SISTEMI EMBEDDED

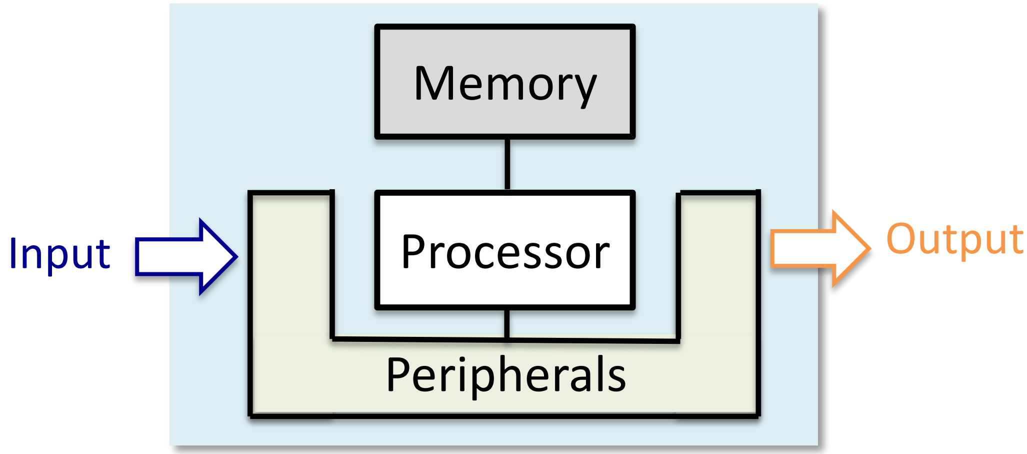
Input/Output
Hardware Abstraction Layer (HAL)

newlib C standard library

Federico Baronti

Last version: 2018021

Input/Output (I/O)



- **Peripheral Interfaces** allow the processor to interact with a variety of I/O devices
  - They “standardize” (in hardware) the different I/O devices, so that they appear to the **processor** as a well-defined set of **registers** (*control*, *status* and *data* regs). The way the programmer must handle these registers depends on how the underlying I/O device works
- How can this standardization (or hardware abstraction) process be further developed in the embedded system **software?**

Standard I/O

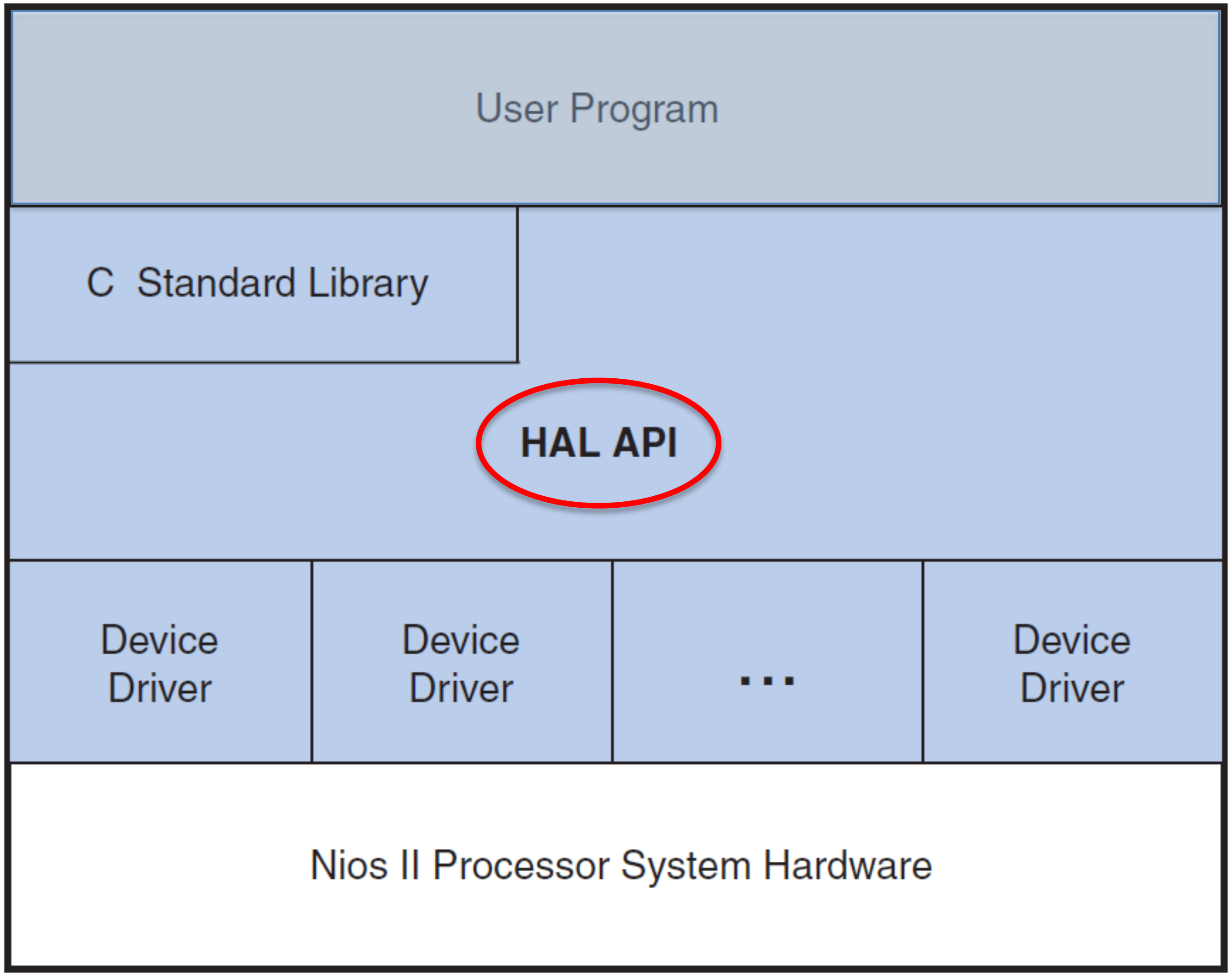
- Consider the following C^{++} program:

```
#include <iostream>
using namespace std;

int main()
{
        int x, y, max;
        cout << "Type two integers: ";
        cin >> x >> y;
        max = (x>y ? x : y);
        cout <<  "The maximum is: " << max << '\n';
}
```

- The interaction with the standard I/O devices is managed by the iostream class and the relevant operating system functions
- Programmer do not need to know how the standard I/O devices work!
- **Can we do something similar when programming the Nios II processor or any other embedded processor?**

Layered software



Hardware Abstraction Layer (1)

- Isolates User Program from hardware implementation
- Uses the services provided by the Device Driver Layer to create a standard interface (API: Application Programming Interface) to the User Program
- Integrates with the *newlib* Standard C Library
  - Peripherals can share the same API (eg. printf(), fopen(), fwrite(), …)
- Automatically generated as part of the **Board Support Package (BSP)** from the specific hardware configuration contained in the SOPC information file (.sopcinfo)

Hardware Abstraction Layer (2)

- Pros:
  - Speed-up software development
  - Code reusability
  - Tolerance to hardware changes during software developing
  - Facilitate parallel development of the software
- Cons:
  - Less optimized code
    - Larger memory footprint
    - Slower performances

Hardware Abstraction Layer (3)

- **HAL additional services:**
  - System Initialization
    - Performs initialization tasks for the processor and the runtime environment before *main*()
      - E.g. Stack Pointer initialization
  - Device Initialization
    - Instantiates and initializes each device in the system before *main*()

Generic Device Models

- **Character-mode devices:** Hardware peripherals that send and/or receive characters serially, such as a UART
- **Timer devices:** Hardware peripherals that count clock ticks and can generate periodic interrupt requests
- **File subsystems:** A mechanism for accessing files stored in physical device(s)
- **Ethernet devices:** Devices that provide access to an Ethernet connection for a networking stack such as the Altera-provided NicheStack® TCP/IP Stack - Nios II Edition
- **Direct memory access (DMA) devices:** Peripherals that perform bulk data transactions from a data source to a destination
- **Flash memory devices:** Nonvolatile memory devices that use a special programming protocol to store data

Benefits of a Device Model

- **HAL defines a set of functions to initialize and access each class of device**
- The programmer can use a **standard API** independent of the device driver implementation
  - E.g. To access character-mode devices and file subsystems, the programmer can use C standard library functions, such as printf(), fopen(),…
- Device driver must provide a set of driver functions according to the device class that are used by the standard API to manipulate the peripheral of the specific class

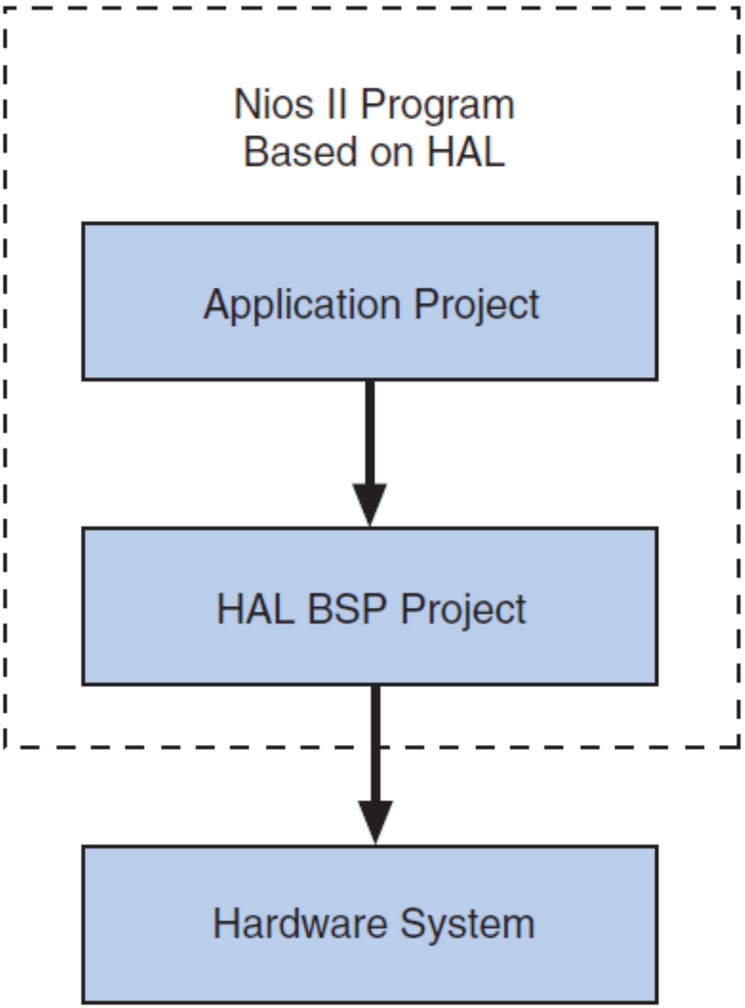
Peripherals supported by HAL (1)

- **Character mode devices:**
  - UART core
  - JTAG UART core
  - LCD 16207 display controller
- **Timer devices:**
  - Timer core
- **File subsystems:**
  - Altera host based file system
  - Altera read-only zip file system
- **Ethernet devices:**
  - Triple Speed Ethernet MegaCore® function
  - LAN91C111 Ethernet MAC/PHY Controller
- **DMA devices:**
  - DMA controller core
  - Scatter-gather DMA controller core
- **Flash memory devices:**
  - Common flash interface compliant flash chips
  - Altera's erasable programmable configurable serial (EPCS) serial configuration device controller

Peripherals supported by HAL (2)

- All peripherals (both from Altera and third party vendors) must provide a header file that defines the peripheral’s low-level interface to hardware
- Some peripherals might not provide device drivers. If drivers are not available, use only the definitions provided in the header files to access the hardware. **Do not use unnamed constants, such as hard-coded addresses, to access a peripheral**
- Some peripherals provide dedicated functions that are not based on the HAL generic device models. For example, Altera provides a general-purpose parallel I/O (PIO) core for use with the Nios II processor system. The PIO peripheral does not fit in any class of generic device models provided by the HAL, and so it provides a header file and a few dedicated functions only

Structure of a project with HAL (1)



Also known as: Your program, or user project
Defined by: **.c**, **.h**, **.S**, **.s** files
Created by: You

Defined by: Nios II BSP settings
Created by: Nios II IDE or Nios II command line tools

Also known as: Nios II processor system, or the hardware
Defined by: **.sopcinfo** file
Created by: System integration tool (Qsys or SOPC Builder)

Structure of a project with HAL (2)

- Two projects:
  - User application project
  - BSP project
- The executable image (.elf) is the result of *building* both projects
- The BSP project incorporates the HAL and the device drivers relevant to the specific hardware system defined by the .sopcinfo file
- The BSP can be updated when the underlying hardware system changes

System description file (1)

- **system.h** contains all information related to the hardware system
  - The hardware configuration of the peripheral
  - The base address
  - Interrupt request (IRQ) information (if any)
  - A symbolic name for the peripheral
- Generated automatically from .sopcinfo file and HAL BSP properties

System description file (2)

- Extracted from **system.h** related to the DE2 Basic Computer

```
/*
 * Pushbuttons configuration
 *
 */
#define ALT_MODULE_CLASS_Pushbuttons altera_up_avalon_parallel_port
#define PUSHBUTTONS_BASE 0x10000050
#define PUSHBUTTONS_IRQ 1
#define PUSHBUTTONS_IRQ_INTERRUPT_CONTROLLER_ID 0
#define PUSHBUTTONS_NAME "/dev/Pushbuttons"
#define PUSHBUTTONS_SPAN 16
#define PUSHBUTTONS_TYPE "altera_up_avalon_parallel_port"
```

System description file (3)

```
/*
 * Interval_timer configuration
 *
 */
#define ALT_MODULE_CLASS_Interval_timer altera_avalon_timer
#define INTERVAL_TIMER_ALWAYS_RUN 0
#define INTERVAL_TIMER_BASE 0x10002000
#define INTERVAL_TIMER_COUNTER_SIZE 32
#define INTERVAL_TIMER_FIXED_PERIOD 0
#define INTERVAL_TIMER_FREQ 50000000u
#define INTERVAL_TIMER_IRQ 0
#define INTERVAL_TIMER_IRQ_INTERRUPT_CONTROLLER_ID 0
#define INTERVAL_TIMER_LOAD_VALUE 6249999ull
#define INTERVAL_TIMER_MULT 0.0010
#define INTERVAL_TIMER_NAME "/dev/Interval_timer"
#define INTERVAL_TIMER_PERIOD 125.0
#define INTERVAL_TIMER_PERIOD_UNITS "ms"
#define INTERVAL_TIMER_RESET_OUTPUT 0
#define INTERVAL_TIMER_SNAPSHOT 1
#define INTERVAL_TIMER_SPAN 32
#define INTERVAL_TIMER_TICKS_PER_SEC 8u
#define INTERVAL_TIMER_TIMEOUT_PULSE_OUTPUT 0
#define INTERVAL_TIMER_TYPE "altera_avalon_timer"
```

HAL API (1)

- Unix-style functions (file access)
  - Facilitate portability of existing programs to Nios II
- HAL API can be further encapsulated by the C standard library
  - E.g. HAL API functions are used by the C standard library defined in **stdio.h** to perform underlying device access
  - Programmer can use both the C standard library or the HAL API functions

HAL API (2)

- Most commonly-used HAL API functions:

int **open**(const char* pathname, int flags, mode_t mode)	Opens a file or device and returns a file descriptor
int **close**(int fd)	Closes the file descriptor fd
int **read**(int fd, void *ptr, size_t len)	Reads a block of data from a file or device
int **write**(int fd, const void *ptr, size_t len)	Writes a block of data to a file or device
off_t **lseek**(int fd, off_t ptr, int whence)	Moves the read/write pointer associated with the file descriptor fd
int **fstat**(int fd, struct stat *st)	Obtains information about the capabilities of an open file descriptor
int **ioctl**(int fd, int req, void* arg)	Allows application code to manipulate the I/O capabilities of a device driver in driver-specific ways

Example (1)

- Using character-mode devices with standard I/O C libraries
  - A character-mode device (e.g. JTAG-UART) can be attached to stdin, stdout, stderr streams (BSP property)
    - printf() is available to access stdout!

```
#include <stdio.h>
int main (){
    printf("Hello world!");
    return 0; /* while(1) ; */
}
```

Example (2)

- Writing characters to the UART device “/dev/uart1”

```
#include <stdio.h>
#include <string.h>
int main(void) {
   char* msg = "hello world";
   FILE* fp;
   fp = fopen ("/dev/uart1", "w");
   if (fp!=NULL) {
      fprintf(fp, "%s",msg);
      fclose (fp);
      }
   return 0;
}
```

Null device

- **/dev/null**
- Included by all HAL-based systems
- It is not connected to any hardware (virtual device)
- Writing to **/dev/null** has no effect and all data are discarded
- Used for safe I/O redirection during system startup and to sink unwanted data

Device implementation

```
// alt_dev.h
typedef struct alt_dev_s alt_dev;

struct alt_dev_s {
  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);
};
```

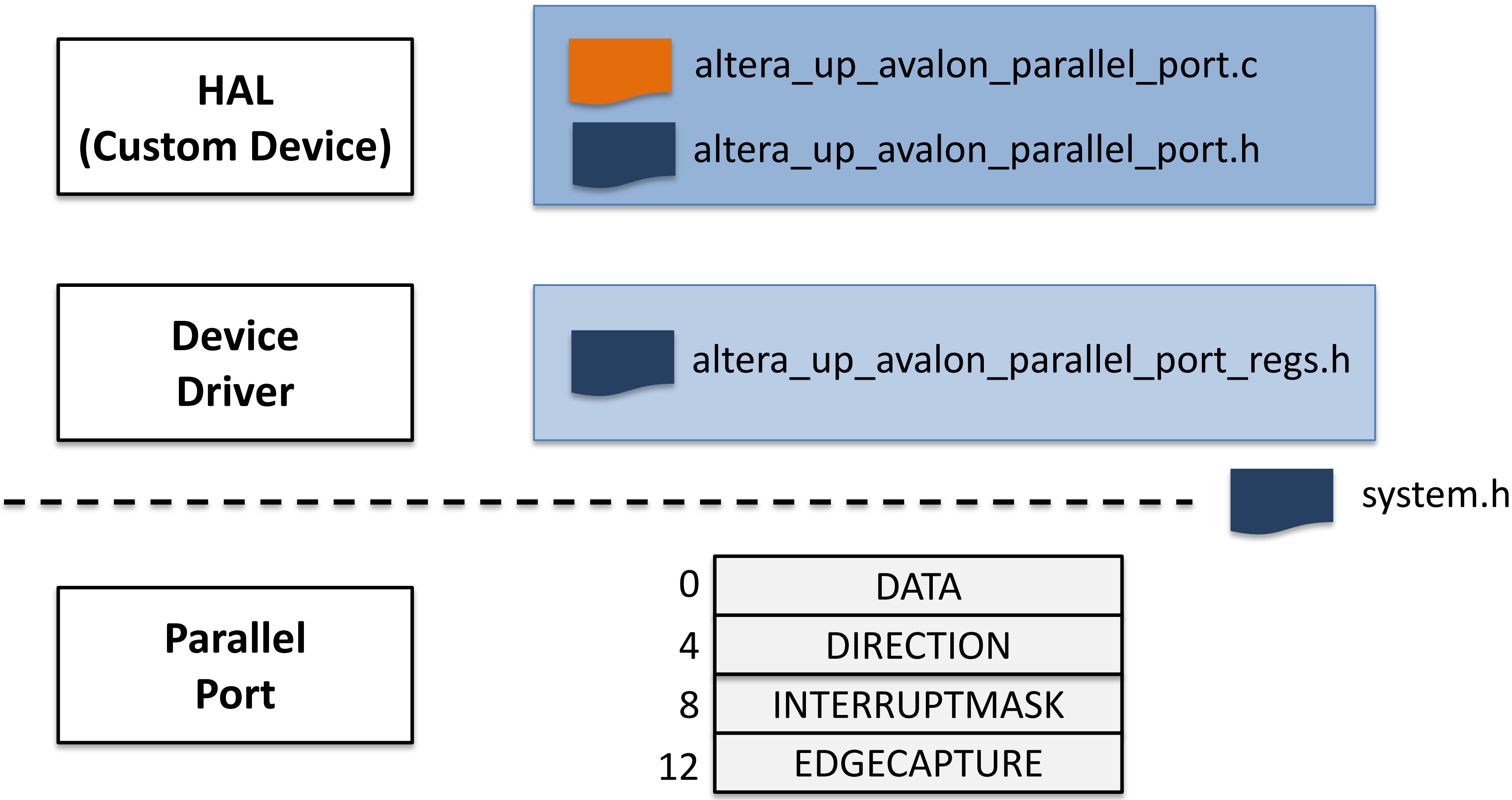
```
// alt_list.h
typedef struct alt_llist_s alt_llist;

struct alt_llist_s {
  alt_llist* next;     /* Pointer to the next element in the list. */
  alt_llist* previous; /* Pointer to the previous element in the
                          list. */
};
```

HAL runtime environment

- alt_sys_init.c
  - Allocates the device structures for the peripherals present in the hardware system
  - Initializes all the devices
- The device structures are managed by a **list**

Parallel Port HAL structure



Parallel Port Device Driver

- **altera_up_avalon_parallel_port_regs.h**
  The programmer should not use this defines!

```
#ifndef __ALTERA_UP_AVALON_PARALLEL_PORT_REGS_H__
#define __ALTERA_UP_AVALON_PARALLEL_PORT_REGS_H__

#include <io.h>

// Data Register
#define ALT_UP_PARALLEL_PORT_DATA                               0
#define IOADDR_ALT_UP_PARALLEL_PORT_DATA(base)                   \
        __IO_CALC_ADDRESS_NATIVE(base, ALT_UP_PARALLEL_PORT_DATA)
#define IORD_ALT_UP_PARALLEL_PORT_DATA(base)                     \
        IORD(base, ALT_UP_PARALLEL_PORT_DATA)
#define IOWR_ALT_UP_PARALLEL_PORT_DATA(base, data)               \
        IOWR(base, ALT_UP_PARALLEL_PORT_DATA, data)

/* ... */

#endif /* __ALTERA_UP_AVALON_PARALLEL_PORT_REGS_H__ */
```

Parallel Port HAL (1)

- **altera_up_avalon_parallel_port.h**
  - Declares/Defines functions/MACROS to manage the device by the user application: open device, read and write data, …
  - Defines ancillary structure and MACROS to be used by the HAL runtime environment for device initialization
- **altera_up_avalon_parallel_port.c**
- Defines:

```
alt_up_parallel_port_dev* alt_up_parallel_port_open_dev(const char* name);

typedef struct alt_up_parallel_port_dev {
  alt_dev dev;
  unsigned int base;
  } alt_up_parallel_port_dev;
```

Parallel Port HAL (2)

```
// test parall port HAL
#include "system.h”
#include "altera_up_avalon_parallel_port.h”

int main() {

    alt_up_parallel_port_dev *slider_dev, *red_leds_dev;

    slider_dev = alt_up_parallel_port_open_dev(SLIDER_SWITCHES_NAME);
    red_leds_dev = alt_up_parallel_port_open_dev(RED_LEDS_NAME);

    while(1) {
        alt_up_parallel_port_write_data(red_leds_dev,
                                        alt_up_parallel_port_read_data(slider_dev));
    }
}
```

newlib C standard Library (1)

- Doc @ https://sourceware.org/newlib/
- Standard functions are divided into groups. Each group has a corresponding header file
- **Standard Utility Functions** (stdlib.h)
  - Includes functions for: dynamic memory allocations, string to number conversion and vice versa, pseudo-random number generation, ….
- **Character Type Macros and Functions** (ctype.h)
  - Includes functions for: classifying characters into several categories (alphabetic, numeric, control characters, whitespace, and so on), or to perform simple character conversions

newlib C standard Library (2)

- **Strings and Memory** (string.h)
  - Includes functions for: string-handling and managing areas of memory
- **Input and Output** (stdio.h)
  - Includes functions for: managing files or other input/output streams. Among these functions are subroutines to generate or scan strings according to some rules specified by means of a format string

Standard I/O with Nios II (1)

- Unformatted input:
  `int getchar();`
  – Get the next single character from the stdin stream
- Unformatted output:
  `int putchar(int ch);`
  – Insert a single character into the stdout stream

Standard I/O with Nios II (2)

- Formatted output:
  int printf(*format string*, *expression list*);
  - Converts the result of the expressions according to the specified format and sends the resulting string to the stdout stream
  - Examples of format specifiers:
    - %d signed integer
    - %u unsigned integer
    - %x hexadecimal unsigned integer
    - %f floating point
    - %c single character
    - %s string

Standard I/O with Nios II (3)

- Formatted input:
  int scanf(*format string, variable list*);
  - Interpreters the stdin characters according to the provided format and assigns the decoded values to the variables (they must be passed as pointers!)
  - Examples of format specifiers:
    - %d signed integer
    - %u unsigned integer
    - %x hexadecimal unsigned integer
    - %f floating point
    - %c single character
    - %s string

Standard I/O with Nios II (4)

- If we recall the initial question "**Can we do something similar also when programming the Nios II processor?**"
- Yes, using standard I/O functions over a character-mode device: printf(), scanf(). (stdin and stout streams must be attached to that device through the BSP editor)

```
#include <stdio.h>

int main() {
	int x, y, max;

	while(1) {
		printf("Type two integers: ");
		scanf("%d %d", &x, &y);
		max = (x>y ? x : y);
		printf( "The maximum is: %d\n", max);
	}
}
```

Putting into practice

- Write a program that uses the JTAG-UART peripheral as standard I/O device in order to make the status of the red LEDs controllable from the remote PC.

References

- Altera "Nios II Software Developer's Handbook," ***n2sw_nii5v2.pdf***
  - Section II. **Hardware Abstraction Layer** (Chapters 5, 6, 7)
  - Section IV. Reference Material (Chapter 14)