PCM-CAN Windows
Device Driver Package

1 Introduction

1.1 The WinSystems PCM-CAN family of devices provides PC/104-compliant Controller Area Network (CAN Bus) peripheral modules. Four different board types provide one or two CAN channels and isolated or non-isolated connections.

1.2 Each board utilizes the NXP SJA-1000 CAN controller ICs. The "SJA1000 Stand-alone CAN controller" data sheet (dated 2000 Jan 04) is referenced multiple times throughout this document.

1.3 This driver only supports the PeliCAN mode of the SJA-1000 CAN controller. This mode supports the CAN 2.0B protocol specification.

1.4 The PCMCAN driver package is designed for and has been verified with 32-bit and 64-bit versions of Microsoft Windows 7, 8, and 10.

2 Installation

2.1 Before loading the Windows operating system, verify that the PCM-CAN board jumpers are configured for the desired I/O base address and interrupt. In the BIOS set-up menus, verify that the selected resources are not used by other devices.

2.2 The driver, support files, and console applications are supplied in a zip file. The following files are included:

- pcmcan.sys – Windows device driver
- pcmcan.inf – Windows installation file
- pcmcan.cat – Windows catalog file
- WdfCoinstaller01011.dll – Windows co-installer
- pcmcanDLL.dll – Windows DLL
- pcmcanDLL.lib – Windows library file
- pcmcanDLL.h – driver include file
- CanDemo.exe – Windows GUI application
- send.cpp – Windows application source
- send.exe – Windows application
- receive.cpp – Windows application source
- receive.exe – Windows application

- vcredist_x86 or vcredist_x64 - Microsoft Visual C++ Redistributable
- Philips Semi SJA-1000 Data Sheet

2.3 Installation is accomplished via the 'Add legacy hardware' selection found in the Action menu of the Windows Device Manager. Navigate to the drive and folder containing the driver files and select *pcmcan.inf*. The Windows installer will copy the *pcmcan.sys* driver file to the appropriate directory in the Windows installation.

2.4 In Device Manager, the PCM-CAN Device will appear under the System Devices item. The desired hardware configuration can be selected under the Resources tab of the PCMCAN Device Properties window. A reboot may be required after resource selection is complete.

2.5 The included console applications, *send.exe* and *receive.exe, c*an be used to verify driver installation and functionality. Usage of the programs is described later in this document. These applications require a CAN test adapter that can transmit and receive CAN 2.0B packets.

3 Driver Overview and Architecture

3.1 The file *pcmcan.sys* is a Windows Driver Foundation kernel-mode (KMDF) driver which facilitates access to the underlying hardware.

3.2 The file *pcmcanDLL.dll* is a Windows Dynamic-Link Library which provides more user-friendly functions to access the device.

3.3 The driver utilizes the I/O Control (IOCTL) Request framework to control the register set of the SJA-1000 CAN controller. Data is passed to and from the driver utilizing input and output buffers.

4 Driver Usage

4.1 The *pcmcanDLL.h* file included in the driver distribution contains the function definitions to be used by an application to communicate with the pcmcan driver. This file is included in Appendix A: pcmcanDLL.h.

4.1 An application calls the function *InitializeSession* to open the driver. This is required before any of the other functions can be called. The following example opens the WinSystems pcmcan driver. If a zero is returned, then the driver has been successfully initialized. Any other returned value indicates that an error has occurred and the device is unusable.

```
if (InitializeSession())
      printf("Error opening device.\n");
```

4.3 Once the driver is initialized, the other functions can be used to control the PCMCAN. Following is a description and sample code for each function. For all functions, if a zero is

returned, the function was successful. Any other returned value indicates that an error or board condition prevented function completion. For some return values, the function can be retried to see if the board condition has changed.

All functions require an input variable specifying the channel selected. Valid options for this variable depend on the board type. For single channel boards, the only valid variable is one. For dual channel boards, the variable can be one or two.

4.3.1 int StartChannel(int channel)

This function initializes the selected channel and configures it to start transmitting and receiving messages.

```
if (StartChannel(1))
       printf("Error starting channel 1\n");
```

4.3.2 int ResetChannel(int channel)

This function resets the selected channel. The specified channel is restored to a known state and any previous configuration is lost. The transmit and receive buffers are both reset and cleared. The StartChannel function must be run for the channel after a reset.

```
if (ResetChannel(2))
       printf("Error resetting channel 2\n");
```

4.3.3 int SetBitRate(int channel , unsigned long bitRate)

Configures the specified channel for the selected bit rate. Any bitrate less than or equal to 1,000,000 (1Mbit./sec) is allowed.

```
#define BITRATE     33333

if (SetBitRate(1, BITRATE))
       printf("Error setting bit rate\n");
```

4.3.4 int WriteDatagram(int channel, unsigned long msgID, unsigned int msgType, int msgLen, unsigned int *data)

This function attempts to send the specified message over the CAN bus. The input variables define the message ID, type, length, and data payload. An enumerated variable type is provided in the *pcmcanDLL.h* file which defines valid values for message type.

```
typedef enum {
   SFF_DATA = 0,    // standard data
   SFF_RTR,         // standard remote
   EFF_DATA,        // extended data
   EFF_RTR,         // extended remote
   MAX_MSG_TYPES
} MsgType;
```

There are two different modes that can be used to transmit messages, polled and interrupt-driven. The mode can be selected by enabling or disabling the Transmit Interrupt Enable bit

in the SJA-1000 Interrupt Enable Register. This can be done using the EnableInterrupts function described in section 4.3.9.

4.3.4.1 Polled Transmission

During a polled transmission, the Transmit Interrupt is disabled. After the WriteDatagram function is called, the SJA-1000 Status Register is read to check the Transmit Buffer Status (TSB) bit. Once that bit is set, the next message can be sent. To confirm that the transmission was successful, the Transmission Complete Status (TCS) bit should also be checked.

The following code demonstrates a polled transmission of a standard data frame followed by a standard remote frame.

```
#define TBS_BIT 0x04 // bit 2

int dllReturn = 0;
unsigned int data[] = { 0xa5, 0xc3 };
unsigned int status_reg;
unsigned int arbit_lost_capture_reg;
unsigned int error_code_capture_reg;

dllReturn = WriteDatagram(1, 0x400, SFF_DATA, sizeof(data)/sizeof(unsigned
int), data);

if (dllReturn != SUCCESS) {
    printf("Error sending data frame\n");
    exit(dllReturn);
}

do {
   dllReturn = ReadStatus(CHANNEL, &status_reg, &arbit_lost_capture_reg,
&error_code_capture_reg);

   if (dllReturn)
   {
       printf("Error reading status\n");
       exit(dllReturn);
   }

   Sleep(1);
} while (!(status_reg & TBS_BIT));

dllReturn = WriteDatagram(1, 0x123, SFF_RTR, 0, NULL);

if (dllReturn != SUCCESS)
    printf("Error sending remote frame\n");
```

4.3.4.2 Interrupt-Driven Transmission

During an interrupt-driven transmission, the Transmit Interrupt is enabled and the WriteDatagram function is called to send the first message. The next message to send can immediately be requested, and if the previous transmission is not complete, this

new message will be buffered. An interrupt is triggered at completion of the current message which will invoke transmission of the next buffered message.

The driver will buffer up to eight messages for future transmission. If the buffer is full, the function will return a XMIT_BUFFER_FULL status. The application can then resend the transmission request when ready.

At any time, the channel may enter a bus-off state due to excessive errors. At this point the device will no longer attempt any transmissions and return a CAN_BUS_OFF status. A channel reset will reset the bus state, but the bus off condition may reoccur if the cause of the bus error is not addressed.

The following code demonstrates an interrupt-driven transmission of an extended data frame followed by an extended remote frame.

```
int dllReturn = 0;
unsigned int data[] = { 0x11, 0x22, 0x33 };

do {
    dllReturn = WriteDatagram(2, 0x1f000000, EFF_DATA, sizeof(data)/
sizeof(unsigned int), data);

    if (dllReturn != SUCCESS && dllReturn != XMIT_BUFFER_FULL) {
        printf("Error sending message\n");
        exit(dllReturn);
    }
} while (dllReturn);

do {
    dllReturn = WriteDatagram(2, 0x12345678, EFF_RTR, 0, NULL);

    if (dllReturn != SUCCESS && dllReturn != XMIT_BUFFER_FULL)
        printf("Error sending message\n");
} while (dllReturn);
```

4.3.5 int ReadDatagram(int channel, unsigned long *msgID, unsigned int *msgType, int *MsgLen, unsigned int *data)

This function attempts to retrieve a receive message on the specified channel. The message information is returned using the four pointers provided in the function parameters. If there is no message to retrieve, the function returns a RCV_BUFFER_EMPTY status.

There are two methods for reading any incoming message: by polling the Receive Buffer Status bit in the status register or allowing receive interrupts to buffer up to 32 messages. The mode can be selected by enabling or disabling the Receive Interrupt Enable bit in the SJA-1000 Interrupt Enable Register. This can be done using the EnableInterrupts function described in section 4.3.9.

4.3.4.1 Polled Receipt

One or more messages are available for retrieval when the Receive Buffer Status (RBS) bit in the Status Register is set. The ReadDatagram function is then called to read the available message. This function can be called until the buffer is empty which is indicated when the RBS bit is cleared. The Receive Interrupt should be disabled in this mode.

The following code demonstrates the polling method on channel 1.

```
#define RBS_BIT 0x01 // bit 0

int dllReturn = 0;
unsigned int status_reg;
unsigned int arbit_lost_capture_reg;
unsigned int error_code_capture_reg;
int msgLen = 0;
unsigned int msgType = 0, data[8] = { 0 };
unsigned long msgID = 0;

do {
    dllReturn = ReadStatus(1, &status_reg, &arbit_lost_capture_reg,
&error_code_capture_reg);

    if (dllReturn)
       printf("Error reading status\n");

    Sleep(1);
} while (!(status_reg & RBS_BIT));

dllReturn = ReadDatagram(1, &msgID, &msgType, &msgLen, data);

if (dllReturn == RCV_BUFFER_EMPTY)
{
    printf("No message to read\n");
}
else if (dllReturn != SUCCESS)
{
    printf("Read error\n");
    exit(dllReturn);
}
```

4.3.4.2 Interrupt-Driven Receipt

To use interrupts to receive messages, the Receive Interrupt must be enabled. When the receive buffer on the SJA-1000 is not empty, the interrupt is generated, and the interrupt handler retrieves a message and moves it to the driver receive buffer. As long as there are messages in the device buffer, the interrupt is continuously triggered.

The ReadDatagram function is called to read messages from the driver receive buffer. The driver architecture prevents this function from being truly interrupt driven. Polling is still required to read packets from the intermediate buffer. A continuous loop can be used to that continuously read the buffer. If no message is available the

function returns a RCV_BUFFER_EMPTY status. The frequency of reading the buffer is application driven but messages can be lost if there is too much latency. If the buffer is full when a new message arrives, the message will be discarded and the Data Overrun Status bit will be set in the Status Register.

The following code demonstrates the interrupt-driven method on channel 2.

```
int dllReturn = 0;
int msgLen = 0;
unsigned int msgType = 0, data[8] = { 0 };
unsigned long msgID = 0;

do
{
    dllReturn = ReadDatagram(2, &msgID, &msgType, &msgLen, data);

    if (dllReturn == RCV_BUFFER_EMPTY)
    {
        printf("No message to read\n");
    }
    else if (dllReturn != SUCCESS && dllReturn != RCV_BUFFER_EMPTY)
    {
        printf("Read error\n");
        exit(dllReturn);
    }

    Sleep(1); // latency between reads
} while (dllReturn); // 0 indicates message has been read
```

4.3.6 SetHwFilter(int channel, int mode, long acceptanceCodes, long acceptanceMasks)

This function configures the acceptance filter which controls which messages are forwarded to the receive buffer. The default state of the filter allows all messages to pass. The filter is defined by four Acceptance Code and four Acceptance Mask Registers. Two different filter modes are selected with the 'mode' input parameter. An enumerated variable type is provided in the *pcmcanDLL.h* file which defines valid values for filter mode: 0 = DUAL and 1 = SINGLE.

Refer to the SJA-1000 Data Sheet on page 44 for more information on the various filter configurations.

The following code demonstrates a single acceptance filter configuration for extended frames. In this example, we want to pass data messages to ID 0x1234XXXX.

```
unsigned long accCodes = 0x91A00000; // 0x1234000 >> 1
unsigned long accMasks = 0x0007FFF8; // bit 2 = RTR bit

if (SetHwFilter(1, SINGLE, accCodes, accMasks))
       printf("Error configuring filter\n");
```

4.3.7 ClearHwFilter(int channel)

This function clears the acceptance filter which controls which messages are forwarded to the receive buffer. This returns the filter to its default state which allows all messages to pass.

```
if (ClearHwFilter(2))
       printf("Error clearing filter\n");
```

4.3.8 SetSwFilter(int channel, unsigned int msgType, long lowIDRange, long highIDRange)

This function configures the software filter which controls which messages are forwarded to the receive buffer. The default state of the filter allows all messages to pass. The filter is defined by byte message type and a low and high ID range. Any message that matches the chosen settings will be passed to the receive buffer. All messages of the selected type that do not fall in the ID range will be discarded. All messages of a different message type are also allowed to pass.

The following code demonstrates a software filter configuration for extended frames. In this example, we want to pass data messages received on channel 2 with ID between 0x0001234 and 0x00005678.

```
#define SWF_LOW_ID  0x00001234
#define SWF_HI_ID   0x00005678

if(SetSwFilter(CHANNEL, EFF_DATA, SWF_LOW_ID, SWF_HI_ID))
       printf("Error adding sw filter\n");
```

4.3.9 ClearSwFilter(int channel)

This function clears the software filter which controls which messages are forwarded to the receive buffer. This returns the filter to its default state which allows all messages to pass.

```
if (ClearSwFilter(2))
       printf("Error clearing filter\n");
```

4.3.10 int ReadStatus(int channel, unsigned int *status_reg, unsigned int *arbit_lost_capture_reg, unsigned int *error_code_capture_reg)

This function returns the current value of three SJA-1000 registers: Status Register, Arbitration Lost Capture Register, and Error Code Capture Register. Refer to the SJA-1000 Data Sheet for specific register bit definitions.

The following code demonstrates how to check the Bus Status bit (bit 7) in the Status Register.

```
#define BS_BIT 0x80 // bit 7

int dllReturn = 0;
unsigned int status_reg;
unsigned int arbit_lost_capture_reg;
unsigned int error_code_capture_reg;
```

```
    dllReturn = ReadStatus(CHANNEL, &status_reg, &arbit_lost_capture_reg,
&error_code_capture_reg);

    if (dllReturn)
    {
        printf("Error reading status registers\n");
        exit(dllReturn);
    }

    if (!(status_reg & BS_BIT))
        printf("Bus is on\n");
    else
        printf("Bus is off\n");
```

4.3.11 int EnableInterrupts(int channel, unsigned int irqMask)

This function allows the user to select which interrupt types will generate a system interrupt. A one on the specific line will enable that interrupt. An enumerated variable is provided that will allow the user to enable or disable the desired interrupts. The following table lists the enumerated values provided. Refer to the SJA-1000 Data Sheet on page 32 for more specific register bit definitions.

Enumerated Interrupt Values
ALL_IRQ_DISABLE
BUS_ERROR_IRQ_ENBL
ARB_LOST_IRQ_ENBL
ERROR_PASSIVE_IRQ_ENBL
WAKE_UP_IRQ_ENBL
DATA_OVERRUN_IRQ_ENBL
ERROR_WARNING_IRQ_ENBL
TRANSMIT_IRQ_ENBL
RECEIVE_IRQ_ENBL
ALL_IRQ_ENABLE

The following example will enable only the Bus Error and Transmit interrupts on channel 1.

```
if (EnableInterrupts(1, BUS_ERROR_IRQ_ENBL | TRANSMIT_IRQ_ENBL))
       printf("Error enabling selected interrupts\n");
```

The following example will disable only the Receive interrupt on channel 2.

```
if (EnableInterrupts(2, ALL_IRQ_ENABLE & ~RECEIVE_IRQ_ENBL))
       printf("Error disabling selected interrupt\n");
```

4.3.12 int GetErrorCounters(int channel, unsigned int *txErrorCounter, unsigned int *rxErrorCounter)

This function returns the current value of the two SJA-1000 error counters: RX Error Counter and TX Error Counter Registers. Refer to the SJA-1000 Data Sheet on page 38 for specific register descriptions.

The following code demonstrates how to read the transmit error counter on channel 2.

```
int dllReturn = 0;
unsigned int txErrCntr, rxErrCntr;

dllReturn = GetErrorCounters(2, &txErrCntr, &rxErrCntr);

if (dllReturn)
    printf("Error reading error counters\n");
else
    printf("Transmit Errors = %d\n", txErrCntr);
```

4.3.13 int CloseSession(void)

This function is used to disable the pcmcan device and close the driver when complete. If a zero is returned, the driver is closed. Otherwise there was an error and the driver is still open.

```
if (CloseSession())
       printf("Error closing driver\n");
```

4.4 Every function returns a zero or a positive integer value indicating success or failure. If a zero is returned, the function has completed successfully. If a failure occurs, the specific value returned provides more clarity as to the failure mechanism.

4.4.1 DRIVER_ERROR (1)

This error indicates that some function within the driver has failed. This error indicates that one of the IOCTL calls within the driver itself has not completed successfully. Using Windows Device Manager, verify that the driver is loaded and has no resource conflicts. Verify that the jumper settings on the device match those selected in Device Manager.

4.4.2 INVALID_HANDLE (2)

This error indicates that the driver has not initialized or closed. The driver attempts to obtain a handle to the PCM-CAN device, and this error indicates that the handle was not obtained. Verify that the driver has loaded successfully.

4.4.3 INVALID_PARAMETER (3)

This error indicates that one of the parameters in a DLL function is out of bounds. Check your parameter definitions.

4.4.4 XMIT_BUFFER_FULL (4)

This error indicates that the transmit buffer is full and will only be returned if the transmit interrupt is enabled. If a transmission is attempted and this value is returned, there is no space to store this message for future transmission. The transmission can be continuously retried until space is once again available.

4.4.5 RCV_BUFFER_EMPTY (5)

This error indicates that the receive buffer is empty. If a read message is attempted and this value is returned, there is no received message currently available to return.

4.4.6 CAN_BUS_OFF (6)

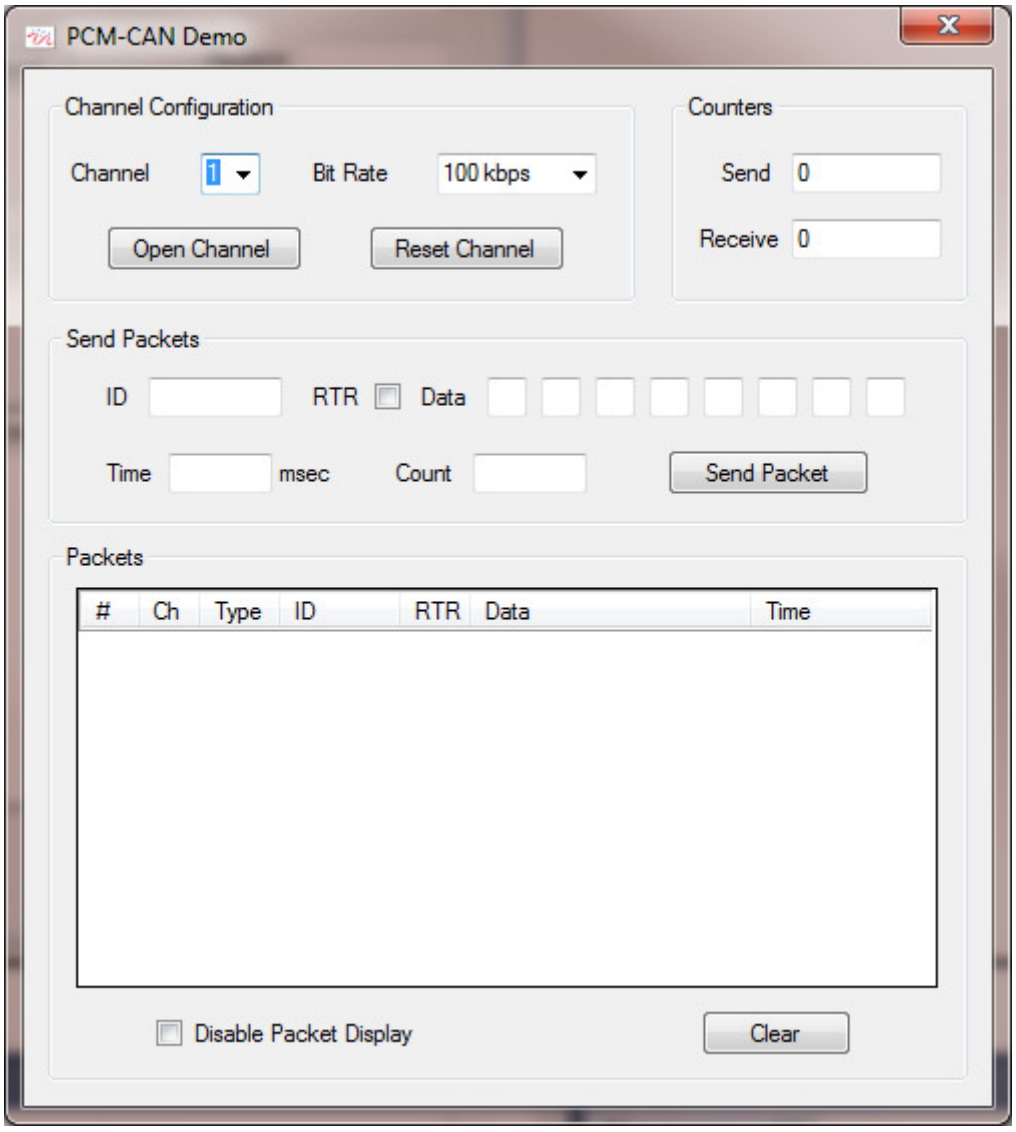
This error indicates that the CAN bus is currently in the 'bus off' mode. No transmission is allowed when in this state.

5 Sample Applications

There are three applications provided to exercise the PCM-CAN device. There is one Windows GUI application, *CanDemo*, and two sample Windows console applications provided in the driver package, *send* and *receive*. The source code for both console applications is provided in the Appendix section.

5.1 CanDemo

The *CanDemo* application is a Windows .NET application that allows the PCM-CAN device to send and receive packets on a CAN bus. A summary of each control follows the GUI screen snapshot.



5.1.1 Channel Configuration Group

5.1.1.1 Channel
Selects the desired channel to use – either 1 or 2. If using a single channel board, selecting channel two will result in an error.

5.1.1.2 Bit Rate
Selects the desired bit rate – 1 Mbps, 500 kbps, 250 kbps, or 100 kbps.

5.1.1.3 Open Channel
When pressed, the selected channel will be open with the selected bit rate.

5.1.1.4 Reset Channel
When pressed, the selected channel will be reset. If a new bit rate is selected, this will be reset also.

5.1.2 Counters Group

5.1.2.1 Send
Maintains the total number of packets sent.

5.1.2.2 Receive
Maintains the total number of packets received.

5.1.3 Send Packets Group

5.1.3.1 ID
Enter a hex value for the desired ID. If the ID uses three or less digits and is less than a value of 0x800, 11-bit mode is used during transmission. Otherwise 29-bit mode is used.

5.1.3.2 RTR
If this box is checked, an RTR packet is transmitted. The data fields are ignored for this type of packet.

5.1.3.3 Data
Enter up to eight bytes of data for transmission. Any bytes that are blank will be ignored. If you want to send only one byte, enter data in the first box and leave all others blank.

5.1.3.4 Time
If this box has a non-zero value, transmissions will occur at the defined rate in milli-seconds.

5.1.3.5 Count
If this box has a non-zero value, each transmission will send a number of packets equal to the value entered.

5.1.3.6 Send Packet

When pressed, the defined packet will be transmitted. If the Count box has a value, that number of the defined packet will be sent. If the Time box has a value, this button starts and stops the periodic transmissions.

5.1.4 Packets Group

This window will display all packets sent and received. It holds up to 2000 packets and then is cleared. The packet # column displays the packet number.

5.1.4.1 Disable Packet Display

If sending packets at a high rate, the display of packets can negatively affect device performance. By checking this box, the display can be disabled.

5.1.4.2 Clear

When pressed, the display is cleared as well as the Send and Receive packet counters.

5.2 send

The *send* sample application is a simple program that continuously transmits six unique CAN packets. It is configured to run at a bus speed of 500 kbps on channel 1. The program can be built to send packets when the buffer is available or store packets and use interrupts to transmit.

5.3 receive

The *receive* sample application is a simple program that continuously receives and displays CAN packets. It is configured to run at a bus speed of 500 kbps on channel 1. The program can be built to retrieve packets by polling the receive buffer or use interrupts to buffer the packets. The code can also provide 1 or 2 hardware filters and/or a software filter.

Appendix A: pcmcanDLL.h

```
//*****************************************************************************
//
//     Copyright 2017 by WinSystems Inc.
//
//*****************************************************************************
//
//     Name    : pcmcanDLL.h
//
//     Project : PCMCAN Windows DLL
//
//     Author  : Paul DeMetrotion
//
//*****************************************************************************
//
//       Date        Rev                    Description
//     --------    -------     ------------------------------------------
//     05/22/17      2.0           Original Release of DLL
//     06/22/17      2.1           Driver Improvements
//
//*****************************************************************************

#ifndef _PCMCAN_DLL_H_
  #define _PCMCAN_DLL_H_

#if defined DLL_EXPORT
#define DECLDIR __declspec(dllexport)
#else
#define DECLDIR __declspec(dllimport)
#endif

#define DLL_DEBUG

extern "C"
{
    DECLDIR int InitializeSession();
    DECLDIR int StartChannel(int channel, unsigned long bitRate);
    DECLDIR int ResetChannel(int channel);
    DECLDIR int SetBitRate(int channel, unsigned long bitRate);
    DECLDIR int WriteDatagram(int channel, unsigned long msgID, unsigned int msgType, int
msgLen, unsigned int *data);
    DECLDIR int ReadDatagram(int channel, unsigned long *msgID, unsigned int *msgType,
int *msgLen, unsigned int *data);
    DECLDIR int SetHwFilter(int channel, int mode, long acceptanceCodes, long
acceptanceMasks);
    DECLDIR int ClearHwFilter(int channel);
    DECLDIR int SetSwFilter(int channel, unsigned int msgType, long lowIDRange, long
highIDRange);
    DECLDIR int ClearSwFilter(int channel);
    DECLDIR int ReadStatus(int channel, unsigned int *status_reg, unsigned int
*arbit_lost_capture_reg, unsigned int *error_code_capture_reg);
    DECLDIR int EnableInterrupts(int channel, unsigned int irqMask);
    DECLDIR int GetErrorCounters(int channel, unsigned int *txErrorCounter, unsigned int
*rxErrorCounter);
    DECLDIR int CloseSession();
}
```

```
typedef enum {
    SUCCESS = 0,
    DRIVER_ERROR,
    INVALID_HANDLE,
    INVALID_PARAMETER,
    XMIT_BUFFER_FULL,
    RCV_BUFFER_EMPTY,
    CAN_BUS_OFF
} ErrorCodes;

typedef enum {
    STANDARD = 0,
    EXTENDED
} FrameFormatBit;

typedef enum {
    DATA = 0,
    REMOTE
} RTRBit;

typedef enum {
    SFF_DATA = 0,
    SFF_RTR,
    EFF_DATA,
    EFF_RTR,
    MAX_MSG_TYPES
} MsgType;

typedef enum {
    DUAL = 0,
    SINGLE
} FilterMode;

typedef enum {
    ALL_IRQ_DISABLE = 0x00,
    RECEIVE_IRQ_ENBL = 0x01,
    TRANSMIT_IRQ_ENBL = 0x02,
    ERR_WARNING_IRQ_ENBL = 0x04,
    DATA_OVERRUN_IRQ_ENBL = 0x08,
    WAKE_UP_IRQ_ENBL = 0x10,
    ERROR_PASSIVE_IRQ_ENBL = 0x20,
    ARB_LOST_IRQ_ENBL = 0x40,
    BUS_ERROR_IRQ_ENBL = 0x80,
    ALL_IRQ_ENABLE = 0xFF
} InterruptMask;

#endif
```

Appendix B: send.cpp

```
//****************************************************************************
//
//     Copyright 2017 by WinSystems Inc.
//
//****************************************************************************
//
//     Name    : send.cpp
//
//     Project : PCMCAN Windows Console Application
//
//     Author  : Paul DeMetrotion
//
//****************************************************************************
//
//       Date        Rev                    Description
//     --------    -------     -----------------------------------------
//     06/23/17      1.0           Original Release
//
//****************************************************************************

#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include "pcmcanDLL.h"

#define MAJOR_VER   1
#define MINOR_VER   0

#define CHANNEL     1
#define BITRATE     500000

#define IRQ         'Y'  // Y or N

// tramsit IDs
#define SFF_ID1     0x400
#define SFF_ID2     0x0ab
#define SFF_ID3     0x123
#define EFF_ID1     0x1f000000
#define EFF_ID2     0x00aabbcc
#define EFF_ID3     0x12345678

#define TBS_BIT     0x04

#if IRQ == 'N'
void waitForXmit(void)
{
    int dllReturn = 0;
    unsigned int status_reg;
    unsigned int arbit_lost_capture_reg;
    unsigned int error_code_capture_reg;

    do
    {
```

```
        dllReturn = ReadStatus(CHANNEL, &status_reg, &arbit_lost_capture_reg,
&error_code_capture_reg);

        if (dllReturn)
        {
            printf("Error reading status\n");
            exit(dllReturn);
        }

        Sleep(1);
    } while (!(status_reg & TBS_BIT));
}
#endif

int _tmain(int argc, _TCHAR* argv[])
{
    int dllReturn = 0;
    unsigned int data1[] = { 0 };
    unsigned int data2[] = { 0xa5, 0xc3 };
    unsigned int data4[] = { 0x01, 0x02, 0x03, 0x04 };
    unsigned int data6[] = { 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF };
    unsigned int data8[] = { 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88 };
    long pckts_sent = 0;

    printf("PCM-CAN Application : send\n");
    printf("Version %d.%d\n\n", MAJOR_VER, MINOR_VER);

    dllReturn = InitializeSession();

    if (dllReturn)
    {
        printf("Error initializing session\n");
        exit(dllReturn);
    }

    dllReturn = StartChannel(CHANNEL, BITRATE);

    if (dllReturn)
    {
        printf("Error starting channel %d\n", CHANNEL);
        exit(dllReturn);
    }

#if IRQ == 'N'
    dllReturn = EnableInterrupts(CHANNEL, ALL_IRQ_ENABLE & ~TRANSMIT_IRQ_ENBL);

    if (dllReturn)
    {
        printf("Error disabling transmit interrupt\n");
        exit(dllReturn);
    }
#else
    dllReturn = EnableInterrupts(CHANNEL, ALL_IRQ_ENABLE);

    if (dllReturn)
    {
        printf("Error enabling all interrupts\n");
        exit(dllReturn);
```

```
    }
#endif

    printf("Sending CAN packets ...\n");

    while (1)
    {
        Sleep(1);

        do {
            dllReturn = WriteDatagram(CHANNEL, EFF_ID1, EFF_DATA, sizeof(data4) /
sizeof(unsigned int), data4);

            if (dllReturn != SUCCESS && dllReturn != XMIT_BUFFER_FULL)
            {
                printf("Error sending message\n");
                CloseSession();
                exit(dllReturn);
            }
        } while (dllReturn);

#if IRQ == 'N'
        waitForXmit();
#endif

        Sleep(1);

        do {
            dllReturn = WriteDatagram(CHANNEL, SFF_ID1, SFF_DATA, sizeof(data8) /
sizeof(unsigned int), data8);

            if (dllReturn != SUCCESS && dllReturn != XMIT_BUFFER_FULL)
            {
                printf("Error sending message\n");
                CloseSession();
                exit(dllReturn);
            }
        } while (dllReturn);

#if IRQ == 'N'
        waitForXmit();
#endif

        Sleep(1);

        do {
            dllReturn = WriteDatagram(CHANNEL, EFF_ID2, EFF_RTR, 0, NULL);


            if (dllReturn != SUCCESS && dllReturn != XMIT_BUFFER_FULL)
            {
                printf("Error sending message\n");
                CloseSession();
                exit(dllReturn);
            }
        } while (dllReturn);

#if IRQ == 'N'
```

```
        waitForXmit();
#endif

        Sleep(1);

        do {
            dllReturn = WriteDatagram(CHANNEL, SFF_ID2, SFF_DATA, sizeof(data6) /
sizeof(unsigned int), data6);

            if (dllReturn != SUCCESS && dllReturn != XMIT_BUFFER_FULL)
            {
                printf("Error sending message\n");
                CloseSession();
                exit(dllReturn);
            }
        } while (dllReturn);

#if IRQ == 'N'
        waitForXmit();
#endif

        Sleep(1);

        do {
            dllReturn = WriteDatagram(CHANNEL, EFF_ID3, EFF_DATA, sizeof(data2) /
sizeof(unsigned int), data2);

            if (dllReturn != SUCCESS && dllReturn != XMIT_BUFFER_FULL)
            {
                printf("Error sending message\n");
                CloseSession();
                exit(dllReturn);
            }
        } while (dllReturn);

#if IRQ == 'N'
        waitForXmit();
#endif

        Sleep(1);

        do {
            dllReturn = WriteDatagram(CHANNEL, SFF_ID3, SFF_RTR, 0, NULL);

            if (dllReturn != SUCCESS && dllReturn != XMIT_BUFFER_FULL)
            {
                printf("Error sending message\n");
                CloseSession();
                exit(dllReturn);
            }
        } while (dllReturn);

#if IRQ == 'N'
        waitForXmit();
#endif

        pckts_sent += 6;
```

```
        printf("Packets sent: %ld\n", pckts_sent);
    }

    CloseSession();

    return dllReturn;
}
```

Appendix C: receive.cpp

```
//****************************************************************************
//
//     Copyright 2017 by WinSystems Inc.
//
//****************************************************************************
//
//     Name    : receive.cpp
//
//     Project : PCMCAN Windows Console Application
//
//     Author  : Paul DeMetrotion
//
//****************************************************************************
//
//       Date        Rev                    Description
//     --------    -------     -----------------------------------------
//     06/23/17      1.0           Original Release
//
//****************************************************************************

#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include "pcmcanDLL.h"

#define MAJOR_VER   1
#define MINOR_VER   0

#define CHANNEL     1
#define BITRATE     500000

#define IRQ         'Y'  // Y or N
#define HWFILTER    0    // 0 or 1 or 2
#define SWFILTER    0    // 0 or 1

// hw filters
#define ACC_CODES1  0x91A00000  // range 0x12340000
#define ACC_MASKS1  0x0007FFFC  //    to 0x1234ffff
#define ACC_CODES2  0x8000c000  // range 0x10000000
#define ACC_MASKS2  0x3FFF1FFF  //    to 0x1bffffff

// sw filters
#define SWF_LOW_ID  0x00001234
#define SWF_HI_ID   0x00005678

#define RBS_BIT     0x01

#if IRQ == 'N'
void waitForMsg(void)
{
    int dllReturn = 0;
    unsigned int status_reg;
    unsigned int arbit_lost_capture_reg;
    unsigned int error_code_capture_reg;
```

```
    do
    {
        dllReturn = ReadStatus(CHANNEL, &status_reg, &arbit_lost_capture_reg,
&error_code_capture_reg);

        if (dllReturn)
        {
            printf("Error reading status\n");
            exit(4);
        }

        Sleep(1);
    } while (!(status_reg & RBS_BIT));
}
#endif

int _tmain(int argc, _TCHAR* argv[])
{
    int dllReturn = 0;
    unsigned long pckts_rcvd = 0;
    unsigned long msgID = 0;
    int msgLen = 0;
    unsigned int msgType = 0, data[8] = { 0 };

    printf("PCM-CAN Application : receive\n");
    printf("Version %d.%d\n\n", MAJOR_VER, MINOR_VER);

    dllReturn = InitializeSession();

    if (dllReturn)
    {
        printf("Error initializing session\n");
        exit(dllReturn);
    }

    dllReturn = StartChannel(CHANNEL, BITRATE);

    if (dllReturn)
    {
        printf("Error starting channel\n");
        exit(dllReturn);
    }

#if IRQ == 'N'
    dllReturn = EnableInterrupts(CHANNEL, ALL_IRQ_ENABLE & ~RECEIVE_IRQ_ENBL);

    if (dllReturn)
    {
        printf("Error disabling transmit interrupt\n");
        exit(dllReturn);
    }
#else
    dllReturn = EnableInterrupts(CHANNEL, ALL_IRQ_ENABLE);

    if (dllReturn)
    {
        printf("Error enabling all interrupts\n");
```

```
        exit(dllReturn);
    }
#endif

#if HWFILTER == 1
    dllReturn = SetHwFilter(CHANNEL, SINGLE, ACC_CODES1, ACC_MASKS1);

    if (dllReturn)
    {
        printf("Error adding hw filter\n");
        exit(dllReturn);
    }
#elif HWFILTER == 2
    dllReturn = SetHwFilter(CHANNEL, DUAL, ACC_CODES2, ACC_MASKS2);

    if (dllReturn)
    {
        printf("Error adding hw filters\n");
        exit(dllReturn);
    }
#else
    dllReturn = ClearHwFilter(CHANNEL);

    if (dllReturn)
    {
        printf("Error clearing hw filter\n");
        exit(dllReturn);
    }
#endif

#if SWFILTER == 1
    dllReturn = SetSwFilter(CHANNEL, EFF_DATA, SWF_LOW_ID, SWF_HI_ID);

    if (dllReturn)
    {
        printf("Error adding sw filter\n");
        exit(dllReturn);
    }
#else
    dllReturn = ClearSwFilter(CHANNEL);

    if (dllReturn)
    {
        printf("Error clearing sw filter\n");
        exit(dllReturn);
    }
#endif

    while (1)
    {
#if IRQ == 'N'
        waitForMsg();

        dllReturn = ReadDatagram(CHANNEL, &msgID, &msgType, &msgLen, data);
#else

        do
        {
```

```
            dllReturn = ReadDatagram(CHANNEL, &msgID, &msgType, &msgLen, data);

            if (dllReturn != SUCCESS && dllReturn != RCV_BUFFER_EMPTY)
            {
                printf("Read error\n");
                exit(dllReturn);
            }

            Sleep(1);
        } while (dllReturn != 0); // 0 indicates message has been read

#endif

        if (msgType == SFF_DATA)
        {
            printf("Received standard data message:\n");
            printf("  ID      = 0x%03x\n", msgID);
            printf("  Length  = %d\n", msgLen);
            for (int i = 0; i < msgLen; i++)
                printf("  Data[%d] = 0x%02x\n", i, data[i]);
            printf("\n");
        }
        else if (msgType == EFF_DATA)
        {
            printf("Received extended data message:\n");
            printf("  ID      = 0x%08x\n", msgID);
            printf("  Length  = %d\n", msgLen);
            for (int i = 0; i < msgLen; i++)
                printf("  Data[i] = 0x%02x\n", data[i]);
            printf("\n");
        }
        else if (msgType == SFF_RTR)
        {
            printf("Received standard remote message:\n");
            printf("  ID      = 0x%03x\n", msgID);
            printf("\n");
        }
        else if (msgType == EFF_RTR)
        {
            printf("Received extended remote message:\n");
            printf("  ID      = 0x%08x\n", msgID);
            printf("\n");
        }

        pckts_rcvd++;
        printf("Packets received : %d\n\n", pckts_rcvd);
    }

    CloseSession();

    return 0;
}
```