INTRODUCTION

The Microchip PIC18XXX8 family of microcontrollers provide an integrated Controller Area Network (CAN) solution along with other PICmicro® features. Although originally intended for the automotive industry, CAN is finding its way into other control applications. In CAN, a protocol message with highest priority wins the bus arbitration and maintains the bus control. For minimum message latency and bus control, messages should be transmitted on a priority basis.

Because of the wide applicability of the CAN protocol, developers are faced with the often cumbersome task of dealing with the intricate details of CAN registers. This application note presents a software library that hides the details of CAN registers, and discusses the design of the CAN driver with prioritized Transmit buffer implementation. This software library allows developers to focus their efforts on application logic, while minimizing their interaction with CAN registers.

If the controller has heavy transmission loads, it is advisable to use software Transmit buffers to reduce message latency. Firmware also supports user defined Transmit buffer size. If the defined size of a Transmit buffer is more than that available in hardware (3), the CAN driver will use 14 bytes of general purpose RAM for each extra buffer.

For details about the PIC18 family of microcontrollers, refer to the PIC18CXX8 Data Sheet (DS30475), the PIC18FXX8 Data Sheet (DS41159), and the PICmicro® 18C MCU Family Reference Manual (DS39500).

CAN MODULE OVERVIEW

The PIC18 family of microcontrollers contain a CAN module that provides the same register and functional interface for all PIC18 microcontrollers.

The module features are as follows:
- Implementation of CAN 1.2, CAN 2.0A and CAN 2.0B protocol
- Standard and extended data frames
- 0 - 8 bytes data length
- Programmable bit rate up to 1 Mbit/sec
- Support for remote frame
- Double-buffered receiver with two prioritized received message storage buffers
- Six full (standard/extended identifier) acceptance filters: two associated with the high priority receive buffer, and four associated with the low priority receive buffer
- Two full acceptance filter masks, one each associated with the high and low priority receive buffers
- Three transmit buffers with application specified prioritization and abort capability
- Programmable wake-up functionality with integrated low-pass filter
- Programmable Loopback mode and programmable state clocking supports self-test operation
- Signaling via interrupt capabilities for all CAN receiver and transmitter error states
- Programmable clock source
- Programmable link to timer module for time-stamping and network synchronization
- Low Power SLEEP mode
FIGURE 1: CAN BUFFERS AND PROTOCOL ENGINE BLOCK DIAGRAM

BUFFERS

PROTOCOL ENGINE

TRANSMIT BYTE SEQUENCER

MESSAGE QUEUE CONTROL

RECEIVE SHIFT

CRC CHECK

CRC GENERATOR

TRANSMIT LOGIC

BIT TIMING LOGIC

RECEIVE ERROR COUNTER

TRANSMIT ERROR COUNTER

BUS OFF

PROTOCOL FINITE STATE MACHINE

RX ERR COUNT

TX ERR COUNT

RX ERR CNT

TX ERR CNT

MSGREQ

TXRT

TXERR

TXBUFF

MESSAGE

RX ERR RCNT

TX ERR RCNT

En Pas
Bus Arbitration and Message Latency

In the CAN protocol, if two or more bus nodes start their transmission at the same time, message collision is avoided by bit-wise arbitration. Each node sends the bits of its identifier and monitors the bus level. A node that sends a recessive identifier bit, but reads back a dominant one, loses bus arbitration and switches to Receive mode. This condition occurs when the message identifier of a competing node has a lower binary value (dominant state = logic 0), which results in the competing node sending a message with a higher priority. Because of this, the bus node with the highest priority message wins arbitration, without losing time by having to repeat the message. Transmission of the lower priority message is delayed until all high priority traffic on the bus is finished, which adds some latency to the message transmission. This type of message latency cannot be avoided.

Depending on software driver implementation, additional latency can be avoided by proper design of the driver. If CAN is working at low bus utilization, then the delay in message transmission is not a concern because of arbitration. However, if CAN bus utilization is high, unwanted message latency can be reduced with good driver design.

To illustrate this point, let us examine latency that occurs because of the implementation of driver software. Consider the case when a buffer contains a low priority message in queue and a high priority message is loaded. If no action is taken, the transmission of the high priority message will be delayed until the low priority message is transmitted. A PIC18CXX8 device provides a workaround for this problem.

In PIC18CXX8 devices, it is possible to assign priority to all transmit buffers, which causes the highest priority message to be transmitted first and so on. By setting the transmit buffer priority within the driver software, this type of message latency can be avoided.

Additionally, consider the case where all buffers are occupied with a low priority message and the controller wants to transmit a high priority message. Since all buffers are full, the high priority message will be blocked until one of the low priority messages is transmitted. The low priority message will be sent only after all the high priority messages on the bus are sent. This can considerably delay the transmission of high priority messages.

How then, can this problem be solved? Adding more buffers may help, but most likely the same situation will occur. What then, is the solution? The solution is to unload the lowest priority message from the transmit buffer and save it to a software buffer, then load the transmit buffer with the higher priority message. To maintain bus control, all n Transmit buffers should be loaded with n highest priority messages. Once the transmit buffer is emptied, load the lower priority message into the transmit buffer for transmission. To do this, intelligent driver software is needed that will manage these buffers, based on the priority of the message (Lower binary value of identifier -> Higher priority, see "Terminology Conventions" on page 5). This method minimizes message latency for higher priority messages.
Macro Wrappers

One of the problems associated with assembly language programming is the mechanism used to pass parameters to a function. Before a function can be called, all parameters must be copied to a temporary memory location. This becomes quite cumbersome when passing many parameters to a generalized function. One way to facilitate parameter passing is through the use of “macro wrappers”. This new concept provides a way to overcome the problems associated with passing parameters to functions.

A macro wrapper is created when a macro is used to “wrap” the assembly language function for easy access. In the following examples, macros call the same function, but the way they format the data is different. Depending on the parameters, different combinations of macro wrappers are required to fit the different applications.

Macro wrappers for assembly language functions provide a high level ‘C-like’ language interface to these functions, which makes passing multiple parameters quite simple. Because the macro only deals with literal values, different macro wrappers are provided to suit different calling requirements for the same functions.

For example, if a function is used that copies the data at a given address, the data and address must be supplied to the function.

EXAMPLES

Using standard methods, a call to the assembly language function CopyDataFunc might look like the macro shown in Example 1.

Example 2: CODE WITH MACRO WRAPPER

```
#define Address 0x1234
CopyData 0x56, Address
```

The code in Example 3 shows variable data stored in DataLoc.

Example 3: CODE WITHOUT MACRO WRAPPER

```
#define Address 0x1234

UDATA
TempWord RES 02
DataLoc RES 01

banksel TempWord
movlw low(Address)
movwf TempWord
movlw high(Address)
movwf TempWord+1
banksel DataLoc
movf DataLoc,W
call CopyDataFunc
```

Using a macro wrapper, the code shown in Example 4 supplies the memory address location for data instead of supplying the data value directly.

Example 4: CODE WITH MACRO WRAPPER

```
#define Address 0x1234

UDATA
DataLoc RES 01
CopyData_IDDataLoc, AddressLoc
```

The code in Example 5 shows one more variation using a macro wrapper for the code of both variable arguments.

Example 5: CODE WITH MACRO WRAPPER

```
UDATA
AddressLoc RES 02
DataLoc RES 01

CopyData_ID_IA DataLoc, AddressLoc
```

To summarize, the code examples previously described call for the same function, but the way they format the data is different. By using a macro wrapper, access to assembly functions is simplified, since the macro only deals with literal values.
PIC18XXX8 CAN FUNCTIONS

All PIC18XXX8 CAN functions are grouped into the following three categories:

- Configuration/Initialization Functions
- Module Operation Functions
- Status Check Functions

The following table lists each function by category, which are described in the following sections.

### TABLE 1: FUNCTION INDEX

<table>
<thead>
<tr>
<th>Function</th>
<th>Category</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANInitialize</td>
<td>Configuration/Initialization</td>
<td>6</td>
</tr>
<tr>
<td>CANSetOperationMode</td>
<td>Configuration/Initialization</td>
<td>8</td>
</tr>
<tr>
<td>CANSetOperationModeNoWait</td>
<td>Configuration/Initialization</td>
<td>9</td>
</tr>
<tr>
<td>CANSetBaudRate</td>
<td>Configuration/Initialization</td>
<td>10</td>
</tr>
<tr>
<td>CANSetReg</td>
<td>Configuration/Initialization</td>
<td>12</td>
</tr>
<tr>
<td>CANSendMessage</td>
<td>Module Operation</td>
<td>16</td>
</tr>
<tr>
<td>CANReadMessage</td>
<td>Module Operation</td>
<td>19</td>
</tr>
<tr>
<td>CANAbortAll</td>
<td>Module Operation</td>
<td>22</td>
</tr>
<tr>
<td>CANGetTxErrorCount</td>
<td>Status Check</td>
<td>23</td>
</tr>
<tr>
<td>CANGetRxErrorCount</td>
<td>Status Check</td>
<td>24</td>
</tr>
<tr>
<td>CANIsBusOff</td>
<td>Status Check</td>
<td>25</td>
</tr>
<tr>
<td>CANIsTxPassive</td>
<td>Status Check</td>
<td>26</td>
</tr>
<tr>
<td>CANIsRxPassive</td>
<td>Status Check</td>
<td>27</td>
</tr>
<tr>
<td>CANIsRxReady</td>
<td>Status Check</td>
<td>28</td>
</tr>
<tr>
<td>CANIsTxReady</td>
<td>Status Check</td>
<td>30</td>
</tr>
</tbody>
</table>

### Terminology Conventions

The following applies when referring to the terminology used in this application note.

### TABLE 2: TERMINOLOGY CONVENTIONS

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$xyz$Func</td>
<td>Used for original assembly language functions.</td>
</tr>
<tr>
<td>$xyz$</td>
<td>The macro that will accept all literal values.</td>
</tr>
<tr>
<td>$xyz_I$ (First letter of argument)</td>
<td>The macro that will accept the memory address location for variable implementation.</td>
</tr>
<tr>
<td>$xyz_D$ (First letter of argument)</td>
<td>The macro that expects the user is directly copying the specified parameter at the required memory location by assembly function.</td>
</tr>
</tbody>
</table>
CONFIGURATION/INITIALIZATION FUNCTIONS

CANInitialize
This function initializes the PIC18 CAN module by the given parameters.

Function
CANInitializeFunc

Input
m_SJW
SJW value as defined in the PIC18CXX8 data sheet (must be between 1 and 4).
m_BRP
BRP value as defined in the PIC18CXX8 data sheet (must be between 1 and 64).
m_PHSEG1
PHSEG1 value as defined in the PIC18CXX8 data sheet (must be between 1 and 8).
m_PHSEG2
PHSEG2 value as defined in the PIC18CXX8 data sheet (must be between 1 and 8).
m_PROPSEG2
PROPSEG value as defined in the PIC18CXX8 data sheet (must be between 1 and 8).
m_Flags1
Flag value of type CAN_CONFIG_FLAGS.
This parameter can be any combination (AND'd together) of the following values:

<table>
<thead>
<tr>
<th>TABLE 3: CAN_CONFIG_FLAGS VALUES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
</tr>
<tr>
<td>----------------------</td>
</tr>
<tr>
<td>CAN_CONFIG_DEFAULTS</td>
</tr>
<tr>
<td>CAN_CONFIG_PHSEG2_</td>
</tr>
<tr>
<td>PRG_ON</td>
</tr>
<tr>
<td>CAN_CONFIG_PHSEG2_</td>
</tr>
<tr>
<td>PRG_OFF</td>
</tr>
<tr>
<td>CAN_CONFIG_LINE_</td>
</tr>
<tr>
<td>FILTER_ON</td>
</tr>
<tr>
<td>CAN_CONFIG_LINE_</td>
</tr>
<tr>
<td>FILTER_OFF</td>
</tr>
<tr>
<td>CAN_CONFIG_SAMPLE_</td>
</tr>
<tr>
<td>ONCE</td>
</tr>
<tr>
<td>CAN_CONFIG_SAMPLE_</td>
</tr>
<tr>
<td>THRICE</td>
</tr>
<tr>
<td>CAN_CONFIG_ALL_MSG</td>
</tr>
<tr>
<td>CAN_CONFIG_VALID_XTD_MSG</td>
</tr>
<tr>
<td>CAN_CONFIG_VALID_STD_MSG</td>
</tr>
<tr>
<td>CAN_CONFIG_ALLVALID_MSG</td>
</tr>
</tbody>
</table>

Note 1: If a definition has more than one bit, position symbol provides information for bit masking. ANDing it with the value will mask all the bits except the required one. Status information is not provided, since the user needs to use ANDing and ORing to set/get value.
Return Values
None

Pre-condition
None

Side Effects
All pending CAN messages are aborted.

Remarks
This function does not allow the calling function to specify receive buffer mask and filter values. All mask registers are set to 0x00, which essentially disables the message filter mechanism. If an application requires the message filter operation, it must perform initialization in discrete steps. See CANSetReg for more information.

Macro
CANInitialize SJW, BRP, PHSEG1, PHSEG2, PROPSEG, Flags

Input
SJW
SJW value as defined in the PIC18CXX8 data sheet (must be between 1 and 4).

BRP
BRP value as defined in the PIC18CXX8 data sheet (must be between 1 and 64).

PHSEG1
PHSEG1 value as defined in the PIC18CXX8 data sheet (must be between 1 and 8).

PHSEG2
PHSEG2 value as defined in the PIC18CXX8 data sheet (must be between 1 and 8).

PROPSEG
PROPSEG value as defined in the PIC18CXX8 data sheet (must be between 1 and 8).

Flags
Flag value of type CAN_CONFIG_FLAGS, as previously described.

Example 1
;Initialize for 125kbps@20MHz, all valid messages
CANInitialize 1, 5, 7, 6, 2, CAN_CONFIG_ALL_VALID_MSG

Example 2
;Initialize for 125kbps@20MHz, valid extended message and line filter on
CANInitialize 1, 5, 7, 6, 2, CAN_CONFIG_LINE_FILTER_ON & CAN_CONFIG_VALID_XTD_MSG
**CANSetOperationMode**

This function changes the PIC18 CAN module Operation mode.

**Function**

CANSetOperationModeFunc

**Input**

\[ W \text{ reg} \]

Value of type CAN_OP_MODE.

This parameter must be only one of the following values:

**TABLE 4: CAN_OP_MODE VALUES**

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN_OP_MODE_NORMAL</td>
<td>Normal mode of operation</td>
</tr>
<tr>
<td>CAN_OP_MODE_SLEEP</td>
<td>SLEEP mode of operation</td>
</tr>
<tr>
<td>CAN_OP_MODE_LOOP</td>
<td>Loopback mode of operation</td>
</tr>
<tr>
<td>CAN_OP_MODE_LISTEN</td>
<td>Listen Only mode of operation</td>
</tr>
<tr>
<td>CAN_OP_MODE_CONFIG</td>
<td>Configuration mode of operation</td>
</tr>
</tbody>
</table>

**Return Values**

None

**Pre-condition**

None

**Side Effects**

If CAN_OP_MODE_CONFIG is requested, all pending messages will be aborted.

**Remarks**

This is a blocking function. It waits for a given mode to be accepted by the CAN module and then returns the control. If a non-blocking call is required, see the CANSetOperationModeNoWait function.

**Macro**

CANSetOperationMode OpMode

**Input**

OpMode

Value of type CAN_OP_MODE.

This parameter must be only one of the values listed in Table 4.

**Example**

```
...  
CANSetOperationMode CAN_OP_MODE_CONFIG
; Module is in CAN_OP_MODE_CONFIG mode.
...  
```
CANSetOperationModeNoWait
This macro changes the PIC18 CAN module Operation mode.

Macro
CANSetOperationModeNoWait

Input
W reg
    Value of type CAN_OP_MODE.
    This parameter must be only one of the values listed in Table 4.

Return Values
None

Pre-condition
None

Side Effects
If CAN_OP_MODE_CONFIG is requested, all pending messages will be aborted.

Remarks
This is a non-blocking function. It requests a given mode of operation and immediately returns the control. Caller must make sure that the desired mode of operation is set before performing any mode specific operation. If a blocking call is required, see the CANSetOperationMode function.

Example
...
CANSetOperationModeNoWait CAN_OP_MODE_CONFIG
CANSetBaudRate
This function programs the PIC18 CAN module for given bit rate values.

Function
CANSetBaudRateFunc

Input
m_SJW
SJW value as defined in the PIC18CXX8 data sheet (must be between 1 and 4).
m_BRP
BRP value as defined in the PIC18CXX8 data sheet (must be between 1 and 64).
m_PHSEG1
PHSEG1 value as defined in the PIC18CXX8 data sheet (must be between 1 and 8).
m_PHSEG2
PHSEG2 value as defined in the PIC18CXX8 data sheet (must be between 1 and 8).
m_PROPSEG2
PROPSEG value as defined in the PIC18CXX8 data sheet (must be between 1 and 8).
m_Flags1
Flag value of type CAN_CONFIG_FLAGS.
This parameter can be any combination (AND’d together) of the values listed in Table 3.

Return Values
None

Pre-condition
PIC18 CAN module must be in the Configuration mode or else given values will be ignored.

Side Effects
None

Remarks
None

Macro
CANSetBaudRate SJW, BRP, PHSEG1, PHSEG2, PROPSEG, Flags

Input
SJW
SJW value as defined in the PIC18CXX8 data sheet (must be between 1 and 4).
BRP
BRP value as defined in the PIC18CXX8 data sheet (must be between 1 and 64).
PHSEG1
PHSEG1 value as defined in the PIC18CXX8 data sheet (must be between 1 and 8).
PHSEG2
PHSEG2 value as defined in the PIC18CXX8 data sheet (must be between 1 and 8).
PROPSEG
PROPSEG value as defined in the PIC18CXX8 data sheet (must be between 1 and 8).
Flags
Flag value of type CAN_CONFIG_FLAGS as previously described.
Example

... CANSetOperationMode CAN_OP_MODE_CONFIG

; Set 125bps at 20MHz oscillator frequency
CANSetBaudRate 1, 5, 7, 6, 2,
CAN_CONFIG_SAMPLE_ONCE &
CAN_CONFIG_PHSEG2_PEG_OFF &
CAN_CONFIG_LINE_FILTER_ON

CANSetOperationMode CAN_OP_MODE_NORMAL

...
CANSetReg

This function sets the PIC18 CAN module mask/filter values for the given receive buffer.

Function

CANSetRegFunc

Input

FSR0H:FSR0L
Starting address of 32-bit buffer to be updated.

Reg1:Reg1+3
32-bit mask/filter value that may correspond to 11-bit Standard Identifier or 29-bit Extended Identifier, with binary zero padded on left. Reg1 = LL, Reg1+1 = LH, Reg1+2 = HL and Reg1+3 = HH byte (see "Terminology Conventions" on page 5).

m_Flags1
Type of message Flag.
This parameter must be only one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
<th>Bit(s)</th>
<th>Position</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN_CONFIG_STD_MSG</td>
<td>Standard Identifier message</td>
<td>1</td>
<td>CAN_CONFIG_MSG_TYPE_BIT_NO</td>
<td>Set</td>
</tr>
<tr>
<td>CAN_CONFIG_XTD_MSG</td>
<td>Extended Identifier message</td>
<td>1</td>
<td>CAN_CONFIG_MSG_TYPE_BIT_NO</td>
<td>Clear</td>
</tr>
</tbody>
</table>

Return Values
None

Pre-condition
PIC18 CAN module must be in the Configuration mode or else given values will be ignored.

Side Effects
None

Remarks
None

Macro

CANSetReg RegAddr, val, Flags

Input
RegAddr
This parameter must be only one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN_MASK_B1</td>
<td>Receive Buffer 1 mask value</td>
</tr>
<tr>
<td>CAN_MASK_B2</td>
<td>Receive Buffer 2 mask value</td>
</tr>
<tr>
<td>CAN_FILTER_B1_F1</td>
<td>Receive Buffer 1, Filter 1 value</td>
</tr>
<tr>
<td>CAN_FILTER_B1_F2</td>
<td>Receive Buffer 1, Filter 2 value</td>
</tr>
<tr>
<td>CAN_FILTER_B2_F1</td>
<td>Receive Buffer 2, Filter 1 value</td>
</tr>
<tr>
<td>CAN_FILTER_B2_F2</td>
<td>Receive Buffer 2, Filter 2 value</td>
</tr>
<tr>
<td>CAN_FILTER_B2_F3</td>
<td>Receive Buffer 2, Filter 3 value</td>
</tr>
<tr>
<td>CAN_FILTER_B2_F4</td>
<td>Receive Buffer 2, Filter 4 value</td>
</tr>
</tbody>
</table>
val
32-bit mask/filter value that may correspond to 11-bit Standard Identifier, or 29-bit Extended Identifier, with binary zero padded on left.

Flags
Value of CAN_CONFIG type.
This parameter must be only one of the values listed in Table 6.

Macro
CANSetReg_IF RegAddr, val, FlagsReg

Input
RegAddr
This parameter must be only one of the values listed in Table 6.

val
32-bit mask/filter value that may correspond to 11-bit Standard Identifier, or 29-bit Extended Identifier, with binary zero padded on left.

FlagsReg
Memory Address location that contains the Flag information. This parameter must be only one of the values listed in Table 6.

Macro
CANSetReg_IV RegAddr, Var, Flags

Input
RegAddr
This parameter must be only one of the values listed in Table 6.

Var
Starting address of 32-bit buffer containing mask/filter value. Buffer storage format should be Low -> High (LL:LH:HL:HH) byte (see “Terminology Conventions” on page 5).
32-bit mask/filter value that may correspond to 11-bit Standard Identifier, or 29-bit Extended Identifier, with binary zero padded on left.

Flags
Value of CAN_CONFIG type. This parameter must be only one of the values listed in Table 6.

Macro
CANSetReg_IV_IF RegAddr, Var, FlagsReg

Input
RegAddr
This parameter must be only one of the values listed in Table 6.

Var
Starting address of 32-bit buffer containing mask/filter value. Buffer storage format should be Low -> High (LL:LH:HL:HH) byte (see “Terminology Conventions” on page 5).
32-bit mask/filter value that may correspond to 11-bit Standard Identifier, or 29-bit Extended Identifier, with binary zero padded on left.

FlagsReg
Memory Address location that contains the Flag information. This parameter must be only one of the values listed in Table 6.
Macro
CANSetReg DREG IV_IF Var, FlagsReg

Input
FSR0H:FSR0L
FSR0 contains starting address of 32-bit buffer to be updated. This buffer must be of the mask/filter type. The starting address is the address of the SIDH register for that mask/filter.

Var
Starting address of 32-bit buffer containing mask/filter value. Buffer storage format should be Low -> High (LL:LH:HL:HH) byte (see "Terminology Conventions" on page 5).
32-bit mask/filter value that may correspond to 11-bit Standard Identifier, or 29-bit Extended Identifier, with binary zero padded on left.

FlagsReg
Memory Address location that contains the Flag information. This parameter must be only one of the values listed in Table 6.

Macro
CANSetReg DREG DV_IF FlagsReg

Input
FSR0H:FSR0L
FSR0 contains starting address of 32-bit buffer to be updated. This buffer must be of the mask/filter type. The starting address is the address of the SIDH register for that mask/filter.

Reg1:Reg1+3
Starting address of 32-bit buffer containing mask/filter value. Buffer storage format should be Low -> High (Reg1 = LL:Reg1+1 = LH:Reg1+2 = HL:Reg1+3 = HH) byte (see "Terminology Conventions" on page 5).
32-bit mask/filter value that may correspond to 11-bit Standard Identifier, or 29-bit Extended Identifier, with binary zero padded on left.

FlagsReg
Memory Address location that contains the Flag information. This parameter must be only one of the values listed in Table 6.

Example

```
... CANSetReg CAN_MASK_B1, 0x00000001, CAN_STD_MSG
CANSetReg CAN_MASK_B2, 0x00008001, CAN_XTD_MSG
CANSetReg CAN_FILTER_B1 F1, 0x0000, CAN_STD_MSG
CANSetReg CAN_FILTER_B1 F2, 0x0001, CAN_STD_MSG
CANSetReg CAN_FILTER_B2 F1, 0x8000, CAN_XTD_MSG
CANSetReg CAN_FILTER_B2 F2, 0x8001, CAN_XTD_MSG
CANSetReg CAN_FILTER_B2 F3, 0x8002, CAN_XTD_MSG
CANSetReg CAN_FILTER_B2 F4, 0x8003, CAN_XTD_MSG

UDATA
Flags RES 01

;Memory location Flags contains configuration flags
;information (Indirect Flag info (pointer to Flag))

CANSetReg IF CAN_MASK_B1, 0x00000001, Flags
```
UDATA
IDVal RES 04
;32-bit memory location IDVal contains 32-bit mask
;value (Indirect value info (pointer to value))
CANSetReg_IV CAN_MASK_B2, IDVal, CAN_XTD_MSG

UDATA
Flags RES 01
IDVal RES 04
;32-bit memory location IDVal contains 32-bit mask
;value (Indirect value info (pointer to value))
;Memory location Flags contains configuration flags
;information (Indirect Flag info (pointer to Flag))
CANSetReg_IV_IF CAN_FILTER_B1_F1, IDVal, Flags

UDATA
Flags RES 01
IDVal RES 04
;32-bit memory location IDVal contains 32-bit mask
;value (Indirect value info (pointer to value))
;Memory location Flags contains configuration flags
;information (Indirect Flag info (pointer to Flag))
movlw low(RxF0SIDH)
movwf FSR0L
movlw high(RxF0SIDH)
movwf FSR0H
;Because of above or some other operation FSR0
;contains starting address of buffer (xxxxSIDH reg.)
;for mask/filter value storage.
CANSetReg_DREG_IV_IF IDVal, Flags

UDATA
Flags RES 01
;32-bit memory location IDVal contains 32-bit mask
;value (Indirect value info (pointer to value))
;Memory location Flags contains configuration flags
;information (Indirect Flag info (pointer to Flag))
movlw low(RxF0SIDH)
movwf FSR0L
movlw high(RxF0SIDH)
movwf FSR0H
;Because of above or some other operation FSR0
;contains starting address of buffer (xxxxSIDH reg.)
;for mask/filter value storage.
;Reg1:Reg1+3 contains 32-bit ID value.
CANSetReg_DREG_DV_IF Flags
MODULE OPERATION FUNCTIONS

CANSendMessage
This function copies the given message to one of the empty transmit buffers and marks it as ready to be transmitted.

Function
CANSendMessageFunc

Input

Reg1:Reg1+3
32-bit identifier value that may correspond to 11-bit Standard Identifier, or 29-bit Extended Identifier, with binary zero padded on left. Exact number of bits to use depends on m_TxFlags. Buffer storage format should be Low -> High (Reg1 = LL:Reg1+1 = LH:Reg1+2 = HL:Reg1+3 = HH) byte (see “Terminology Conventions” on page 5).

FSR1H:FSR1L
Starting address of data buffer.

m_DataLength
Number of bytes to send.

m_TxFlags
Value of type CAN_TX_MSG_FLAGS.
This parameter can be any combination (AND’d together) of the following group values:

TABLE 7: CAN_TX_MSG_FLAGS VALUES

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
<th>Bit(s)</th>
<th>Position</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN_TX_STD_FRAME</td>
<td>Standard Identifier message</td>
<td>1</td>
<td>CAN_TX_FRAME_BIT_NO</td>
<td>Set</td>
</tr>
<tr>
<td>CAN_TX_XTD_FRAME</td>
<td>Extended Identifier message</td>
<td>1</td>
<td>CAN_CONFIG_MSG_TYPE_BIT_NO</td>
<td>Clear</td>
</tr>
<tr>
<td>CAN_TX_NO_RTR_FRAME</td>
<td>Regular message - not RTR</td>
<td>1</td>
<td>CAN_TX_RTR_BIT_NO</td>
<td>Set</td>
</tr>
<tr>
<td>CAN_TX_RTR_FRAME</td>
<td>RTR message</td>
<td>1</td>
<td>CAN_TX_RTR_BIT_NO</td>
<td>Clear</td>
</tr>
</tbody>
</table>

Return Values
W = 1, if the given message was successfully placed in one of the empty transmit buffers.
W = 0, if all transmit buffers were full.

Pre-condition
None

Side Effects
None

Remarks
None

Macro
CANSendMessage msgID, DataPtr, DataLength, Flags

msgID
32-bit identifier value that may correspond to 11-bit Standard Identifier, or 29-bit Extended Identifier, with binary zero padded on left. Exact number of bits to use depends on Flags.
DataPtr

Pointer to zero or more of data bytes to send.

DataLength

Number of bytes to send.

Flags

Value of type CAN_TX_MSG_FLAGS.

This parameter can be any combination (AND’d together) of the group values listed in Table 7.

Macro

CANSendMessage_IID_IDL_IF  msgIDPtr, DataPtr, DataLngthPtr, FlagsReg

msgIDPtr

Starting address of memory location containing 32-bit message ID. Buffer storage format should be Low -> High (LL:LH:HL:HH) byte (see "Terminology Conventions" on page 5).

32-bit identifier value that may correspond to 11-bit Standard Identifier, or 29-bit Extended Identifier, with binary zero padded on left. Exact number of bits to use depends on FlagsReg.

DataPtr

Pointer to zero or more of data bytes to send.

DataLngth

Memory Address location having data of number of bytes to send.

FlagsReg

Memory Address location that contains the Flag information. Flags must be of type CAN_TX_MSG_FLAGS.

This parameter can be any combination (AND’d together) of the group values listed in Table 7.

Example A

UDATA

MessageData RES 02

call  CANIsTxReady
bnc TxNotRdy
movlw 0x01
movwf MessageData ;Copy Data byte 1
movlw 0x02
movwf MessageData+1 ;Copy Data byte 2

CANSendMessage 0x20,
MessageData,
2,
CAN_TX_STD_FRAME &
CAN_TX_NO_RTR_FRAME

TxNotRdy:
;All Buffer are full, Try again
Example B

UDATA
MessageData RES 02

movlw 0x01
movwf MessageData ;Copy Data byte 1
movlw 0x02
movwf MessageData+1 ;Copy Data byte 2

CANSendMessage 0x20,
MessageData, 2,
CAN_TX_STD_FRAME &
CAN_TX_NO_RTR_FRAME
addlw 0x00 ;Check for return value 0 in W
bz TxNotRdy ;Buffer Full, Try again
Message is copied in buffer for Transmission. It will be
transmitted based on priority and pending messages in
buffers
nop ;Application specific code

TxNotRdy:
;All Buffer are full, Message was not copied in buffer for
;Transmission...

UDATA
MessageData RES 02
Idval RES 04
DataLength RES 01
Flags RES 01

call CANIsTxReady
bnc TxNotRdy
movlw 0x01
movwf MessageData ;Copy Data byte 1
movlw 0x02
movwf MessageData+1 ;Copy Data byte 2
movwf DataLength ;Set Data length to 2

;IDval contains 32-bit message ID and Flags
;contains TX Flags info.
CANSendMessage_IID_IDL_IF
Idval,
MessageData,
DataLength,
Flags

TxNotRdy:
;All Buffer are full, Try again
**CANReadMessage**

This function copies the new available message to the user supplied buffer.

**Function**

CANReadMessageFunc

**Input**

FSR0H:FSR0L

Starting address for received data storage.

**Output**

Temp32Data:Temp32Data+3

Received Message ID. Buffer storage format is Low -> High (LL:LH:HL:HH) byte (see "Terminology Conventions" on page 5).

32-bit identifier value that may correspond to 11-bit Standard Identifier, or 29-bit Extended Identifier, with binary zero padded on left.

**DataLen**

Number of bytes received.

**m_RxFlags**

Value of type CAN_RX_MSG_FLAGS.

This parameter can be any combination (AND'd together) of the following values. If a flag bit is set, the corresponding meaning is TRUE; if cleared, the corresponding meaning is FALSE.

**TABLE 8: CAN_RX_MSG_FLAGS VALUES**

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
<th>Bit(s)</th>
<th>Position</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN_RX_FILTER_1, CAN_RX_FILTER_2, CAN_RX_FILTER_3, CAN_RX_FILTER_4, CAN_RX_FILTER_5, CAN_RX_FILTER_6</td>
<td>Receive buffer filter that caused this message to be accepted.</td>
<td>3</td>
<td>CAN_RX_FILTER_BITS</td>
<td></td>
</tr>
<tr>
<td>CAN_RX_OVERFLOW</td>
<td>Receive buffer overflow condition</td>
<td>1</td>
<td>CAN_RX_OVERFLOW_BIT_NO</td>
<td>Set</td>
</tr>
<tr>
<td>CAN_RX_INVALID_MSG</td>
<td>Invalid message</td>
<td>1</td>
<td>CAN_RX_INVALID_MSG_BIT_NO</td>
<td>Set</td>
</tr>
<tr>
<td>CAN_RX_XTD_FRAME</td>
<td>Extended message</td>
<td>1</td>
<td>CAN_RX_XTD_FRAME_BIT_NO</td>
<td>Set</td>
</tr>
<tr>
<td>CAN_TX_RTR_FRAME</td>
<td>RTR message</td>
<td>1</td>
<td>CAN_RX_RTR_FRAME_BIT_NO</td>
<td>Set</td>
</tr>
<tr>
<td>CAN_RX_DBL_BUFFERED</td>
<td>This message was double-buffered</td>
<td>1</td>
<td>CAN_RX_DBL_BUFFERED_BIT_NO</td>
<td>Set</td>
</tr>
</tbody>
</table>

**Return Values**

W = 1, if new message was copied to given buffer.

W = 0, if no new message was found.

**Pre-condition**

id, Data, DataLen and MsgFlags pointers must point to valid/desired memory locations.

**Side Effects**

None

**Remarks**

This function will fail if there are no new message(s) to read. Caller may check the return value to determine new message availability, or may call CANIsRxReady function.
Macro

CANReadMessage msgIDPtr, DataPtr, DataLngth, Flags

msgIDPtr
Starting address of 32-bit buffer for message ID storage. Buffer storage format is Low -> High (LL:LH:HL:HH) byte (see "Terminology Conventions" on page 5). 32-bit identifier value that may correspond to 11-bit Standard Identifier, or 29-bit Extended Identifier, with binary zero padded on left.

DataPtr
Starting address of data buffer for storage of received data byte.

DataLngth
Address of the memory location for storage of number of bytes received.

Flags
Address of the memory location for storage of number of bytes received.

Value of type CAN_RX_MSG_FLAGS.
This parameter can be any combination (AND’d together) of the values listed in Table 8. If a flag bit is set, the corresponding meaning is TRUE; if cleared, the corresponding meaning is FALSE.

Example A

UDATA
NewMessage RES 04
NewMessageData RES 08
NewMessageLen RES 01
NewMessageFlags RES 01
RxFilterMatch RES 01

call CANIsRxReady
bnc RxNotRdy

CANReadMessage NewMessage,
NewMessageData,
NewMessageLen,
NewMessageFlags

banksel NewMessageFlags

btfsc NewMessageFlags,CAN_RX_OVERFLOW_BIT_NO
bra RxOvrFlow ;Branch to Logic for Rx
;overflow occurred.

btfsc NewMessageFlags,CAN_RX_INVALID_MSG_BIT_NO
bra RxInvldMsg ;Branch to Logic for Invalid
;Message received

btfsc NewMessageFlags,CAN_RX_XTD_FRAME_BIT_NO
nop ;Logic for Extended frame
;nop ;Else logic for standard
;nop ;frame received

btfsc NewMessageFlags,CAN_RX_RTR_FRAME_BIT_NO
bra RxRTRFrame ;Branch to Logic for RTR
;frame received

nop ;Regular frame received
movlw      CAN_RX_FILTER_BITS
andwf      NewMessageFlags,W
movwf      RxFilterMatch   ;Save matched Filter ;number

RxNotReady:
;Receive buffer is empty, Wait for new message

Example B

UDATA
NewMessage    RES   04
NewMessageData  RES   08
NewMessageLen   RES   01
NewMessageFlags RES   01
RxFilterMatch   RES   01

CANReadMessage NewMessage,
               NewMessageData,
               NewMessageLen,
               NewMessageFlags

xorlw      0x01   ;Check for Success code
bnz       RxNotReady

banksel    NewMessageFlags

btfsc     NewMessageFlags,CAN_RX_OVERFLOW_BIT_NO
bra       RxOvrFlow   ;Branch to Logic for Rx
          ;overflow occurred.

btfsc     NewMessageFlags,CAN_RX_INVALID_MSG_BIT_NO
bra       RxInvldMsg  ;Branch to Logic for Invalid
          ;Message received

btfsc     NewMessageFlags,CAN_RX_XTD_FRAME_BIT_NO
nop       ;Logic for Extended frame
          ;received
nop       ;Else logic for standard
          ;frame received

btfsc     NewMessageFlags,CAN_RX_RTR_FRAME_BIT_NO
bra       RxRTRFrame  ;Branch to Logic for RTR
          ;frame received
nop       ;Regular frame received

movlw      CAN_RX_FILTER_BITS
andwf      NewMessageFlags,W
movwf      RxFilterMatch   ;Save matched Filter ;number

RxNotReady:
;Receive buffer is empty, Wait for new message
CANAbortAll

This macro aborts all pending messages from the PIC18 CAN module. See the PIC18CXX8 Data Sheet for rules regarding message abortion.

Macro
CANAbortAll

Input
None

Return Values
None

Pre-condition
None

Side Effects
None

Remarks
None

Example
...
CANAbortAll
...

STATUS CHECK FUNCTIONS

CANGetTxErrorCount

This macro returns the PIC18 CAN transmit error count, as defined by BOSCH CAN Specifications, in WREG. See the PIC18CXX8 Data Sheet for more information.

**Macro**

`CANGetTxErrorCount`

**Input**

None

**Return Values**

WREG contains the current value of transmit error count.

**Pre-condition**

None

**Side Effects**

None

**Remarks**

None

**Example**

```
UDATA
TxErrorCount RES 01
...
CANGetTxErrorCount ;Returns error count in W
banksel TxErrorCount
movwf TxErrorCount
...
```
CANGetRxErrorCount

This macro returns the PIC18 CAN receive error count, as defined by BOSCH CAN Specifications, in WREG. See the PIC18CXX8 Data Sheet for more information.

Macro
CANGetRxErrorCount

Input
None

Return Values
WREG contains the current value of receive error count.

Pre-condition
None

Side Effects
None

Remarks
None

Example
UDATA
RxErrorCount RES 01
...
CANGetRxErrorCount ; Returns error count in W
banksel RxErrorCount
movwf RxErrorCount
...

**CANIsBusOff**

This function returns the PIC18 CAN module On/Off state.

**Function**

CANIsBusOff

**Input**

None

**Return Values**

Carry C = 1, if PIC18 CAN module is in the Bus Off state.
Carry C = 0, if PIC18 CAN module is in the Bus On state.

**Pre-condition**

None

**Side Effects**

None

**Remarks**

None

**Example**

```plaintext
... call CANIsBusOff()
bnc CANBusNotOff
nop ;CAN Module is in Bus off state

CANBusNotOff
nop ;CAN Module isn’t in Bus off state
```
CANIsTxPassive

This function returns the PIC18 CAN transmit error status, as defined by BOSCH CAN Specifications. See the PIC18CXX8 Data Sheet for more information.

Function
CANIsTxPassive

Input
None

Return Values
Carry C = 1, if the PIC18 CAN module is in transmit error passive state.
Carry C = 0, if the PIC18 CAN module is not in transmit error passive state.

Pre-condition
None

Side Effects
None

Remarks
None

Example

...  call CANIsTxPassive()
bnc CANIsNotTxPassive
nop ;CAN Module is in Transmit Passive
     ;state

CANIsNotTxPassive
nop ;CAN Module isn't in Tx Passive
     ;state

...
**CANIsRxPassive**

This function returns the PIC18 CAN receive error status, as defined by BOSCH CAN Specifications. See the PIC18CXX8 Data Sheet for more information.

**Function**

CANIsRxPassive

**Input**

None

**Return Values**

- Carry C = 1, if the PIC18 CAN receive module is in receive error passive state.
- Carry C = 0, if the PIC18 CAN receive module is not in receive error passive state.

**Pre-condition**

None

**Side Effects**

None

**Remarks**

None

**Example**

```assembly
... call CANIsRxPassive()
  bnc CANIsNotRxPassive
  nop ;CAN Module is in Receive Passive
       ;state, Do Something

  CANIsNotRxPassive
  nop ;CAN Module isn’t in Rx Passive
       ;state

...```

© 2002 Microchip Technology Inc.
**CANIsRxReady**

This function returns the PIC18 CAN receive buffer(s) readiness status.

**Function**

CANIsRxReady

**Input**

None

**Return Values**

- Carry C = 1, if at least one of the PIC18 CAN receive buffers is full.
- Carry C = 0, if none of the PIC18 CAN receive buffers are full.

**Pre-condition**

None

**Side Effects**

None

**Remarks**

None

**Example**

```assembly
UDATA
NewMessage   RES  04
NewMessageData RES  08
NewMessageLen  RES  01
NewMessageFlags RES  01
RxFilterMatch  RES  01

call CANIsRxReady
bnc RxNotRdy

CANReadMessage NewMessage,
               NewMessageData,
               NewMessageLen,
               NewMessageFlags

banksel NewMessageFlags

btfsc NewMessageFlags,CAN_RX_OVERFLOW_BIT_NO
    bra RxOvrFlow ;Branch to Logic for Rx overflow occurred.

btfsc NewMessageFlags,CAN_RX_INVALID_MSG_BIT_NO
    bra RxInvldMsg ;Branch to Logic for Invalid Message received

btfsc NewMessageFlags,CAN_RX_XTD_FRAME_BIT_NO
    nop ;Logic for Extended frame received

btfsc NewMessageFlags,CAN_RX_RTR_FRAME_BIT_NO
    bra RxRTRFrame ;Branch to Logic for RTR frame received
```
nop ;Regular frame received

movlw CAN_RX_FILTER_BITS
andwf NewMesageFlags,W
movwf RxFilterMatch ;Save matched Filter ;number

RxNotReady
;Receive buffer is empty, wait for new message
...

CANIsTxReady
This function returns the PIC18 CAN transmit buffer(s) readiness status.

Function
CANIsTxReady

Input
None

Return Values
Carry C = 1, if at least one of the PIC18 CAN transmit buffers is empty.
Carry C = 0, if none of the PIC18 CAN transmit buffers are empty.

Pre-condition
None

Side Effects
None

Remarks
None

Example

<table>
<thead>
<tr>
<th>UDATA</th>
<th>RES</th>
<th>02</th>
</tr>
</thead>
<tbody>
<tr>
<td>MessageData</td>
<td>RES</td>
<td>02</td>
</tr>
</tbody>
</table>

```assembly
    call CANIsTxReady
    bnc TxNotRdy
    movlw 0x01
    movwf MessageData
    movlw 0x02
    movwf MessageData+1
    CANSendMessage 0x20,
                    MessageData,
                    2,
                    CAN_TX_STD_FRAME &
                    CAN_TX_NO_RTR_FRAME

TxNotRdy:
; All Buffer are full, Try again
...```
PIC18 CAN FUNCTIONS
ORGANIZATION AND USAGE

These functions were developed for Microchip MPLAB® using MPLINK™ Object Linker; however, they can easily be ported to any assembler supporting linking for PIC18 devices.

Source code for the PIC18XXX8 CAN module is divided into the following three files:

- CAN18xx8.asm
- CAN18xx8.inc
- CANDef.inc

To employ these CAN functions in your project, perform the following steps:

1. Copy “CAN18xx8.asm”, “CANDef.inc” and “CAN18xx8.inc” files to your project source directory.
2. Include “CAN18xx8.asm” file in your project as an asm source file.
3. Add #include “CAN18xx8.inc” line in each source file that will be calling CAN routines.
4. By default, CAN interrupt priority is high. CAN interrupt can be assigned a lower priority by defining CANIntLowPrior in CANDef.inc. User must call CANISR function from the respective interrupt vector to service CAN transmit interrupt.
5. Firmware implements user defined size of transmit buffer. User can define size of software transmit buffer to increase the buffer size that is available in hardware (3). In that case, it will use 14 bytes of general purpose RAM for each extra buffer. User should define required extra software buffer size in CANDef.inc at MAX_TX_SOFT_BUFFER.

FIGURE 2: PIC18 CAN MODULE INITIALIZATION PROCEDURE

```
Start Can Module Initialization

Initialize CAN Module by Calling CANInitialize with Desired Bit Rate and CONFIG Flags

Is Message Filtering Required? Yes No

End CAN Initialization

Set CONFIG Operation Mode by Calling CANSetOperationMode

Set Desired Mask and Filter Values by Calling CANSetMask and CANSet Filter

Set Normal Operation Mode by Calling CANSetOperationMode

End CAN Initialization
```
SAMPLE APPLICATION PROGRAM USING THE PIC18 CAN LIBRARY

An application program that uses the PIC18 CAN functions must follow certain initialization steps, as shown in Figure 2.

The following is a portion of a sample application program that requires all CAN Standard Identifier messages to be accepted.

EXAMPLE 6: ALL IDENTIFIER MESSAGES ACCEPTED

```
;UDATA
NewMessage    RES  04
NewMessageData RES  08
NewMessageLen  RES  01
NewMessageFlags RES  01
RxFilterMatch  RES  01
MessageData    RES  02

;Application specific initialization code

;Initialize CAN module with no message filtering
CANInitialize 1, 5, 7, 6, 2, CAN_CONFIG_ALL_VALID_MSG

Loop:
call CANIsRxReady ;Check for CAN message
bnc RxNotRdy

CANReadMessage NewMessage,
    NewMessageData,
    NewMessageLen,
    NewMessageFlags

banksel NewMessageFlags

btfsr NewMessageFlags,CAN_RX_OVERFLOW_BIT_NO
bra RxOvrFlow ;Branch to Logic for Rx
            ;overflow occurred.

btfsr NewMessageFlags,CAN_RX_INVALID_MSG_BIT_NO
bra RxInvldMsg ;Branch to Logic for Invalid
            ;Message received

btfsr NewMessageFlags,CAN_RX_XTD_FRAME_BIT_NO
nop ;Logic for Extended frame received

nop ;Else logic for standard frame received

btfsr NewMessageFlags,CAN_RX_RTR_FRAME_BIT_NO
bra RxRTRFrame ;Branch to Logic for RTR
            ;frame received

nop ;Regular frame received

movlw CAN_RX_FILTER_BITS
andwf NewMessageFlags,W
movwf RxFilterMatch ;Save matched Filter number

RxNotReady:
;Receive buffer is empty, Wait for new message
```
EXAMPLE 6: ALL IDENTIFIER MESSAGES ACCEPTED (Continued)

; Process received message
...

; Transmit a message due to previously received message or
; due to application logic itself.
call CANIsTxReady
bnc TxNotRdy
movlw 0x01
movwf MessageData ; Copy Data byte 1
movlw 0x02
movwf MessageData+1 ; Copy Data byte 2

CANSendMessage 0x20,
MessageData,
2,
CAN_TX_STD_FRAME &
CAN_TX_NO_RTR_FRAME

TxNotRdy:
; All Buffer are full, Try again

; Other application specific logic
...

goto Loop ; Do this forever

; End of program
The following is a portion of a sample application program that requires only a specific group of CAN Standard Identifier messages to be accepted:

**EXAMPLE 7: SPECIFIC IDENTIFIER MESSAGES ACCEPTED**

```
UDATA
NewMessage   RES  04
NewMessageData RES  08
NewMessageLen  RES  01
NewMessageFlags RES  01
RxFilterMatch RES  01
MessageData   RES  02

;Application specific initialization code

;Initialize CAN module with no message filtering
CANInitialize 1, 5, 7, 6, 2, CAN_CONFIG_ALL_VALID_MSG

;Set OperationMode
CANSetOperationMode CAN_OP_MODE_CONFIG

;Set Buffer Mask 1 value
CANSetReg CAN_MASK_B1, 0x0000000f, CAN_STD_MSG

;Set Buffer Mask 2 value
CANSetReg CAN_MASK_B2, 0x000000f0, CAN_STD_MSG

;Set Buffer 1, Filter 1 value
CANSetReg CAN_FILTER_B1_F1, 0x00000001, CAN_STD_MSG

;Set Buffer 1, Filter 2 value
CANSetReg CAN_FILTER_B1_F2, 0x00000002, CAN_STD_MSG

;Set Buffer 2, Filter 1 value
CANSetReg CAN_FILTER_B2_F1, 0x00000010, CAN_STD_MSG

;Set Buffer 2, Filter 2 value
CANSetReg CAN_FILTER_B2_F2, 0x00000020, CAN_STD_MSG

;Set Buffer 3, Filter 3 value
CANSetReg CAN_FILTER_B2_F3, 0x00000030, CAN_STD_MSG

;Set Buffer 4, Filter 4 value
CANSetReg CAN_FILTER_B2_F4, 0x00000040, CAN_STD_MSG

Loop:
    call CANIsRxReady ;Check for CAN message
    bnc RxNotRdy
    CANReadMessage NewMessage, NewMessageData, NewMessageLen, NewMessageFlags
    banksel NewMessageFlags
    btfsb NewMessageFlags,CAN_RX_OVERFLOW_BIT_NO
    bra RxOvrFlow ;Branch to Logic for Rx
    ;overflow occurred.
    btfsb NewMessageFlags,CAN_RX_INVALID_MSG_BIT_NO
    bra RxInvldMsg ;Branch to Logic for Invalid
    ;Message received
```
EXAMPLE 7: SPECIFIC IDENTIFIER MESSAGES ACCEPTED (Continued)

```
btfsc NewMessageFlags,CAN_RX_XTD_FRAME_BIT_NO
  nop ; Logic for Extended frame received
  nop ; Else logic for standard frame received
btfsc NewMessageFlags,CAN_RX_RTR_FRAME_BIT_NO
  bra RxRTRFrame ; Branch to Logic for RTR frame received
  nop ; Regular frame received
movlw CAN_RX_FILTER_BITS
andwf NewMessageFlags,W
movwf RxFilterMatch ; Save matched Filter number

RxNotReady:
  ; Receive buffer is empty, Wait for new message

  ; Process received message

  ; Transmit a message due to previously received message or
due to application logic itself.
call CANIsTxReady
  bnc TxNotRdy
  movlw 0x01
  movwf MessageData ; Copy Data byte 1
  movlw 0x02
  movwf MessageData+1 ; Copy Data byte 2
  CANSendMessage 0x20, MessageData, 2,
  CAN_TX_STD_FRAME &
  CAN_TX_NO_RTR_FRAME

TxNotRdy:
  ; All Buffers are full, Try again

  ; Other application specific logic

  goto Loop ; Do this forever

; End of program
```
CONCLUSION

The CAN library provided in this application note can be used in any application program that needs an interrupt controlled mechanism to implement CAN transmission and a simple polling mechanism to implement CAN reception. This library can be used as a reference to create prioritized receive buffer CAN communication. Macro wrappers, provided for the functions described, may not be sufficient for all requirements. Using the code provided in this application note, users can develop their own wrappers to fit their needs.

APPENDIX A: SOURCE CODE

Due to size considerations, the complete source code for this application note is not included in the text.

A complete version of the source code, with all required support files, is available for download as a Zip archive from the Microchip web site, at:

www.microchip.com
Note the following details of the code protection feature on PICmicro® MCUs.

- The PICmicro family meets the specifications contained in the Microchip Data Sheet.
- Microchip believes that its family of PICmicro microcontrollers is one of the most secure products of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the PICmicro microcontroller in a manner outside the operating specifications contained in the data sheet. The person doing so may be engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as “unbreakable”.
- Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our product.

If you have any further questions about this matter, please contact the local sales office nearest to you.

Trademarks

The Microchip name and logo, the Microchip logo, KEELoq, MPLAB, PIC, PICmicro, PICSTART and PRO MATE are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

FilterLab, microID, MXDEV, MXLAB, PICMASTER, SEEVAL and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A.

dsPIC, dsPICDEM.net, ECONOMONITOR, FanSense, FlexROM, fuzzyLAB, In-Circuit Serial Programming, ICSP, ICEPIC, microPort, Migratable Memory, MPASM, MPLIB, MPLINK, MPSIM, PICC, PICDEM, PICDEM.net, rPIC, Select Mode and Total Endurance are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

Serialized Quick Turn Programming (SQTP) is a service mark of Microchip Technology Incorporated in the U.S.A.

All other trademarks mentioned herein are property of their respective companies.

© 2002, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.

Printed on recycled paper.
WORLDWIDE SALES AND SERVICE

AMERICAS
Corporate Office
2335 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200 Fax: 480-792-7277
Technical Support: 480-792-7627
Web Address: http://www.microchip.com

Rocky Mountain
2335 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7966 Fax: 480-792-4338

Atlanta
500 Sugar Mill Road, Suite 200
Atlanta, GA 30350
Tel: 770-640-0034 Fax: 770-640-0307

Boston
2 Lan Drive, Suite 120
Westford, MA 01886
Tel: 978-692-3848 Fax: 978-692-3821

Chicago
333 Pierce Road, Suite 180
Itasca, IL 60143
Tel: 630-285-0071 Fax: 630-285-0075

Dallas
4570 Westgrove Drive, Suite 160
Addison, TX 75001
Tel: 972-818-7423 Fax: 972-818-2924

Detroit
Tri-Atria Office Building
32255 Northwestern Highway, Suite 190
Farmington Hills, MI 48334
Tel: 248-538-2250 Fax: 248-538-2260

Kokomo
2767 S. Albright Road
Kokomo, Indiana 46902
Tel: 765-864-8360 Fax: 765-864-8387

Los Angeles
18201 Von Karman, Suite 1090
Irvine, CA 92612
Tel: 949-263-1888 Fax: 949-263-1338

New York
150 Motor Parkway, Suite 202
Hauppauge, NY 11788
Tel: 631-273-2705 Fax: 631-273-5395

San Jose
Microchip Technology Inc.
2107 North First Street, Suite 590
San Jose, CA 95131
Tel: 408-436-7950 Fax: 408-436-7955

Toronto
6285 Northview Drive, Suite 108
Mississauga, Ontario L4V 1X5, Canada
Tel: 905-673-0699 Fax: 905-673-6509

ASIA/PACIFIC
Australia
Microchip Technology Australia Pty Ltd
Suite 22, 41 Rawson Street
Epping 2121, NSW
Australia
Tel: 61-2-9868-6733 Fax: 61-2-9868-6755

China - Beijing
Microchip Technology Consulting (Shanghai) Co., Ltd., Beijing Liaison Office
Unit 915
Beijing, 100027, No. China
Tel: 86-10-85282100 Fax: 86-10-85282104

China - Chengdu
Microchip Technology Consulting (Shanghai) Co., Ltd., Chengdu Liaison Office
Rm. 2401, 24th Floor, Ming Xing Financial Tower
No. 88 TIDU Street
Chengdu 610016, China
Tel: 86-28-86766200 Fax: 86-28-86766599

China - Fuzhou
Microchip Technology Consulting (Shanghai) Co., Ltd., Fuzhou Liaison Office
Unit 28F, World Trade Plaza
No. 71 Wusi Road
Fuzhou 350001, China
Tel: 86-591-7503506 Fax: 86-591-7503521

Japan
Microchip Technology Japan K.K.
Benex S-1 6F
3-18-20, Shin yokohama
Kohoku-Ku, Yokohama-shi
Kanagawa, 222-0033, Japan
Tel: 81-45-471-6166 Fax: 81-45-471-6122

Korea
Microchip Technology Korea
168-1, Youngbo Bldg. 3 Floor
Samsung-Dong, Gangnam-Ku
Seoul, Korea 135-882
Tel: 82-2-554-7200 Fax: 82-2-558-5934

Singapore
Microchip Technology Singapore Pte Ltd.
200 Middle Road
#07-02 Prime Centre
Singapore, 189980
Tel: 65-6334-8870 Fax: 65-6334-8850

Taiwan
Microchip Technology (Barbados) Inc.,
Taiwan Branch
11F-3, No. 207
Tung Hua North Road
Taipei, 105, Taiwan
Tel: 886-2-2717-7175 Fax: 886-2-2545-0139

EUROPE

Austria
Microchip Technology Austria GmbH
Durlisstrasse 2
A-4600 Wels
Austria
Tel: 43-7242-2244-399 Fax: 43-7242-2244-393

Denmark
Microchip Technology Nordic ApS
Regus Business Centre
Lautrup høj 1-3
Ballerpur DK-2750 Denmark
Tel: 45 4420 9895 Fax: 45 4420 9910

France
Microchip Technology SARL
Parc d’Activite du Moulin de Massy
43 Rue du Saule Trapu
Batiment A - ler Etage
91300 Massy, France
Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79

Germany
Microchip Technology GmbH
Steinheilstrasse 10
D-85737 Ismaning, Germany
Tel: 49-89-627-144-0 Fax: 49-89-627-144-44

Italy
Microchip Technology SRL
Centro Direzionale Colleoni
Palazzo Taurus 1 V. Le Colleoni 1
20041 Agrate Brianza
Milan, Italy
Tel: 39-039-65791-1 Fax: 39-039-6899883

United Kingdom
Microchip Ltd.
505 Eskdale Road
Winnersh Triangle
Wokingham
Berkshire, England RG41 5TU
Tel: 44 118 921 5869 Fax: 44-118 921-5820

08/01/02