UIO48 Device Driver Package
2.6.38 Kernel-Based Linux

1 Introduction

1.0 The UIO48 Device Driver Package consists of a Linux Device Driver, an application programming interface library, and example application programs

1.1 The driver was built and tested on a Linux version 2.6.38-8 generic based Ubuntu 11.04 Natty Narwhal distribution.

1.2 The driver is for use with WinSystems, Inc. Digital I/O boards and Single Board Computers (SBC) that utilize our exclusive digital I/O ASIC dubbed the WS16C48. The WS16C48 has 48 configurable digital I/O points. Input points can be configured to use our unique "Event-Sense Interrupt-Generation" mechanism. The "Event-Sense Interrupt-Generation" mechanism will raise an interrupt when a specified event occurs on the associated input point. This mechanism dispenses with the traditional input-point "polling-loop" mechanism, and in so doing uses significantly fewer processor cycles. These reclaimed processor cycles can be put to use sampling more I/O points or devoted to high-level computational and/or logging tasks.

The following WinSystems, Inc. Products incorporate at least one (1) WS16C48 UIO functional block:

I/O Cards: PCM-UIO48A, PCM-UIO96B
SBC: EPX-C380, EBC-Z5, EBC-855. EBC-C3PLUS, EPX-855, PPM-LX800-G

1.3 This driver is provided on an 'as-is' basis and no warranty as to usability or fitness of purpose is inferred or claimed.

1.4 WinSystems, Inc. does not provide support for the modification of this driver. Customer application specific queries can be sent to: support@winsystems.com.

1.5 This work is provided under the terms of the GNU General Public License (GPL).

2 Installations and Build

2.0 The device driver and the sample applications are provided in source code form as a compressed zipped folder.

2.1 It will be necessary to become the root user to build and install the driver and device nodes.

2.2 The MAJOR number for this device is allocated dynamically. A static number can be assigned by editing the *uio48_init_major* variable at the beginning of ***uio48.c***.

2.3 To create the device driver Loadable Kernel Module and the sample applications in a command shell, execute: ***make all***. The device driver Loadable Kernel Module ***uio48.ko*** is created and moved to the appropriate kernel driver directory. The file access permissions are set to allow access by all users and groups; they may be changed manually as desired. The two sample programs ***flash*** and ***poll*** are also built.

make install will install the kernel driver to a kernel directory and create dependencies.
make uninstall will remove the kernel driver from the kernel directory.
make flash will create the flash sample program.
make poll will create the poll sample program.
make clean will remove objects created by the build.
make spotless will forcibly remove all artifacts of the build.

2.4 The device driver can be loaded with the provided initialization script ***uio48_load*** or manually. In either case modprobe is used to install the driver. The I/O base address and IRQ assignments should be specified as arguments to modprobe and should match the jumper settings on I/O cards or the SBC BIOS settings. Executing:

modprobe uio48.ko io=0x200 irq=10

This will install the driver, which supports up to four devices. Multiple devices can be loaded by executing the same command with multiple parameter settings.

modprobe uio48.ko io=0x200. 0x220 irq=10, 11

Interrupts can usually not be shared across boards, although the driver will attempt it anyway.

The *uio48_load* script can be added to the /etc/rc.local file to load the driver automatically on boot.

3 Driver Usage

3.0 The WS16C48 ASIC is accessed utilizing byte-wide I/O access instructions. The device driver model that is most similar is the "Character" model. Block oriented, File I/O, and random access operations on these devices (read, write, and seek) are very inefficient, and may not always give the desired results. The driver was designed to use ioctl as its principal programming interface.

3.1 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 include uio48.h and link to uio48io.o.

3.2 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.

3.3 Application Programming Interface

An object file containing the Application Programming Interface utilized by user level programs to access the Kernal Loadable Module device driver driven devices is created as part of the build procedure.

3.3.1 int read_bit(int chip_number, int bit_number)

This function takes as an argument the chip_number (1-4) 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.

3.3.2 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.

3.3.3 int set_bit(int chip_number, int bit_number)

This function takes arguments of chip_number (1-4) 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.

3.3.4 int clr_bit(int chip_number, int bit_number)

This function takes the arguments chip_number (1-4) 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.

3.3.5 int enab_int(int chip_number, int bit_number, int polarity)

This function takes three arguments. The chip_number (1-4), 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.

3.3.6 int disab_int(int chip_number, int bit_number)

This function disables polarity-sensing interrupts on the specified chip_number (1-4) 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.

3.3.7 int clr_int(int chip_number, int bit_number)

This function takes as arguments the chip_number (1-4) 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 the driver was installed using insmod with no IRQ specification 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.

3.3.8 int get_int(int chip_number)

This function takes a single argument of the chip_number (1-4) 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.

3.3.9 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 will sleep 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 though it is asleep. 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.

3.3.10 int read_int_pending(int chip_number)

This function takes a single argument of the chip_number (1-4) and returns either the 8-bit contents of the INT_PENDING Register of the WSUIO48 device or -1 if the chip_number was invalid or inaccessible.

3.3.11 int clr_int_id(int chip_number, int port_number)

This function takes as arguments the chip_number (1-4) and the port_number (0-2) corresponding to the interrupt ID byte to be cleared. Return value is 0 on success or -1 if the chip

is inaccessible or invalid. This function writes a zero to the specified INT_ID register. This has the effect of clearing all pending interrupts for the specified port.

3.3.12 int read_byte(int chip_number, int port_number)

This function takes as arguments the chip_number (1-4) and the port_number (0-5) and returns either the 8-bit contents of the register offset specified by port_number on the WSUIO48 device or -1 if the chip_number was invalid or inaccessible.

3.3.13 int write_byte(int chip_number, int port_number, int val)

This function takes as arguments the chip_number (1-4), the port_number (0-5), and the value to be written to the register offset specified by port_number on the WSUIO48 device. Return value is 0 on success or -1 if the chip is inaccessible or invalid.

3.3.14 int lock_port(int chip_number, int port_number)

This function takes as arguments the chip_number (1-4) and the port_number (0-5) to be locked on the WSUIO48 device. This prevents ()2.368(0)-10.96(-6166333(_)-10.956(u95641(u)-0.9564181.f

4 Sample Programs

4.1 Flash

The “Flash” sample application is a simple program that illustrates how to set and clear output points. All I/O points are programmed to be outputs and cleared. Then starting with the least significant bit, each bit is set and after a fixed interval, cleared. The sequence is then repeated indefinitely. The Flash executable is built as part of the driver build, but may be built separately by: ***make flash***.

4.2 Poll

The “Poll” sample application is quite a step-up from “Flash” in complexity. It uses the POSIX threads capability of Linux to create two 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. The Poll executable may be built separately by: ***make poll.***

NOTE: In both of these sample programs the -static switch is used with gcc. This causes all library functions used, to be contained within 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.