INTRODUCTION

This technical brief provides a general overview of how Controller Area Network (CAN) interrupt events are handled by the dsPIC30F Digital Signal Controller (DSC) hardware, including how a typical user application might process these events. It also describes an application scenario in which some special interrupt handling methods may need to be implemented by the user applications.

CAN INTERRUPTS OVERVIEW

A dsPIC30F DSC device can have up to two CAN modules, referred to as CAN1 and CAN2. Each CAN module generates a different interrupt, with its own dedicated interrupt vector, interrupt enable bit (C1IE or C2IE), interrupt status flag (C1IF or C2IF), and interrupt priority bits (C1IP<2:0> or C2IP<2:0>).

A CAN interrupt can be generated by one or more of the eight possible events. Each of these eight events has its own interrupt event enable bit (in the C1INTE or C2INTE register) and its own interrupt event flag (in the C1INTF or C2INTF register). A certain CAN event can cause an interrupt only if the corresponding event’s interrupt enable bit is set. However, the CAN event flag is set when the event occurs whether the corresponding event’s interrupt enable bit is set or not.

The eight event flags that can trigger a CAN interrupt in the dsPIC30F are as follows:

- RX0IF: Data received in receive buffer 0
- RX1IF: Data received in receive buffer 1
- TX0IF: Data has been completely transmitted from transmit buffer 0
- TX1IF: Data has been completely transmitted from transmit buffer 1
- TX2IF: Data has been completely transmitted from transmit buffer 2
- WAKIF: CAN module has woken up from Disable or Sleep mode on detection of CAN bus activity
- IVRIF: Invalid Message Received
- ERRIF: This error event is a logical OR of the following eight possible transmission or reception error conditions
  - EWARN: Transmitter or receiver warning
  - TXWAR: Transmitter error warning
  - RXBP: Receiver error passive state
  - TXBP: Transmitter error passive state
  - TXBO: Transmitter bus off state
  - RX1OVR: Receive buffer 1 data overflow
  - RX0OVR: Receive buffer 0 data overflow

Refer to the “dsPIC30F Family Reference Manual” (DS70046), which is available from the Microchip Web site (www.microchip.com) for details about each of the individual error conditions and states.

The individual event flag states in the end of the previous instruction cycle and the beginning of the current instruction cycle, are compared by the CAN interrupt event handling logic using a logical XOR operation. Therefore, an interrupt request is generated only if a previously clear individual interrupt event is set in the current instruction cycle.

Also, the individual event flags and enable bits are decoded to generate the ICODE<2:0> status bits that indicate which interrupt event has occurred most recently. For example, the ICODE<2:0> bits can be used in an Interrupt Service Routine (ISR) to conditionally execute (using a jump table) different event handlers.
CAN INTERRUPT HANDLING IN A USER APPLICATION

A typical CAN ISR can be expected to perform the following sequence of operations:

- Inspect each interrupt event flag in the lower byte of the C1INTF (or C2INTF) register
  - If the bit being tested is found to be set, take the appropriate action depending on the application requirements
  - If the bit is found to be clear, generally no action is needed
  - In either case, clear the corresponding flag before inspecting the next one
- Clear any C1INTF (or C2INTF) event flags for which the corresponding event interrupts have been disabled (i.e., events that are not required to be processed by the particular application)
- Clear the global CAN interrupt flag (C1IF or C2IF)
- Return from the ISR function

Example 1 shows the general structure of a CAN ISR based on the approach outlined above.

Alternatively, the ISR can directly inspect the ICODE<2:0> bits (C1CTRL<3:1> or C2CTRL<3:1>). In this case, only one interrupt event would be processed during each execution of the ISR. This alternative interrupt handling sequence is shown in Example 2.

It should be noted that the generation of the C1IF (or C2IF) interrupt request is "event-triggered". This implies that the C1IF bit will be set only at the time any of the C1INTF bits have been set. In the context of a CAN interrupt handler routine, this means it is possible that (depending on the rate and timing at which interrupt events occur) certain interrupt events might not be processed if either of the above methods is used. Specifically, consider the following sequence of events:

- The last check of the C1INTF register has already been performed by the ISR software
- A new interrupt event occurs (e.g., the reception of a new message), resulting in a bit being set in the C1INTF register
- The ISR software, unaware that a new interrupt event has just occurred, clears the C1IF bit

On returning from the ISR, the C1IF bit remains clear (because it was cleared after the last interrupt event), even though one of the C1INTF bits is set (i.e., C1INTF has a non-zero value). As the C1IF interrupt is event-triggered, no new CAN interrupt request is generated, essentially resulting in an interrupt event being "lost" due to the relative timing of interrupt events and interrupt handler operations.

The scenario described above can be avoided by simply inspecting the values of the C1INTF and C1INTE registers (similarly, C2INTF and C2INTE) before clearing the C1IF (or C2IF) flag. If any of the enabled individual interrupt event flags is found to be cleared, then the C1IF bit is not cleared; thus, C1IF is cleared only if (C1INTF & C1INTE) is zero. An example of this modified approach is shown in Example 3.

Alternatively, the user application could wait in a loop within the ISR until all enabled interrupt events are found to be clear, but this is a relatively inefficient method.

Note: If the ERRIF bit was found set, the cause(s) of the error can be determined by inspecting the specific error flags in the upper byte of C1INTF (or C2INTF)

- If the bit is found to be clear, generally no action is needed
  - In either case, clear the corresponding flag before inspecting the next one

The scenario described above can be avoided by simply inspecting the values of the C1INTF and C1INTE registers (similarly, C2INTF and C2INTE) before clearing the C1IF (or C2IF) flag. If any of the enabled individual interrupt event flags is found to be cleared, then the C1IF bit is not cleared; thus, C1IF is cleared only if (C1INTF & C1INTE) is zero. An example of this modified approach is shown in Example 3.

Alternatively, the user application could wait in a loop within the ISR until all enabled interrupt events are found to be clear, but this is a relatively inefficient method.
EXAMPLE 1: CAN ISR EXAMPLE 1

```c
void __attribute__((interrupt, no_auto_psv)) _C1Interrupt(void)
{
    if (C1INTFbits.TX0IF) C1INTFbits.TX0IF = 0;
    if (C1INTFbits.TX1IF) C1INTFbits.TX1IF = 0;
    if (C1INTFbits.TX2IF) C1INTFbits.TX2IF = 0;
    if (C1INTFbits.RX0IF) { C1INTFbits.RX0IF = 0; // Add code to read buffer 0
                            C1RX0CONbits.RXFUL = 0;  }
    if (C1INTFbits.RX1IF) { C1INTFbits.RX1IF = 0; // Add code to read buffer 1
                            C1RX1CONbits.RXFUL = 0;  }
    if (C1INTFbits.WAKIF) C1INTFbits.WAKIF = 0; // Add wake-up handler code
    if (C1INTFbits.ERRIF) C1INTFbits.ERRIF = 0; // Add error handler code
    if (C1INTFbits.IVRIF) C1INTFbits.IVRIF = 0;
    IFS2bits.C1IF = 0; // Clear CAN1 interrupt flag before returning
}
```

EXAMPLE 2: CAN ISR EXAMPLE 2

```c
void __attribute__((interrupt, no_auto_psv)) _C1Interrupt(void)
{
    switch (C1CTRLbits.ICODE) {
        case 7: C1INTFbits.WAKIF = 0; break;
        case 6: C1INTFbits.RX0IF = 0; // Add code to read buffer 0
                 C1RX0CONbits.RXFUL = 0;
                 break;
        case 5: C1INTFbits.RX1IF = 0; // Add code to read buffer 1
                 C1RX1CONbits.RXFUL = 0;
                 break;
        case 4: C1INTFbits.TX0IF = 0; break;
        case 3: C1INTFbits.TX1IF = 0; break;
        case 2: C1INTFbits.TX2IF = 0; break;
        case 1: C1INTFbits.ERRIF = 0; // Add error handler code
                 break;
    }
    IFS2bits.C1IF = 0; // Clear CAN1 interrupt flag before returning
}
```

EXAMPLE 3: MODIFIED CAN ISR

```c
void __attribute__((interrupt, no_auto_psv)) _C1Interrupt(void)
{
    if (C1INTFbits.TX0IF) C1INTFbits.TX0IF = 0;
    if (C1INTFbits.TX1IF) C1INTFbits.TX1IF = 0;
    if (C1INTFbits.TX2IF) C1INTFbits.TX2IF = 0;
    if (C1INTFbits.RX0IF) { C1INTFbits.RX0IF = 0; // Add code to read buffer 0
                            C1RX0CONbits.RXFUL = 0;  }
    if (C1INTFbits.RX1IF) { C1INTFbits.RX1IF = 0; // Add code to read buffer 1
                            C1RX1CONbits.RXFUL = 0;  }
    if (C1INTFbits.WAKIF) C1INTFbits.WAKIF = 0; // Add wake-up handler code
    if (C1INTFbits.ERRIF) C1INTFbits.ERRIF = 0; // Add error handler code
    if (C1INTFbits.IVRIF) C1INTFbits.IVRIF = 0;
    if ( (C1INTF & C1INTE) == 0 ) IFS2bits.C1IF = 0;
}
```

CONCLUSION

The CAN interrupt architecture in dsPIC30F Digital Signal Controller (DSC) devices provides a versatile mechanism for efficiently responding to various transmission, reception, and error events. This technical brief has outlined two typical interrupt handler structures. A simple and reliable technique of eliminating the possibility of lost interrupt events has also been described.
APPENDIX A: SOURCE CODE

Software License Agreement

The software supplied herewith by Microchip Technology Incorporated (the “Company”) is intended and supplied to you, the Company’s customer, for use solely and exclusively with products manufactured by the Company.

The software is owned by the Company and/or its supplier, and is protected under applicable copyright laws. All rights are reserved. Any use in violation of the foregoing restrictions may subject the user to criminal sanctions under applicable laws, as well as to civil liability for the breach of the terms and conditions of this license.

THIS SOFTWARE IS PROVIDED IN AN “AS IS” CONDITION. NO WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. THE COMPANY SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.
Note the following details of the code protection feature on Microchip devices:

• Microchip products meet the specification contained in their particular Microchip Data Sheet.

• Microchip believes that its family of products is one of the most secure families 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 Microchip products in a manner outside the operating specifications contained in Microchip’s Data Sheets. Most likely, the person doing so is 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 products. Attempts to break Microchip’s code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Information contained in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION, INCLUDING BUT NOT LIMITED TO ITS CONDITION, QUALITY, PERFORMANCE, MERCHANTABILITY OR FITNESS FOR PURPOSE. Microchip disclaims all liability arising from this information and its use. Use of Microchip devices in life support and/or safety applications is entirely at the buyer’s risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights.

Trademarks

The Microchip name and logo, the Microchip logo, Accuron, dsPIC, KEELoO, KEELoO logo, MPLAB, PIC, PICmicro, PICSTART, rPIC and SmartShunt are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

FilterLab, Linear Active Thermistor, MXDEV, MXLAB, SSEEVAL, SmartSensor and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Analog-for-the-Digital Age, Application Maestro, CodeGuard, dsPICDEM, dsPICDEM.net, dsPICWorks, dsSPEAK, ECAN, ECONOMONITOR, FanSense, In-Circuit Serial Programming, ICSP, ICEPIC, Mindi, MiWi, MPASM, MPLAB Certified logo, MPLIB, MPLINK, mTouch, PICkit, PICDEM, PICDEM.net, PICtail, PICtail logo, PowerCal, PowerInfo, PowerMate, PowerTool, REAL ICE, rLAB, Select Mode, Total Endurance, UNI/O, WiperLock and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

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.

© 2009, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.

Printed on recycled paper.