Contemporary Control
Systems, Inc.

CANISA/CAN104 Driver

Driver for DOS version 1.1

Table of Contents

This page intentionally left blank

Overview

Contemporary Controls CAN Driver

CAN Driver Overview

The Contemporary Controls CAN Driver (CCCD) provides a software interface for sending and receiving messages over the Controller Area Network (CAN). It includes functions for device initialization, sending and receiving CAN packets, and for reporting device status. It operates in either interrupt or polled mode.

The driver supports the Contemporary Controls PC104XX (for the PC104 Bus), and PCXXX (for the PC ISA bus). Multiple boards may be used simultaneously by a single application. These boards use the Phillips SJA1000 CAN controller chip which is initialized to operate in PeliCAN ™ mode. PeliCAN mode is compatible with the CAN 2.0B specification and includes provisions for extended frames, long identifiers, and improved status reporting.

A single function call is used to initialize a board. This function is accepts the interrupt number, I/O address, baudrate, the received message filter, and the size of the send and receive buffers. Once initialized the application can send messages at will. Incoming messages that pass the receive message filter are stored in a buffer for retrieval by the application.

The driver does not implement any specific protocol. It can be used, however, to send packets over any of the popular CAN networks (DeviceNet, etc.). The included CANTALK application, for instance, uses DeviceNet Group 3 messages to communicate between nodes on a DeviceNet network.

Provided Files

Along with the driver, a sample application is also provided in both source code and compiled forms. The table below lists the provided files.

File	Description
CANTALK.EXE	DOS sample driver application that allows ASCII messages to be sent/received between two or more computers.
CAN_DRIV.H	Include file that defines how to interface to the driver.
CAN_DRIV.C	Driver source code.
PCTIMER.H,C	Timer support function for CAN_DRIV.C.
CANTALK.CPP	C++ source code file for CANTALK.EXE sample application.
COMPORT.H,C SCRN_FCN.H,C SCREEN.HPP,CPP	Screen display and support functions for CANTALK.CPP.

Installation

The only installation required is to copy CAN_DRIV.C, CAN_DRIV.H, PCTIMER.C, and PCTIMER.H to the directory in which they will be used. You will include these files into your C or C++ project.

History

The driver history is listed in the table below:

Item	Version	Date	Comments
CAN_DRIV.H,C	1.0	5.23.00	Initial Release.
Manual	1.0	5.23.00	Initial Release

References

[1] SJA1000 Data Sheet, Phillips Corporation 1997

[2] AN97076 Application Note: SJA1000 Stand-alone CAN controller; Phillips Corporation.

[3] CAN Bus Specification; Robert BOSCH GmbH.

Driver Operation

The purpose of the driver is to provide a means to allow DOS applications to send and receive CAN packets. Best performance is achieved when the driver is operated in interrupt mode; both incoming and outgoing packets are automatically buffered.

There are only three functions necessary to send and receive packets; 'canInit()', 'canSend()', and 'canRecv()'. The 'canInit()' function takes care of initializing the SJA100. It returns a pointer to an initialized 'canObject' structure that is passed to all other functions.

Both 'canSend()' and 'canRecv()' operate on 'canBuffer' structures. These structures are used to store header and packet data.

The 'canSend()' function is used to send packets out on the CAN network. This function buffers packets automatically, so you can send a series of packets without waiting for each one to be sent.

The 'canRecv()' function returns TRUE if a message has been received and data has been transferred to the 'canBuffer' structure.

CAN Bus Considerations

Although the driver tries to abstract the network as much as possible, some understanding of CAN must be attained in order to effectively utilize the driver.

Packets sent over a CAN network have an 'identifier' field. Each node examines the identifier field of each packet that is sent on the network. A node will only accept a packet if the identifier field matches the criteria programmed into registers within the CAN chip. The 'acceptance mask' register contains a value which identifies which bits of the identifier field are relevant. The 'acceptance code' register contains a value which identifies whether the relevant bits should be '1' or '0'. Only those packets whose identifier field, after masking and ANDing, matches the acceptance code will be accepted and stored within the CAN chip.

The original CAN specification (Part A) had an 11 bit identifier field and 8 bit acceptance mask and code registers. The acceptance mask and code applied to the top 8 bits of the identifier which meant that of the 2048 possible identifier values, you could only mask for 256 of them; the bottom 3 bits didn't matter.

The SJA1000 CAN chip from Phillips implements the Part B CAN specification which allows extended frames with 29 bit identifiers. When sending standard frames, the SJA1000 allows you to have dual identifier masks and codes, you can discriminate among all 11 bits of the standard identifier, and you can even extend the identifier process into the first two data bytes within a packet.

DeviceNet, for instance, uses only standard frames. The included 'cantalk.exe' program uses standard frames designed to look like 'Group 3' DeviceNet packets, so that it can peacefully co-exist on a DeviceNet network. The boards are set up, however, to use an identifier which includes the first data byte in the identifier process. The first data byte in a cantalk packet is used to specify the destination node. Since this data byte is programmed to become part of the identifier field, only those Group 3 packets with matching MAC ID's are accepted by a node. This means that there can be a maximum of 7 data bytes transferred in one packet (because the first data byte has become part of the identifier).

Programming the identifier masks and codes is probably the hardest part of using the SJA100 in PeliCan ™ mode. This CAN driver is capable of handling standard and extended frames with single and dual identifiers. Refer to the CANTALK source code for an example on how to set up the identifier masks and codes.

Driver Data Structures

canObject

```
//
// CAN object structure.
//
typedef struct canObject
{
        char    inService;                      // Set when initialized.
        char    irqUsed;                        // Set when using interrupts.
        char    irq;                            // IRQ number (0-15).
        short   port;                           // Base address of SJA1000.
        short   oldImrMask;                     // Original PIC IMR mask.
        #ifdef __cplusplus                      // Original interrupt handler.
        void (interrupt far *oldHandler)( ... );
        #else
        void (interrupt far *oldHandler)();
        #endif
        uchar   modeReg;                        // Mode control bits.
                                                //
        char    txRunning;                      // Transmit operation in process.
        uchar   errorFlags;                     // Accumulated error bits.
        ushort  errorCnt;                       // Number of error ints.
        ushort  errorIdx;                       // Error log index.
        uchar   errorLog[MAX_ERROR];            // Error log.
        char    rxOverrun;                      // Set if overrun.
        char    timeTrig;                       // Set if next receive int to be 'zero' time.
        long    timeZero;                       // 'Zero point' for time stamps.
        //
        ushort  rxBufSize;                      // Max number of receive 'canBuffer'.
        ushort  rxCount;                        // Number of 'canBuffer' in rx buffer.
        ushort  rxIn;                           // Index for putting messages in rx buffer.
        ushort  rxOut;                          // Index for taking messages out of rx buffer.
        struct  rawCanBuffer huge       *rxBuf;// Pointer to receive buffer.
        //
        ushort  txBufSize;                      // Max number of transmit 'canBuffer'.
        ushort  txCount;                        // Number of 'canBuffer' in tx buffer.
        ushort  txIn;                           // Index for putting messages in tx buffer.
        ushort  txOut;                          // Index for taking messages out of tx buffer.
        struct  rawCanBuffer huge       *txBuf; // Pointer to transmit buffer.
};
```

A ‘canObject’ structure is used for each CAN board in a system. Each board is identified by a pointer it’s ‘canObject’ structure, and this pointer is passed as the first parameter to all CAN board functions. This structure is created and initialized by the ‘canInit()’ function.

This structure stores information used by the other CAN board functions.

canFilter

```
//
// Structure used to set acceptance filters.
//
typedef struct canFilter
{
        char    type;           // Type of filter; STD_SNGL, etc.
                                //
                                // Filter 1.
                                //
        long    idMask1;        // Identifier mask.
        long    idCode1;        // Identifier matching value.
        char    rtrMask1;       // RTR bit mask(s).
        char    rtrCode1;       // RTR bit value(s).
                                //
                                // Filter 2.
                                //
        long    idMask2;        // Identifier mask.
        long    idCode2;        // Identifier matching value.
        char    rtrMask2;       // RTR bit mask(s).
        char    rtrCode2;       // RTR bit value(s).
                                //
                                // Data Filter.
                                //
        uchar   dataMask[2];    // Data byte(s) mask.
        uchar   dataCode[2];    // Data byte(s) value.
};
```

The ‘canFilter’ structure is used to define the type of filter to be used for received packets. There are four types of filters that can be used with the SJA1000 chip. These are shown as #define statements in ‘can_driv.h’:

STD_SNGL	Standard frame, single identifier: Filtering on one 11 bit identifier, RTR bit, and two data bytes.
STD_DUAL	Standard frame, dual identifier: Filtering on two 11 bit identifiers, two RTR bits, and one data byte for each filter.
EXT_SNGL	Extended frame, single identifier: Filtering on one 29 bit identifier, one RTR bit.
EXT_DUAL	Extended frame, dual identifier: Filtering on two 16 bit identifiers only (no RTR bits).

The ‘canFilter’ structure can define all four filters. Defining filters can be one of the most confusing tasks in dealing with the CAN network. A couple important points to remember are:

1. The mask bit values are ‘0’ when you want the bit to match, and ‘1’ when you don’t care about the bit status.

2. Even if you have a STD_SNGL filter like those used with DeviceNet, the chip will use the first two data bytes in the packet for filtering. If you don’t intend to use the data bytes for filtering, make sure the mask bit values are set to ‘1’ (set both dataMask[0] and dataMask[1] to 0xff).

canBuffer

```
//
// Structure used for receive and transmit operations.
//
typedef struct canBuffer
{
        ulong   time;           // Time stamp returned by RX service.
        char    type;           // 0 = Standard Frame, 1 = Extended Frame.
        long    id;             // Identifier: 11 or 29 bits.
        char    rtr;            // Remote Transmission Request bit.
        char    cnt;            // Number of bytes in message.
        uchar   data[8];        // RX/TX data.
};
```

The ‘canBuffer’ structure is used for both sending and receiving packets.

When sending, you assign the type, id, rtr, cnt, and data bytes. You do not assign a value to ‘time’. A pointer to the ‘canBuffer’ you have just filled out is passed to ‘canSend()’, which unloads the data and prepares a packet to be sent.

When receiving, you pass a pointer to a ‘canBuffer’ to ‘canRecv()’. If a packet is waiting the ‘canBuffer’ will be filled out with data from the received packet. All fields will be filled by ‘canRecv()’, including the time stamp (see ‘canTime()’ function and Timer Module section).

Driver Functions

canInit()

```
                              // Initializes CAN chip and data structure(s).
                              // Returns pointer to 'canObject' if success.
                              // Returns NULL pointer if failure.
                              //
struct canObject *canInit     ( short irq ,
                                ushort ioAddress ,
                                short baudrate ,
                                struct canFilter *filter,
                                short rxBufSize ,
                                short txBufSize );
```

This function creates and initializes a 'canObject' structure, initializes the CAN hardware, and prepares the CAN board for sending and receiving. It is passed the PC interrupt number (0-15), the base PC I/O address of the CAN card, the baudrate, a pointer to an initialized 'canFilter' structure, and the number of packets that can be held in the receive and transmit buffers. Memory is allocated for the 'canObject' and the packet buffers, the 'canObject' data elements are initialized to default values, and the hardware is initialized.

Note that an interrupt number of '0' will place the CAN driver in polled mode. In this mode, transmitted packets will not be buffered, and the 'canSend()' function will wait until data can be loaded into the CAN chip. Received packets, if not retrieved from the CAN chip before its buffer overflows, will be lost.

Valid baudrates are:

```
//
// CAN baud rates.
// Passed as parameter to 'canSetBaud()'.
//
#define             BAUD_125KB      1
#define             BAUD_250KB      2
#define             BAUD_500KB      3
```

If successful, the function returns a pointer to the initialized 'canObject'. You will use this pointer as the first parameter to all other CAN driver functions. It returns a NULL pointer if memory can't be allocated or if the hardware cannot be initialized. If a null pointer is returned, examine the global variable 'canInitStatus' for the cause of failure:

```
//
// Initialization failure codes.
// If 'canInit()' fails, one of these will be in 'canInitStatus'.
//
#define             INIT_OK         0       // No error.
#define             INIT_NOCHIP     1       // Can't find/reset chip.
#define             INIT_BADIRQ     2       // Bad irq number.
#define             INIT_MALLOC_CP  3       // Can't allocate memory for object.
#define             INIT_MALLOC_TX  4       // Can't allocate memory for tx buffer.
#define             INIT_MALLOC_RX  5       // Can't allocate memory for rx buffer.
```

canSend()

```
                        // Transmits a message.
                        // Returns SUCCESS, or FAILURE if buffer full.
                        //
short   canSend         ( struct canObject *cp , struct canBuffer *txBuf );
```

This function transmits a packet on the CAN network. You pass a pointer to an initialized ‘canObject’ and a pointer to an initialized ‘canBuffer’. The data in the ‘canBuffer’ is either sent directly to the CAN chip or is placed in the transmit buffer if the CAN chip is busy transmitting another packet.

It returns SUCCESS if the packet can be queued for transmit; FAILURE if the transmit buffer is full.

Note that SUCCESS does not mean that the packet has been sent. You can look at the value of ‘canObject.txCount’ to determine if all packets in the transmit buffer have been loaded into the CAN chip for transmission; this data value will be ‘0’ if no packets remain in the transmit buffer. You must then use the ‘canStatus()’ function and look at the STAT_TXCOMPLETE bit to determine if the last packet has been successfully transmitted.

canRecv()

```
                        // Retrieves a packet; fills 'rxBuf' with data.
                        // Returns SUCCESS if packet waiting and data transferred
                        // to 'rxBuf'.
                        // Returns FAILURE if no packet available.
                        //
short   canRecv         ( struct canObject *cp , struct canBuffer *rxBuf );
```

This function retrieves a packet from the receive buffer if one is available. You pass a pointer to an initialized ‘canObject’ and a pointer to an empty ‘canBuffer’. If a packet is waiting in the receive buffer (or within the CAN chip in polled mode), data will be transferred to the ‘canBuffer’ structure.

The value of the time stamp is described in the ‘canTime()’ function and the Timer Module section below.

It returns SUCCESS if a packet was available; FAILURE if no packet was waiting.

canDestroy()

```
                        // Disables CAN chip, releases allocated memory,
                        // and unhooks interrupt.
                        //
void    canDestroy      ( struct canObject *cp );
```

This function disables the CAN hardware, releases memory allocated for the ‘canObect’, releases memory allocated for the receive and transmit buffers, and unhooks the interrupt used by the CAN board. It should be called for each ‘canObject’ before your program exits.

canReset()

```
                        // Puts CAN chip in reset mode
                        // Returns SUCCESS or FAILURE if chip can't be reset.
                        //
short   canReset        ( struct canObject *cp );
```

This function puts the CAN chip in reset mode, effectively disconnecting it from the CAN network. Use this function to remove the chip from the network.

It returns SUCCESS if chip can be reset; FAILURE if chip can't be put into reset mode.

canEnable()

```
                        // Brings CAN chip out of reset mode.
                        // Returns SUCCESS or FAILURE if chip can't be brought
                        // out of reset.
                        //
short   canEnable       ( struct canObject *cp );
```

This function brings the CAN chip back online after 'canReset()' has been called.

It returns SUCCESS if the chip can be brought out of reset mode; FAILURE otherwise.

canSetBaud()

```
                        // Sets baud rate.
                        // Returns SUCCESS or FAILURE if invalid baud rate.
                        //
short   canSetBaud      ( struct canObject *cp , short baudrate );
```

This function is used to change the baud rate of the CAN chip. It puts the chip into reset, changes the baud rate, and the brings the chip out of reset. Refer to the BAUD_ #defines in CAN_DRIV.H for valid baud rates:

```
//
// CAN baud rates.
// Passed as parameter to 'canSetBaud()'.
//
#define             BAUD_125KB      1
#define             BAUD_250KB      2
#define             BAUD_500KB      3
```

It return SUCCESS if the baud rate can be changed; FAILURE otherwise.

canSetFilter()

```
                        // Sets acceptance filter for received packets.
                        // Returns SUCCESS or FAILURE if invalid filter type.
                        //
short   canSetFilter    ( struct canObject *cp , struct canFilter *filter );
```

This function is used to change the acceptance filter for received packets. You pass it a pointer to an initialized 'canFilter' structure containing the new filter. This data is transferred to the CAN chip. The CAN chip is put into reset mode, the new filter is loaded, and the chip is brought out of reset mode.

It returns SUCCESS if the new filter can be loaded; FAILURE if invalid filter type.

canStatus()

```
                        // Returns status informaton.
                        // Refer to STAT_XXX #defines above.
                        //
ushort  canStatus       ( struct canObject *cp );
```

Returns the status of the CAN chip. Refer to the STAT_ #defines in CAN_DRIV.H for the meaning of each bit in the returned value:

```
//
// CAN status bits.
// Bitfield definitions for value returned by 'canStatus()'.
//
#define             STAT_RXRDY      0x01    // Message waiting in RX buffer.
#define             STAT_OVERRUN    0x02    // Receive data overrun.
#define             STAT_TXRDY      0x04    // TX buffer will accept packet.
#define             STAT_TXCOMPLETE 0x08    // TX buffer empty and TX complete.
#define             STAT_ERROR      0x10    // Chip reports error condition.
#define             STAT_BUSOFF     0x20    // Chip is in bus-off condition.
```

canError()

```
                        // Returns, and then clears, accumulated error status
                        // bits.
                        //
ushort  canError        ( struct canObject *cp );
```

Returns, and then sets to '0', the value of 'canObject.errorFlags'. This structure member is the accumulated 'OR' value of the error interrupt bits, excluding RI_BIT and TI_BIT:

```
//
// CAN Interrupt bitmasks; used for interrupt enable & status.
//
#define             RI_BIT          0x01            // Receive.
#define             TI_BIT          0x02            // Transmit.
#define             EI_BIT          0x04            // Error warning.
#define             DOI_BIT         0x08            // Data overrun.
#define             WUI_BIT         0x10            // Wake-up.
#define             EPI_BIT         0x20            // Error Passive.
#define             ALI_BIT         0x40            // Arbitration lost.
#define             BEI_BIT         0x80            // Bus error.
```

canTxPurge()

```
                        // Discard any packets in tx buffer waiting to be sent.
                        //
void    canTxPurge      ( struct canObject *cp );
```

This function discards any transmit packets waiting to be sent. It does not stop a transmission in progress or disable the transmit function on the CAN chip.

canRxPurge()

```
                        // Discard any packets in rx buffer waiting to be read.
                        //
void    canRxPurge      ( struct canObject *cp );
```

This function discards any packets in the receive buffer. It does not disable the receive function on the CAN chip.

canRecover()

```
                        // Recovers from bus-off (or other error condition).
                        // Resets the chip, purges tx & rx packets, and takes the
                        // chip out of reset.
                        //
void    canRecover      ( struct canObject *cp );
```

This function is used to recover from a fatal system error. It disables all CAN interrupts, puts the CAN chip in reset mode, purges all transmit and receive packets, aborts any transmit operation in progress, takes the CAN chip out of reset, and then re-enables interrupts.

Use this function to recover from a bus-off condition or other serious network error.

canTime()

```
                        // Performs timer functions needed for time stamp.
                        // 1 = Reset time stamp to '0'.
                        // 2 = Reset time stamp to '0' when next packet
                        //     is received.
                        // See TIME_XXX #defines above.
                        //
void    canTime         ( struct canObject *cp , short funtion );
```

This function is used to modify the time stamp value returned in the ‘time’ field of the ‘canBuffer’ structure by the receive function. The value passed as the ‘function’ argument determines what action is taken:

```
//
// Time function codes.
// These are used by 'canTime()' to determine which function to perform.
//
#define     TIME_RESET      0x01    // Resets time stamp to zero.
                                    // Used to define a starting point for time
                                    // stamps.
#define     TIME_TRIG       0x02    // Sets trigger so that next received packet
                                    // gets a time stamp of '0'.
                                    // Used to start a sequence of time stamps
                                    // starting from '0'.
```

When a receive interrupt occurs, the value of ‘canBuffer.time’ is loaded with a time stamp. The value of the time stamp is computed from the difference between the global variable ‘tmrTics’ (from the timer module) and ‘canObject.timeZero’.

The TIME_RESET function loads the current value of ‘tmrTics’ into ‘canObject.timeZero’ so that subsequent time stamps will be relative to when this function was called (the current time).

The TIME_TRIG function sets the ‘canObject.timeTrig’ flag so that during the next receive interrupt the value of ‘canObject.timeZero’ will be loaded with ‘tmrTics’. This will cause the next received packet to have a time stamp of ‘0’, and all subsequent packets will have time stamps reflecting a value relative to that next receive interrupt.

Note that the time stamp function will not be active unless the system timer interrupt is hooked by ‘tmrHook()’.

Refer to the section regarding the Timer Module for hooking the timer interrupt.

Timer Module

The timer module is defined by PCTIMER.H and PCTIMER.C. The features in this module are used by the can driver functions to establish and record time stamps for received packets. The timer module allows you to increase the frequency of the system timer interrupt to achieve much higher timing resolution than normally possible.

tmrHook()

In order for the time stamp function to operate, you must first “hook” the system timer interrupt routine. This is done by the ‘tmrHook()’ function:

```
//
// This routine installs the timer object system timer tic handler.  It is
// installed at the "top of the chain" of any other system timer tic
// handlers (for PC timer 0).  It can increase the frequency of system
// timer tics.  It calls the previous timer tic handler on (approximately)
// the same rate as the PC standard (18.2 tics/second).
// Value 'newMult' specifies new timer operating rate as an integer
// multiple of the basic PC rate of 18.2 tics/second.
// Sample values and their associated time periods:
//
//  1:      18.2 tics/second        54.925 mS/tic (PC standard)
//  2:      36.4 tics/second        27.463 mS/tic
//  55:     1001 tics/second         998 uS/tic
//  110:    2002 tics/second        499 uS/tic
//
void tmrHook( int newMult );
```

After calling ‘tmrHook()’ with an appropriate clock divisor value, the system timer tic is redirected to the timer module. Each timer interrupt, the global variable ‘tmrTics’ is incremented. This value is used by the receive interrupt routine to establish time stamps for received packets.

The timer module also allows you to implement general purpose high resolution timers for other purposes in your program. Refer to PCTIMER.H for a description of the available timer functions.

tmrUnhook()

Before your application returns to DOS, it will be necessary to “unhook” the system timer interrupt using the ‘tmrUnhook()’ function:

```
// This routine MUST be called when program terminates (if the timer interrupt
// is hooked) to remove the timer tic interrupt handler from the "chain".
// Interrupt handler goes away when program terminates, and if the interrupt
// vector is still set to our (now nonexistent) routine the PC goes off into
// the weeds.
//
void tmrUnhook( void );
```

Failure to call ‘tmrUnhook()’ will guarantee a system crash.