

icc 1995

2nd international CAN Conference

in London (United Kingdom)

Sponsored by

**Motorola Semiconductor
National Semiconductor
Philips Semiconductors**

Organized by

CAN in Automation (CiA)

international users and manufacturers group

Am Weichselgarten 26

D-91058 Erlangen

Phone +49-9131-69086-0

Fax +49-9131-69086-79

Email: headquarters@can-cia.de

URL: <http://www.can-cia.de>

Martin Embacher
VLSI Design Engineer
National Semiconductor
Embedded Technologies Division Europe

Cost efficient and customizable microcontroller solution replaces dedicated protocol controllers in low speed CAN network applications.

The CAN (Controller Area Network) is one of today's most widely accepted car networking systems. Various protocol implementations are available from different suppliers. Dedicated protocol controllers - Full-CAN controllers - are found as system bus interfaces connected to a main CPU or integrated into them. Yet in some applications, particularly in the low speed arena, these devices don't meet the price target or offer the flexibility required by the system designer. This paper outlines the application interfaces available for the CAN protocol, gives an overview to National Semiconductors CAN chips and it demonstrates in a practical example how these products can help to minimise the cost of Full-CAN controller applications while increasing the flexibility of such systems.

Introduction

CAN is designed to address the needs for a highly reliable protocol with maximum throughput for interconnecting multiple autonomous controller modules within harsh industrial or automotive applications. The need for such a system arose first when more and more electronic modules were introduced to the automobiles resulting in huge amounts of wires being laid out within a car to perform interconnection between control modules and the sensors/actuators. The first objective of the CAN system was to reduce these kilometers of cabling and thereby reducing system cost by saving wiring effort. Additionally the system had to have maximum reliability as basically all functions within a car could introduce a safety risk. Next to obviously important functions as motor management and anti blocking systems also comfort electronic functions can lead to unsafe operation of cars. Taking a faulty electronically controlled driver seat as an example this becomes more clear. Only assume due to a fault the seat is suddenly moving while the car runs at high speed.

Basic and Full CAN implementations

Many Semiconductor suppliers implemented various versions of a CAN user interface. Even the protocol remains the same Basic-CAN implementations provide only the basic functionality of a CAN interface with the capability to buffer only one message with a limited acceptance filter. Though, an additional burden is placed on the CPU since it has to perform message filtering next to their regular task. Full CAN controllers extend this basic features by not only implementing the protocol - moreover they implement a complete message 'server' capable of automatically receiving and transmitting multiple messages on the CAN bus without interrupting the systems main CPU if it is not necessary. Figure 1. shows the Basic-CAN interface with the extension for Full-CAN from the programmers point of view. The hardware interface to the processor is provided with either serial links or a parallel interface with message data (identifier, control and data) being accessed on memory mapped address locations, so the user 'sees' message data and control information.

Typical multiplex systems consists of one or more of each protocol implementations connected to each other over a common bus. The Full-CAN is used where the CPU has to perform a magnitude of other tasks and where communication needs to be highly independent from the rest of the software. A Basic-CAN controller is used in areas in which the CPU has some spare performance to assist the communication work. Full-CAN implementations with their higher level of functionality, require a larger silicon area than a Basic CAN implementation, which translates directly into higher prices. Though, from a systems designers point of view it might be desirable to use as many Full-CAN controllers as possible to free up the CPU for applications tasks and provide free processing resources for future or different features. Common to both implementations is that they can process at least one receive and one transmit message object completely autonomous - which results in a specific number of registers being required on the CAN block to store the information. Lastly, fully autonomous CAN modules, commonly known as SLIO (Serial Linked

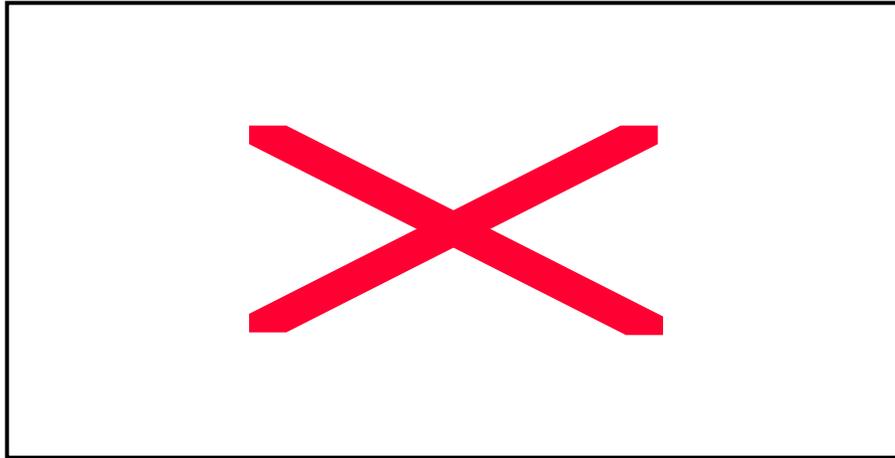


Figure 1: CAN programming model

I/O) devices are also build. Those devices integrate a quasi Full-CAN controller with self sufficient simple I/O capabilities having no need for a main CPU within the module and therefore no dedicated programming requirements. With these devices a simple CAN module can be designed by only developing the input and output circuitry required for the specific application. All I/O control is then provided via the CAN network. Examples are National Semiconductors MM57C360 and MM57C362, which integrate a CAN cell, a message sequencer and I/O drivers. With this functionality the device is perfectly suited for simple modules like keypads or light clusters.

National's COPCAN interface

With the implementation of the COPCAN on the COP8_ microcontroller family National Semiconductor has addressed especially the high implementation costs of previous CAN modules. By reducing the amount of registers required to implement the CAN protocol. The driving factors was cost on the one side and the idea that not all applications require the high speed feature all CAN implementation known so far offered. To reduce cost the object buffer for both receive and transmit was reduced from 10 bytes (2 identifier + 8 data bytes) to 4 bytes (2 identifier + 2 data bytes). The 'reminder' of the CAN interface, error management, BTL was not changed in order to achieve full compatibility. With the reduction of registers the interface is no longer capable to process messages with more than two bytes of data independently, data needs to be provided by the main processor in time. This register reduction however has no influence to the interface performance in low speed (< 125kbit/s) applications as the processor has enough time to store/provide the data when required. Interrupt flags indicate to the CPU when the processor needs to provide data to the COPCAN interface. This results in a ratio between the maximum possible bus speed and the time the processor needs to save and provide data which will now be explained in more detail. On the one side the COP8 microcontroller core features an instruction cycle time of $1 \mu s = 1 \text{ tC}$ with an external clock of 10 MHz. Most instructions take one tC to execute. On the other side with a bus speed of 125 kBit/s, typical for low speed applications, one byte time on the CAN bus takes a minimum of: $8 \text{ (bit)} * (1 / 125 \text{ kHz}) (\mu s) = 64 \mu s$, without the possible stuff bits. With a given interrupt latency time of 20 tC maximum (including transfer of control instructions) this leaves 44 μs to store the receive data or write new data into the transmit register. Figure 2 shows the timing of a message reception with four data bytes. It can be seen from the picture that adding data bytes to the frame would neither introduce a critical path nor decrease the processors free time.

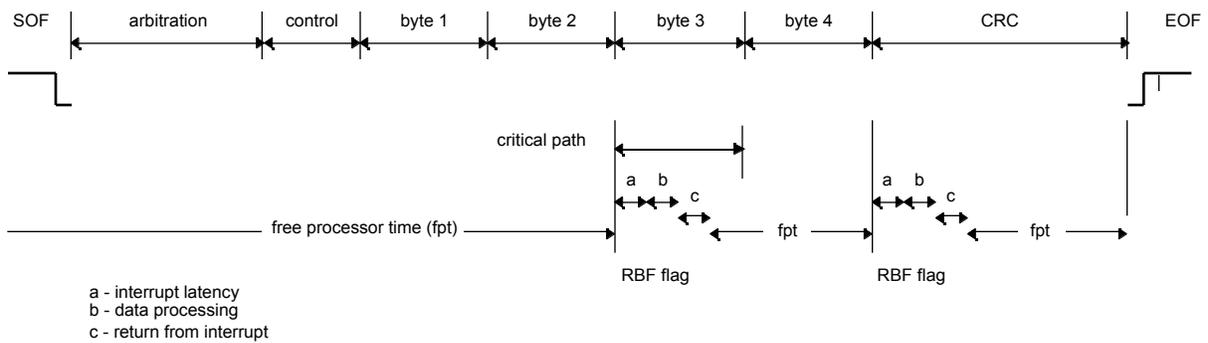


Figure 8: Message Processing with the COPCAN interface

In this example the CPU's usage to store the received data is given as apx. 20 tC. The critical path is to read the first receive buffer byte after the receive buffer full flag (RBF) is set and before it gets overwritten by new incoming data. During the free processor time other application tasks can be executed. Basically the same example is valid for a transmitter. The main difference is that transmitting data is mostly synchronus to the programmes execution where receiving is asynchronus. Thus the transmission of data is not time critical. Also using a high speed link (>125kBit/s) is possible for applications which don't need to receive more than two bytes of data.

Implementing a Full-CAN processor with an COP8 microcontroller

National Semiconductors COP8_ microcontroller core contains beside the pure CPU a serial synchronous microwire interface and a processor independent Timer. Additional functional blocks like the COPCAN interface, Timers, USART, A/D convertors and various sizes of ROM and RAM can be added with the Configurable Controller Methodology (CCM). Today several standard parts are offered of which two feature the COPCAN interface. The COP884BC and the COP888EB. All COP8_ microcontroller family members are also available as one time programmable (OTP) devices. The following section shows how a protocol processor, providing a customizable Full-CAN interface, can be integrated with the COP8 family. A block diagram of the setup with the microcontroller and the main CPU is shown in Figure 2.

Additional customized options like time stamp for received messages or timed automatic transmission of

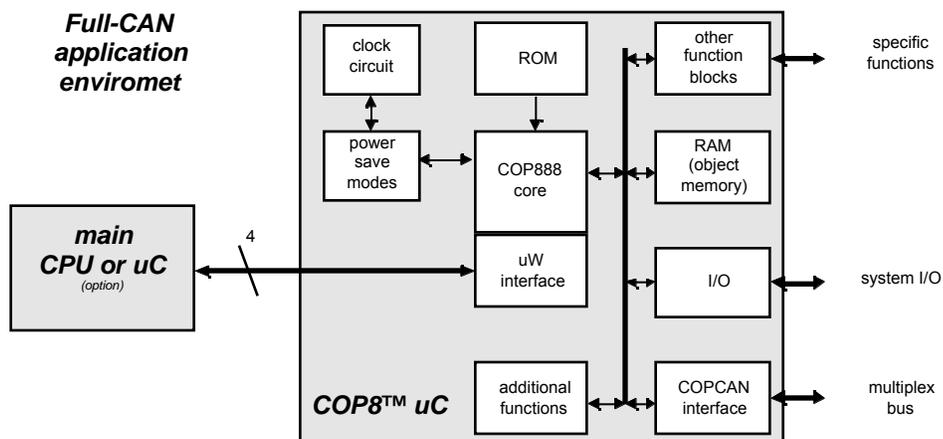


Figure 2: COP8 based Full-CAN interface

data can be integrated by simply altering the software. In addition several data processing tasks, i.e. automatic keyboard scanning can be integrated - thus reducing the overhead on the main CPU and freeing up processing resources. Another advantage of having a second CPU in the system is automatic diagnostics either with a specific protocol or with the watchdog circuit integrated on the COP8_. Finally, the power save features of the COP8_ microcontrollers help to minimize power consumption in the application by gradually switching off modules - including the main CPU. The multi-input wake-up feature allows multiple sources to return from the save mode to the active mode. The interface to the main CPU can be chosen to be provided with standard I/O ports of the microcontroller, the microwire interface or if very high speed communication is required with a newly developed high speed serial link.

In this example, however, the microwire interface is used. Communication is done with three wires and one handshake signal. The data from the main CPU to the microcontroller is transmitted in packets of eight bit with a customizable protocol. The microwire interface can be programmed to generate an interrupt every eight clocks applied to the SK. Thus indicating the master CPU wants to exchange some commands or data with the microcontroller. After the COP8_ reads out the data from the microwire register it returns an acknowledge signal to the main processor, by toggling the handshake line. Figure 9 shows a block diagram of the COP8_ microcontroller linked with the main CPU. The instructions stored in the COP8_ ROM firstly execute the protocol between the master and the COP8_, secondly they process CAN messages and thirdly they filter out unwanted data.

The software of the application is divided into several tasks which allow easy customization. A main loop continuously polls various flags. These are set by the microcontrollers hardware, like the system timer, the multi-sourced external interrupt/wake-up or by the interrupt handlers e.g. of the microwire interface or CAN interface. The microwire interrupt indicates a main CPU communication request. The CAN interrupts are receive, transmit and error. All of them are leading to a separate interrupt vector within the COP8_ memory. Upon detecting one flag to be set the program branches to the certain subroutine. This program structure is chosen to ensure fast response times for the time critical communication parts CAN and microwire. A flowchart of the main routine is found in Figure 10 together with the CAN receive interrupt handler.

The communication request flag is set as soon as the microwire received the first byte, e.g. indicating the

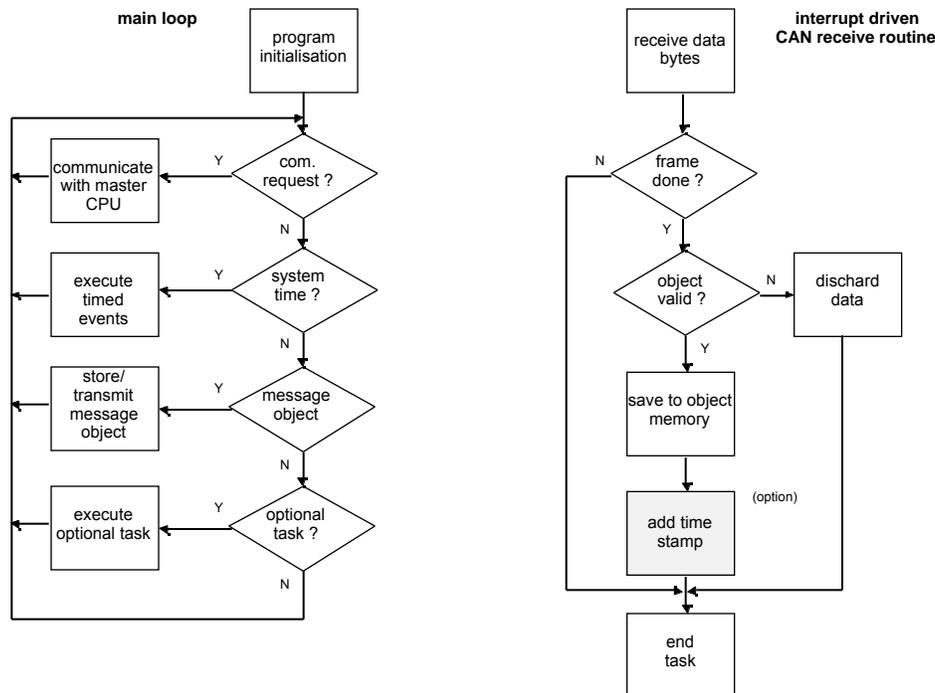


Figure 3. Main program and CAN receive interrupt handler

command for the COP8_, this data was read out the uW shift register and the handshake signal set, to indicate the main CPU the possibility to read or write the next data byte. Within the communication subroutine these data bytes are exchanged with the main processor. The system time can be generated

by the idle timer's pending flag. This flag is set every 4096 tC on the COP884BC and it can be programmed to be set every 4k, 8k, 16k or 32k tC on the COP888EB. Secondly, the system time can be generated with a free programmable 16 bit autoreload timer T1, for increased flexibility. Timed events, like automatic transmission of a CAN message or a software real time clock are then executed. CAN message objects are handled by a subroutine as described later. Finally, flags for optional tasks can be included and polled within the main loop in order to comprise additional features.

The CAN receive interrupt routine stores received data in receive buffers located within the RAM (please refer to Figure 8.). For this data storage special memory locations - base page RAM - are used as indirect addressing operations in this area are executed faster than if used on the remaining RAM area. After a complete message object is received and no errors occurred a flag is set. Optionally the systems time can be stored as well to allow verification of the creation time for a specific message in real time systems. Then the message object is filtered and stored into its final location within the message object handlers. One receive message object handler is shown in Figure 11 and described below. CAN transmit interrupts work similar to to the receive routine on a different interrupt vector. Additions to the transmit schedule routine - which are not offered with standard Full-CAN chips - can be done as well. For example a transmit object may be verified to be sent within a specific time or within a certain number of retrys if they're likely to loose arbitration on a highly frequented bus. Another example is the automatic transmission of the systems (real) time in order to have a common time base over the network.

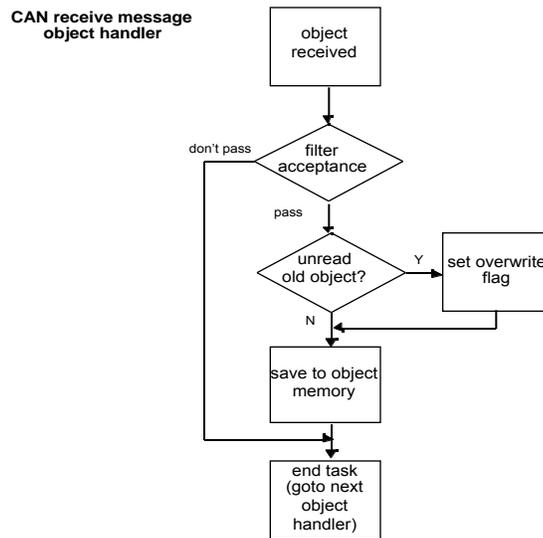


Figure 4: CAN receive object handler

The CAN receive object handler is called after a CAN message was satisfactory received. One object handler verifies the received message object with its specific acceptance filter and stores the data if the messages identifier matches. Else the next receive object handler is called until all possible receive objects are verified. A flag may be set to indicate to the main CPU afterwards the reception of a message. If new data for one object is received a flag is set to indicate the data overwrite. The number of possible message objects to be stored is only limited by the processors RAM.

Software example

After theoretically outlining the implementation of the protocol processors software, this section provides an example in COP8 assembly language to instantiate one receive object handler. The program is written in the form of a macro which allows multiple message handlers to be used within one program by simply calling the macro several times. The program uses 10 or 12 bytes of RAM for every message object and two global status bytes. One to indicate the reception of a message (rx_status) and one to indicate the overrun condition if new data is received before it was transmitted to the main controller (rx_overwrite).

The parameters to the macro are the number of the message object and a pointer to the message object memory. The program is executed as shown in figure 4. After initializing some pointers (lines 004 and 005) the received messages identifier is compared with the identifier of the current object (lines 006 to 013). As the identifier now matches it is checked if the objects data was read by the main processor. If the

data was not read the overrun flag is set (lines 014 and 015). Finally, the received data is copied into the objects memory (line 016 to 033).

```
(001) .macro rx_object, obj_number, msg_obj
(002) .local ; local variables used
(003) $message_filter:
(004) ld b, #msg_obj ; point to msg_obj
(005) ld x, #rx_buffer ; point to receive buffer
(006) ld a, [x+] ; get receive identifier
(007) ifne a, [b] ; compare with object id
(008) jp $end_msg ; if fail - then end
(009) $idlc_test:
(010) ld a, [b+] ; increment rx buff pointer
(011) ld a, [x+] ; get remaining receive id
(012) ifne a, [b] ; compare with object id
(013) jp $end_msg ; if fail - then end
(014) ifbit obj_number, rx_status ; data received before ?
(015) sbit obj_number, rx_overwrite ; then indicate
(016) andsz a, #0x0f ; mask data length code
(017) jp $copy_loop ; and copy data
(018) jp $end_obj ; if dlc == 0 then end
(019) x a, byte_count ; save dlc to byte counter
(020) $copy_loop:
(021) ld a, [x+] ; read bytes
(022) x a, [b+] ; and save to memory
(023) drsz byte_count ; decrement counter
(024) jp $copy_loop ; ..until done
(025) $end_obj:
(026) ld b, #msg_time ; point to time stamp
(027) ld a, system_time_high ; get time high byte
(028) x a, [b+] ; and save
(029) ld a, system_time_low ; get time low byte
(030) x a, [b] ; and save
(031) sbit obj_number, rx_status ; indicate receive
(032) $end_msg: ; end
(033) .endm ;
```

This macro uses 41 bytes of ROM and takes a maximum of 19 tC until the acceptance of one message is filtered.

Conclusion

This paper explained the different programming models of CAN chips. It showed how a Full-CAN controller to move and shape the information contained within the messages can be implemented at low cost. Finally, the flexibility of a microcontroller solution with customizable software compared to a fixed chip solution was outlined.