ST10X167/ST10F168

Programming ST10X167/ST10F168 CAN interrupt drivers

APPLICATION NOTE

This application note describes the CAN interrupt drivers of the ST10X167/ST10F168 and provides programming examples that can be used to define interrupt schemes and write interrupt drivers.

Interrupt sources, the way the sources of interrupts are identified, and the two methods of handling interrupts are described: one using the hardware features of the CAN module and the other through polling internal sources.

Programming the CAN interrupt drivers through CAN hardware features uses RXIE and TXIE bits in the Message Control Register of each message object. Whenever a message is transmitted or received by a message object, the corresponding interrupt is serviced according to its priority (based on the value of INTID). This method requires minimum CPU overhead and is the preferred method for most applications.

CAN polling generates an interrupt whenever a successful transmission or reception occurs. Polling is high in CPU overhead, as the CPU is interrupted every time a message is acknowledged on the CAN bus. Therefore, programming the interrupt driver using polling is only recommended for small networks.

Sample programs are provided for each method and can used as examples.

Table of Contents

1 Interrupt sources & identification - - - - - - - - - - - - - - - - 3
1.1 Global interrupt sources - 3
1.2 Individual interrupt sources - - - - - - - - - - - - - - - - - - 3
1.3 Identifying interrupt sources - - - - - - - - - - - - - - - - - - 4

2 Handling interrupts - 5
2.1 Message-specific interrupts - - - - - - - - - - - - - - - - - - 5
2.2 Busoff interrupts - 7
2.3 Message 15 interrupts - 8

3 Programming through CAN hardware features - - - - - - - - 9

4 Programming through polling - - - - - - - - - - - - - - - - 12

5 Appendix1: Register description - - - - - - - - - - - - - - - 14
5.1 Global interrupt control register - - - - - - - - - - - - - - - 14
5.2 CAN control/status register - - - - - - - - - - - - - - - - - 15

6 Appendix 2: Message object register definition - - - - - - 18
6.1 can_obj.h - 18
6.2 can_16x.h - 18

1 Interrupt sources & identification

Interrupts generated by the CAN module can come from different sources: ‘global interrupt sources’ and ‘individual interrupt sources’. Although all CAN interrupt sources can be individually masked, only 1 global interrupt request is sent to the CPU.

1.1 Global interrupt sources

To enable a CAN interrupt to the CPU, set both bit IE in the CAN Control/Status Register and bit XP0IE in the CAN Global Interrupt Control Register XP0IC. The CAN global interrupt uses the same Global Interrupt Control Register XP0IC as other on-chip peripherals. Refer to “Appendix1: Register description” on page 14 for a description of the Global Interrupt Control Register XP0IC and the CAN Control/Status Register.

1.2 Individual interrupt sources

The CAN controller distinguishes 3 different individual interrupt sources:

Status interrupts

Status interrupts are generated after a status change of the CAN module indicated by the flags in the status part (high byte) of the Control/Status Register. Status interrupts are enabled by setting bit SIE in the Control/Status Register.

Status interrupt are caused by:

- Successful transmission from the CAN module (TxOK is set) of any message from message objects.
- Reception of a message on the CAN bus - RxOK is set after an acknowledge of a CAN frame from the CAN module - this CAN frame may not correspond to any message object identifier.
- Occurrence of an error on the CAN bus during a message transfer (LEC bit field is updated); this error may not come from the CAN module *itself,* but has at least been detected by the CAN module. LEC code 7 is not used and may be written to 7 by the CPU to check for updates.

TxOK, RxOK, LEC can be read from the Control/Status Register.

Note *Enabling the Status Change Interrupt, causes an interrupt to the CPU every time a message is acknowledged on the CAN bus. For this reason, Status Change Interrupt should be disabled in most applications except where a close monitor on the bus activity is required (e.g. after an early warning).*

Error Interrupts

Error Interrupts are generated once predefined error conditions are reached. Error interrupts are enabled by setting bit EIE in the Control/Status Register. Error interrupts are caused by:

- Warning status (EWRN is set) indicates that least one of the error counters has reached the error warning limit of 96.
- Busoff (BOFF is set) indicates that the CAN controller is in busoff state.

EWRN and BOFF can be read from the Control/Status Register.

Message specific interrupts

Message specific interrupts are generated by each message object. They are individually enabled by setting bits TXIE (for transmission) and/or RXIE (for reception) in the CAN Message Control Register (MCR_Mn) located at address EFn0h (where n is the number of the corresponding message object).

Message specific interrupts are caused by:

- Reception of a message (data frame or remote) into the corresponding message object.
- Successful transmission (data frame or remote) of the corresponding message object.

Bit ‘INTPND’ in each message object indicates whether the corresponding message object has generated an interrupt request.

Note *Refer to “Message-specific interrupts” on page 5 for further information on decoding message specific interrupts.*

1.3 Identifying interrupt sources

The value of INTID code in the Interrupt Register identifies the interrupt source; the interrupt with the lowest number has the highest priority. To update the INTID value, read the status partition.

- INTID = 0: indicates that no (or no more) interrupt/s is/are pending.
- INTID = 1: indicates that a Status Change Interrupt or an Error Interrupt is pending.
- INTID = 2: indicates the reception of a message by message object 15. Refer to “Message 15 interrupts” on page 8
- INTID = 3 to 16: indicate a Message Specific Reception or Transmission from message object 1 to 14. The INTID value is 2+n, where n is the code of the highest priority object which generated an interrupt request.

2 Handling interrupts

CAN interrupt-source priority decreases with increasing INTID code. The INTID code is taken into account when identifying the interrupt source. For example, if bits SIE and TXIE have been set, the successful transmission of 1 message can cause 2 independent interrupt requests. While a status interrupt (highest priority) is serviced, a message specific interrupt remains pending (bit INTPND of the corresponding object is only cleared by writing a 0 value in the Message Control register).

There are 2 ways to handle interrupts: one method relies on the hardwired priority of the message object, the other method uses polling. These methods are described in Chapter 3: *Programming through CAN hardware features* on page 9 and Chapter 4: *Programming through polling* on page 12.

Although these 2 methods identify the interrupt source in different ways, they handle lower level events in the same way (message specific interrupts, busoff interrupt, message object 15 interrupt). These lower level events are described in the following sections.

2.1 Message-specific interrupts

When a message object causes an interrupt, the message object can be identified in 2 ways:

- from the INTID bit value
- by polling bit INTPND of all message objects

The direction of bit DIR in the Message Configuration Register must be considered in the decoding phase:

Message objects with the message direction = ‘receive’ (bit DIR is reset)

These can be caused by:

- Successful transmission of a remote frame (TXIE must have been set).
- Successful reception of a data frame (RXIE must have been set).

The bit ‘New data’ can be used to differentiate between the 2 interrupt causes:

- New data = 1: reception of a data frame.
- New data = 0: successful transmission of a remote frame.

Message objects with message direction = ‘transmit’ (bit DIR is set)

These can be caused by:

- Reception of a remote frame with same identifier (RXIE must have been set).
- Successful transmission of a data frame following a CPU request or remote request (TXIE must have been set).

If an interrupt is caused by the successful transmission of a data frame, a remote frame with the same identifier may still be received by the CAN *before* the CPU enters the interrupt routine. **Hint:** disable the reception interrupt (clear RXIE) for message objects with direction = ‘transmit’ as long as CPUUPD remains at 0.

Changing the message configuration

To change the message configuration during normal operation, set MSGVAL to ‘not valid’ (bit MSGVAL =0). When the new configuration is set into the message object registers, bit MSGVAL can be reset.

Example - software decoding of message specific interrupts

Note *This example assumes that a ‘switch on INTID’ instruction is placed at the beginning of the interrupt driver:*

```
case (n+2):                          // Message n specific interrupt
    if (Mn_MFR & 0x0008)             // direction = transmit
    {
      if (Mn_MCR & 0x8000)           // remote received
                          {.....}    // procedure to handle answer to remote
      else
                          {.....}    // procedure to handle transmit interrupts
                                     // RXIE is cleared as long as CPUUPD=0
    }
    else                             // direction = receive
    {
      if (Mn_MCR & 0x0200)           // New data is set
                          {.....}    // procedure to handle data receive interrupt
      else
                          {.....}    // a remote frame was successfully transmitted
    }
    Mn_MCR = 0xFFFDh;                // reset INTPND
    break;
```

2.2 Busoff interrupts

Each transfer-error detected on the CAN bus (though not necessarily generated by the CAN module) increments of one of the 2 error counters (receive error counter and transmit error counter). If one of these counters reaches the value of 96, bit EWRN is set. If enabled by bits EIE and IE, an error interrupt is generated. If the transmit error counter exceeds the value of 255, bit BOFF is set. If enabled by bits EIE and IE, this generates an error interrupt. Furthermore, when the device goes into the bus-off state, bit INIT is set and all actions on the bus are stopped immediately.

The bus-off recovery sequence can be initiated from the bus-off state. During the bus recovery sequence, the CAN module monitors the recovery sequence.

- The bus-off recovery sequence is started by resetting bit INIT.
- Bit0error is set every time a sequence of 11 recessive bits is detected. Bit0error indicates that the CAN bus is not stuck at dominant, or not continuously disturbed. If bits SIE and IE are set, a status interrupt is generated after each sequence.
- The recovery sequence cannot be shortened by setting or resetting bit INIT.

```
case 1:                          // INTPID=0x01 for status and error interrupt
     {
     if (control & 0x08)         // error interrupt are enabled
          {if (status & 0x40)    // EWRN is set
                  { ......}      // procedure for Early warning
          if (status &0x80)      // BOFF is set
                  { .......}     // bus-off procedure
          }
     break;
```

2.3 Message 15 interrupts

The message object 15 provides a basic CAN feature: all incoming messages with identifiers which do not match other message objects, can be stored in message object 15. A double buffer system prevents data loss when 2 messages arrive in rapid succession, and gives the CPU time for message handling.

The message 15 double buffer:

Buffers can be allocated to the CPU or CAN module on CPU or CAN module request, providing 2 handling methods:

- The CPU releases the buffer just after it has been read, leaving the CAN module to allocate a buffer when a new message is stored. For example:

```
case 2:                                  // INTPID = 0x02 for message15
        {
        .....                            // message 15 interrupt routine
        M15_MCR = 0x55DFh;               // release buffer
        M15_MCR = 0xFFFDh;               // reset INTPND
        }
        break;
```

- The CPU releases the buffer that it has previously read from, just before reading from a new buffer. For example:

```
case 2:                                  // INTPID = 0x02 for message15
        {
        M15_MCR = 0x55DFh;               // release buffer
        M15_MCR = 0xFFFDh;               // reset INTPND
        .....                            // message 15 interrupt routine
        }
        break;
```

The buffer can be released from the CPU by resetting bits NEWDAT and RMTPND of the Message Control Register (writing value = 0x55DFh).

3 Programming through CAN hardware features

Programming the CAN interrupt drivers through CAN hardware features uses bits RXIE and TXIE in the Message Control Register of each message object. Whenever a message is transmitted or received by a message object, the corresponding interrupt is serviced according to its priority (based on the value of INTID). This method uses the hardwired priority scheme of the CAN module, and requires minimum CPU overhead. This section provides a sample program for programming through CAN hardware features.

Hints:

- If the Status Change Interrupt is enabled, the CPU will be interrupted every time a message is acknowledged on the CAN bus. *Disable the Status Change Interrupt*!
- RXOK is cleared by the Status Change Interrupt Routine. Therefore, *do not use RXOK to identify the cause of interrupts*.
- TXOK can be used to identify the cause of interrupt if Status Change Interrupt is disabled, and if TXOK is cleared by every interrupt routine of message objects with direction = 'transmit'. For clarity, the example provided below assumes that Status Change Interrupt is enabled.

There are two ways to use the sample program provided below:

- First, the software interrupt driver repeats itself along as there is an active interrupt in the CAN module.
- Second, the software driver is run once and handles only one interrupt. This allows the interrupt controller to re-arbitrate interrupts of the same priority level but not of a higher priority group.

```
// CAN interrupt driver
//
#include <canr_16x.h>              // see desc. of include file in Section 6
#include <reg167.h>
interrupt(0x40)                    // CAN Module IR
void CAN_IT (void)
{
    unsigned short n;
    unsigned char status, intid, control;

    while (intid=IR)               //reload intid with IR,
                                   // continue loop if intid != 0
                                   //'while'can be removed to run  IT driver once
```

```
        {
status=SR control = CR;            // copy status + control REG to variable
     SR=0;                         // clear status register
     switch (intid)
 {
    case 1:                        // Status Change Interrupt
       if (CR & 0x04)              // if SIE is set (status interrupts)
       {
         if (status & 0x08)        // transmit interrupt
         {......}
         if (status & 0x10)        // receive interrupt
         {......}
         if (status & 0x07)        // erroneous transfer interrupt
         {......}
    }
       if (CR & 0x08)              // if EIE is set (error interrupts)
       {
         if (status & 0x40)        // error counter warning
         {......}
         if (status & 0x80)        // bus-off situation
         CR = (CR & 0xfe);         // recover from BOFF (clear INIT)
         {......}                  // remaining part of the BOFF procedure
    }
      break;

    case 2:                         // INTPID = 0x02 for message15 receive INT
         {                          // see Section 2.3
         .....                      // message 15 interrupt routine
         M15_MCR = 0x55DFh;         // release buffer
         M15_MCR = 0xFFFDh;         // reset INTPND
         }
         break;

    case 3:                         // Message 1 Interrupt
         if (M1_MFR & 0x0008)       // direction = transmit
         {
          if (M1_MCR & 0x8000)      // remote received
                     {.....}        // procedure to handle answer to remote
          else
                     {.....}        // procedure to handle transmit interrupts
                                    // RXIE shall be cleared as long as CPUUPD=0
         }
```

```
          else                      // direction = receive
          {
           if (M1_MCR & 0x0200)     // New data is set
                      {.....        // procedur to handle data receive interrupt
           else
                      {.....        // a remote frame  successfully transmitted
          }
          M1_MCR = 0xFFFDh;         // reset INTPND
          break;

block to be repeated (from 'case 3:' to 'case 16:')

     case (n+2):                    // Message n Interrupt
          if (Mn_MFR & 0x0008)      // direction = transmit
          {
           if (Mn_MCR & 0x8000)     // remote received
                      {.....        // procedure to handle answer to remote
           else
                      {.....}       // procedure to handle transmit interrupts
                                    // RXIE shall be cleared as long as CPUUPD=0
          }
          else                      // direction = receive
          {
           if (Mn_MCR & 0x0200)     // New data is set
                      {.....        // procedur to handle data receive interrupt
           else
                      {.....        // a remote frame successfully transmitted
          }
          Mn_MCR = 0xFFFDh;         // reset INTPND
          break;

        }
     }
}
```

4 Programming through polling

Programming the interrupt driver using polling generates an interrupt whenever a successful transmission or reception occurs. This is done by setting bit SIE of the control register. RXOK and/or TXOK is/are set when a message object transmits or receives a message. The CAN module sets RXOK whenever a message is acknowledged on the CAN bus.

Once in the interrupt routine, the CPU polls bit INTPND of each message object. The message object polling sequence defines the message object priority, independent of the existing hardware priority scheme defined for INTID.

Polling is high in CPU overhead as the CPU is interrupted every time a message is acknowledged on the CAN bus. Therefore, programming the interrupt driver using polling is only recommended for small networks.

This section provides a sample program for programming through polling.

```
// CAN interrupt driver
//
#include <canr_16x.h>               // inc file to define CAN global registers
#include <reg167.h>                 // standard include file for 167
#include <can_obj.h>                // inc file to define CAN message object reg
                                    // see Section 6 for description.

interrupt(0x40)                     // CAN Module IR
void CAN_IT (void)
{
   unsigned short n;
   unsigned char status, intid, control;

             {
 status=SR;  control = CR;          // copy status + control REG to variable
      SR=0;                         // clear status register
             if (M15_MCR & 0x0002)
                        {.....}     // procedure to handle message 15 interrupts
                                    // see example below
             if (M1_MCR & 0x0002)
                        {.....}      // procedure to handle message 1 interrupts

             ...
             if......
             ...
```

```
            if (M14_MCR & 0x0002)
                        {.....}     // procedure to handle message 14 interrupts
            }
}
```

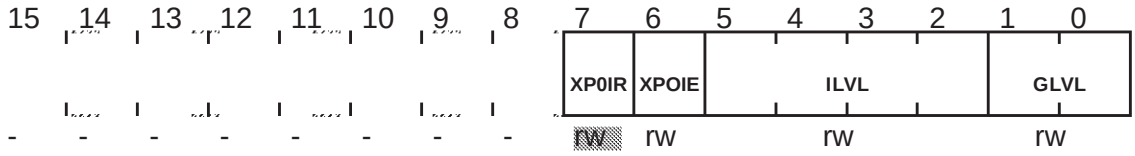
Then, for each message object, procedure to handle interrupt and decode can be the same as the one proposed before:

```
            if (Mn_MFR & 0008)      // direction = transmit
            {
             if (Mn_MCR & 0x8000)   // remote received
                        {.....}     // procedure to handle answer to remote
             else
                        {.....}     // procedure to handle transmit interrupts
                                    // RXIE shall be cleared as long as CPUUPD=0
            }
            else                    // direction = receive
            {
             if (Mn_MCR & 0x0200)   // New data is set
                        {.....}     // procedur to handle data receive interrupt
             else
                        {.....}     // a remote frame successfully transmitted
            }
            Mn_MCR = 0xFFFDh;       // reset INTPND
            break;
```

5 Appendix1: Register description

5.1 Global interrupt control register

XP0IC (F186h / C3h) **<ESFR area>** **Reset Value: - - 00h**



Bit	Function
GLVL	**Group Level** Defines the internal order for simultaneous requests of the same priority. 3: Highest group priority 0: Lowest group priority
ILVL	**Interrupt Priority Level** Defines the priority level for the arbitration of requests. F_H: Highest priority level 0_H: Lowest priority level
XP0IE	**Interrupt Enable Control Bit (individually enables/disables a specific source)** ‘0’: Interrupt request is disabled ‘1’: Interrupt Request is enabled
XP0IR	**Interrupt Request Flag** ‘0’: No request pending ‘1’: This source has raised an interrupt request

Table 1 XP0IC Interrupt control register

5.2 CAN control/status register

Control / Status Register (EF00h) **XReg** **Reset Value: XX01h**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BOFF	EWRN	-	RXOK	TXOK	LEC			0[1)]	CCE	0	0	EIE	SIE	IE	INIT
r	r	r	rw	rw	rw			rw	rw	r	r	rw	rw	rw	rw

Bit	Function (Control Bits)
INIT	**Initialization** 1: Software initialization of the CAN controller. While INIT is set, all message transfers are stopped. Setting INIT does not change the config registers and does not stop transmission or reception of a message in progress. The INIT bit is also set by hardware, following a BUSOFF condition; the CPU then needs to reset INIT to start the bus recovery sequence. 0: Disable software initialization of the CAN controller; on INI completion, the CAN waits for 11 consecutive recessive bits before taking part in bus activities.
IE	**Interrupt Enable** 1: Global interrupt enable from CAN module. 0: Global interrupt disable from CAN module. Does not affect status updates.
SIE	**Status Change Interrupt Enable** 1: Enables interrupt generation when a message transfer (reception or transmission is successfully completed or CAN bus error is detected (and registered in LEC is the status partition. 0: Disable status change interrupt.
EIE	**Error Interrupt Enable** 1: Enables interrupt generation on a change of bit BOFF or EWARN in the status partition. 0: Disable error interrupt.

Table 2 CAN Control/Status register

Bit	Function (Control Bits)
CCE	**Configuration Change Enable** 1: Allows CPU access to the bit timing register 0: Disables CPU access to the bit timing register
1)	**Test Mode (Bit 7)** Make sure that bit 7 is cleared when writing to the Control Register. Writing a 1 during normal operation may lead erroneous device behavior.
LEC	**Last Error Code** This field holds a code which indicates the type of the last error occurred on the CAN bus. If a message has been transferred (reception or transmission) without error, this field will be cleared. Code “7” is unused and may be written by the CPU to check for updates. 0: **No Error** 1: **Stuff Error:** More than 5 equal bits in a sequence have occurred in a part of a received message where this is not allowed. 2: **Form Error:** A fixed format part of a received frame has the wrong format. 3: **AckError:** The message this CAN controller transmitted was not acknowledged by another node 4: **Bit1Error:** During the transmission of a message (with the exception of the arbitration field), the device wanted to send a *recessive* level (“1”), but the monitored bus value was *dominant* 5: **Bit0Error:** During the transmission of a message (or acknowledge bit, active error flag, or overload flag), the device wanted to send a *dominant* level (“0”), but the monitored bus value was *recessive*. During *busoff* recovery this status is set each time a sequence of 11 *recessive* bits has been monitored. This enables the CPU to monitor the proceeding of the busoff recovery sequence (indicating the bus is not stuck at *dominant* or continuously disturbed). 6: **CRCError:** The CRC check sum was incorrect in the message received.
TXOK	**Transmitted Message Successfully** Indicates that a message has been transmitted successfully (error free and acknowledged by at least one other node), since this bit was last reset by the CPU (the CAN controller does not reset this bit!).

Table 2 CAN Control/Status register

Bit	Function (Control Bits)
RXOK	**Received Message Successfully** Indicates that a message has been received successfully, since this bit was last reset by the CPU (the CAN controller does not reset this bit!).
EWRN	**Error Warning Status** Indicates that at least one of the error counters in the EML has reached the error warning limit of 96.
BOFF	**Busoff Status** Indicates when the CAN controller is in busoff state (see EML).

Table 2 CAN Control/Status register

6 Appendix 2: Message object register definition

The files below defines the CAN module main registers that are used by the interrupt driver. Other message object specific registers, such as the arbitration registers, can also be user defined.

6.1 can_obj.h

```
//      can_obj.h

// Define Message object specific registers
#define M1_MCR   *(unsigned int *) 0xef10         // message control reg.1
#define M1_MFR   *(unsigned int *) 0xef16         // message configuration reg.1
#define M2_MCR   *(unsigned int *) 0xef20         // message control reg.2
#define M2_MFR   *(unsigned int *) 0xef26         // message configuration reg.2
#define M3_MCR   *(unsigned int *) 0xef30         // message control reg.3
#define M3_MFR   *(unsigned int *) 0xef36         // message configuration reg.3
#define M4_MCR   *(unsigned int *) 0xef40         // message control reg.4
#define M4_MFR   *(unsigned int *) 0xef46         // message configuration reg.4

....

#define M15_MCR  *(unsigned int *) 0xeff0         // message control reg. 15
#define M15_MFR  *(unsigned int *) 0xeff6         // message config reg.15
```

6.2 can_16x.h

```
//       can_16x.h

#define  CR     *(unsigned char*) 0xEF00
#define  SR     *(unsigned char*) 0xEF01
#define  IR     *(unsigned char*) 0xEF02
```

Note *All software provided here follows the TASKING CAN library register naming convention. Register naming may be different for another tool chain.*