# Replacing Dedicated Protocol Controllers with Code Efficient and Configurable Microcontrollers—Low Speed CAN Network Applications

National Semiconductor
Application Note 1048
Martin Embacher
May 1997



## BACKGROUND

*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 Application Note outlines the application interfaces available for the CAN protocol, gives an overview to National Semiconductor's CAN devices and demonstrates in a practical example how these products can help to minimize 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 where introduced to the automobiles, resulting in huge amounts of wires being lain 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 reduce 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 such as motor management and anti-blocking systems, 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 its 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 system's main CPU if it is not necessary. *Figure 1* shows the Basic-CAN interface with the extension for Full-CAN from the programmer's point of view.

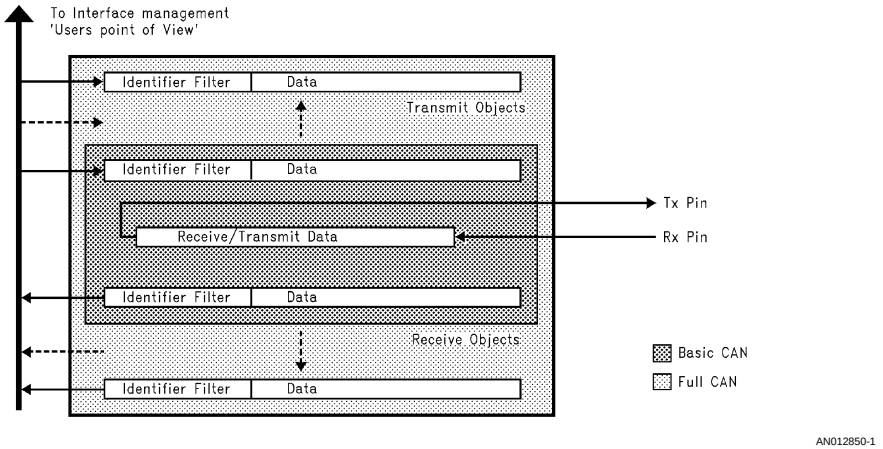



**FIGURE 1. CAN Programming Model**

COP8™, MICROWIRE/PLUS™ and WATCHDOG™ are trademarks of National Semiconductor Corporation.

The hardware interface to the processor is provided with either serial links or a paralell interface with message data (identifier, control and data) being accesed on memory mapped address locations, so the user “sees” message data and control information.

Typical multiplex systems consist of one or more of each protocol implementation connected to each other over a common bus. The Full-CAN is used where the CPU has to perform a magnitute 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 designer’s 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 autonomously—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 I/O) devices are also available. These 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 circurity required for the specific application. All I/O control is then provided via the CAN network. (See Application Note 1073 “SLIO-CAN; CAN-Linked I/O based on COP884BC”).

## NATIONAL’S COPCAN INTERFACE

With the implementation of the COPCAN on the COP8™ microcontroller family, National 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 were 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 “remainder” 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 on the interface performance in low speed (<125 kbit/s) applications, since 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 µs = 1 $t_C$ with an external clock of 10 MHz. Most instructions take one $t_C$ 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 (bit) * (1/125 kHz) (µs) = 64 µs, without the possible stuff bits. With a given interrupt latency time of 20 $t_C$ 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 processor’s free time.

In this example, the CPU’s usage to store the received data is given as approximately 20 $t_C$. 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 exceuted. Basically the same example is valid for a transmitter. The main difference is that transmitting data is mostly synchronus to the program’s excecution where receiving is asynchronous. Thus, the transmission of data is not time-critical. Also, using a high speed link (>125 kbit/s) is possible for applications which don’t need to receive more than two bytes of data.

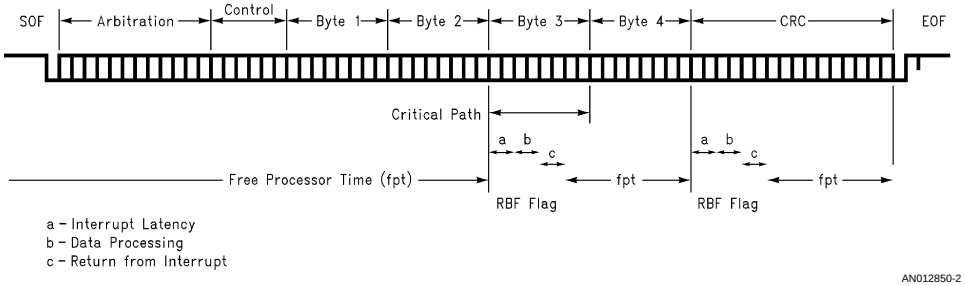



**FIGURE 2. Message Processing with the COPCAN Interface**

## IMPLEMENTING A Full-CAN PROCESSOR WITH A COP8 MICROCONTROLLER

National’s COP8 microcontroller core contains, beside the pure CPU, a serial synchronous MICROWIRE/PLUS™ 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 3*. Additional customized options, like time stamp for received messages or timed automatic transmission of 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/PLUS interface or, if very high speed communication is required, with a newly developed high-speed serial link.

In this example, however, the MICROWIRE/PLUS 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 bits with a customizable protocol. The MICROWIRE/PLUS 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/PLUS register, it returns an acknowledge signal to the main processor, by toggling the handshake line. *Figure 3* shows a block diagram of the COP8 microcontroller linked with the main CPU. The instructions stored in the COP8 ROM first execute the protocol between the master and the COP8, then process CAN messages, and finally 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 microcontroller's hardware, like the system timer, the muti-sourced external interrupt/wake-up or by the interrupt handlers (e.g., of the MICROWIRE/PLUS interface or CAN interface). The MICROWIRE/PLUS 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/PLUS. A flowchart of the main routine is found in *Figure 4*, together with the CAN receive interrupt handler.

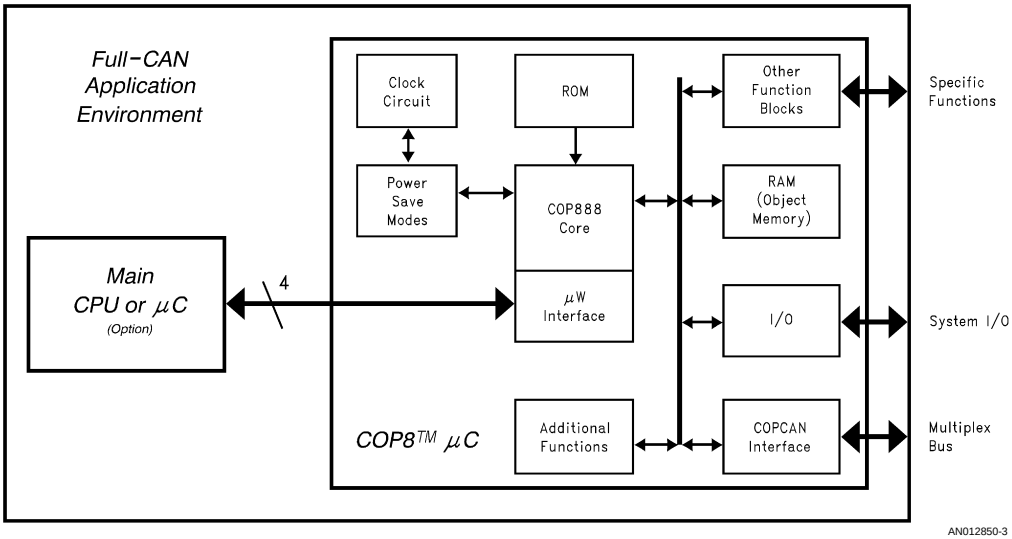



**FIGURE 3. COP8-Based Full-CAN Interface**

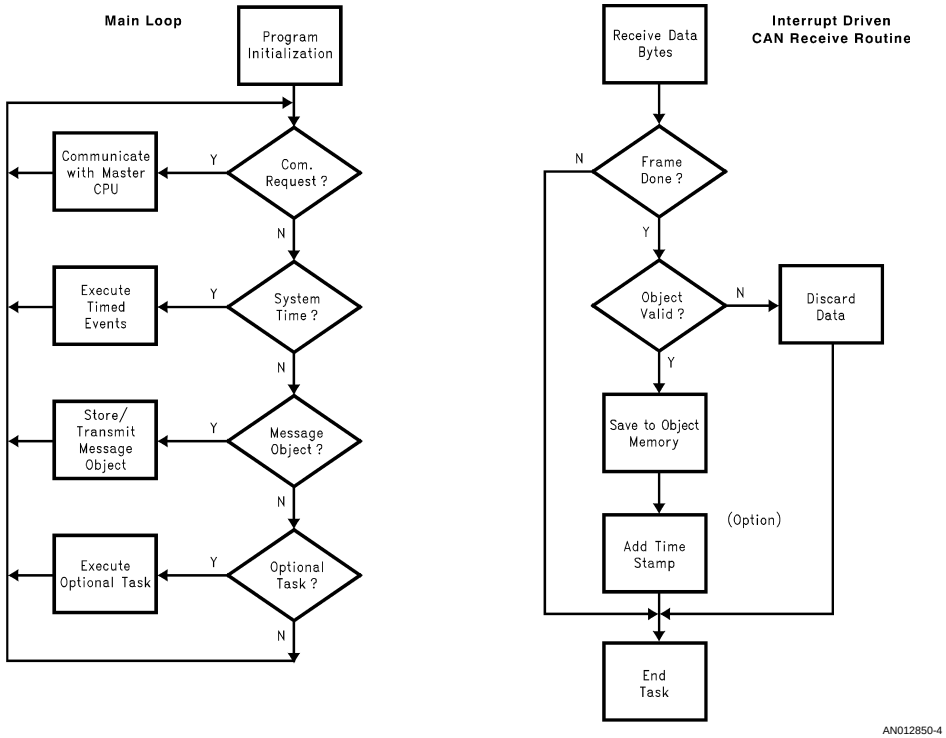



**FIGURE 4. Main Program and CAN Receive Interrupt Handler**

The communication request flag is set as soon as the MICROWIRE/PLUS received the first byte, indicating the command for the COP8. This data was read from the µW shift register and the handshake signal set, to indicate the possibility to read or write the next data byte to the main CPU. 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 $t_C$ on the COP884BC and it can be programmed to be set every 4k, 8k, 16k or 32k $t_C$ on the COP888EB. Secondly, the system time can be generated with a free programmable 16 bit auto-reload 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 (*Figure 2*). For this data storage, special memory locations—base page RAM—are used as indirect addressing operations in this area and 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 system's 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 5* and described below. CAN transmit interrupts work similarly 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 retries if they're likley to lose arbitration on a highly frequented bus. Another example is the automatic transmission of the system's (real) time in order to have a common time base over the network.

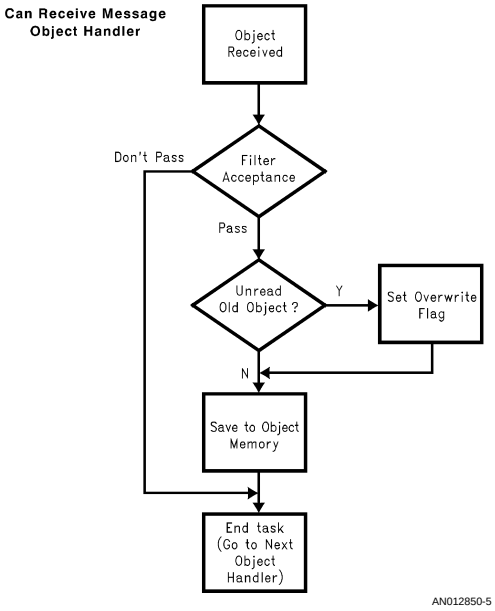



**FIGURE 5. CAN Receive Object Handler**

The CAN receive object handler is called after a CAN message was satisfactorily received. One object handler verifies the received message object with its specific acceptance filter and stores the data if the message's identifier matches. Otherwise, the next receive object handler is called until all possible receive objects are verified. Afterwards, a flag may be set to indicate the reception of a message by the main CPU. 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 processor's RAM.

## SOFTWARE EXAMPLE

After theoretically outlining the implementation of the protocol processor's software, this section provides an example in COP8 assembly language to instanciate 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).

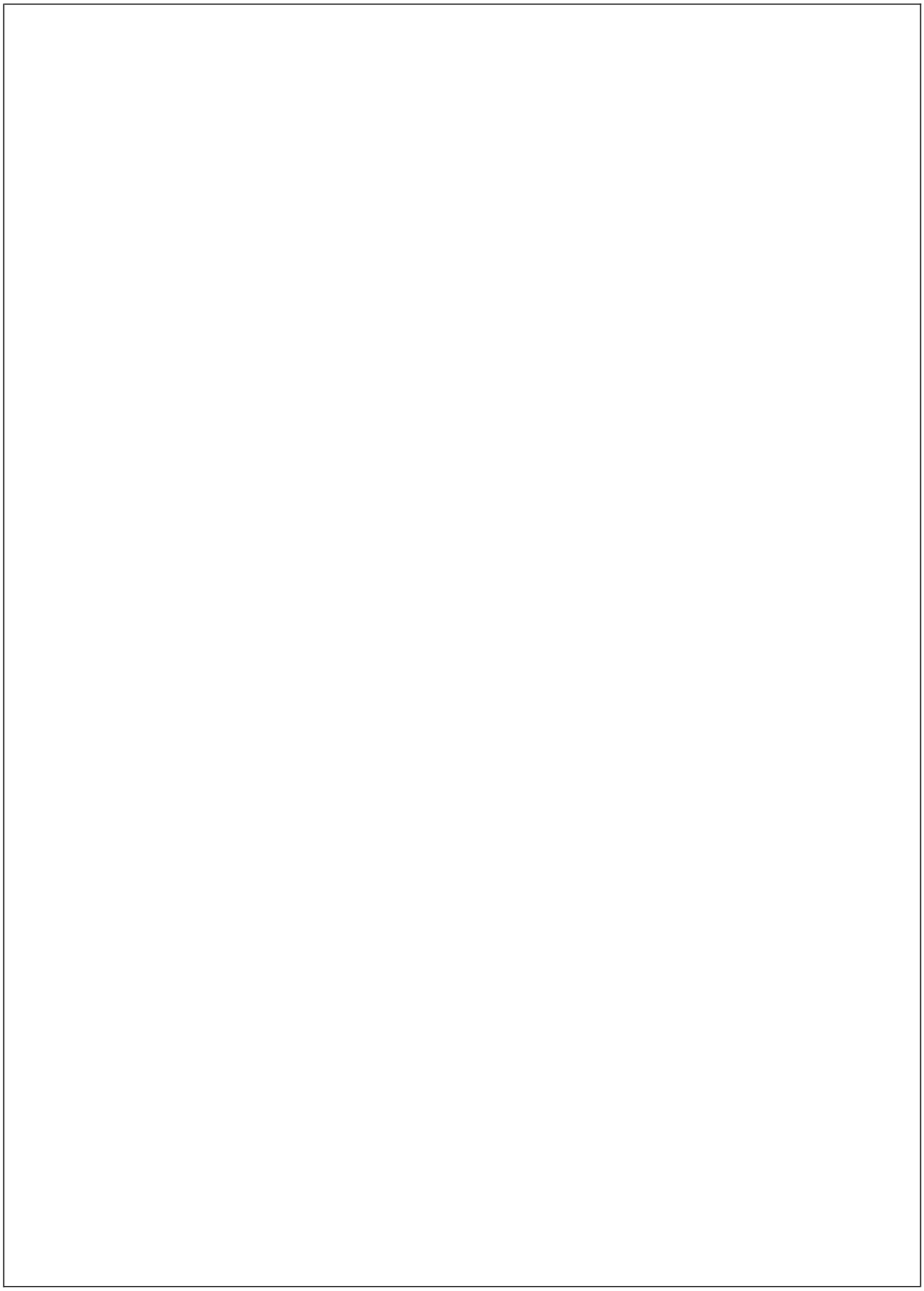
The parameters to the macro are the number of the message object and a pointer to the message object memory. The program is executed as shown in *Figure 5*. After initializing some pointers (lines 004 and 005) the received message's identifier is compared with the identifier of the current object (lines 006 to 013). As the identifier now matches, it is checked if the object's 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 object's memory (line 016 to 033).

This macro uses 41 bytes of ROM and takes a maximum of 19 $t_C$ 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.

```
(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__overwite  ; 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                               ;
```



**LIFE SUPPORT POLICY**

NATIONAL'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS WRITTEN APPROVAL OF THE PRESIDENT OF NATIONAL SEMICONDUCTOR CORPORATION. As used herein:

1. Life support devices or systems are devices or systems which, (a) are intended for surgical implant into the body, or (b) support or sustain life, and whose failure to perform when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.
2. A critical component in any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

**National Semiconductor Corporation**
Americas
Tel: 1-800-272-9959
Fax: 1-800-737-7018
Email: support@nsc.com

www.national.com

**National Semiconductor Europe**
Fax: +49 (0) 1 80-530 85 86
Email: europe.support@nsc.com
Deutsch Tel: +49 (0) 1 80-530 85 85
English Tel: +49 (0) 1 80-532 78 32
Français Tel: +49 (0) 1 80-532 93 58
Italiano Tel: +49 (0) 1 80-534 16 80

**National Semiconductor Hong Kong Ltd.**
13th Floor, Straight Block,
Ocean Centre, 5 Canton Rd.
Tsimshatsui, Kowloon
Hong Kong
Tel: (852) 2737-1600
Fax: (852) 2736-9960

**National Semiconductor Japan Ltd.**
Tel: 81-3-5620-6175
Fax: 81-3-5620-6179

National does not assume any responsibility for use of any circuitry described, no circuit patent licenses are implied and National reserves the right at any time without notice to change said circuitry and specifications.