Embedded Systems

Altera NIOS II Software Development:
Part II

Using Timer Devices

- Timer devices are hardware peripherals that count clock ticks and can generate periodic interrupt requests
- A timer device can be used to provide a number of time-related facilities, such as:
  - HAL system clock, alarms, the time-of-day, and time measurement
- The HAL API provides two types of timer device drivers:
  - System clock driver: Supports alarms, such as you would use in a scheduler
  - Timestamp driver: Supports high-resolution time measurement
- The HAL-specific API functions for accessing timer devices are defined in sys/alt_alarm.h and sys/alt_timestamp.h

System Clock Driver

- The HAL system clock driver provides a periodic heartbeat, causing the system clock to increment on each beat
- Software can use the system clock facilities to execute functions at specified times, and to obtain timing information
- A specific hardware timer peripheral is defined as the system clock device within the BSP settings

System Clock Driver (continued)

- The HAL provides implementations of the following standard UNIX functions:
  - gettimeofday(), settimeofday(), and times()
- The times returned by these functions are based on the HAL system clock
- The system clock measures time in clock ticks
- The HAL system clock is not the same as the clock signal driving the NIOS II processor hardware
- The period of a HAL system clock tick is generally much longer than the hardware system clock. system.h defines the clock tick frequency
  - This is a coarse grain timer

Timer Excerpt from system.h

```
/*
 * sys_clk_timer configuration
 *
 */

#define ALT_MODULE_CLASS_sys_clk_timer altera_avalon_timer
#define SYS_CLK_TIMER_ALWAYS_RUN 0
#define SYS_CLK_TIMER_BASE 0x8000160
#define SYS_CLK_TIMER_COUNTER_SIZE 32
#define SYS_CLK_TIMER_FIXED_PERIOD 0
#define SYS_CLK_TIMER_FREQ 60000000u
#define SYS_CLK_TIMER_IRQ 8
#define SYS_CLK_TIMER_IRQ_INTERRUPT_CONTROLLER_ID 0
#define SYS_CLK_TIMER_LOAD_VALUE 599999ull
#define SYS_CLK_TIMER_MULT 0.0010
#define SYS_CLK_TIMER_NAME "/dev/sys_clk_timer"
#define SYS_CLK_TIMER_PERIOD 10.0
#define SYS_CLK_TIMER_PERIOD_UNITS "ms"
#define SYS_CLK_TIMER_RESET_OUTPUT 0
#define SYS_CLK_TIMER_SNAPSHOT 1
#define SYS_CLK_TIMER_SPAN 32
#define SYS_CLK_TIMER_TICKS_PER_SEC 100u
#define SYS_CLK_TIMER_TIMEOUT_PULSE_OUTPUT 0
#define SYS_CLK_TIMER_TYPE "altera_avalon_timer"
```


System Clock Driver (continued)

- The current value of the system clock can be obtained by calling the alt_nticks() function
  - Returns the elapsed time in system clock ticks since reset
- The system clock rate, in ticks per second, can be obtained by calling the function alt_ticks_per_second()
- The HAL timer driver initializes the tick frequency when it creates the instance of the system clock

- The standard UNIX function gettimeofday() is available to obtain the current time
- First calibrate the time of day by calling settimeofday()
- In addition, times() function can be used to obtain information about the number of elapsed ticks
  - The prototypes for these functions appear in times.h

Alarms

- Alarms can be considered part of the exception processing system
- User functions can be registered to be executed at a specified time using the HAL alarm facility
- A software program registers an alarm by calling the function:

alt_alarm_start():

```
int alt_alarm_start (alt_alarm* alarm,
                     alt_u32    nticks,
                     alt_u32    (*callback) (void* context),
                     void*      context);
```


Alarms (continued)

- The function callback() is called after nticks have elapsed
- The input argument context is passed as the input argument to callback() when the call occurs
  - The HAL does not use the context parameter
  - It is only used as a parameter to the callback() function
- User code must allocate the alt_alarm structure, pointed to by the input argument alarm
  - This data structure must have a lifetime that is at least as long as that of the alarm
  - The best way to allocate this structure is to declare it as a static or global
    - For a static variable, the last value of the variable is preserved between successive calls to a function
- alt_alarm_start() initializes *alarm

Alarms (continued)

- The callback function can reset the alarm
- The return value of the registered callback function is the number of ticks until the next call to callback
  - A return value of zero indicates that the alarm should be stopped
- You can manually cancel an alarm by calling alt_alarm_stop()
- One alarm is created for each call to alt_alarm_start()
  - Multiple alarms can run simultaneously

Using a Periodic Alarm Callback Function

```
#include <stdio.h>
#include <stddef.h>
#include "sys/alt_alarm.h"
#include "alt_types.h"

// The callback function.

alt_u32 my_alarm_callback (void* context)
{
/* This function is called once per second */
    printf("Alarm called\n");
return alt_ticks_per_second();
}

int main()
{
  /* The alt_alarm must persist for the duration of the alarm. */
  static alt_alarm alarm;

  printf("Hello from Nios II!\n");
  if (alt_alarm_start (&alarm, alt_ticks_per_second(), my_alarm_callback, NULL) < 0)
  {
      printf("No system clock available\n");
  }
  while(1){}
  return 0;
}
```

Timestamp Driver

- You may need to measure time intervals with a degree of accuracy greater than that provided by HAL system clock ticks
- The HAL provides high resolution timing functions using a timestamp driver
- A timestamp driver provides a monotonically increasing counter that you can sample to obtain timing information
- The HAL only supports one timestamp driver in the system
- You specify a hardware timer peripheral as the timestamp device by manipulating BSP settings
- The Altera-provided timestamp driver uses the timer that you specify

Timestamp Driver (continued)

- If a timestamp driver is present, the following functions are available:
  - alt_timestamp_start()
  - alt_timestamp()
- Calling alt_timestamp_start() starts the counter running
- Subsequent calls to alt_timestamp() return the current value of the timestamp counter
- Calling alt_timestamp_start() again resets the counter to zero
- The behavior of the timestamp driver is undefined when the counter reaches $(2^{32} - 1)$

Timestamp Driver (continued)

- The rate at which the timestamp counter increments can be obtained by calling the function alt_timestamp_freq()
- This rate is typically the hardware frequency of the NIOS II processor system
  - Usually millions of cycles per second
- The timestamp driver functions are defined (prototyped) in the **alt_timestamp.h** header file

Timestamp Driver Example (part 1)

```
#include <stdio.h>
#include "sys/alt_timestamp.h"
#include "alt_types.h"

int func1() {
    int i,j,k;
    i=j+k;
    return 0;
}

int func2() {
    int i,j,k;
    i=j*k;
    return 0;
}
```

Timestamp Driver Example (part 2)

```
int main (void)
{
    alt_u32 time1;
    alt_u32 time2;
    alt_u32 time3;
    if (alt_timestamp_start() < 0) {
        printf ("No timestamp device available\n");
    }
    else {
        time1 = alt_timestamp();
        func1(); /* first function to monitor */
        time2 = alt_timestamp();
        func2(); /* second function to monitor */
        time3 = alt_timestamp();
        printf ("time in func1 = %u ticks\n", (unsigned int) (time2 - time1));
        printf ("time in func2 = %u ticks\n", (unsigned int) (time3 - time2));
        printf ("Number of ticks per second = %u\n", (unsigned int)alt_timestamp_freq());
    }
    return 0;
}
```


Basic Parallel Input/Output (PIO) Devices

- The HAL provides basic functions for low level read/write access to PIO devices
  - IOWR_ALTERA_AVALON_PIO_DATA
  - IORD_ALTERA_AVALON_PIO_DATA
- Device names are defined in **system.h**
- MACROS for accessing devices via low level register I/O read/write functions are defined in **altera_avalon_pio_regs.h**

Typical PIO Output Device Defined in system.h

```
#define ALT_MODULE_CLASS_led_pio altera_avalon_pio
#define LED_PIO_BASE 0x8000190
#define LED_PIO_BIT_CLEARING_EDGE_REGISTER 0
#define LED_PIO_BIT_MODIFYING_OUTPUT_REGISTER 0
#define LED_PIO_CAPTURE 0
#define LED_PIO_DATA_WIDTH 2
#define LED_PIO_DO_TEST_BENCH_WIRING 0
#define LED_PIO_DRIVEN_SIM_VALUE 0x0
#define LED_PIO_EDGE_TYPE "NONE"
#define LED_PIO_FREQ 60000000u
#define LED_PIO_HAS_IN 0
#define LED_PIO_HAS_OUT 1
#define LED_PIO_HAS_TRI 0
#define LED_PIO_IRQ -1
#define LED_PIO_IRQ_INTERRUPT_CONTROLLER_ID -1
#define LED_PIO_IRQ_TYPE "NONE"
#define LED_PIO_NAME "/dev/led_pio"
#define LED_PIO_RESET_VALUE 0x3
#define LED_PIO_SPAN 16
#define LED_PIO_TYPE "altera_avalon_pio"
```


Typical PIO Input Device Defined in system.h

```
#define ALT_MODULE_CLASS_button_pio altera_avalon_pio
#define BUTTON_PIO_BASE 0x8000180
#define BUTTON_PIO_BIT_CLEARING_EDGE_REGISTER 0
#define BUTTON_PIO_BIT_MODIFYING_OUTPUT_REGISTER 0
#define BUTTON_PIO_CAPTURE 1
#define BUTTON_PIO_DATA_WIDTH 4
#define BUTTON_PIO_DO_TEST_BENCH_WIRING 1
#define BUTTON_PIO_DRIVEN_SIM_VALUE 0xf
#define BUTTON_PIO_EDGE_TYPE "RISING"
#define BUTTON_PIO_FREQ 60000000u
#define BUTTON_PIO_HAS_IN 1
#define BUTTON_PIO_HAS_OUT 0
#define BUTTON_PIO_HAS_TRI 0
#define BUTTON_PIO_IRQ 0
#define BUTTON_PIO_IRQ_INTERRUPT_CONTROLLER_ID 0
#define BUTTON_PIO_IRQ_TYPE "EDGE"
#define BUTTON_PIO_NAME "/dev/button_pio"
#define BUTTON_PIO_RESET_VALUE 0x0
#define BUTTON_PIO_SPAN 16
#define BUTTON_PIO_TYPE "altera_avalon_pio"
```

Example HAL-Based PIO Operations

- Example PIO data write

```
volatile alt_u8 led;
/* Turn the LEDs on. Active low output*/
led = 0x00;
IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, led & 0xF);
```

- Example PIO data read

```
/* Unpressed buttons are active high by default */
volatile alt_u8 buttons;
buttons=IORD_ALTERA_AVALON_PIO_DATA(BUTTON_PIO_BASE);
```


Using Flash Devices

- The HAL provides a generic device model for nonvolatile flash memory devices
- Flash memories use special programming protocols to store (write) data
- The HAL API provides functions to write data to flash memory
- For example, you can use these functions to implement a flash-based file subsystem
- The HAL API also provides functions to read flash, although it is generally not necessary
- For most flash devices, programs can treat the flash memory space as simple memory when reading, and do not need to call special HAL API functions

HAL API for Flash Devices

- The API provides two levels of access to a flash device
- Simple flash access
  - Functions that write buffers to flash and read them back at the block level
  - In writing, if the buffer is less than a full block, these functions erase preexisting flash data above and below the newly written data
- Fine-grained flash access
  - Functions that write buffers to flash and read them back at the buffer level
  - In writing, if the buffer is less than a full block, these functions preserve preexisting flash data above and below the newly written data
  - This functionality is generally required for managing a file subsystem
- The API functions for accessing flash devices are defined in **sys/alt_flash.h**

Simple Flash Access

- This interface consists of the functions
  - alt_flash_open_dev()
  - alt_write_flash()
  - alt_read_flash(), and
  - alt_flash_close_dev()
- A flash device is opened by calling alt_flash_open_dev(), which returns a file handle to a flash device
- This function takes a single argument that is the name of the flash device, as defined in **system.h**

Simple Flash Access (continued)

- Use the alt_write_flash() function to write data to the flash device
- The prototype is:

```
int alt_write_flash( alt_flash_fd* fd,
                     int offset,
                     const void* src_addr,
                     int length )
```

- A call to this function writes to the flash device identified by the handle **fd**
- The driver writes the data starting at **offset** bytes from the base of the flash device
- The data written comes from the address pointed to by **src_addr**, and the amount of data written is **length**

Simple Flash Access (continued)

- The alt_read_flash() function is used to read data from the flash device
- The prototype is:

```
int alt_read_flash(   alt_flash_fd* fd,
                      int offset,
                      void* dest_addr,
                      int length )
```

- A call to alt_read_flash() reads from the flash device with the handle **fd**, **offset** bytes from the beginning of the flash device
- The function writes the data to location pointed to by **dest_addr**, and the amount of data read is **length**
- For most flash devices, you can access the contents as standard memory, making it unnecessary to use alt_read_flash()

Simple Flash Access (continued)

- The function alt_flash_close_dev() takes a file handle and closes the device
- The prototype for this function is:

```
void alt_flash_close_dev(alt_flash_fd* fd )
```


Block Erasure or Corruption

- Generally, flash memory is divided into blocks
- alt_write_flash() might need to erase the contents of a block before it can write data to it
- In this case, it makes no attempt to preserve the existing contents of the block
- This action can lead to unexpected data corruption (erasure), if you are performing writes that do not fall on block boundaries
- If you wish to preserve existing flash memory contents, use the fine-grained flash functions

Block Erasure Example

- Assume an 8-kilobyte (KB) flash memory comprising two 4-KB blocks
- Assume the following action is performed:
  - First write 5 KB of all 0xAA to flash memory at address 0x0000, and then write 2 KB of all 0xBB to address 0x1400
- After the first write succeeds (at time t(2)), the flash memory contains 5 KB of 0xAA, and the rest is empty (that is, 0xFF)
- Then the second write begins, but before writing to the second block, the block is erased
- At this point, t(3), the flash contains 4 KB of 0xAA and 4 KB of 0xFF
- After the second write finishes, at time t(4), the 2 KB of 0xFF at address 0x1000 is corrupted

Block Erasure Example

Example of Writing Flash and Causing Unexpected Data Corruption

Address	Block	Time t(0)	Time t(1)	Time t(2)	Time t(3)	Time t(4)
			First Write		Second Write	
		Before First Write	After Erasing Block(s)	After Writing Data 1	After Erasing Block(s)	After Writing Data 2
0x0000	1	??	FF	AA	AA	AA
0x0400	1	??	FF	AA	AA	AA
0x0800	1	??	FF	AA	AA	AA
0x0C00	1	??	FF	AA	AA	AA
0x1000	2	??	FF	AA	FF	FF *(1)*
0x1400	2	??	FF	FF	FF	BB
0x1800	2	??	FF	FF	FF	BB
0x1C00	2	??	FF	FF	FF	FF

(1) Unintentionally cleared to FF during erasure for second write.

Using Simple Flash API Functions

```
#include <stdio.h>
#include <string.h>
#include "sys/alt_flash.h"
#define BUF_SIZE 1024

int main ()
{
  alt_flash_fd* fd;
  int           ret_code;
  char          source[BUF_SIZE];
  char          dest[BUF_SIZE];

  /* Initialize the source buffer to all 0xAA */
  memset(source, 0xAA, BUF_SIZE);

  fd = alt_flash_open_dev("/dev/ext_flash");
```


Using Simple Flash API Functions

```
  if (fd!=NULL)
  {
    ret_code = alt_write_flash(fd, 0, source, BUF_SIZE);
    if (ret_code==0)
    {
      ret_code = alt_read_flash(fd, 0, dest, BUF_SIZE);
      if (ret_code==0)
      {
        /*
         * Success.
         * At this point, the flash is all 0xAA and we
         * have read that all back to dest
         */
      }
    }
    alt_flash_close_dev(fd);
  }
  else
  {
    printf("Cannot open flash device\n");
  }
  return 0;
}
```

Fine-Grained Flash Access

- Three additional functions provide complete control for writing flash contents at the highest granularity:
  - alt_get_flash_info()
  - alt_erase_flash_block()
  - alt_write_flash_block()
- By the nature of flash memory, you cannot erase a single address in a block
- You must erase (that is, set to all ones) an entire block at a time
- Writing to flash memory can only change bits from 1 to 0; to change any bit from 0 to 1, you must erase the entire block along with it

Fine-Grained Flash Access (continued)

- To alter a specific location in a block while leaving the surrounding contents unchanged, you must:
  - read out the entire contents of the block to a buffer,
  - alter the value(s) in the buffer,
  - erase the flash block, and finally
  - write the whole block-sized buffer back to flash memory
- The fine-grained flash access functions automate this process at the flash block level

alt_get_flash_info()

- alt_get_flash_info() gets the number of erase regions, the number of erase blocks in each region, and the size of each erase block
- The function prototype is as follows:

```
int alt_get_flash_info (
  alt_flash_fd* fd,
  flash_region** info,
  int* number_of_regions )
```

- If the call is successful, on return the address pointed to by **number_of_regions** contains the number of erase regions in the flash memory, and ***info** points to an array of **flash_region** structures
- This array is part of the file descriptor

The flash_region Structure

- The flash_region structure is defined in **sys/alt_flash_types.h**
- The data structure is defined as follows:

```
typedef struct flash_region
{
  int offset;              /* Offset of this region from start of the flash */
  int region_size;         /* Size of this erase region */
  int number_of_blocks;    /* Number of blocks in this region */
  int block_size;          /* Size of each block in this erase region */
} flash_region;
```

- With the information obtained by calling alt_get_flash_info(), you are in a position to erase or program individual blocks of the flash device

alt_erase_flash()

- alt_erase_flash() erases a single block in the flash memory
- The function prototype is as follows:

```
int alt_erase_flash_block ( alt_flash_fd* fd,
                            int offset,
                            int length )
```

- The flash memory is identified by the handle **fd**
- The block is identified as being **offset** bytes from the beginning of the flash memory, and the block size is passed in **length**

alt_write_flash_block()

- alt_write_flash_block() writes to a single block in the flash memory
- The prototype is:

```
int alt_write_flash_block( alt_flash_fd* fd,
                           int block_offset,
                           int data_offset,
                           const void *data,
                           int length)
```

- This function writes to the flash memory identified by the handle **fd**
- It writes to the block located **block_offset** bytes from the start of the flash device
- The function writes **length** bytes of data from the location pointed to by **data** to the location **data_offset** bytes from the start of the flash device

Using the Fine-Grained Flash Access API Functions

```
#include <string.h>
#include "sys/alt_flash.h"
#include "alt_types.h"
#include "system.h"
#define BUF_SIZE 100

int main (void)
{
    flash_region* regions;
    alt_flash_fd* fd;
    int number_of_regions;
    int ret_code;
    char write_data[BUF_SIZE];

    /* Set write_data to all 0xa */
    memset(write_data, 0xA, BUF_SIZE);

    fd = alt_flash_open_dev(EXT_FLASH_NAME);
```


Using the Fine-Grained Flash Access API Functions

```
    if (fd)
    {
        ret_code = alt_get_flash_info(fd, &regions, &number_of_regions);
        if (number_of_regions && (regions->offset == 0))
        {
            /* Erase the first block */
            ret_code = alt_erase_flash_block(fd,
            regions->offset,
            regions->block_size);
            if (ret_code == 0) {
                /*
                * Write BUF_SIZE bytes from write_data 100 bytes to
                * the first block of the flash
                */
                ret_code = alt_write_flash_block (
                fd,
                regions->offset,
                regions->offset+0x100,
                write_data,
                BUF_SIZE );
            }
        }
    }
    return 0;
}
```