

NIOS ii API 常用函数解析

IOADDR_ALTERA_AVALON_PIO_DATA(base)

IORD_ALTERA_AVALON_PIO_DATA(base)

读取以“base”为基地址的 PIO 数据寄存器中的数据

IOWR_ALTERA_AVALON_PIO_DATA(base, data)

写入以“base”为基地址的 PIO 数据寄存器中的“data”数据

IOADDR_ALTERA_AVALON_PIO_DIRECTION(base)

IORD_ALTERA_AVALON_PIO_DIRECTION(base)

读取以“base”为基地址的 PIO 方向寄存器中的数据

IOWR_ALTERA_AVALON_PIO_DIRECTION(base, data)

写入以“base”为基地址的 PIO 方向寄存器中的“data”数据

IOADDR_ALTERA_AVALON_PIO_IRQ_MASK(base)

IORD_ALTERA_AVALON_PIO_IRQ_MASK(base)

读取以“base”为基地址的 PIO 中断任务寄存器中的数据

IOWR_ALTERA_AVALON_PIO_IRQ_MASK(base, data)

写入以“base”为基地址的 PIO 中断任务寄存器中的“data”数据

IOADDR_ALTERA_AVALON_PIO_EDGE_CAP(base)

IORD_ALTERA_AVALON_PIO_EDGE_CAP(base)

读取以“base”为基地址的 PIO 沿寄存器中的数据

IOWR_ALTERA_AVALON_PIO_EDGE_CAP(base, data)

写入以“base”为基地址的 PIO 沿寄存器中的“data”数据

IOADDR_ALTERA_AVALON_PIO_SET_BIT(base)

IORD_ALTERA_AVALON_PIO_SET_BITS(base)

读取以“base”为基地址的 PIO 标志位寄存器中的数据

IOWR_ALTERA_AVALON_PIO_SET_BITS(base, data)

写入以“base”为基地址的 PIO 标志位寄存器中的“data”数据

IOADDR_ALTERA_AVALON_PIO_CLEAR_BITS(base)

IORD_ALTERA_AVALON_PIO_CLEAR_BITS(base)

读取以“base”为基地址的 PIO 清除标志寄存器中的数据

IOWR_ALTERA_AVALON_PIO_CLEAR_BITS(base, data)

写入以“base”为基地址的 PIO 清除标志寄存器中的“data”数据

IO 操作函数

函数原型: IORD(BASE, REGNUM)

输入参数: BASE 为寄存器的基地址, REGNUM 为寄存器的偏移量

函数说明: 从基地址为 BASE 的设备中读取寄存器中偏移量为 REGNUM 的单元里面的值。寄存器的值在地址总线的范围之内。

返回值: —

函数原型: IOWR(BASE, REGNUM, DATA)

输入参数: BASE 为寄存器的基地址, REGNUM 为寄存器的偏移量, DATA 为要写入的数据

函数说明: 往偏移量为 REGNUM 寄存器中写入数据。寄存器的值在地址总线的范围之内。

返回值: —

函数原型: IORD_32DIRECT(BASE, OFFSET)

输入参数: BASE 为寄存器的基地址, OFFSET 为寄存器的偏移量

函数说明: 从地址位置为 BASE+OFFSET 的寄存器中直接读取 32Bit 的数据

返回值: —

函数原型: IORD_16DIRECT(BASE, OFFSET)

输入参数: BASE 为寄存器的基地址, OFFSET 为寄存器的偏移量

函数说明: 从地址位置为 BASE+OFFSET 的寄存器中直接读取 16Bit 的数据

返回值: —

函数原型: IORD_8DIRECT(BASE, OFFSET)

输入参数: BASE 为寄存器的基地址, OFFSET 为寄存器的偏移量

函数说明: 从地址位置为 BASE+OFFSET 的寄存器中直接读取 8Bit 的数据

返回值: —

函数原型: IOWR_32DIRECT(BASE, OFFSET, DATA)

输入参数: BASE 为寄存器的基地址, REGNUM 为寄存器的偏移量, DATA 为要写入的数据

函数说明: 往地址位置为 BASE+OFFSET 的寄存器中直接写入 32Bit 的数据

返回值: —

函数原型: IOWR_16DIRECT(BASE, OFFSET, DATA)

输入参数: BASE 为寄存器的基地址, REGNUM 为寄存器的偏移量, DATA 为要写入的数据

函数说明: 往地址位置为 BASE+OFFSET 的寄存器中直接写入 16Bit 的数据

返回值: —

函数原型: IOWR_8DIRECT(BASE, OFFSET, DATA)

输入参数: BASE 为寄存器的基地址, REGNUM 为寄存器的偏移量, DATA 为要写入的数据

函数说明：往地址位置为 BASE+OFFSET 的寄存器中直接写入 8Bit 的数据
返回值： —

Dma:

函数原型：int alt_dma_rxchan_close (alt_dma_rxchan rxchan)

输入参数：rxchan 为接收信道

函数说明：函数 alt_dma_rxchan_close () 通知系统：应用程序已经完成 DMA 接收信道 rxchan，目前执行是成功的

返回值： 成功返回为 0，反之为 -1

函数原型：alt_dma_rxchan_depth(alt_dma_rxchan dma)

输入参数：dma

函数说明：函数 alt_dma_rxchan_depth () 返回传送到特别 DMA 的最大数量(深度)的接收请求

返回值： DMA 的最大数量

函数原型：int alt_dma_rxchan_ioctl (alt_dma_rxchan dma, int req, void * arg)

输入参数：dma 直接存储器名，req 为请求操作的列举，arg 由请求决定

函数说明：通过 DMA 接收信道执行设备的具体 I/O 操作

返回值： 成功返回请求具体值，反之返回为负数

请求类型

请求类型 请求类型说明

ALT_DMA_SET_MODE_8 传输以 8Bit 为单位的数据，arg 值忽略

ALT_DMA_SET_MODE_16 传输以 16Bit 为单位的数据，arg 值忽略

ALT_DMA_SET_MODE_32 传输以 32Bit 为单位的数据，arg 值忽略

ALT_DMA_SET_MODE_64 传输以 64Bit 为单位的数据，arg 值忽略

ALT_DMA_SET_MODE_128 传输以 128Bit 为单位的数据，arg 值忽略

ALT_DMA_TX_ONLY_ON (1) 软件控制下只能发送

ALT_DMA_TX_ONLY_OFF (1) 自定义模式，软件控制下可以接收，发送

ALT_DMA_RX_ONLY_ON (1) 软件控制下只能接收

ALT_DMA_RX_ONLY_OFF (1) 自定义模式，软件控制下可以接收，发送

函数原型：alt_dma_rxchan alt_dma_rxchan_open (const char* name)

输入参数：name 为常数字符指针，如/dev/dma_0

函数说明：为 DMA 接收信道获得一个 alt_dma_rxchan 描述符

返回值： 成功返回非 0，反之返回为 0

函数原型：int alt_dma_rxchan_prepare (alt_dma_rxchan dma, void* data, alt_u32 length, alt_rxchan_done * done, void* handle)

输入参数：dma 使用的信道；data 接收数据位置的指针；length 最大的接收数据长度；done 一旦数据被接收，调用返回函数；handle，非透明值传到 done

函数说明：发送一个接收请求到 DMA 接收信道，

返回值： 成功返回 0，反之返回为负数

函数原型: `int alt_dma_rxchan_reg (alt_dma_rxchan_dev * dev)`

输入参数: `dev` 接收信道设备名

函数说明: 给系统寄存 DMA 接收信道

返回值: 成功返回 0, 反之返回为负数

函数原型: `int alt_dma_txchan_close (alt_dma_txchan txchan)`

输入参数: `txchan` 发送信道名

函数说明: 通知系统: 应用程序已经完成 DMA 发送信道 `txchan`

返回值: 成功返回 0, 反之返回为负数

函数原型: `int alt_dma_txchan_ioctl (alt_dma_txchan dma, int req, void * arg)`

输入参数: `dma` 直接存储器名; `req` 为请求操作的列举; `arg` 请求的额外参数, 由请求决定

函数说明: 通过 DMA 发送信道执行设备的具体 I/O 操作

返回值: 成功返回请求具体值, 反之返回为负数

函数原型: `alt_dma_txchan alt_dma_txchan_open (const char* name)`

输入参数: `name` 为常数字符指针, 如 `/dev/dma_0`

函数说明: 为 DMA 发送信道获得一个 `alt_dma_rxchan` 描述符

返回值: 成功返回非 0, 反之返回为 0

函数原型: `int alt_dma_txchan_reg (alt_dma_txchan_dev* dev)`

输入参数: `dev` 接收信道设备名

函数说明: 给系统寄存 DMA 发送信道

返回值: 成功返回 0, 反之返回为负数

函数原型: `int alt_dma_txchan_send (alt_dma_txchan dma, const void* from,`

`alt_u32 length, alt_txchan_done* done, void* handle)`

输入参数: `dma` 使用的信道; `data` 接收数据位置的指针; `length` 最大的接收数据长度; `done` 一旦数据被接收, 调用返回函数; `handle`, 非透明值传到 `done`

函数说明: 发送一个发送请求到 DMA 发送信道,

返回值: 发送成功返回 0, 反之返回为负数

函数原型: `nt alt_dma_txchan_space (alt_dma_txchan dma)`

输入参数: `dma` 直接存储器名

函数说明: 返回被传送到具体 DMA 发送信道的发送请求数目

返回值: 返回发送请求数目

Flash

函数原型: `int alt_erase_flash_block(alt_flash_fd* fd, int offset, int length)`

输入参数: `fd` 为具体的 flash 设备; `offset` 擦除的 flash 模块的偏移量; `length` 擦除的 flash 模块的长度

函数说明：擦除单独的一个 flash 模块
返回值： 发送成功返回 0，反之返回为负数

函数原型：void alt_flash_close_dev(alt_flash_fd * fd)

输入参数：fd 为具体的 flash 设备

函数说明：关闭 flash 设备

返回值： 一

函数原型：alt_flash_fd * alt_flash_open_dev(const char* name)

输入参数：

函数说明：打开 flash 设备。一旦打开，函数 alt_write_flash() 用来写入，函数 alt_read_flash() 用来读取数据，或者使用函数 alt_get_flash_info(), alt_erase_flash_block(), alt_write_flash_block(), 控制单个模块

返回值： 失败返回 0，成功其他值

函数原型：int alt_get_flash_info(alt_flash_fd* fd, flash_region ** info,

int* number_of_regions)

输入参数：fd flash 设备；info 指向 flash_region 结构体的指针；number_of_regions

函数说明：得到擦除 flash 区域的细节

返回值： 发送成功返回 0，反之返回为负数

函数原型：int alt_read_flash(alt_flash_fd* fd, int offset, void* dest_addr, int length)

输入参数：dest_addr 目标地址指针

函数说明：从 flash 偏移量为 offset 字节开始读取数据，写入到目标地址 dest_addr 中

返回值： 成功返回 0，反之为非 0

函数原型：int alt_write_flash(alt_flash_fd* fd, int offset, const void* src_addr, int length)

输入参数：src_addr 源地址；fd, flash 设备；offset 偏移量；length 字节长度

函数说明：写数据到 flash 中，要写的数据在源地址 src_addr 中

返回值： 成功返回 0，反之为非 0

函数原型：int alt_write_flash_block(alt_flash_fd* fd, int block_offset, int data_offset,

const void *data, int length)

输入参数：fd; data_offset 起始写数据的偏移量；length 为要写数据的长度

函数说明：写入到一个已擦除的 flash 模块

返回值： 成功返回 0，反之为非 0

Irq

函数原型: `alt_irq_context alt_irq_disable_all (void)`

输入参数: `void`

函数说明: 禁止所有中断

返回值: 传递的值作为随后的函数调用的输入参数

函数原型: `void alt_irq_enable_all (alt_irq_context context)`

输入参数: 先前调用函数 `alt_irq_disable_all (void)` 的返回值,

函数说明: 启动所有中断

返回值: 一

函数原型: `int alt_irq_enabled (void)`

输入参数: `void`

函数说明: 启动中断

返回值: 禁止中断返回 0, 反之为非 0

函数原型: `int alt_irq_register (alt_u32 id, void* context, void (*isr) (void*, alt_u32))`

输入参数: `id`, 32 位无符号数, 中断使能; `context` 和 `id` 是 `isr` 的两个输入参数; 中断激活时调用 `isr`

函数说明: 寄存一个 `isr`

返回值: 成功返回 0, 反之为非 0

函数原型: `int alt_write_flash(alt_flash_fd* fd, int offset, const void* src_addr, int length)`

输入参数: `src_addr` 源地址; `fd`, flash 设备; `offset` 偏移量; `length` 字节长度

函数说明: 写数据到 flash 中, 要写的在源地址 `src_addr` 中

返回值: 成功返回 0, 反之为非 0

函数原型: `int alt_write_flash_block(alt_flash_fd* fd, int block_offset, int data_offset, const void *data, int length)`

输入参数: `fd`; `data_offset` 起始写数据的偏移量; `length` 为要写数据的长度

函数说明: 写入到一个已擦除的 flash 模块

返回值: 成功返回 0, 反之为非 0

函数原型: `int close (int filedes)`

输入参数: `filedes`, 描述符

函数说明: 标准的 UNIX 函数 `close()`, 关闭文件描述符 `filedes`

返回值: 成功返回 0, 反之为 -1

函数原型: `int open (const char* pathname, int flags, mode_t mode)`
输入参数: `pathname`, 路径名; `flags`, `O_RDONLY` 或 `O_WRONLY` 或 `O_RDWR`, 分别对应着只读, 只写, 或读写操作; `mode`, 使用许可说明
函数说明: 打开文件或设备, 返回一个文件描述符 (读写中使用的非负整数)
返回值: 成功返回文件描述符, 反之返回 -1

函数原型: `int read(int file, void *ptr, size_t len)`
输入参数: `file` 文件描述符; `ptr` 为读数据的位置指针, `len` 读数据的长度, 单位为字节
函数说明: 从文件或设备中读取数据块
返回值: 成功返回读取的字节数, 反之返回 -1

函数原型: `clock_t times (struct tms *buf)`
输入参数: `buf` 结构体指针
函数说明: 兼容 `newlib`, `tms` 的结构体指针如下:

```
type struct
{clock_t tms_utime;
clock_t tms_stime;
clock_t tms_cutime;
clock_t tms_sutime;
};
```


`tms_utime`: CPU 索取用户指令的执行时间
`tms_stime`: CPU 索取由系统表示的过程的执行时间
`tms_cutime`: 所有子进程 `tms_utime` 和 `tms_cutime` 的时间之和
`tms_sutime`: 所有子进程 `tms_stime` 和 `tms_sutime` 的时间之和
返回值: 返回时钟数, 没有时钟则返回 0

函数原型: `int usleep (int us)`
输入参数: `us`, 单位为微秒
函数说明: 直到 `us` 微秒后才解除阻塞, 即其功能相当于延时 `us` 微秒
返回值: 成功返回 0, 反之为 -1, 有错误发生显示错误发生原因

函数原型: `int wait(int *status)`
输入参数: `status` 进程状态指针
函数说明: 功能是等候所有子进程退出, 由于 HAL 不支持分散子进程, 函数立即返回
返回值: `status` 内容清 0, 表明没有子进程; 返回值为 -1, 且 `errno` 置为 `ECHILD`, 表明没有子进程等候

函数原型: `int write(int file, const void *ptr, size_t len)`
输入参数: `file` 文件描述符; `ptr` 为读数据的位置指针, `len` 读数据的长度, 单位为字节
函数说明: 往文件或设备写入数据块,
返回值: 成功返回写入的字节数, 也可能少于请求的长度; 反之返回 -1, 万一有错误发生, `errno` 被设置为发生的原因

数据的标准类型

类型 说明

alt_8 符号 8 位整数

alt_u8 无符号 8 位整数

alt_16 符号 16 位整数

alt_u16 无符号 16 位整数

alt_32 符号 32 位整数

alt_u32 无符号 32 位整数

下面为自己整理

函数原型: `int fopen (char * file_name, way_use);`

输入参数: `file_name` 文件名, `way_use` 使用文件方式, 比如 `r`, `w` 分别对应着读写

函数说明: 打开文件, 对其进行某种文件操作

返回值: 打不开则出错, 返回一个空指针 `NULL`

函数原型: `int fclose (fp)`

输入参数: `fp` 的定义为: `FILE *fp`

函数说明: 关闭文件 `fp`

返回值: 成功返回 0, 反之为 -1 (EOF)

函数原型: `int fread(void *ptr, int size, int count, FILE * fp);`

输入参数: `buffer` 为指针; 是读入数据地存放地址; `size` 读字节数; `count` 读字节数地数目; `fp` 文件型指针

函数说明: 从一个流中读取数据

返回值: 成功返回值为 `count`

函数原型: `int fwrite(void *ptr, int size, int count, FILE *fp)`

输入参数: `buffer` 为指针; 是读入数据地存放地址; `size` 读字节数; `count` 读字节数地数目; `fp` 文件型指针,

函数说明: 写内容到流中

返回值: 成功返回值为 `count`

函数原型: `int fprintf(FILE *fp, char *format[, argument,...]);`

输入参数: `fp` 文件型指针; `format` 格式字符串; `[, argument,...]` 输出列表, 如:

`fprintf(fp, "%d,%f", i, t)`

函数说明: 传送格式化输出到一个流中

返回值: 一

函数原型: `int fscanf(FILE * fp, char *format[, argument,...])`

输入参数: `fp` 文件型指针; `format` 格式字符串; `[, argument,...]` 输入列表, 如:

`fscanf(fp, "%d,%f", i, t)`

函数说明：从一个流中执行格式化输入

返回值：—

函数原型：int fputc(int ch, FILE *fp)

输入参数：ch 字符；fp：文件型指针

函数说明：送一个字符到一个流中

返回值：成功返回字符，反之返回-1（EOF）

函数原型：int fgetc(FILE *fp);

输入参数：fp：文件型指针

函数说明：从流中读取字符

返回值：遇到文件结束返回-1（EOF）

函数原型：int putw(int w, FILE *fp)

输入参数：w：字符或字；fp：文件型指针

函数说明：把一字符或字送到流中

返回值：—

函数原型：int getw(FILE *fp)

输入参数：fp：文件型指针

函数说明：从流中取一整数

返回值：—

函数原型：int rewind(FILE *fp)

输入参数：fp：文件型指针

函数说明：将文件指针重新指向一个流的开头

返回值：—

函数原型：int fseek(FILE *fp, long offset, int fromwhere);

输入参数：fp：文件型指针；offset：long 型偏移量；fromwhere：起始点
起始点为 0，1，2 分别代表文件开始，当前位置，文件末尾

函数说明：重定位流上的文件指针

返回值：—

函数原型：int ferror(FILE *fp)

输入参数：fp：文件型指针

函数说明：检测流上的错误

返回值：未出错返回值为 0，反之为非 0

函数原型：long ftell(FILE *fp)

输入参数：fp：文件型指针

函数说明：返回当前文件指针，得到当前位置

返回值：返回值为-1 表示出错，反之为非 0

函数原型：void clearerr(FILE *fp)

输入参数: fp: 文件型指针
函数说明: 复位错误标志
返回值: 出错为非 0, 反之为 0

函数原型: char *fgets(char *string, int n, FILE *fp)
输入参数: string: 字符串指针; fp: 文件型指针
函数说明: 从流中读取一字符串, 但只从文件输入 n-1 个字符, 后一个为 '\0'
结束标志位
返回值: -

函数原型: int fputs(char *string, FILE *fp)
输入参数: string: 字符串指针; fp: 文件型指针
函数说明: 送一个字符串到一个流中
返回值: -

函数原型: int feof(FILE *fp)
输入参数: fp: 文件型指针
函数说明: 检测流上的文件结束符
返回值: -

Nios II IDE Command Line Tools

Tool Descriptor

nios2-create-system-library 创建一个新系统库工程
nios2-create-application-project 创建一个 C/C++ 应用库工程
nios2-build-project 使用 Nios II IDE 编译工程, 创建或更新文件编写来编译工程, 该操作工程必须是存在当前的 Nios II IDE 工作区间

nios2-import-project 导入一个以前创建的 Nios II IDE 工程到当前的工作区间

nios2-delete-project 从 Nios II IDE 工作区间删除工程

Altera Command-Line Tools

Tool Descriptor

nios2-download 为调试或运行下载代码到目标处理器
nios2-flash-programmer 编程数据到目标板的 flash 存储器上
nios2-gdb-server 通过 TCP, 用目标 Nios II 处理器把 GNU 调试器远程的串口协议分组翻译为共同测试行动小组 (JTAG) 的事务
nios2-terminal 用 JTAG 通用异步收发机 (UART) 执行终止 Nios II 系统里面的 I/O
validate_zip 核实指定的 zip 文件是否兼容 Altera 只读 zip 文件系统

File Conversion Utilities

Utility Descriptor

bin2flash 为下载到 flash 存储器上, 将二进制文件转换为 .flash 文件
elf2dat 为适应 Verilog HDL 硬件仿真, 将 .elf 可执行文件格式转换为 .dat 文件格式
elf2flash 为下载到 flash 存储器上, 将 .elf 可执行文件格式转换为 .flash 文件
elf2hex 将 .elf 可执行文件格式转换为 Intel.hex 文件格式
elf2mem 在指定的 Nios II 系统中为存储设备生成存储内容
elf2mif
将 .elf 可执行文件格式转换为 Quartus II
内存初始化文件 (.mif) 格式
flash2dat
为适应 Verilog HDL 硬件仿真, 将 .flash 可执行文件格式转换为 .dat 文件格式
mk-nios2-
signal-tap-mnemonic-table 获得一个 .elf 文件和 SOPC Builder 系统文件 (.ptf), 创建一个 .stp 包含 Nios II 子令集记忆表和 Altera's SignalTap II
logic 分析仪符号的文件
sof2flash
为下载到 flash 存储器上, 将 FPGA 配置文件 (.sof) 转换为 .flash 文件

Backward Compatibility Tools

Tool Descriptor

nios2-build 基于传统 SDK 库的编译和链接软件工程
nios2-run 下载程序到 Nios II 处理器, 终止 I/O 的变成
nios2-debug
下载程序到 Nios II 处理器, 启动洞察力的调试器
nios2-console
打开 FS2 命令行接口 (CLI), 连接到 Nios II 处理器

IORD_16DIRECT(BASE, OFFSET)

从地址位置为 BASE+OFFSET 的寄存器中直接读取 16Bit 的数据

IORD_8DIRECT(BASE, OFFSET)

从地址位置为 BASE+OFFSET 的寄存器中直接读取 8Bit 的数据

IOWR_32DIRECT(BASE, OFFSET, DATA)

往地址位置为 BASE+OFFSET 的寄存器中直接写入 32Bit 的数据

IOWR_16DIRECT(BASE, OFFSET, DATA)

往地址位置为 BASE+OFFSET 的寄存器中直接写入 16Bit 的数据

IOWR_8DIRECT(BASE, OFFSET, DATA)

往地址位置为 BASE+OFFSET 的寄存器中直接写入 8Bit 的数据

IORD(BASE, REGNUM)

从基地址为 BASE 的设备中读取偏移量为 REGNUM 的寄存器里面的值。寄存器的值在地址总线的范围之内。

IOWR(BASE, REGNUM, DATA)

BASE 为基地址, 往偏移量为 REGNUM 寄存器中写入数据。寄存器的值在地址总线

的范围之内。

IORD_32DIRECT(BASE, OFFSET)

BASE 为寄存器的基地址，OFFSET 为寄存器的偏移量。

从地址位置为 BASE+OFFSET 的寄存器中直接读取 32Bit 的数据

IORD_16DIRECT(BASE, OFFSET)

从地址位置为 BASE+OFFSET 的寄存器中直接读取 16Bit 的数据

IORD_8DIRECT(BASE, OFFSET)

从地址位置为 BASE+OFFSET 的寄存器中直接读取 8Bit 的数据

IOWR_32DIRECT(BASE, OFFSET, DATA)

往地址位置为 BASE+OFFSET 的寄存器中直接写入 32Bit 的数据

IOWR_16DIRECT(BASE, OFFSET, DATA)

往地址位置为 BASE+OFFSET 的寄存器中直接写入 16Bit 的数据

IOWR_8DIRECT(BASE, OFFSET, DATA)

往地址位置为 BASE+OFFSET 的寄存器中直接写入 8Bit 的数据