PCM-CAN Module
CAN 2.0A Driver Package
Ubuntu Linux 12.04 LTS

1 Introduction

1.0 This PCM-CAN Driver Package consists of a Linux Device Driver and an example application program that support CAN version 2.0A. This version of the driver does not support extended CAN IDs.

1.1 The driver was built and tested with Ubuntu Linux 12.04 LTS, kernel version 3.2.0-29-generic-pae.

1.2 The driver is for use with WinSystems PCM-CAN boards.

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

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 Installation and Build

2.0 The device driver and the sample application are provided in a *.tar* file.

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 statically set to 65. A different static number can be assigned by editing the MAJOR variable at the beginning of the ***Makefile***.

2.3 To create the device driver Loadable Kernel Module in a command shell, execute: `make all`. The device driver Loadable Kernel Module *CAN_LinuxDrv.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.

`make install` - Installs the kernel driver to a kernel directory and create dependencies.
`make uninstall` - Removes the kernel driver from the kernel directory.
`make clean` - Removes objects created by the build.
`make spotless` - Forcibly removes all artifacts of the build.

2.4 The device driver can be loaded with the provided initialization script `CAN_LinuxDrv_load` or manually. In either case `modprobe` is used to install the driver. The I/O base address and IRQ assignments are parameters passed to the IOCTL initialization call from the application and should match the jumper settings on the board as described in the PCM-CAN manual.

The `CAN_LinuxDrv_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-CAN 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 *CAN_LinuxDrv.o* implements the `ioctl` interface and presents to the application a set of standard C functions that may be called through the Linux operating system character driver interface (`open`, `ioctl`, and `close`).

3.2 Interrupts are handled entirely in the driver and are transparent to the user application.

3.3 Application Programming Interface

3.3.1 Before any of the `ioctl` functions can be called, the driver must obtain an open "file" handle for accessing the PCM-CAN device through the kernel character driver model interface. This is accomplished through the following Linux system call:

```
hDevice = open("/dev/CAN_LinuxDrv", O_RDWR, S_IRWXU);
```

3.3.2 IOCTL_CANDRV_NUMPORT

The first call to the driver after it is opened needs to obtain the number of ports supported by the current device. This must be correct because the driver can crash your system if this is configured incorrectly. The sample code is as follows:

```
nPorts.numPorts = 2; this number is 1 or 2 depending on which CAN board you
have

error  =  ioctl((unsigned int)hDevice,
          IOCTL_CANDRV_NUMPORT,
          (unsigned long)&nPorts);
```

3.3.3 INIT_CAN_DEVICE

The device must then be initialized with the following code. This initializes the PCM-CAN hardware and sets up the IO port space and the IRQ resources. The device takes 32 I/O addresses per CAN channel so care must be taken as to where this I/O space is positioned. The

parameter `portOffset` is provided for the possibility of future hardware changes. But this is physically fixed in the hardware on any given version. The `bitRate` parameter is a decimal bits-per-second value. The "baud" rate value to be loaded into the device is determined in the driver.

```
INIT_DATA init;
init.baseIO       = (unsigned char *)0x00000600;
init.portOffset   = 0x0020;
init.bitRate      = 500000;
init.irq          = 10;

error = ioctl((unsigned int)hDevice,
         INIT_CAN_DEVICE,
         (unsigned long)&init);
```

3.3.4 IOCTL_CANDRV_WRITE

To write data across the CAN bus requires the following code. This call requires the `messageID`, `messageLEN`, and `messageData` parameters. The `messageLEN` is provided for user control, however all basic CAN messages are 8 bytes in length. Using another size could result in unpredictable results. Additionally, the channel number (0 or 1) is provided. All receive and transmit activities are channel based.

```
TPCANMsg wrMessage;

// The PCM-CAN driver currently only supports standard format messages
// with 8 bytes of data. So this example is the most common standard
// version.
wrMessage.ID      = 0x0300;
wrMessage.LEN     = 8;
wrMessage.DATA[0] = 1;
wrMessage.DATA[1] = 2;
wrMessage.DATA[2] = 3;
wrMessage.DATA[3] = 4;
wrMessage.DATA[4] = 5;
wrMessage.DATA[5] = 6;
wrMessage.DATA[6] = 7;
wrMessage.DATA[7] = 8;

wrData.message = (unsigned char *)&wrMessage;
wrData.channel = channel;

error = ioctl((unsigned int)hDevice,
        IOCTL_CANDRV_WRITE,
        (unsigned long)&wrData);
```

3.3.5 IOCTL_CANDRV_READ

To read data across the CAN bus requires the following code. This call requires the `channel` parameter and a buffer pointer to receive the datagram. As the comments state, the messages received from the hardware are interrupt driven and queued in the driver for the application to ask for them as desired. When the function returns `ERR_NO_DATAGRAM_PRESENT`, this tells the application that there are no more received datagrams queued.

```
READ_DATA rdData;
```

```
// The device driver is interrupt driven but user level code must call
// the driver to receive the message. The ReadDatagram call must be made
// for each message read. It will return ERR_NO_DATAGRAM_PRESENT
// when there is nothing for it to return.

rdData.channel = channel;
rdData.message = (unsigned char *)&rdMessage;

error = ioctl((unsigned int)hDevice,
        IOCTL_CANDRV_READ,
        (unsigned long)&rdData);
```

3.3.6 IOCTL_GET_VERSION

This Linux system call will cause the driver to return its software version number in the `verData` variable.

```
VERSION_DATA verData;

error = ioctl((unsigned int)hDevice,
        IOCTL_GET_VERSION,
        (unsigned long)&verData);
```

3.3.7 When the device is no longer needed, the driver must release the "file" handle obtained earlier. This is accomplished through the following Linux system call.

```
close(hDevice)
```

4 Sample Program

4.1 CanMonitor

CanMonitor is a graphical interface application that illustrates the capabilities of the WinSystems CAN driver. This application is built with the Qt Creator SDK. The runtime portion of the Qt environment was rebuilt to link statically so that the executable would be a standalone application. For this application to run, the device driver must be installed correctly and the software configuration must match the corresponding hardware configuration. This point is critical because incorrect configuration of the system can result in system crash or lockup. The base I/O address and IRQ of the WinSystems PCM-CAN card are configured via jumpers. The galvanically isolated version of the board requires jumpers to control who provides power to the CAN bus. This is complex and must be correct for the system to work.

Once the device driver and hardware are correctly installed and configured, you can run CanMonitor. Once the CanMonitor window is up, the first thing that must be done is software configuration. No other buttons will do anything until the software configuration is complete. To get started, click on the <*Set Parameters*> button. <*Set Parameters*> opens a dialog window containing buttons and boxes that allow you to enter the configuration details of your hardware. Once all of these are set, click the <*Ok*> button and the system will initialize the driver and save the configuration. After configuration, all the other buttons on the screen are active. You can always come back to the <*Set Parameters*> window again later to change the configuration but some parameters may require the device to be closed and reopened if they are changed. This will interrupt the CAN bus.

In the configuration dialog, there is a time interval option. The default value is set to zero which configures the <*Send Message(s)*> button to send exactly one message each time it is pressed. If a non-zero value is entered, the <*Send Message(s)*> button will cause the application to send messages at intervals based on the value entered. This value is in milliseconds. Please note that Linux is not a real-time operating system by default so the actual time interval can vary somewhat from your set value.

The <*Stop Sending*> button is only functional if messages are being sent repeatedly on an interval. This button will stop the sending process.

The functions of the remaining buttons are fairly obvious. They clear the various display windows. The <*Exit*> button will exit the application.

The <*Get Device Driver Version*> button simply returns the version number for the loaded driver.

All the information buttons display the returned contents in the information window.