CANPC527D Isolated 1 Mb/s Full CAN Interface Board User,s Manual

KASKOD

1999

Sankt-Peteterburg

189625, S-Petrsburg, Pavlovsk,
Filtrovskoy road, 3
tel.: (812) 466-5784, 476-0795
Fax: (812) 465-3519
E-mail : cascod@online.ru
kaskod@spb.cityline.ru

http://www.kaskod.ru

Table of Contents Page

INTRODUCTION

This user‘s manual describes the operation of the CANPC527D Can-bus interface board.

Some of the key properties of the CANPC527D include:

- 1 Mb/s maximum datarate (fully programmable)
- Full CAN-functionally 2.0 B
- Intel 82527 CAN-bus controller
- Galvanically isolated physical interface with CAN transceiver
- +5V only operation

The following paragraphs briefly describe the major features of the CANPC527D. A more detailed discussion is included in Chapter 3 (Hardware description) and in Chapter 4 (Boad operation and programming). The boad set-up is described in Chapter 1 (Board Setting). A full description of the Intel 82527 CAN-controller is included in Chapter 4.

CAN-bus controller

The CANPC527D CAN-bus interface is implemented using the Intel 82527 chip. This controller supports CAN Specification 2.0B. This versatile chip supports standard and extended Data and remote frames as follows: A Programmable Global Message Identifier Mask; 15 message objects of 8-byte Data Length and a Programmable Bit Rate. This fully integrated chip supports all the functinality of the CAN-bus protocol.

Physical Interface

Industrial environments require galvanic isolation and bus filtering to provide reliable data communication and safety. CANPC527D has option for the physical interface.

The galvanically isolated physical interface is implemented using high speed optocouplers and a DC/DC converter. To protect the input from radiated bus noise a special balanced bus filter is used. This filter is designed to meet EMI requirements.

Mechanical description

The CANPC527D is designed on a ISA form factor. An easy mechanical interface to both PC/104 and EUROCARD systems can be achieved. PUT your CANPC527D directly on a ISA compatible computer using the onboard mounting holes.

Connector description

For the switching on two identical connectors are used P1 (channel 1) and P2 (channel 2).

Pinout is as follows:

TABLE 1-1

contact	*signal*
1	nc
2	BUS_L
3	GND
4	nc
5	nc
6	nc
7	BUS_H
8	nc
9	VCC

Explanation:

nc	–	not connected (may be used anyhow)
BUS_L	–	bus active low (relatevely GND)
BUS_H	–	bus active high (relatevely GND)
GND	–	may be not used
VCC	–	power supply voltage +5V (in this version is not used)

Terminal Resistors

CAN Standard implies terminal resistors at the end-nodes of the net. If in your specific configuration channel 1 is the end-node, then setting of J2 is needed, if it is channel 2 then J3.

What commes with your board

You will receive the following items in youe CANPC527D package:

- CANPC527D CAN-bus interface module
- Software diskette with C source code for CAN bus interfacing for DOS
- User‘s manual

Board accessories

In addition to the items included in your CANPC527D delivery, several software and hardware accessories are available. Call your distributor for more information on these accessories and for help in choosing the best items to support your distributed control system.

- Application software
  - Third party high level protocol drivers
  - WIN 95/98 and WIN NT drivers and driver source code

CHAPTER 1 - BOARD SETTING

The CANPC527D CAN bus interface board has jumper settings which can be changed to suit your applicacation and host computer memory configuration. The factory setting are listed and shown in the diagram in the beginning of this chapter.

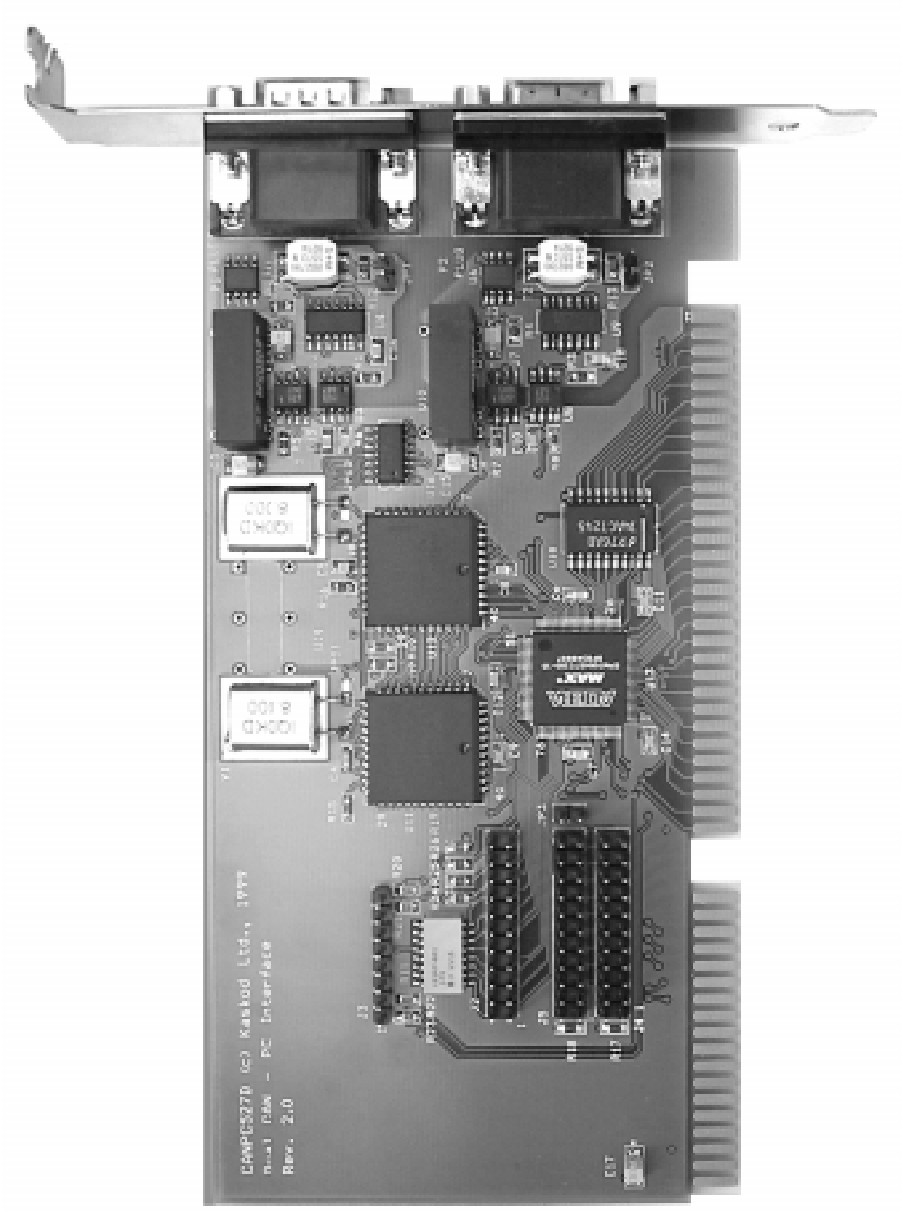

Fig.1

CANPC527D contains 2 identical optoisolated CAN 2.0B specification channels. First channel registers begin from base address, the second - base address+100h. Each channel can generate its interrupt request with its interrupt vector, or both channels can use one vector. In this case interrupt request program must process both chanels simultaneously.

Base Address Jumpers

For base address setting **J2** is used.
J2 consists of 11 sections, and each section is associated with its address line.
Note: Section 0 (connectors 1-2) is reserved for S version.

Sections are arranged as following:

TABLE 1-2

Section 0 (connectors 1-2)	–	reserved
Section 1 (connectors 3-4)	–	A9
Section 2 (connectors 5-6	–	A10
Section 3 (connectors 7-8	–	A11
Section 4 (connectors 9-10)	–	A12
Section 5 (connectors 11-12)	–	A13
Section 6 (connectors 13-14)	–	A14
Section 7 (connectors 15-16)	–	A15
Section 8 (connectors 17-18)	–	A16
Section 9 (connectors 19-20)	–	A17
Section 10 (connectors 21-22)	–	A18

Thus base address possible range setting is: 080000h-0FFE00h.
Explanation: Board starts working when address line signal A19 is set high. Connection corresponds to high level.

Examples:

To set base address 0C8000h - connect 21 and 22 in Section10, 15 and 16 in Section 7 (FACTORY SETTING).
To set base address 0D1E00h -

connect 21 and 22 in Section 10,	17 and 18 in Section 8
connect 1 and 2 in Section 1	3 and 4 in Section 2
connect 5 and 6 in Section 3,	7 and 8 in Section 4

Interrupt request number setting

J4, J5 is used for the selection of interrupt request.

J4, J5 consists of 11 sections.

Followin interrupts are available

Channel number	*J4*	*J5*
resistor insert	1-2	1-2
15	3-4	3-4
14	5-6	5-6
12	7-8	7-8
11	9-10	9-10
10	11-12	11-12
7	13-14	13-14
6	15-16	15-16
5	17-18	17-18
4	19-20	19-20
3	21-22	21-22

Other setting of jumpers switch off the interrupt requests.

CHAPTER 2 - BOARD INSTALLATION

Board Installation

Keep your board in its antistatic bag until you are ready to install it to your system! When removing it from the bag, hold the board at the edges and do not touch the components or connectors. Please handle the board in an antistatic environment and use a rounded workbench for testing and handling of your hardware.

Before installing the board in your computer, check the jumper setting. Chapter 1 reviews the factory settings and how to change them. If you need to change any setting, refer to the appropriate instructions in Chapter 1. Note that incompatible jumper settings can result in unpredictable board operation and erratic response.

General installation guidelines:

1. Turn OFF the power to your computer and all devices connected to CANPC527D

2. Touch the grounded metal housing of your computer to discharge any antistatic buildup and then remove the board from its antistatic bag.

3. Hold the board by the edge and install it in an enclosure or place it on the table on an antistatic surface.

4. Connect the board to the CAN fieldbus interface header connectors. Make sure the polarity of the cable is correct. Both connectors are identical, one of these headers may be used to bring the CAN bus to the one channel; the other connector interfaces for second channel.

CHAPTER 3 - HARDWARE DESCRIPTION

Chapter 3 - Hardware discription describes the major features of the CANPC527D: the Intel 82527 CAN bus controller, Galvanic isolation of the CAN-bus.

Figure 3-1 shows the general block diagram of the CANPC527D. This chapteer describes the major features of the CANPC527D: the Intel 82527 CAN-bus controller, Galvanic isolation of the CAN-bus, the Fiberoptic interface, the Onboard configuration EEPROM, and Digital I/O.

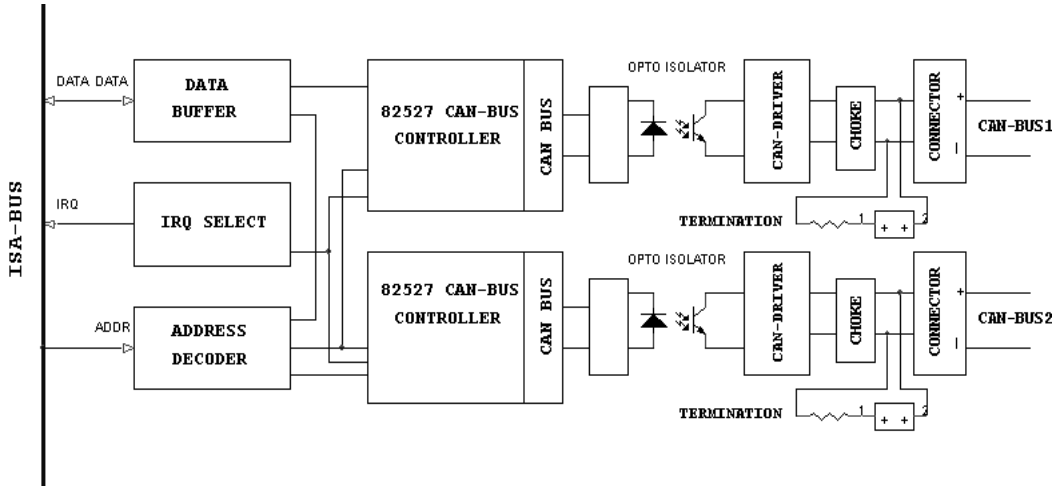

Fig. 3-1. CANPC527D Block diagram

82527 CAN bus controller

Reference note:

(Intel publication CAN Architectural Overview, Automotive Products Databook)

The 82527 CAN controller consists of six functional block. The CPU interface logic manages the communication to the host computer. The CAN controller interface to the CAN bus and implements the protocol rules of the CAN protocol for the transmission and reception of messages. The RAM is the physical interface layer between the host CPU and the CAN bus. One eight bit I/O port provides low speed I/O capabilities.

The 82527 RAM provides storage for 15 message objects of 8 byte length. Each message object has a unique identifier and can be configured to either transmit of to receive except for the last message object. The last message object is a receive-only buffer with a special mask design to allow selected groups of different message identifiers to be received.

Each message identifier contains control and status bits. A message object with a direction set for receive will send a remote frame by requesting a message transmission. A message set as transmit will be configured to automatically send a data frame whenever a remote frame with a matching identifier is received over the CAN bus. All message objects have separate transmit and receive interrupt and status bits, allowing the CPU full flexibility in detecting when a remote frame has been sent or received.

The 82527 also implements a global masking feature for acceptance filtering. This feature allows the user to globally mask, or “don`t care”, any identifier bits of the incoming message. This mask is programmable to allow the user to design an application specific message identification strategy. There are separate global mask for standard and extended frames.

The incoming message first passes through the global mask and is matched to the identifiers in the message objects 1-14. If there is no identifier match then the message passes through the local mask in message object 15. the local mask allows a large number of infrequent messages to be received by the 82527. Message object 15 is also buffered to allow the CPU time to service a message received.

Galvanic isolation of the CAN-bus

The galvanic isolation of the CANPC527D is implemented using

1) Optocouplers for data transmission and
2) a DC/DC converter to supply power to the CAN bus and the physical interface circuitry.

The high speed optocouplers are directly connected to the 82527. the optocouplers drive CAN bus transceiver. A special balanced CAN bus choke is used not only to improve immunity to bus noise, but also to protect the bus transceiver. This filter assists in conforming to the EN5502 Radiated Emision test requirements.

CHAPTER 4 - BOARD OPERATION AND PROGRAMMING

This chapter shows you how to program and use your CANPC527D. It provides a complete detailed description of the memory map and a detailed discussion of programming operations to aid you in programming. The full functionality of the CANPC527D is described in the datasheet reprint from Intel on the 82527 CAN controller chip.

Defining the Memory Map

The memory map of the CANPC527D occupies 512 bytes of host PC low memory space. This window is freely selectable by the user as described in Chapter 1 (table 1-2). After setting the base address you have access to the internal resources of the 82527 CAN controller chip, as is described in the next sections reprinted from the Intel 82527 manual.

Interrupts

- What is an interrupt?

An interrupt is an event that causes the processor in your computer to temporerily halt its current process and execute another routine. Upon completion of the new routine, control is returned to the original routine at the point where its execution was interrupted.

Interrupts are a very flexible way of dealing with asynchronous events. Keyboard activity is good example; your computer cannot predict when you might press a key and it would be a waste of processor time to do nothing while waiting for a keystroke to occur. Thus the interrupt scheme is used and the processor proceeds with other tasks.When a keystroke occurs, the keyboard “interrupt” the processor, the processor then gets the keyboard data which is then placed into the memory. It then returns to what it was doing before the interrupt occurred. Other common devices that use interrupts are Network boards, A/D boards, serial ports etc.

Your CANPC527D can interrupt the main processor when a message is received or transmitted if interrupts are enabled on the CANPC527D board. By using interrupts you can write powerful code to interface to your CAN network.

- Interrupt request lines

To allow different peripheral devices to generate interrupts on the same computer, the PC AT bus has interrupt request channels (IRQ‘s). A rising edge transition on one of these lines will be latched into the interrupt controller. The interrupt controller checks to see if the interrupts are to be acknowledged from that IRQ and, if another interrupt is being processed, it decides if the new request should supercede the one in progress or if it has to wait until the one in progress is done. The priority level of the interrupt is determined by the number of the IRQ; IRQ0 has the highest priority IRQ15 the lowest. Many of the IRQ‘s are used by the standard system resources. IRQ0 is dedicated for the internal timer, IRQ1 is dedicated to the keyboard input, IRQ3 for serial port COM2 and IRQ4 for serial port COM1. Often interrupt 5 and 7 are free for the user.

- 8259 Programmable Interrupt Controller

The chip responsible for handling interrupt requests in PC is the 8259 Interrupt Controller. To use interrupts you will need to know how to read and set the 8259‘s internal interrupt mask register (IMR) and how to send the end-interrupt (EOI) command to acknowledge the 8259 interrupt controller.

-Interrupt Mask Register (IMR)

Each bit in the interrupt mask register (IMR) contains the mask status of the interrupt line. If a bit is set (equal to 1), then the corresponding IRQ is masked, and it will not generate an interrupt. If a bit is cleared (equal to 0), then the corresponding IRQ is not masked, and it can then generate an interrupt. the interrupt mask register is programmed through **port 21h.**

-End-of-Interrupt (EOI) Command

After an interrupt service routine is complete, the 8259 Interrupt Controller must be acknowledged by **writing the value 20h to port 20h.**

-What exactly happens when an interrupt occurs?

Understanding the sequence of events when an interrupt is triggered is necessary to correctly write interrupt handlers. When an interrupt request line is driven high by a peripheral device (such as the CANPC527D), the interrupt controller checks to see if interrupts are enabled for that IRQ, and then checks to see if other interrupts are active or requested and determines which interrupt has priority. The interrupt controller then interrupts the processor. The current code segment (CS), instruction pointer (IP), and flags are pushed onto the system stack., and a new set of CS and IP are loaded from the lowest 1024 bytes of memory.

This table is referred to as the interrupt vector table and each entry to this table is called an interrupt vector. Once the new CS and IP are loaded from the interrupt vector table, the processor starts to execute code from the new Code Segment (CS) and from the new Instruction Pointer (IP). When the interrupt routine is completed the old CS and IP are popped from the system stack and the program execution continues from the point it was interrupted.

- Using Interrupt in your Program

Adding interrupt support to your program is not as difficult as it may seem, especially when programming under DOS. The following discussion will cover programming under DOS. Note, that even the smallest mistake in your interrupt program may cause the computer to hang up and will only restart after a reboot. This can be frustrating and time-consuming.

- Writing an Interrupt Service Routine (ISR)

The first step in adding interrupts to your software is to write an interrupt service routine (ISR). This is the routine that will be executed automatically each time an interrupt request occurs for the specified IRQ. An ISR is different from other subroutines or procedures. First, on entrance the processor registers must be pushed onto the stack before anything else! Second, just before exiting the routine, you must clear the interrupt on the CANPC527D by writinf to the 82527 CAN controller, and write the EOI command to the interrupt controller. Finally, when exiting the interrupt routine the processor registers must be popped from the system stack and you must execute the IRET assembly instruction. This instruction pops the CS, IP and processor flags from the system stack. These were pushed onto the stack when entering the ISR.

Most compilers allow you to identify a function as an interrupt type and will automatically add these instructions to your ISR with one exception: most compilers do not automatically add the EOI command to the function, you must do it yourself. Other than this and a few exceptions discussed below, you can write your ISR as any code routine. It can call other functions and procedures in your program and it can access global data. If you are writting your first ISR, we recommend you stick to the basics; just something that enables you to veify you have entered the ISR and executed it successfully. For example: set a flag in your ISR and in your main program check for the flag.

Note: If you choose to write your ISR in in-line Assembly, you must push and pop registers correctly, and exit the routine with the IRET instruction instead of the RET instruction.

There are a few precautions you must consider when writing ISR`s. The most important is, do not use any DOS functions or functions that call DOS functions from an interrupt routine. DOS is not reentrant; that is, a DOS function cannot call itself. In typical programming, this will not happen because of the way DOS is written. But what about using interrupts? Then, you could have the situationn such as this in your program. If DOS function X is being executed when an interrupt occurs and the interrupt routine makes a call to the DOS function X, then function X is essentially being called while active. Such cases will cause the computer to crash. DOS does not support such opperation. A general rule is, that do not call any functions that use the screen, read keyboard input and any file I/O routines should not be used in ISR‘s.

The same problem of reentrancy exists for many floating point emulators as well, meaning you should avoid floating point mathematical opperations in your ISR.

> *Note,* that the problem of reentrancy exists, no matter what programming language you use. Even, if you are writing your ISR in Assembly language, DOS and many floating point emulators are not reentrant. Of couse, there are way to avoid this problem, such as those which involve checking if any DOS functions are currently active when your ISR is called, but such solutions are beyond the scope of this manual.

The second major concern when writing ISR‘s is to make them as short as possible in term of execution time. Spending long times in interrupt service routines may mean that other important interrupt are not serviced. Also, if you spend too long in your ISR, it may be called again before you have exited. This will lead to your computer hanging up and will require a reboot.

Your ISR should have the following structure:

- Push any procesor registers used in your ISR. Most C compiler do this automatically
- Put the body of your routine here
- Read interrupt status register of the 82527 chip on your CANPC527D board
- Clear the interrupt bit by writing to the 82527 CAN controller
- Issue the EOI command to the 8259 by writing 20h to address 20h
- Pop all registers. Most C compilers do this automatically

The following C example shows what the shell of your ISR should be like:

```
/*---------------------------------------------------------------------------
| Function: new_IRQ_handler
| Inputs:   Nothing
| Returns:  Nothing              _Sets the interrupt flag for the EVENT.
\---------------------------------------------------------------------------*/
void interrupt far new_IRQ_handler(void)
{
      IRQ_flag = 1                  // Indicate to main process interrupt has occurred
      {
      //Your program code should be here
      }
                                    //Read interrupt status registers
                                    //Clear interrupt
      outp(0x20, 0x20);         /*Acknowledge the interrupt controller.*/
}
```

- Saving the Startup Interrupt Mask Register (IMR) and interrupt vector

The next step after writing the ISR is to save the startup state of the interrupt mask register (IMR) and the original interrupt vector you are using. The IMR is located in address 21h. The interrupt vector you will be using is located in the interrupt vector table which is an array of 4-byte pointers (addresses) and it is locate in the first 1024 bytes of the memory (Segment 0 offset 0). You can read this value directly, but it is a better practice to use DOS function 35h (get interrupt vector) to do this. Most C compilers have a special function available for doing this. The vectors for the hardware interrupts on the XT-bus are vectors 8-15., where IRQ0 uses vector 8 and IRQ7 uses vector 15. Thus if your CANPC527D is using IRQ5 it corresponds to vector number 13.

Before you install your ISR, temporerily mask out the IRQ you will be using. This prevents the IRQ from requesting an interrupt while you are installing and initializing your ISR. To mask the IRQ, read the current IMR at I/O port 21h, and set the bit that corresponds to tout IRQ. the IMR is arranged so that bit 0 is for IRQ0 and bit 7 for IRQ7. See the paragraph entitled Interrupt Mask Register (IMR) earlier in this discussion for help in determining your IRQ‘s bit. After setting the bit, write the new value to I/O port 21h.
with the startup IMR saved and the interrupts temporarily disabled, you can assing the interrupt vector to point to your ISR. Again you can overwrite the appropriate entry in the vector table with a direct memory write, but this is not recommended. Instead use the DOS function 25h (Set Interrupt Vector) or, if your compiler provides it, the library routine for setting up interrupt vectors. Remember, that interrupt vector 8 corresponds to IRQ0, vector 9 for IRQ1 etc.

If you need to program the source of your interrupts, do that next. For example, if you are using transmitted or received messages as an interrupt source, program it to do that.

Finally, clear the mask bit for your IRQ in the IMR. This will enable your IRQ.

- Common Interrupt mistakes

- Remember, hardware interrupt are from 8-15, XT IRQ‘s are numbered 0-7

- Forgetting to clear the IRQ mask bit in the IMR
- Forgetting to send the EOI command after ISR code. Disables further interrupts.

Example on Interrupt vector table setup in C-code:

```
void far_interrupt new_IRQ1_handler(void);      /* ISR function prototype */
#define IRQ1_VECTOR      3                      /* Name for IRQ */
void (interrupt far*old_IRQ1_dispatcher)
      (es,ds,di,si,bp,sp,bx,dx,cx,ax,ip,cs,flags);   /* Variable to store old IRQ_Vecotr */
void far_interrupt new_IRQ1_handler(void);

/*---------------------------------------------------------------------------
| Function: init_irq_handlers
| Inputs:    Nothing
| Returns:   Nothing
| Purpose:   Set the pointers in the interrupt table to point to
\            our functions ie. setup for ISR`s.
\---------------------------------------------------------------------------*/
void init_irq_handlers(void)
{
      _disable();
            old_IRQ1_handler = _dos_getvect(IRQ1_VECTOR + 8);
            _dos_setvect(IRQ1_VECTOR + 8, new_IRQ1_handler);
            Gi_old_mask = inp(0x21);
            outp(0x21,Gi_old_mask & ~(1 << IRQ1_VECTOR));
      _enable();
}



/*---------------------------------------------------------------------------
| Function: restore do this before exiting program
| Inputs:    Nothing
| Returns:   Nothing
| Purpose:   Restore interrupt vector table
\---------------------------------------------------------------------------*/
void restore(void)
{
      /* Restore the old vector */
      _disable();

            _dos_setvect(IRQ1_VECTOR + 8, new_IRQ1_handler);
            outp(0x21,Gi_old_mask);
      _enable();
}
```

APPENDIX A. CANPC527D Specifications

Host Interface

Memory mapped into low memory, occupies 512 bytes
Jumper-selectable base address, 11 jumpers
8-bit data bus, 16 bit ISA bus connector

CAN Interface

- Galvanically isolated transceiver with 1 Mb/s datarate
- Timing parameters and speed of bus programmable
- Balanced CAN-bus choke for low EMI emissions
- Jumper selectable 120 Ohm onboard termination resistor

Connectors

Galvanically isolated CAN bus ISO 111898 compliant

Electrical

Operating temperature range	-40^{o}C to $+70^{o}$C (option)
Supply voltage	+5V only
Power consuption	125 mA

APPENDIX B. WHAT IS CAN INTERFACE?

Overview

The Controller Area Network (the CAN bus) is a serial data communications bus for real-time applications.

CAN operates at data rates of up to 1 Megabits per second and has excellent error detection and confinement capabilities.

CAN was originally developed by the German company Robert Bosch for use in the car industry to provide a cost-effective communications bus for in-car electronics and as as alternative to expensive and cumbersome wiring looms.

Now, because of its proven reliability and robustness, CAN is being used in many other automation and industrial applications.

CAN is now an international standard and is documented in ISO 11898 (for high-speed applications) and ISO 11519 (for lower-speed applications).

[In the UK - If you want to buy copies of the ISO standards - contact the British Standards Organisation on 0181 996-7000, or on 01908 221166.]

Low-cost CAN controllers and interface devices are available as off-the-shelf parts from several of the leading semiconductor manufacturers. Custom built devices and popular microcontrollers with embedded CAN controllers are also available.

There are many CAN-related system development packages. Hardware interface cards and easy-to-use software packages provide system designers, builders and maintainers with a wide range of design, monitoring, analysis, and test tools.

CAN in Cars

The Need

To satisfy the customer's wishes for greater safety, comfort, and convenience, and to comply with increasingly stringent government requirements for improved pollution control and reduced fuel consumption, the car industry has developed many electronic systems. Anti lock Braking Systems, Engine Management Systems, Traction Control, Air Conditioning Control, central door locking, and powered seat and mirror controls are just some examples.

The complexity of these control systems, and the need to exchange data between them meant that more and more hard-wired, dedicated signal lines had to be provided. Sensors had to be duplicated if measured parameters were needed by different controllers. Apart from the cost of the wiring looms needed to connect all these components together, the physical size of the wiring looms sometimes made it impossible to thread them around the vehicle (to control panels in the doors, for example).

In addition to the cost, the increased number of connections posed serious reliability, fault diagnosis, and repair problems during both manufacture and in service.

A new solution was needed.

The Solution

The Robert Bosch company (a highly regarded supplier of components and sub systems to the automotive industry) provided the answer in the mid 1980s by specifying the Controller Area Network (CAN).

Bosch defined the protocol (subsequently standardised internationally as ISO11898 and ISO 11519) and also licensed a number of companies to allow the design and manufacture of CAN-compliant semiconductor controllers and other devices.

Using the CAN protocol, such controllers, sensors, and actuators communicate with each other, in real-time, at speeds of up to 1Megabit per second, over a two wire serial data bus.

In addition, CAN networks:

- Are cost effective to design and implement
- Will continue to operate in harsh environments
- Are easy to configure and modify
- Automatically detect data transmission errors
- Provide an environment that enables the centralised diagnosis of faults – during design, or in-service

Industrial Applications of CAN

CAN controllers and interface chips are physically small. They are available as low-cost, off-the-shelf components. They will operate at high, real-time speeds, and in harsh environments. All these properties have led to CAN being used in a wide range of applications other than the car industry.

The benefits of reduced cost and improved reliability that the car industry gains by using CAN are now available to manufacturers of a wide range of products.

For example:

Marine control and navigation systems
Elevator control systems
Agricultural machinery
Production line control systems
Machine tools
Large optical telescopes
Photo copiers
Medical systems
Paper making and processing machinery
Packaging machinery
Textile production machinery
and even toys for children

Using CAN to network controllers, actuators, sensors, and transducers, manufacturers of all the above-mentioned computer controlled products have benefited from:

Reduced design time (readily available, multi sourced components, and tools)

Lower connection costs (lighter, smaller cables)

Improved reliability (fewer connections.)

Safety

The safety-related aspects of using CAN in cars attracted the attention of manufacturers of medical systems. Because of the inherent reliability of the data transmission and the stringent safety requirements that need to be built into medical equipment such as X-ray machines, CAN is now used in a range of these systems.

User Groups

To cater for the growth in the use of CAN and to provide a forum for discussion, several User Groups have been formed. One of the first to be formed was the CAN Textile Users Group, but the principal international Users Group is CAN in Automation (CiA).

CAN is now being used in an increasing number of applications in the automotive world and in many other industrial applications. The simplicity, robustness, and error detection capabilities of CAN make it an ideal communications technology for systems where reliability, low cost, and ease of design and configuration are required.

Standardisation through ISO 11898 provides true “plug ‘n play” usage.

Implementations of CAN

Communication is identical for all implementations of CAN. However, there are two principal hardware implementations.

The two implementation are known as Basic CAN and Full CAN.

The terms Basic CAN and Full CAN must not be confused with the terms Standard CAN (the 11 bit identifier, or Version 2.0A data format) and Extended CAN (the 29 bit identifier, or Version 2.0B data format). Suitably configured, each implementation (Basic or Full CAN) can handle both Standard and Extended data formats.

Basic CAN

In Basic CAN configurations there is a tight link between the CAN controller and the associated microcontroller. The microcontroller, which will have other system related functions to administer, will be interrupted to deal with every CAN message.

Full CAN

Full CAN devices contain additional hardware to provide a message “server” that automatically receives and transmits CAN messages without interrupting the associated microcontroller. Full CAN devices carry out extensive acceptance filtering on incoming messages, service simultaneous requests, and generally reduce the load on the microcontroller.

Network Sizes

The number of nodes that can exist on a single network is, theoretically, limited only by the number of available identifiers. However, the drive capabilities of currently available devices imposes greater restrictions. Depending on the device types, up to 32 or 64 nodes per network is normal, but at least one manufacturer now provides devices that will allow networks of 110 nodes.

Data Rate vs Bus Length

The rate of data transmission depends on the total overall length of the bus and the delays associated with the transceivers. For all ISO11898 compliant devices running at 1Mbit/sec speed, the maximum possible bus length is specified as 40 Metres, For longer bus lengths it is necessary to reduce the bit rate. To give some indication of this the following numbers are from the DeviceNet features list:

500 K bits per second at 100 metres (328 ft)
250 K bits per second at 200 metres (656 ft)
125 K bits per second at 500 metres (1640 ft)

How CAN works

Principle

Data messages transmitted from any node on a CAN bus do not contain addresses of either the transmitting node, or of any intended receiving node.

Instead, the content of the message (e.g. Revolutions Per Minute, Hopper Full, X-ray Dosage, etc.) is labelled by an identifier that is unique throughout the network. All other nodes on the network receive the message and each performs an acceptance test on the identifier to determine if the message, and thus its content, is relevant to that particular node.

If the message is relevant, it will be processed; otherwise it is ignored.

Identifiers

The unique identifier also determines the priority of the message. The lower the numerical value of the identifier, the higher the priority.

The higher priority message is guaranteed to gain bus access as if it were the only message being transmitted. Lower priority messages are automatically re-transmitted in the next bus cycle, or in a subsequent bus cycle if there are still other, higher priority messages waiting to be sent.

Bit encoding

CAN uses Non Return to Zero (NRZ) encoding (with bit-stuffing) for data communication on a differential two wire bus. The use of NRZ encoding ensures compact messages with a minimum number of transitions and high resilience to external disturbance.

The physical bus

The two wire bus is usually twisted pair (shielded or unshielded). Flat pair (telephone type) cable also performs well but generates more noise itself, and may be more susceptible to external sources of noise.

Robustness

CAN will operate in extremely harsh environments and the extensive error checking mechanisms ensure that any transmission errors are detected. See the the Errors section for more details.

The ISO11898 standard “Recommends” that bus interface chips be designed so that communication can still continue (but with reduced signal to noise ratio) even if:

- Either of the two wires in the bus is broken
- Either wire is shorted to power
- Either wire is shorted to ground

Depending on the design and configuration of the system, if both wires are broken at the same location, some limited functionality may still be achievable in each of the sub-systems created by the breaks.

Network Flexibility and Expansion

The content-oriented nature of the CAN messaging scheme delivers a high degree of flexibility for system configuration.

New nodes that are purely receivers, and which need only existing transmitted data, can be added to the network without the need to make any changes to existing hardware or software.

Measurements needed by several controllers can be transmitted via the bus, thereby removing the need for each controller to have its own individual sensor.

How Arbitration Works on the CAN Bus

In any system, some parameters will change more rapidly than others. For example – parameters that change quickly could be the RPM of a car engine, or the current floor level of an elevator (US) - lift (UK). Slower changing parameters may be the temperature of the coolant of a car engine.

It is likely that the more rapidly changing parameters need to be transmitted more frequently and, therefore, must be given a higher priority.

To cater for real-time data communication, this requires not only a fast data transmission rate, but also a rapid bus allocation mechanism to deal with occasions when more than one node may be trying to transmit at the same time.

To determine the priority of messages, CAN uses the established method known as Carrier Sense, Multiple Access with Collision Detect (CSMA/CD) but with the enhanced capability of non-destructive bitwise arbitration to provide collision resolution, and to deliver maximum use of the available capacity of the bus.

Non-Destructive Bitwise Arbitration

The priority of a CAN message is determined by the binary value of its identifier.

The numerical value of each message identifier (and thus the priority of the message) is assigned during the initial phase of system design.

The identifier with the lowest numerical value has the highest priority. Any potential bus conflicts are resolved by bitwise arbitration in accordance with the wired-and mechanism, by which a dominant state (logic 0) over writes a recessive state (logic 1).

The overall result is the same as if the highest priority message was the only message being transmitted. As soon as any lower priority transmitter loses control of the bus via the arbitration mechanism, it automatically becomes a receiver of the message with the highest priority and will not attempt re-transmission until the bus becomes available again.

The Benefits

Non-destructive bitwise arbitration provides bus allocation on the basis of need, and delivers efficiency benefits that can not be gained from either fixed time schedule allocation (e.g. Token ring) or destructive bus allocation (e.g. Ethernet.)

With only the maximum capacity of the bus as a speed limiting factor, CAN will not collapse or lock up. Outstanding transmission requests are dealt with, in their order of priority, with minimum delay, and with maximum possible utilisation of the available capacity of the bus.

Error Detection

CAN implements five error detection mechanisms; three at the message level and two at the bit level.

At the message level:

Cyclic Redundancy Checks (CRC)
Frame Checks
Acknowledgement Error Checks

At the bit level:

Bit Monitoring
Bit Stuffing

Cyclic Redundancy Check

Every transmitted message contains a 15 bit Cyclic Redundancy Check (CRC) code. The CRC is computed by the transmitter and is based on the message content. All receivers that accept the message perform a similar calculation and flag any errors.

Frame Check

There are certain predefined bit values that must be transmitted at certain points within any CAN Message Frame.

If a receiver detects an invalid bit in one of these positions a Form Error (sometimes also known as a Format Error) will be flagged.

Acknowledgement (ACK) Error Check

If a transmitter determines that a message has not been ACKnowledged then an ACK Error is flagged.

Bit Monitoring

Any transmitter automatically monitors and compares the actual bit level on the bus with the level that it transmitted. If the two are not the same, a bit error is flagged.

Bit Stuffing

CAN uses a technique known as bit stuffing as a check on communication integrity.

After five consecutive identical bit levels have been transmitted, the transmitter will automatically inject (stuff) a bit of the opposite polarity into the bit stream.

Receivers of the message will automatically delete (de-stuff) such bits before processing the message in any way.

Because of the bit stuffing rule, if any receiving node detects six consecutive bits of the same level, a stuff error is flagged.

Error Flag

If an error is detected by any node, using any and all of the five mechanisms described above, the node that detects the error aborts the transmission by sending an Error Flag. This prevents any other node from accepting the message and ensures consistency of data throughout the network.

Error Frame

An Error Frame consists of an Error Flag, as described above, and an Error Delimiter. The Error Delimiter is eight recessive bits and only after this period, plus a 3 bit INTermission field, can nodes start to transmit, or attempt re-transmission of aborted messages.

Error Confinement

Error confinement is a mechanism which is understood to be unique to CAN and provides a method for discriminating between temporary errors and permanent failures. Temporary errors may be caused by, spurious external conditions, voltage spikes etc. Permanent failures are likely to be caused by bad connections, faulty cables, defective transmitters or receivers, or long lasting external disturbances.

The general principle only is described here. More detailed information is available in the ISO standard, and in the data sheets from the device manufacturers.

Error Counts

When an error is flagged, error counts are added to one of two dedicated error count registers within each CAN controller on each node.

It’s more complex than stated here, but - in principal - receive errors are given a weighting of 1 and are accumulated in a Receive Error Count register; transmit errors are given a weighting of 8 and accumulated in a Transmit Error Count register.

If errors continue to occur, the error counts continue to increase. Any good messages decrement the Error Count registers and, if no further errors are detected, both Error Counts go back to zero.

The accumulated error counts determine the error status of a node.

Error Active Mode

Nodes usually operate in a state known as Error Active mode. In this condition a node is fully functional and both the Error Count registers contain counts of less than 127.

Error Passive Mode

If the count in either Error Count register in a node exceeds 127, the node will go from Error Active mode to a heightened state of “alert” known as Error Passive mode.

Error Passive nodes can still transmit and receive messages but are restricted in relation to how they flag any errors that they may detect.

The ISO standard (and some of the device data sheets) explain the precise mechanisms in more detail.

Bus Off Mode

If an error condition persists, such that the Transmit Error Count of a device exceeds 255, the device will take itself off the bus by going to BusOff mode. This means that a permanently faulty device will cease to be active on the bus, but communications between the other nodes can continue unhindered.

Error Detection Capabilities

Error detection on CAN is extremely thorough.

Global errors which occur at all nodes are 100% detectable.

For local errors (i.e. errors which may appear at only some nodes) the CRC check alone has the following error detection capabilities:

- Up to 5 single bit errors are 100% detectable, even if the errors are distributed randomly within the code word
- All single bit errors are detected if their total number within the code word is odd

The residual (undetected) error probability of the CRC check alone is 3 x 10 to the power -5.

In conjunction with all the other error checking mechanisms, a more realistic value is 10 to the power -11.

CAN Message Format

Message Frames

In a CAN system, data is transmitted and received using Message Frames. Message Frames carry data from a transmitting node to one, or more, receiving nodes.

The Standard CAN protocol (version 2.0A) supports messages with 11 bit identifiers.

The Extended CAN protocol (version 2.0B) supports both 11 bit and 29 bit identifiers.

Most 2.0A controllers transmit and receive only Standard format messages, although some (known as 2.0B passive) will receive Extended format messages but then ignore them. 2.0B controllers can send and receive messages in both formats.

2.0A Format

A Standard CAN (Version 2.0A) Message Frame consists of seven different bit fields:

- A Start of Frame (SOF) field - which indicates the beginning of a message frame.
- An Arbitration field, containing a message identifier and the Remote Transmission Request (RTR) bit. The RTR bit is used to discriminate between a transmitted Data Frame and a request for data from a remote node.

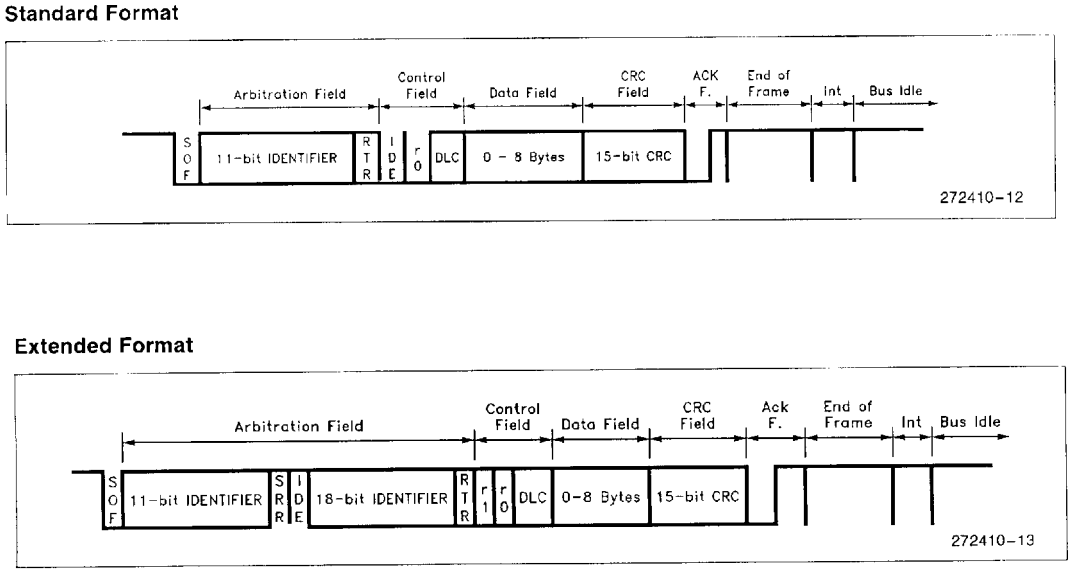

Fig 1. CAN 2.0A Message Frame

A Control Field containing six bits:

* two dominant bits (r0 and r1) that are reserved for future use, and

* a four bit Data Length Code (DLC). The DLC indicates the number of bytes in the Data Field that follows

A Data Field, containing from zero to eight bytes.

The CRC field, containing a fifteen bit cyclic redundancy check code and a recessive delimiter bit

The ACKnowledge field, consisting of two bits. The first is the Slot bit which is transmitted as recessive, but is subsequently over written by dominant bits transmitted from any node that successfully receives the transmitted message. The second bit is a recessive delimiter bit

The End of Frame field, consisting of seven recessive bits.

Following the End Of Frame is the INTermission field consisting of three recessive bits.

After the three bit INTermission period the bus is recognised to be free. Bus Idle time maybe of any arbitrary length including zero.

2.0B Format

The CAN 2.0B format provides a twenty nine (29) bit identifier as opposed to the 11 bit identifier in 2.0A.

Version 2.0B evolved to provide compatibility with other serial communications protocols used in automotive applications in the USA. To cater for this, and still provide compatibility with the 2.0A format, the Message Frame in Version 2.0B has an extended format.

The differences are:

– In Version 2.0B the Arbitration field contains two identifier bit fields. The first (the base ID) is eleven (11) bits long for compatibility with Version 2.0A. The second field (the ID extension) is eighteen (18) bits long, to give a total length of twenty nine (29) bits.

– The distinction between the two formats is made using an Identifier Extension (IDE) bit.

– A Substitute Remote Request (SRR) bit is also included in the Arbitration Field. The SRR bit is always transmitted as a recessive bit to ensure that, in the case of arbitration between a Standard Data Frame and an Extended Data Frame, the Standard Data Frame will always have priority if both messages have the same base (11 bit) identifier.

All other fields in a 2.0B Message Frame are identical to those in the Standard format.

2.0A and 2.0B Compatibility

2.0B controllers are completely backward compatible with 2.0A controllers and can transmit and receive messages in either format.

Note, however, that there are two types of 2.0A controllers:

– The first is capable of transmitting and receiving only messages in 2.0A format. With this type of controller, reception of any 2.0B message will flag an error.

– The second (known as 2.0B passive) is capable of sending and receiving 2.0A messages. They will also acknowledge receipt of 2.0B messages - but then ignore them.

Therefore, within the above mentioned constraints it is possible to use both Version 2.0A (with 2.0B passive capabilities) and 2.0B controllers on a single network.

The number of unique identifiers available to users, on a single 2.0A network, is 2,032 (2 to the power 11 - 2 to the power 4).

Leaving aside the use for compatibility purposes with American buses, the number of unique identifiers available on a 2.0B network is in excess of 500 million!

Bit time

As defined in ISO11898, the nominal time for each bit in a CAN message frame is made up of four non-overlapping time segments as shown below.

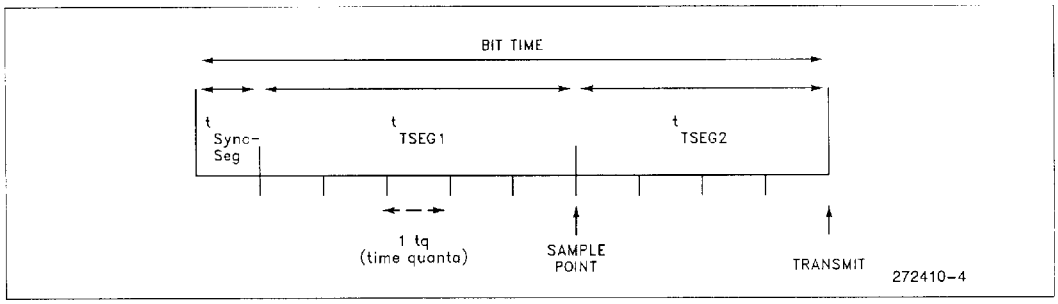

Fig 2. Bit time segments

Sync-seg is the segment that is used to synchronise the nodes on the bus. A bit edge (if there is a data change) is expected during this segment.

Prop-Seg is a period of time that is used to compensate for physical delay times within the network.

Phase-seg1 is a buffer segment that may be lengthened during resynchronisation to compensate for oscillator drift and positive phase differences between the oscillators of the transmitting and receiving node(s).

Phase-seg2 is a buffer segment that may be shortened during resynchronisation (described below) to compensate for negative phase errors and oscillator drift.

The Sample point is always at the end of Phase-seg1 and is the time at which the bus level is read and interpreted as the value of the current bit.

Whether transmitting or receiving, all nodes on a single CAN bus must have the same nominal bit time. Bit time is programmable at each node on a CAN Bus and is a function of the period of the oscillator local to each node, the value that is user-programmed into a Baud Rate Prescaler (BRP) register in the controller at each node, and the programmed number of time quanta per bit.

The BRP register is an ISO11898 requirement,and must be user-programmable with integer values ranging from (at least) 1 to 32.

One time quanta (Also known as the system clock period) is defined as the period of the local oscillator, multiplied by the value in the BRP.

Each of the four time segments in one bit is one or more time quanta long. As stated in the Bosch CAN2 spec:

Sync-seg is always one time quantum long

Prop-seg is programmable from one to eight (or, optionally, more) time quanta long

Phase-seg1 is programmable from one to eight (or, optionally, more) time quanta long

Phase-seg2 is the maximum of Phase-seg1 and the Information Processing Time where the Information Processing Time is less than or equal to 2 time quanta.

Synchronisation

When any node receives a data frame or a remote frame, it is necessary for the receiver to synchronise with the transmitter.

Because there is no explicit clock signal that a CAN system can use as a timing reference, two mechanisms are used to maintain synchronisation.

The first is hard synchronisation and occurs within each receiving controller whenever a falling (recessive-to-dominant) edge is detected during bus-idle time; i.e. at Start-of-Frame (SOF), which is the ONLY time when hard synchronisation occurs.

To compensate for oscillator drift, and phase differences between transmitter and receiver oscillators, additional synchronisation is needed.

So - for subsequent bits in any received frame, if a bit edge does not occur in the Sync-seg segment of bit time, resynchronisation is automatically invoked and will shorten or lengthen the current bit time depending on where the edge occurs. The maximum amount by which the bittime is lengthened or shortened is determined by a user-programmable number of time quanta known as the Synchronisation Jump Width (SJW).

If the edge of the current bit is “late”, i.e. it occurs after Sync-seg but before the sample point, then Phase-seg1 of the current bit is automatically lengthened.

If the edge of the next bit is “early”, i.e. it occurs during Phase-seg2 of the current bit, then Phase-seg2 of the current bit is automatically shortened.