



PCM-VDX PWM Device Driver Package

2.6.32 Kernel-Based Linux

1 Introduction

1.0 The PCM-VDX PWM 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 2.6.32-24-generic based Ubuntu 10.04 Lucid Lynx distribution.

1.2 The applications are for use with WinSystems, Inc. PCM-VDX-1-256 and PCM-VDX-2-512 Single Board Computers (SBC) which provide 16-lines of TTL-compatible digital I/O.

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 and bug reports may be sent to: linux_drivers@winsystems.com.

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

2 Installation and Build

2.0 The device driver and the sample applications are provided in source code form as a gzipped TAR-ball.

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

2.2 All application programs assume that any Multi-Function Port being used has been configured for PWM using the BIOS Setup Utility.

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

2.4 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 *pcmvdxpwm.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, *pulse* 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 pulse will create the pulse 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.

2.5 The device driver can be loaded with the provided initialization script *pcmvdxpwm_load* or manually. In either case modprobe is used to install the driver. Executing:

```
modprobe pcmvdxpwm.ko
```

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

3 Driver Usage

3.0 The PCM-VDX Digital I/O consists of sixteen dedicated programmable I/O pins consisting of two individual 8-bit ports. Each port can be configured as GPIO or Pulse Width Modulation (PWM) outputs.

All GPIO pins are independent and can be configured as inputs or outputs. When configured as outputs, pins have 8 mA drive capability and are unterminated; when configured as inputs, pins are pulled-high with a 75k ohm resistance. Each input pin also supports interrupt triggers.

All PWM pins are independent and can be configured to output a continuous frequency or a fixed number of pulses. The frequency is selected by programming high and low pulse count values. An interrupt can be used to indicate when a pulse count has completed.

The features are configured and controlled utilizing PCI configuration and I/O access instructions.

3.1 The file *pcmvdxpwmio.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 need for dealing with, or understanding how to access the driver through *ioctl*. An application must include *pcmvdxpwm.h* and link to *pcmvdxpwmio.o*.

3.2 Applications using the driver may enable interrupts on any or all of the channels. The application may further specify the event polarity 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 Kernel Loadable Module device driver driven devices is created as part of the build procedure.

3.3.0 **int ioctl_enab_pwm(int channel, unsigned long low, unsigned long high)**

This function requires arguments of channel (0-15) and the 32-bit pulse values, low and high, which program the low and high counts for the waveform. This function configures a continuous waveform. Return value is 0 on success or -1 if the device is inaccessible.

3.3.1 **int ioctl_enab_pwm_rc(int channel, unsigned long low, unsigned long high, unsigned long repeat)**

This function requires arguments of channel (0-15) and the 32-bit pulse values, low and high, which program the low and high counts for the waveform. This function configures a single waveform with the number of pulses equal to the repeat argument. Return value is 0 on success or -1 if the device is inaccessible.

3.3.2 **int ioctl_disab_pwm(int channel)**

This function requires an argument of channel (0-15). This function disables the selected channel. Return value is 0 on success or -1 if the device is inaccessible.

3.3.3 **int ioctl_enab_pwm_int(int channel)**

This function requires an argument of channel (0-15). The device is then armed and completion of the repeat count will cause an interrupt to occur. The driver will buffer these interrupts and hand them to calling programs using either `ioctl_get_pwm_int()` or `ioctl_wait_pwm_int()`. Return value is 0 on success or -1 if the device is inaccessible.

3.3.4 **int ioctl_disab_pwm_int(int channel)**

This function requires an argument of channel (0-15). This function disables interrupts for the selected channel. Return value is 0 on success or -1 if the device is inaccessible.

3.3.5 **int ioctl_clr_pwm_int(int channel)**

This function requires an argument of channel (0-15). This function is ordinarily used within an Interrupt Service Routine to clear an interrupt and re-enable the interrupt for that particular channel. Return value is 0 on success or -1 if the device is inaccessible.

3.3.6 **int ioctl_get_pwm_int(void)**

This function requires no arguments and returns a channel on which an interrupt event has occurred. Returns values of 1 to 16 correspond to Channel 0 to Channel 15. Return value is 0 if no event has occurred on any channel or -1 if the device is not accessible. This function does NOT wait for an event.

3.3.7 **int ioctl_wait_pwm_int(void)**

This function is nearly identical to `ioctl_get_pwm_int()` with one major exception. If there is no error and no event has occurred, the current process will sleep until some event is sensed. 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 `ioctl_get_pwm_int()` there are three possible types of return values: 0 signals that no interrupt occurred, -1 indicates the device is not accessible, and a value between 1 and 16 indicates the channel on which an event sense occurred. This function requires no arguments and returns a channel on which an interrupt event has occurred.

3.3.8 `int ioctl_pwm_set_clk(int freq)`

This function requires an argument of frequency (0-1). This function selects the internal clock speed for the PWM clock of 10 MHz (0) or 50 MH (1). Return value is 0 on success or -1 if the device is inaccessible.

4 Sample Programs

4.0 Pulse

The “Pulse” sample application is a simple program that illustrates how to program waveforms on specific channels in both continuous and non-continuous modes. Using the channel number provided on the command line, waveforms with different frequencies and duty cycles are output in continuous mode. Then a waveform of varying pulse lengths in non-continuous mode is output on the same channel. The Pulse executable is built as part of the driver build but may be built separately by: *make pulse*.

4.1 Poll

The “Poll” sample application is a step-up from “Pulse” in complexity. It uses the POSIX threads capability of Linux to create two sub-processes that are used to monitor the device for interrupts generated on any of 16 channels. 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 of `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*.