

# WinSystems

PCM-UIO48/PCM-UIO96/WS16C48  
Linux Device Driver

Release 1.1 May 6, 2004

## 1 INTRODUCTION

- 1.1 This driver has been built and tested on Linux Kernels 2.2 and 2.4 using the REDHAT 6.22 distribution and the SUSE 7.2 and 8.1 Professional distributions.
- 1.2 The driver supports the WinSystems WS16C48 Digital I/O ASIC with event sense interrupt capability. This chip is present on the WinSystems PCM-UIO48, the PCM-UIO96 and several WinSystems single board computers (SBC's).
- 1.3 This driver is provided 'as-is' and no warranty as to usability or fitness of purpose is claimed.
- 1.4 WinSystems does not provide support for modification to this driver. Bug reports may be sent to [linux\\_drivers@winsystems.com](mailto:linux_drivers@winsystems.com)
- 1.5 This driver is provided under the terms of the GNU General Public License.

## 2 INSTALLATION

- 2.1 The driver code is distributed on an MS-DOS filesystem 1.44MB floppy diskette. The floppy disk should be mounted and the contents copied to the desired development directory.
- 2.2 It will be necessary to become the root user to build the driver and device nodes.
- 2.3 The default MAJOR number for this device is 110. It may be easily changed by changing the definition at the beginning of the *Makefile*. To create the driver, the device nodes, and the sample application type :  
  

```
make all
```
- 2.4 The device driver file *uio48.o*, the device nodes *uio48a* through *uio48f* are created and *chmod* is executed to allow access by all users and groups. These permissions can be changed manually as desired. These device nodes are created in the current directory. New nodes can be created manually in */dev* if desired. Two sample programs are also built.
- 2.5 The current driver must be explicitly loaded either through *init* scripts or manually. In either case *insmod* is used to install the driver. Since the WS16C48 chips used on the WinSystems PCM-UIO48A, PCM-UIO96A and other boards are not plug-n-play, I/O port probing would be problematic at best. I/O address assignments and IRQ assignments must be specified on the command line i.e.

```
insmod uio48.o io=0x320,0x330 irq=5,5
```

This would install the driver with support for 2 chips, The first at I/O address 320H and the second at I/O address 330H. The list can contain up to 6 values for the 6 chips that are supported by the driver separated by commas. Likewise the IRQs are specified on a chip by

chip basis. Interrupts can be shared as long as the hardware supports it (PCM-UIO96). Interrupts can usually not be shared across boards, although the driver will attempt it anyway.

### 3 DRIVER USAGE

- 3.1 The WS16C48 ASIC is accessed in hardware as a byte oriented device. Therefore, the driver is implemented as a character device. Using file I/O i.e. read, write, and seek operations although implemented, are very inefficient, and may not always give the desired results. The driver was designed for maximum versatility using *ioctl* as its principal program interface.
- 3.2 The file *uio48io.c* implements the *ioctl* interface and presents to the application a set of standard C functions that may be called directly from the application without any further need for dealing with, or understanding how to access the driver through *ioctl*. An application must merely include *uio48.h* and link to *uio48io.o* to provide this simple interface.
- 3.3 Applications using the driver may enable interrupts on any or all of the first 24 bits of each device. The application may further specify the polarity of the event, which will trigger the interrupt. Within the driver itself, interrupt events are buffered and handed to waiting processes. Further details on interrupt handling can be seen in the later sections which detail the functions implemented through *ioctl* or by examining the sample programs.

#### 3.4 Function Calls

##### 3.4.1 `int read_bit(int chip_number, int bit_number)`

This function takes as an argument the *chip\_number* (1-6) and the *bit\_number* (1-48) and returns 0 if the input is open or high, 1 if the input is low, and -1 if the chip is inaccessible or invalid.

##### 3.4.2 `int write_bit(int chip_number, int bit_number, int value)`

This function takes arguments similar to *read\_bit* and adds the *value* argument which is either 1 or 0. Writing a 1 to a bit sets the output pin to a low state. Writing a 0 releases the pin so that that it is pulled high. Return value is 0 on success or -1 if the chip is inaccessible or invalid.

##### 3.4.3 `int set_bit(int chip_number, int bit_number)`

This function takes arguments of *chip\_number* (1-6) and *bit\_number* (1-48). The value returned is 0 if successful or -1 if the *chip\_number* is invalid or the chip is not accessible. On success, the output pin associated with the bit is driven low.

##### 3.4.4 `int clear_bit(int chip_number, int bit_number)`

This function takes the arguments *chip\_number* (1-6) and *bit\_number* (1-48). It returns 0 on success and -1 if the *chip\_number* is invalid or the specified chip is not accessible. On success, the output pin associated with the bit is released to a high state.

##### 3.4.5 `int enab_int(int chip_number, int bit_number, int polarity)`

This function takes three arguments. The *chip\_number* (1-6), the *bit\_number* (1-24) and the *polarity* (1= rising edge, 0 = falling edge). The chip is then armed and transitions on the specified bit will cause an interrupt to occur. The driver will buffer up these interrupts and hand them out to calling programs using

either *get\_int()* or *wait\_int()*. Note that the input pins on the WS16C48 are NOT debounced and depending on what type of stimulus is presented to the input pin, the possibility exists for multiple transitions and interrupts to occur. It is the responsibility of the application program to filter or debounce these types of multiple interrupts.

#### 3.4.6 int *disab\_int*(int *chip\_number*, int *bit\_number*)

This function disables polarity sensing interrupts on the specified *chip\_number* (1-6) at the specified *bit\_number* (1-24). A return value of 0 signals success, a return value of -1, indicates an invalid *chip\_number* or an inaccessible chip.

#### 3.4.7 int *clr\_int*(int *chip\_number*, int *bit\_number*)

This function takes as arguments the *chip\_number* (1-6) and the *bit\_number* (1-24) and returns 0 on success or -1 on error. This function is ordinarily not used as the ISR in the driver will clear an interrupt as it responds to it. This function is mostly used in the case where a chip was installed with no IRQ specification at the time the driver was loaded with *insmod* but an application has enabled event sensing anyway. Then an application can make repeated calls to *get\_int()* awaiting an event. When one does occur, this function, *clr\_int()* must be called to re-enable the sense interrupt for that particular bit.

#### 3.4.8 int *get\_int*(int *chip\_number*)

This function takes a single argument of the *chip\_number* (1-6) and returns either 0, if no event was sensed on that chip, -1 if the *chip\_number* was invalid or inaccessible, or a number between 1 and 24 which indicates that an event has occurred on that bit number. This function does NOT wait for an event. It returns immediately with either an error (-1), the top value in the interrupt buffer, or the result of polling the chip's registers for an event sense.

#### 3.4.9 int *wait\_int*(int *chip\_number*)

This function is nearly identical to *get\_int()* with one major exception. If there is no error, and if there is nothing in the event buffer, and if there is nothing in the event sense registers of the specified chip, then the current process is suspended and will remain so until some event is sensed on the specified chip. Certain signals can also awaken the process and cause it to return without an actual event having occurred. This is by design, and allows a process to be terminated even while being suspended deep within the device driver. As with *get\_int()* there are three possible types of return values. 0 signals that no interrupt occurred, -1 indicates an error, and a value between 1 and 24 signifies the bit on which an event sense occurred.

## 4 SAMPLE PROGRAMS

- 4.1 *Flash* is a very simple program that illustrates how to set and clear bits on the PCM-UIO96. It assumes that there are two chips available. We attached a series of LEDs to the output pins and this program sequences through the bits lighting each LED, pausing, turning off the LED and looping until the program is terminated. Flash can be built in two ways. Using the supplied *Makefile* simply type :

```
make flash
```

or you can directly invoke the compiler with :

```
gcc -static flash.c uio48io.o -o flash
```

- 4.2 *poll* is quite a step-up from *flash* in complexity. It uses the POSIX threads capability of Linux to create a couple of sub-processes that are used to monitor two chips for interrupts generated by high to low transitions on any of the first 24 bits of the two chips. Whenever either of the two monitor processes detects an interrupt, a message is displayed, and an event counter is updated. The foreground code simulates a command based user interface. Refer to the source code for *poll.c* for a further discussion of the methodology used in this program. This program demonstrates a simple way to coordinate, in the context of a single application program, with external asynchronous stimulus events. *Poll* can be rebuilt using the supplied *Makefile* with :

```
make poll
```

or by direct invocation of the compiler with :

```
gcc -D_REENTRANT -static poll.c uio48io.o -o poll -lpthread
```

**NOTE :** In both of these sample programs the *-static* switch is used with *gcc*. This causes all of the library functions used to be permanently linked with the executable. This creates a very large executable file. We use this technique because our ELS Linux distribution has a limited number of dynamic libraries present on the Disk-On-Chip and it's possible that the required library, especially in the case of *pthread* might not be present on the target system. It's certainly possible, and even recommended, that if a variety of programs are going to be run on an embedded system, to copy over all of the dynamic libraries necessary for their operation in which case static linking would not be required.