance Counter Report--
Total Time: 140.2867 seconds  (1214372123 clock-cycles)
+---------------+-----+------------+---------------+-----------+
| Section       |  %  | Time (sec) |  Time (clocks)|Occurrences|
+---------------+-----+------------+---------------+-----------+
|DAC Wait       | 13.9|   17.85852|      1517976819|          5 |
+---------------+-----+------------+---------------+-----------+
|MLA loop       | 33.3|   42.86016 |     3643111107|          7 |
+---------------+-----+------------+---------------+-----------+
|synth_frame    | 52.8|   67.87189 |     5768259804|         18 |
+---------------+-----+------------+---------------+-----------+
```

Сравнение Performance Counter и профайлера

- **Performance Counter** измеряет количество тактов тактового импульса, проведенных в указанной области кода
- **Профайлер** периодически считывает значение счетчика команд и определяет, сколько времени было проведено в каждой функции, исходя из количества соответствий счетчика команд ее размещению в памяти программ во время сбора информации

Генерация данных для профайлера

- Для приложения и системной библиотеки
  - Установите **Debug Level** в “On (-g)”
- На странице свойств системной библиотеки
  - Включите **enable_gprof** в BSP Editor
  - Перегенерируйте BSP

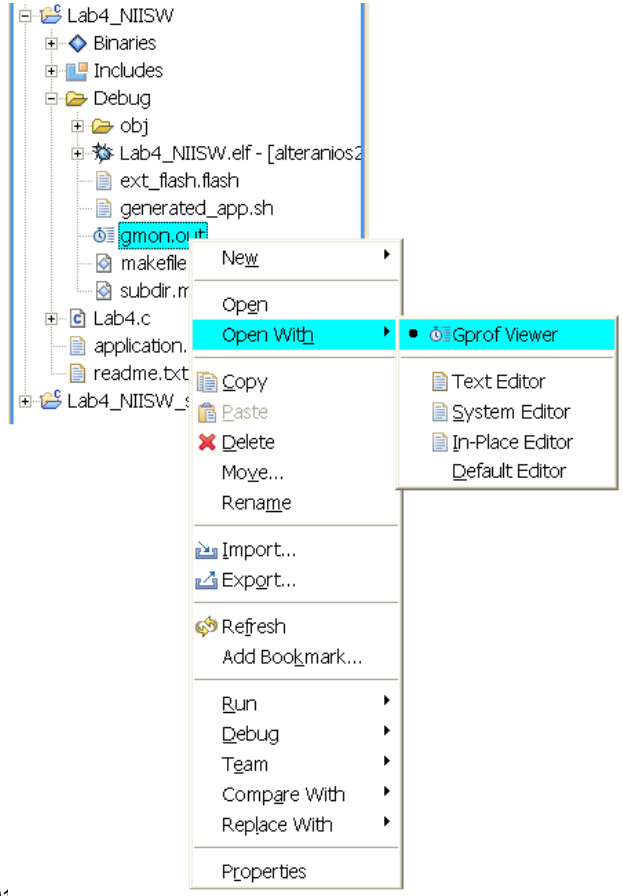


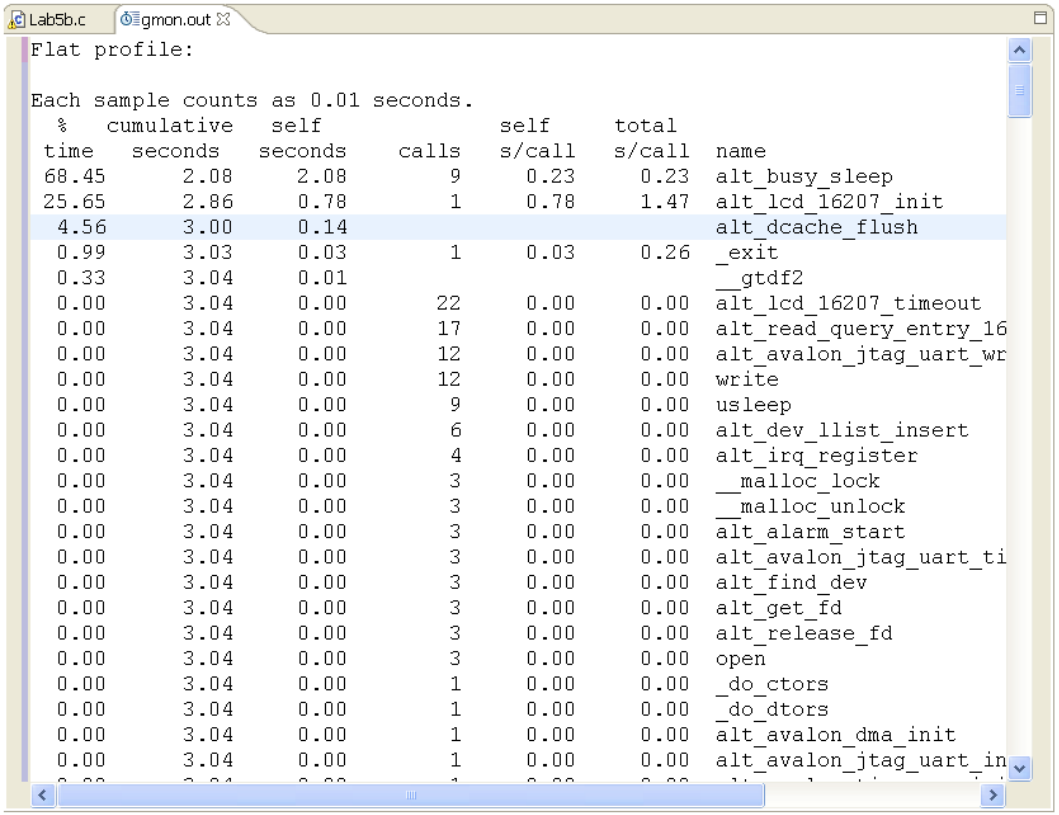
- Выполните очистку проекта и перекомпилируйте его
  - **Clean, затем Rebuild**
- Запустите программу и прогоните исполнение на требуемом наборе значений/воздействий. Программа должна завершить выполнение.
  - В составе проекта будет сгенерирован файл с результатами профилирования **gmon.out**
- Запустите анализ результатов профилирования
  - через **Nios II IDE** или командой **nios2-elf-gprof** в командной строке
- Дополнительная информация:

  <NiosII_Install_Dir>\documents\gnu-tools\binutils\gprof.html

Просмотр результатов в Nios II IDE

- Вызовите просмотр результатов из контекстного меню на файле “**gmon.out**”





Представление данных профайлера

- **Flat Profile**
  - Показывает, сколько времени программа провела в каждой функции и сколько раз функция вызывалась
- **Граф вызовов**
  - Для каждой функции показывает, сколько раз и какие функции вызывались из нее

Flat Profile

```
Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 98.71      2.30     2.30       14     0.16     0.16  alt_busy_sleep
  0.97      2.32     0.02        1     0.02     0.19  _exit
  0.32      2.33     0.01                             fstat
  0.00      2.33     0.00      249     0.00     0.00  alt_irq_handler
  0.00      2.33     0.00      234     0.00     0.00  alt_avalon_timer_sc_irq
  0.00      2.33     0.00      234     0.00     0.00  alt_tick
  0.00      2.33     0.00      196     0.00     0.00  __unpack_d
  0.00      2.33     0.00      142     0.00     0.00  udivmodsi4
  0.00      2.33     0.00      103     0.00     0.00  __pack_d
  0.00      2.33     0.00       78     0.00     0.00  __udivsi3
  0.00      2.33     0.00       64     0.00     0.00  __umodsi3
  0.00      2.33     0.00       35     0.00     0.00  _fpadd_parts
  0.00      2.33     0.00       33     0.00     0.00  __muldf3
  0.00      2.33     0.00       31     0.00     0.00
   __floatsidf
                                                          .
                                .
                                .
```

Call Graph

- Показывает структуру вызовов программы
- Вывод сортируется по времени, проведенному в каждой функции и вызванных ей функциях
- Каждый элемент в таблице содержит:
  - Индекс функции в левом столбце в квадратных скобках
    - Строки перед строкой с индексом в группе показывают, какие функции вызывали данную функцию
    - Строки под строкой с индексом показывают, какие функции вызывала данная функция

Пример Call Graph

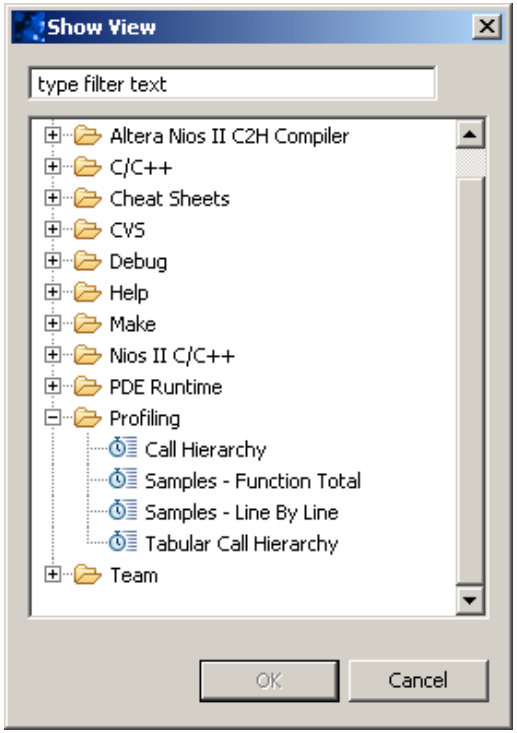
Вывод для первых двух функций:

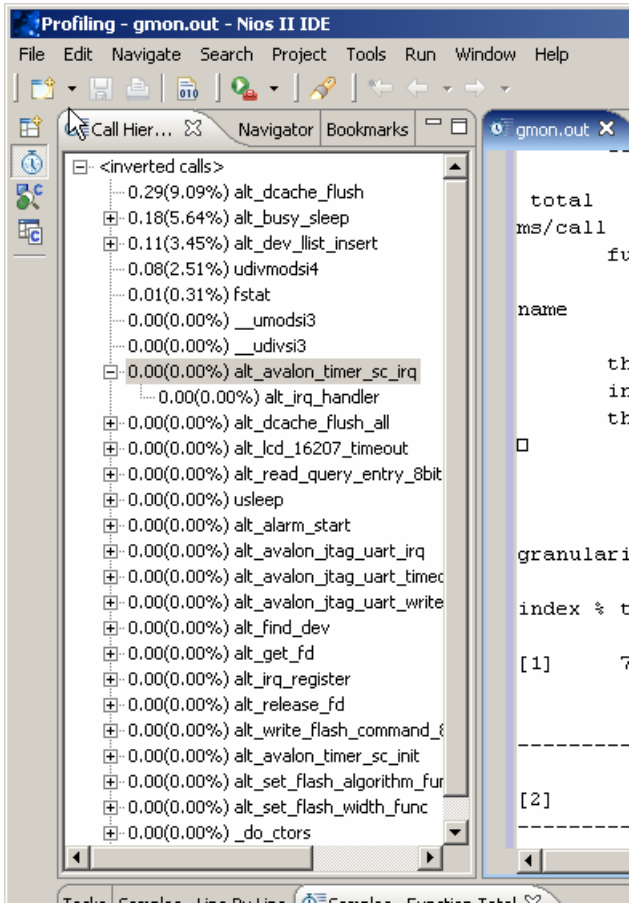
```
index % time    self  children    called     name
                670.33    0.00       4/12   led_intensity_test[4]
               1340.67    0.00       8/12   alt_lcd_16207_init [2]
[1]     99.6 2011.00    0.00       12     alt_busy_sleep [1]
-----------------------------------------------------------------
                  2.00 1340.67       1/1    alt_sys_init [3]
[2]     66.5      2.00 1340.67       1      alt_lcd_16207_init [2]
               1340.67    0.00       8/12   alt_busy_sleep [1]
                  0.00    0.00       8/12   usleep [17]
                  0.00    0.00       1/2    alt_alarm_start [27]
-----------------------------------------------------------------
```

...

Просмотр иерархии вызов

- **Window-> Show View -> Other…**





Дополнительная информация

- Embedded IP User Guide
  - Performance Counter Core
- Application Note AN391
  - “Profiling Nios II Systems”

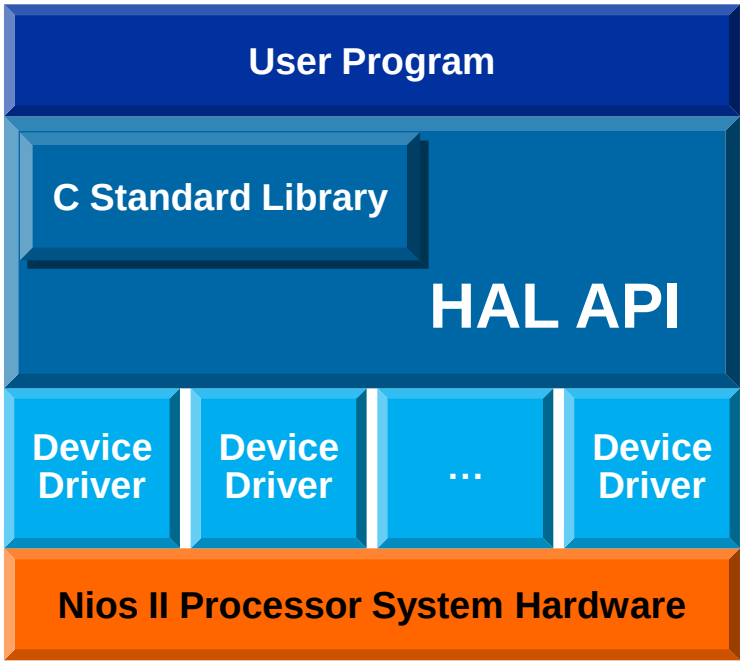
Разработка драйверов устройств

Разработка драйверов

- **Драйверы компонент**
- Типы драйверов
- Разработка драйверов
- Интеграция драйвера в HAL

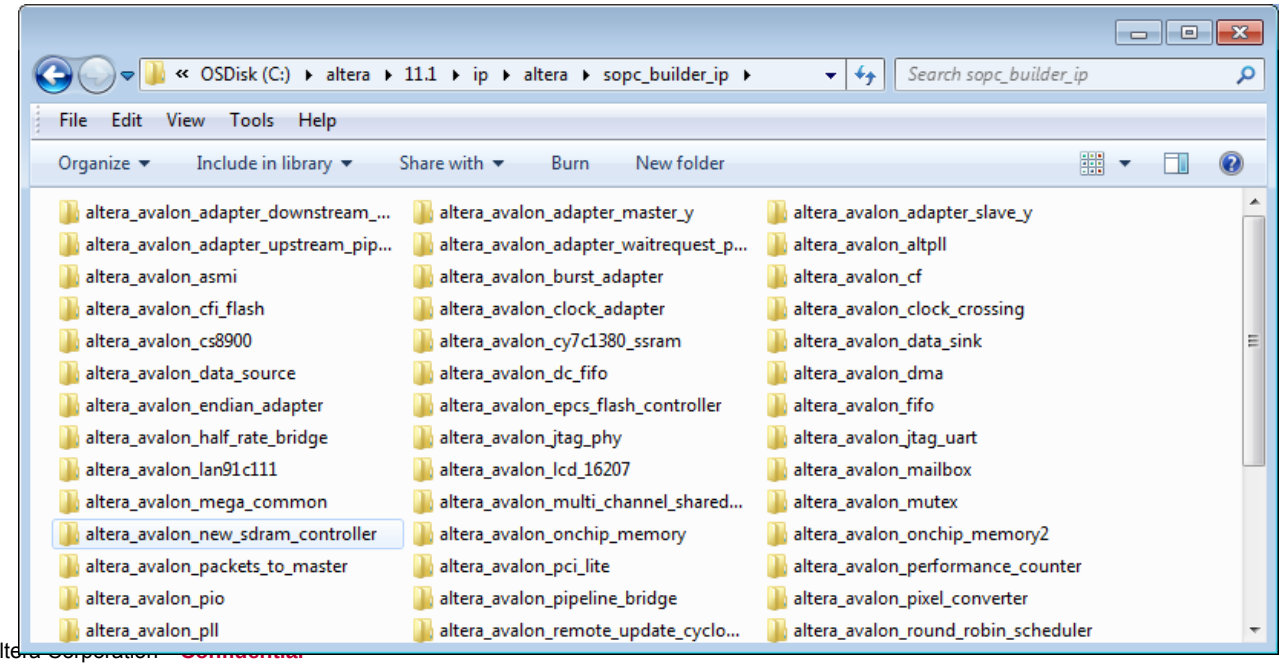
Драйверы периферийных устройств

- Для всех периферийных устройств требуется драйвер
  - Реализует взаимодействие с аппаратурой
  - Осуществляет доступ к регистрам устройства
  - Устраняет необходимость включать низкоуровневый код в код приложения

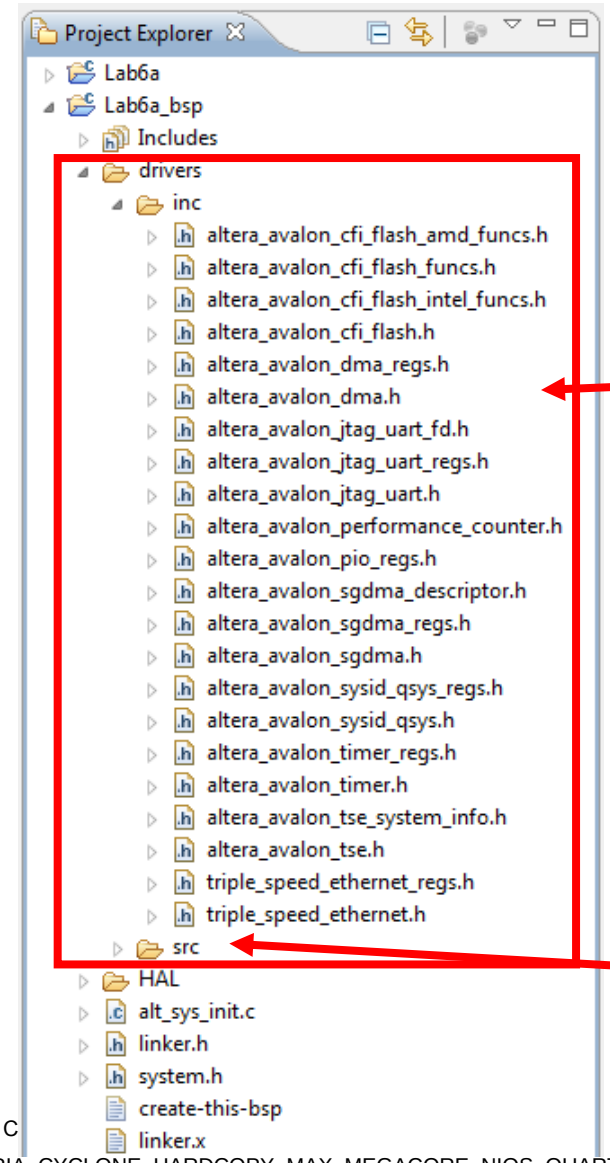


Драйверы стандартных компонент Qsys

- Реализованы для всех стандартных периферийных компонент QSys
  - Автоматически включаются в состав проекта
  - Находятся в **C:\altera\<ver>\ip\altera\sopc_builder_ip**



Драйверы копируются в папку BSP



Папка “drivers/inc” Заголовочные файлы для драйверов всех компонент системы

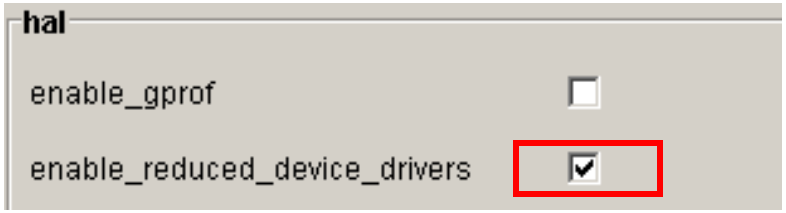
Исходные файлы

Разработка драйверов

- Драйверы компонент
- **Типы драйверов**
- Разработка драйверов
- Интеграция драйвера в HAL

Сложность драйверов

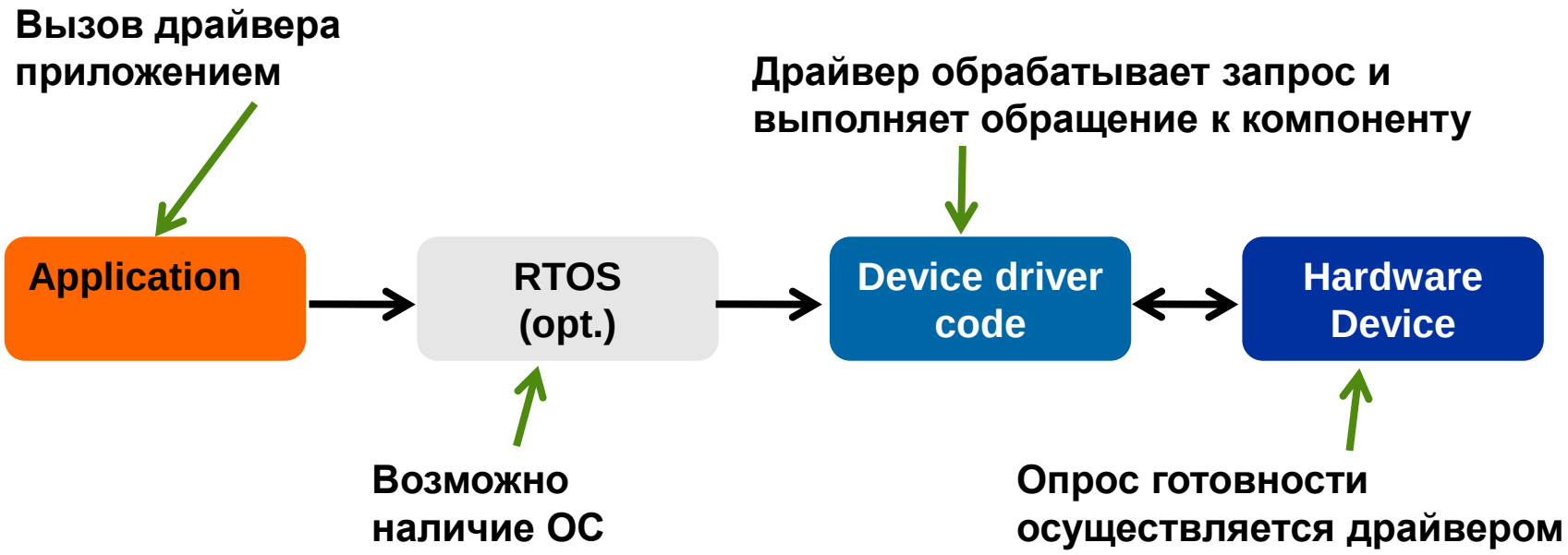
- Зависит от функции и сложности компоненты
  - Может представлять собой заголовочный файл с обращением к регистрам компоненты (Например, файл **avalon_pwm_regs.h** для **avalon_pwm.v**)
  - Может реализовывать более сложные функции для различных редимов работы компоненты
- Для некоторых компонент реализовано два типа драйверов - “Small” и“Fast”
  - “Small” => Синхронный
  - “Fast” => Асинхронный



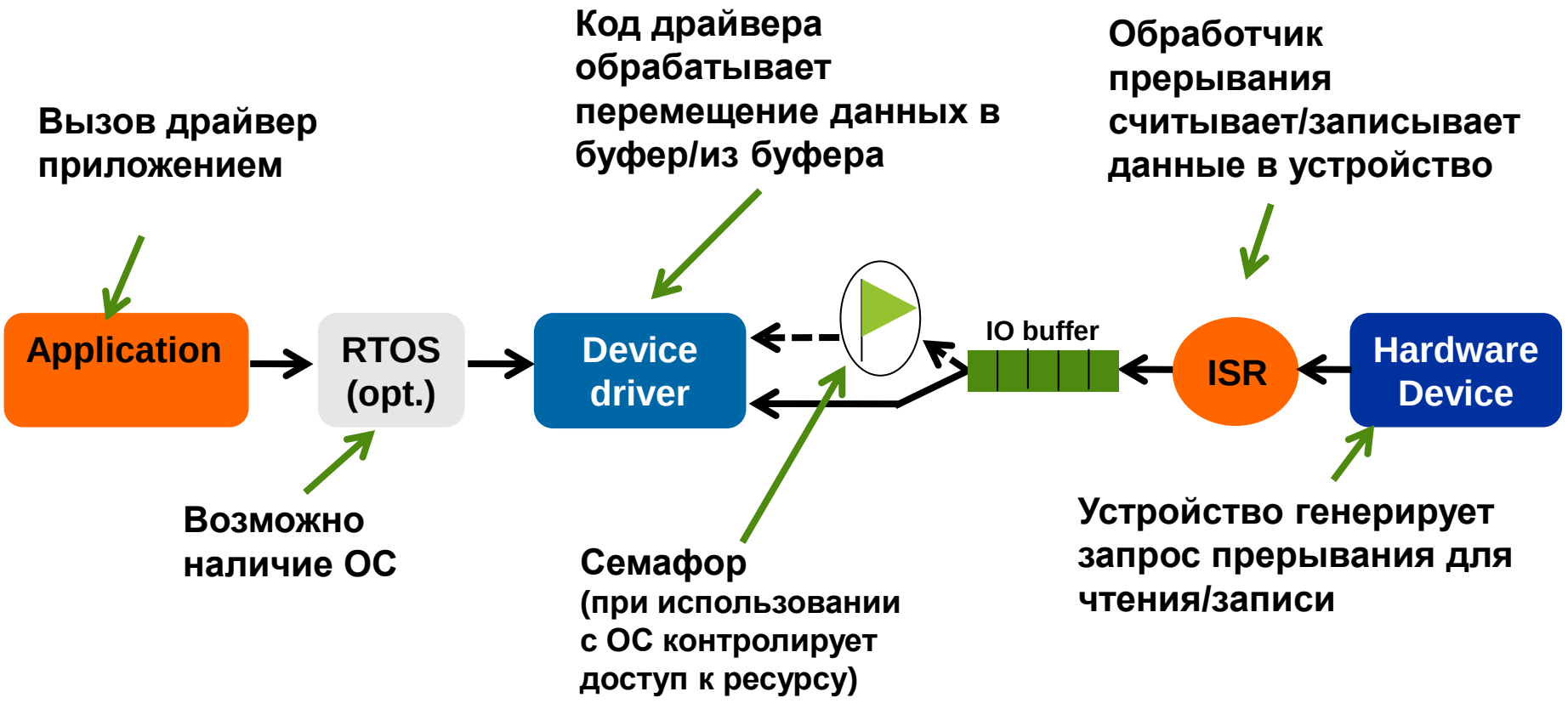
Архитектура драйвера

- Синхронный
  - Более простой, “блокирующий” драйвер
  - При вызове функции драйвера возврат из нее не осуществляется до завершения операции
  - Приложение (или задача ОС) ожидает завершения операции
  - Функция драйвера опрашивает статус готовности устройства
- Асинхронный
  - Более сложный, “неблокирующий” драйвер
  - При вызове функции драйвера возврат из нее осуществляется до завершения операции
  - Приложение (или задача ОС) продолжает выполняться в процессе выполнения операции
  - В драйвере реализуется буфер ввода-вывода
  - Требует большего объема кода
  - Обычно использует обработчик прерывания

Работа синхронного драйвера



Работа асинхронного драйвера



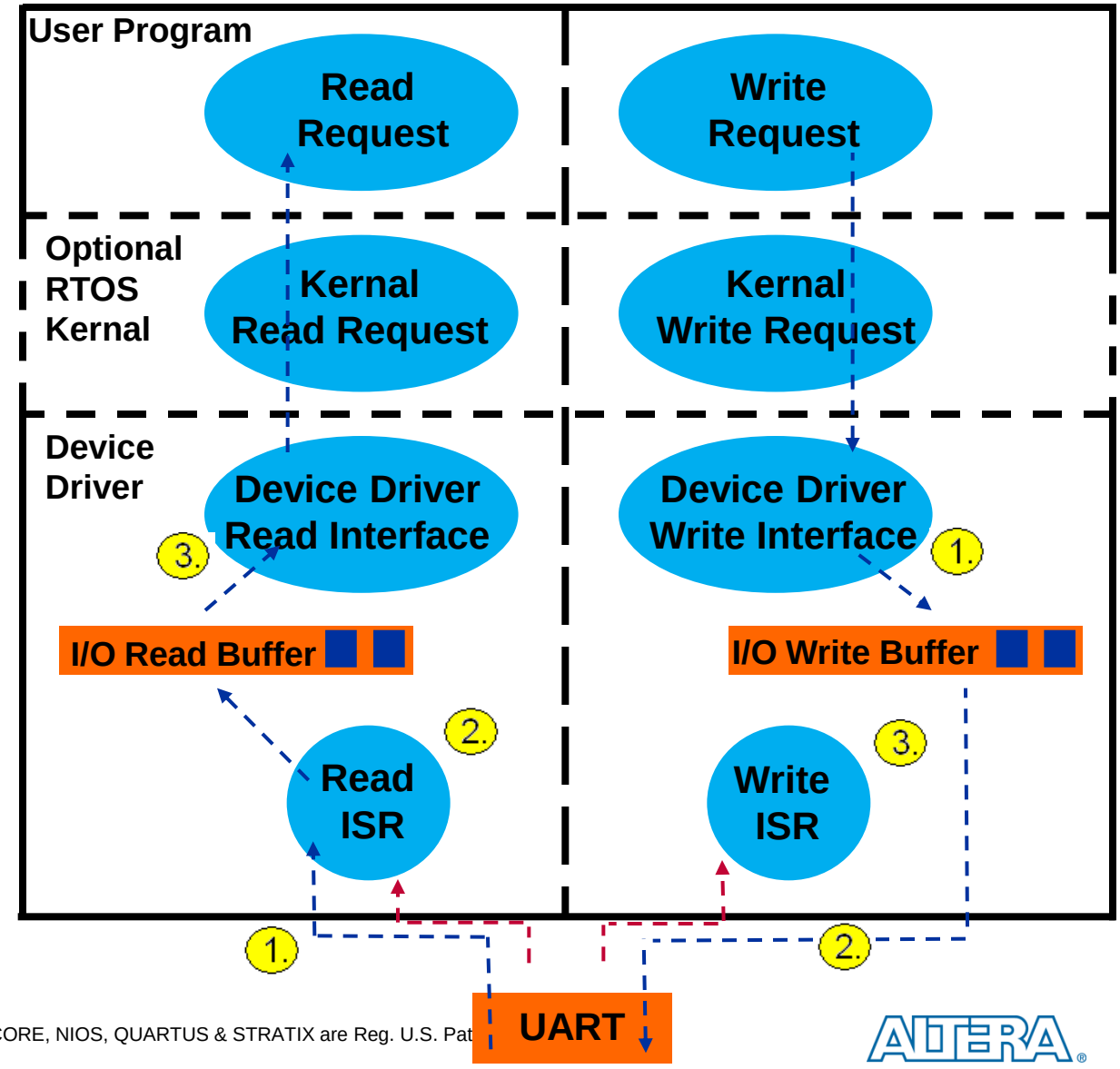
Altera Asynchronous UART Driver

Чтение:

1. UART сигнализирует о наличии данных через прерывание
2. Обработчик считывает данные и помещает в буфер
3. Данные из буфера перемещаются в приложение

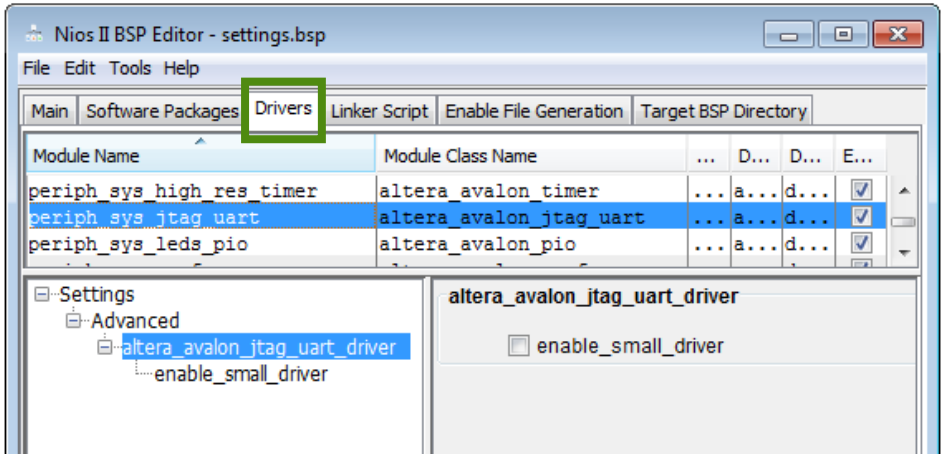
Запись:

1. Драйвер заполняет буфер данными для вывода
2. UART сигнализирует о готовности через прерывание
3. Обработчик прерывания обрабатывает запрос и пересылает байт из буфер в UART



Для одного устройства могут быть реализованы разные модели

- Определяются в настройках BSP Editor
  - JTAG UART
  - UART
  - Common flash interface
  - LCD module controller
  - EPCS serial configuration device



Разработка драйверов

- Драйверы компонент
- Типы драйверов
- **Разработка драйверов**
- Интеграция драйвера в HAL

Разработка драйверов

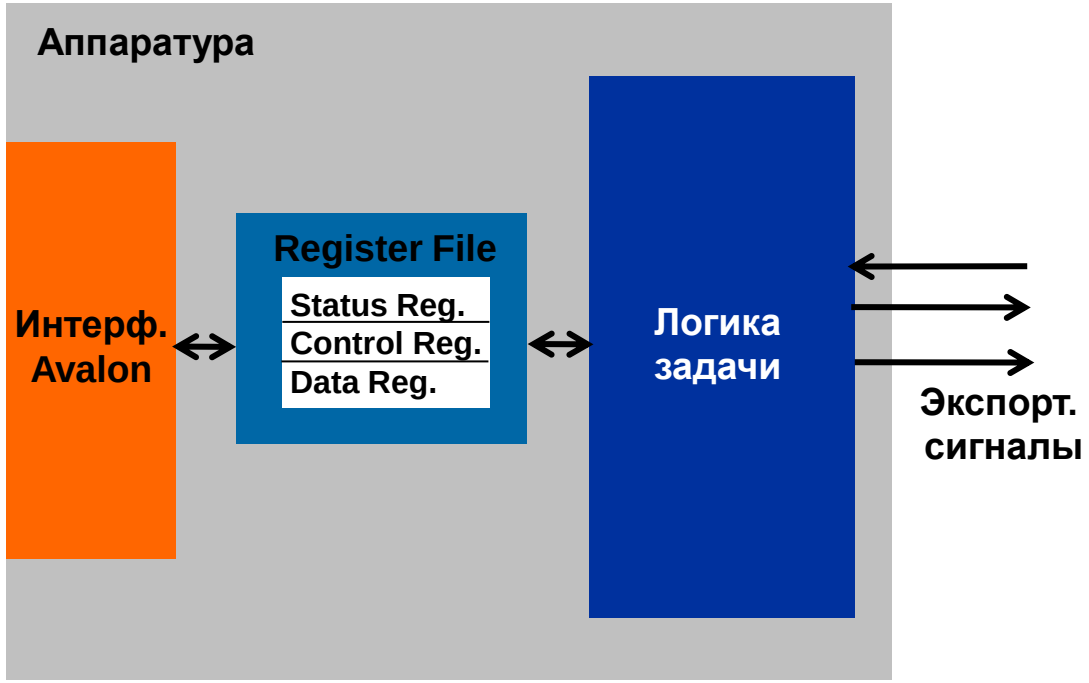
- Выбрать тип и сложность драйвера
- Можно ли использовать существующие сервисы HAL?
  - Использовать предопределенный HAL API?
  - Разработать специальный API компонента?
- На каком уровне интегрировать драйвер в HAL?
  - Многоуровневая (поддержка разных моделей)?
  - Не интегрировать в HAL, а добавить в приложение?
  - Интегрировать в HAL и сделать доступным при формировании BSP любого приложения с этим компонентом?

Шаги разработки

Функции прямого доступа к регистрам через IORD иIOWR

ISR

Функции управления состоянием и передачей данных



Базовые типы Altera HAL

Поддерживаемые разрядности (`#include <stdarg.h>`)	
Тип	**Значение**
alt_8	Signed 8 bit integer
alt_u8	Unsigned 8 bit integer
alt_16	Signed 16 bit integer
alt_u16	Unsigned 16 bit integer
alt_32	Signed 32 bit integer
alt_u32	Unsigned 32 bit integer
alt_64	Signed 64 bit integer
alt_u64	Unsigned 64 bit integer

Типы C/C++ в nios2-elf-gcc

Тип	Значение
char	8 bits
short	16 bits
int	32 bits
long	32 bits
float	32 bits
long long	64 bits

Макросы ввода-вывода в HAL

Макросы ввода-вывода HAL обходят обращения в кэш данных (#include <io.h>)	
Макрос	**Назначение**
IORD(BASE, REGNUM)	Считать значение регистра со смещением REGNUM в устройстве с базовым адресом в карте памяти BASE. Смещение задается в словах.
IOWR(BASE, REGNUM, DATA)	Запись значения DATA в регистр со смещением REGNUM в устройстве с базовым адресомв карте памяти BASE. Смещение задается в словах.
IORD_32DIRECT(BASE, OFFSET)	32-х разрядное чтение по адресу BASE + OFFSET
IORD_16DIRECT(BASE, OFFSET)	16-ти разрядное чтение по адресу BASE + OFFSET
IORD_8DIRECT(BASE, OFFSET)	8-ми разрядное чтение по адресу BASE + OFFSET
IOWR_32DIRECT(BASE, OFFSET, DATA)	Запись 32-х разрядного значения DATA по адресу BASE + OFFSET
IOWR_16DIRECT(BASE, OFFSET, DATA)	Запись 16-ти разрядного значения DATA по адресу BASE + OFFSET
IOWR_8DIRECT(BASE, OFFSET, DATA)	Запись 8-ми разрядного значения DATA по адресу BASE + OFFSET

Определенные классы моделей устройств

- Символьные устройства (UART, LCD)
- Таймеры
- Устройства ПДП (DMA)
- Устройства Ethernet
- Файловые подсистемы (Altera Read-Only Zip FS)
- Устройства Flash памяти

Использование классов моделей устройств

- Для классов моделей устройств HAL определены сервисы, которые они предоставляют
- Сервисы могут быть использованы любым периферийным компонентом, для которого они подходят
  - Каждая модель определяет набор функций драйвера для работы с данным классом устройства
  - Требуется разработать функции с таким интерфейсом для нового компонента
  - Упрощается разработка, так как шаги разработки определены и документированы
  - Для доступа к устройству могут быть использованы существующие функции HAL

Символьные устройства

- Аппаратура, которая посылает/получает символы последовательно (экраны, принтеры, последовательные порты)
- Требует применения структуры класса модели устройства **alt_dev**
- **alt_dev** включает несколько функций доступа к устройству (“device accessor functions”), обеспечивая таким образом интерфейс между устройством и общепринятым HAL API для работы с сиволами
- Обращение осуществляется как к файлу

Структура alt_dev

- Содержит указатели на функции
  - Вызов каждой функции осуществляется в ответ на определенные запросы к HAL
  - См. “C:\altera\<ver>\nios2eds\components\altera_hal\HAL\inc\sys\alt_dev.h”

```
typedef struct {
  alt_llist    llist; /* for internal use */
  const char*  name;
  int (*open)  (alt_fd* fd, const char* name, int flags, int mode);
  int (*close) (alt_fd* fd);
  int (*read)  (alt_fd* fd, char* ptr, int len);
  int (*write) (alt_fd* fd, const char* ptr, int len);
  int (*lseek) (alt_fd* fd, int ptr, int dir);
  int (*fstat) (alt_fd* fd, struct stat* buf);
  int (*ioctl) (alt_fd* fd, int req, void* arg);
} alt_dev;
```

Назначение функций

Описаны в HAL API Reference

Функция	Описание
*open	Открывает устройство при вызове функций **“open()”** или **“fopen()”** для устройства в API файловой системы HAL
*close	Вызывается при вызове функций закрытия устройства **“close()”** и **“fclose()”**
*read	Вызывается при вызове функций **“read()”** или **“fscanf()”** для чтения из устройства
*write	Вызывается при вызове функций **“write()”** или **“fprintf()”** для записи в устройство
*lseek	Перемещает указатель в файле на заданное смещение в байтах
*fstat	Возвращает информацию о возможностях открытого дескриптора файла
*ioctl	Вызывается при вызове функции POSIX **“ioctl()”** для доступа к специальным функциям устройства (не покрывается read/write и т.д.)

Если в таблице задано значение NULL

Поведение по умолчанию	
Функция	**Описание**
*open	Вызовы open() проходят успешно, если не был вызван запрос блокировки TIOEXCL через вызов ioctl()
*close	Вызовы close() для действительного дескриптора файла всегда проходят успешно
*read	Вызовы read() возвращают ошибку
*write	Вызовы write() возвращают ошибку
*lseek	Вызовы lseek() возвращают ошибку
*fstat	Устройство идентифицирует себя как символьное (character mode device)
*ioctl	Если вызов ioctl() не может быть обработан HAL ьез обращения к устройству, возвращается ошибка

Предоставление сервисов устройства

- Драйвер или программа должны создать экземпляр alt_dev
- alt_dev должна быть зарегистрирована в HAL через вызов alt_dev_reg()
  - Описывает устройство как узел в файловой системе HAL
- Процесс может быть автоматизирован (описывается далее)

```
#include <stdio.h>                  // Access std. IO functions
#include "sys/alt_dev.h"     // Access to alt_dev(reg)
```

Код драйвера символьного устройства (1)

```
#include  "sys/alt_dev.h"                    // include "alt_dev.h" in driver
#include  "system.h"
#include  "altera_avalon_mylcd_regs.h"       // Define register access for
#include  <stdio.h>                          // char mode LCD peripheral

typedef struct {                             // Define device structure
  alt_dev dev;                               // for your LCD peripheral
  unsigned int base;                         // here ("my_lcd_dev")
} my_lcd_dev;

/* Write function - "my_lcd_write" */
int my_lcd_write (alt_fd* fd, const char* ptr, int len)
{
  int i;
  my_lcd_dev* dev = (my_lcd_dev*) fd->dev;   // "fprintf()", etc. will map to
  for (i = 0; i < len; i++)                  // this function
  {
    IOWR_ALTERA_AVALON_MYLCD_DATA (dev->base, ptr[i]);
  }                                          // Note:  Could also use actual
  return len;                                // base address of LCD instead
}                                            // of "dev->base" here
```

Код драйвера символьного устройства (2)

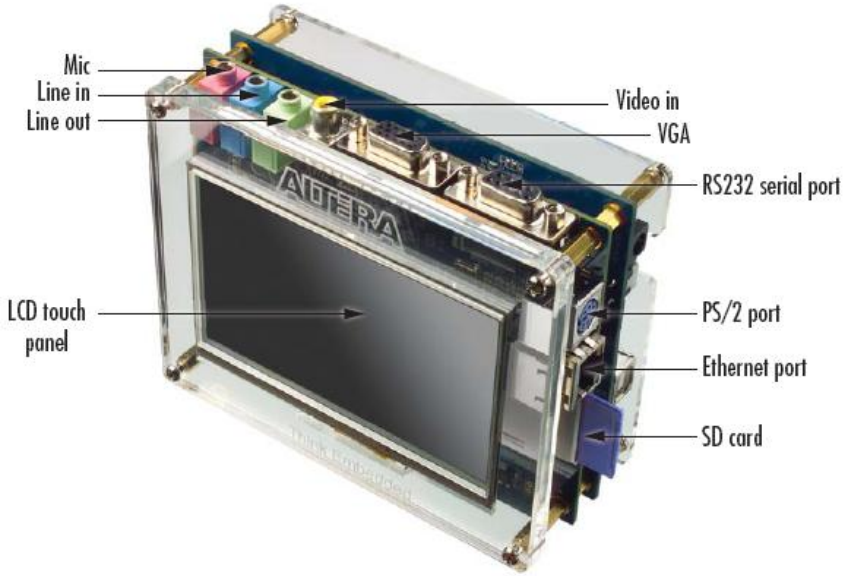
```
int main()
{
    FILE * fd;

    /* Instantiate the device */
    my_lcd_dev lcd_device =
    {                                  // Device instantiated in your C code,
        {                              //with services selected . . .
            ALT_LLIST_ENTRY,           // Instance name here is "lcd_device"
            "/dev/lcd_periph",
            NULL,                      /*open*/
            NULL,                      /*close*/
            NULL,                      /*read*/
            (void*) my_lcd_write,      // "fprintf()" maps to write function
            NULL,                      /*lstat*/
            NULL,                      /*lseek*/
            NULL,                      /*ioctl*/
        },
        LCD_PERIPH_BASE                // system.h
    };  // End of my_lcd_dev
```

Код драйвера символьного устройства (3)

```
    alt_dev_reg(&lcd_device);       // Register device with the HAL
                                    // Makes it available to your program
     /* Use the device */
     fd = fopen ("/dev/lcd_periph", "w");
     fprintf (fd, "hello device number %i\n", 33); // Example Use
     fclose(fd);
     return 0;
}
```

Более сложные драйверы устройств

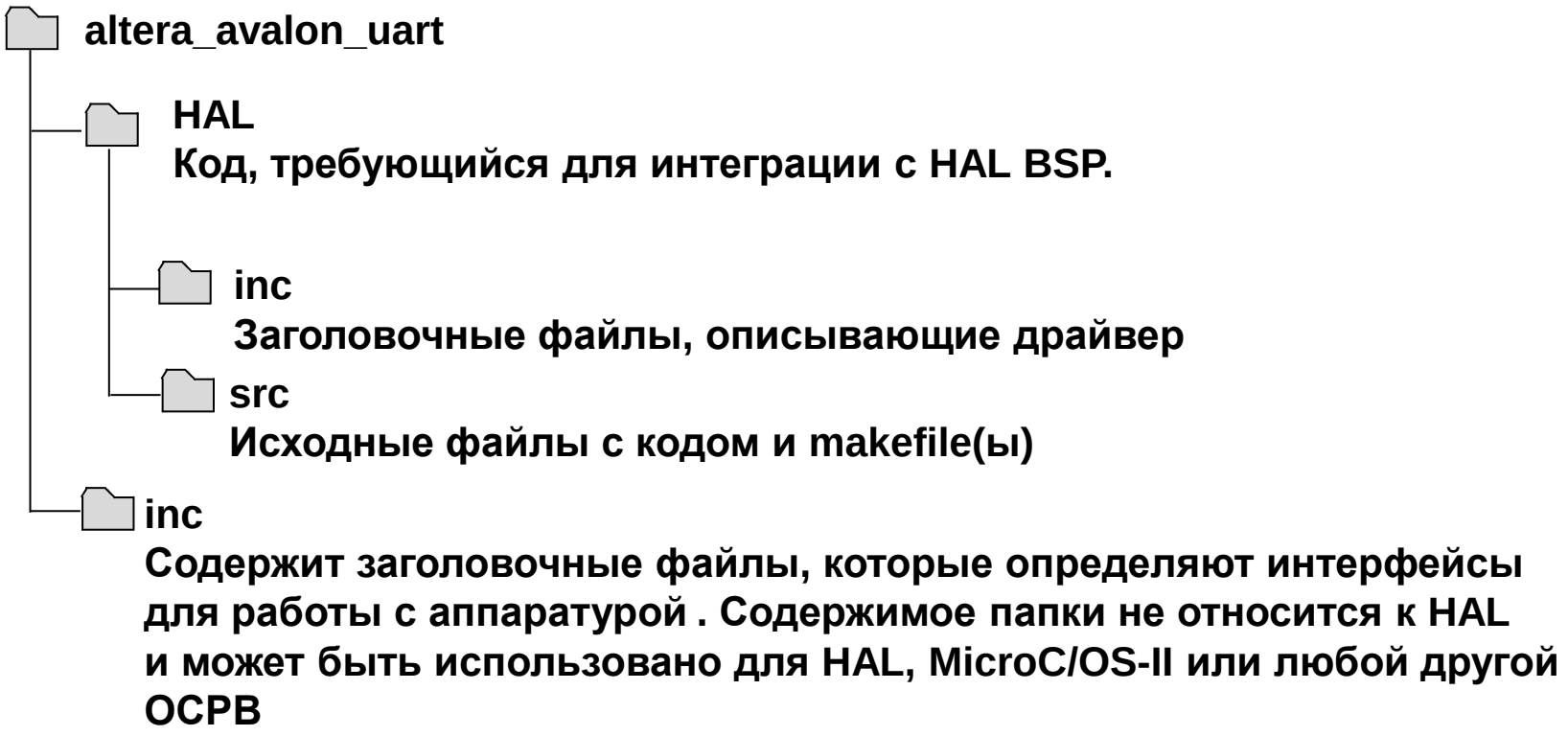
- Могут потребовать дополнительных уровней кода в драйвере (стек)
- Пример – вывод символов и графики на LCD панель



Размещение кода altera_avalon_uart

- Размещение драйвера HAL

  C:\altera\<ver>\ip\altera\sopc_builder_ip\altera_avalon_uart



HAL\src\component.mk

- Для каждого драйвера периферийного устройства требует локального makefile

```
# List all source files supplied by this component.
C_LIB_SRCS   += altera_avalon_uart_fd.c     \
                altera_avalon_uart_init.c   \
                altera_avalon_uart_ioctl.c  \
                altera_avalon_uart_read.c   \
                altera_avalon_uart_write.c
ASM_LIB_SRCS +=
INCLUDE_PATH +=
```

Список исходных файлов на C для включения и сборки с BSP

Список папок для добавлеия в путь поиска заголовочных файлов
Папка <*component*>/HAL/inc добавляется автоматически

Список файлов на ассемблере для включения и сборки с BSP

Файл <имя компонента>_sw.tcl

- Описание драйвера на TCL (Tool Command Language)
  - Используется Nios II SBT для автоматической интеграции драйвера в BSP
  - Команды Tcl описывают файлы драйвера и специальные настройки

```
#
# Source file listings...
#
# C/C++ source files
add_sw_property c_source HAL/src/altera_avalon_uart_fd.c
add_sw_property c_source HAL/src/altera_avalon_uart_init.c
add_sw_property c_source HAL/src/altera_avalon_uart_ioctl.c
add_sw_property c_source HAL/src/altera_avalon_uart_read.c
add_sw_property c_source HAL/src/altera_avalon_uart_write.c
# Include files
add_sw_property include_source HAL/inc/altera_avalon_uart.h
add_sw_property include_source HAL/inc/altera_avalon_uart_fd.h
add_sw_property include_source inc/altera_avalon_uart_regs.h
# This driver supports HAL & UCOSII BSP (OS) types
add_sw_property supported_bsp_type HAL
add_sw_property supported_bsp_type UCOSII
```

Вызов alt_sys_init() интегрирует драйвер в HAL

Включает заголовочные файлы для всех периферийных модулей QSys, перечисленных в файле.sopcinfo

```
#include "altera_avalon_uart.h"
                    :
                    :
ALTERA_AVALON_UART_INSTANCE(MY_UART, my_uart);
                    :
                    :

void alt_sys_init( void )
{
                    :
                    :
   ALTERA_AVALON_UART_INIT(MY_UART, my_uart) ;
                    :
                    :
}
```

Вызов макросов INSTANCE для всех компонент системы

Вызов макросов INIT для всех компонент системы для инициализации устройств

Разработка драйверов

- Драйверы компонент
- Типы драйверов
- Разработка драйверов
- **Интеграция драйвера в HAL**

Интеграция драйвера в HAL

- Интеграция драйвера в HAL позволяет использовать его во всех проектах с использованием данного компонента
  - Лучший выбор для обеспечения портируемости и повторного использования кода

Разработка HAL-совместимого драйвера

- Создайте папку с именем компонента
- Создайте в ней структуру папок
  - inc
  - HAL\inc
  - HAL\src

Эта папка может быть скопирована в папку с компонентами в пути установки Nios II:

C:\altera\<ver>\ip\altera\sopc_builder_ip\components

- Разработайте файлы драйвера с именем компонента в качестве префикса:
  - inc\<имя компонента>_regs.h
  - HAL\inc\<имя компонента>.h
  - HAL\src\<имя компонента>.c
  - HAL\src\component.mk
  - <имя компонента>_sw.tcl

Для создания файлов .mk и _sw.tcl можно использовать в качестве примера файлы других компонент

Разные версии драйвера

- Многоуровневый драйвер
  - Используете команды препроцессора `ifdef`'s для связывания многоуровневой функциональности с настройкой Reduced Device Drivers в свойствах BSP


```
#if defined(ALT_USE_SMALL_DRIVERS) || defined(LCD_PERIPH_SMALL)
/* Use "Small" Version of the Driver */
                                        .
                                        .
                                        .
#else
/* Using the "fast" version of the driver */
                                        .
                                        .
                                        .
```

Рекомендации по разработке драйверов

- Разработайте драйвер в коде приложения
  - Разработайте заголовочный файл для обращения к ренистрам компонента
  - Используйте структуры для подходящей модели класса устройства
  - Проведите тестирование и отладку всех вызовов функций (инициализации и доступа к устройству)
- Используйте имена, соответствующие требованиям HAL, для:
  - Имен папок и исходных файлов
  - Регистров устройства
  - Функций доступа в исходном коде
- Интегрируйте драйвер в HAL
    - Можно использовать altera_avalon_uart как пример