Software ISO 7816 I/O Line Implementation

Features

- ISO 7816-3 compliant (direct convention)
- Byte reception and transmission with parity check
- Retransmission on error detection
- Automatic reception at the end of transmission
- Dynamically programmable baud rate (PTS)
- ARM Procedure Call Standard (APCS) compliant

Introduction

ISO 7816-3 is the standard for electronic signals and transmission protocols for IC cards with contacts. It defines the characteristics of the signals used to communicate with a Smart Card including the I/O line which allows messages to be exchanged. The AT91M40400 does not have dedicated hardware to manage the ISO 7816 I/O line, but because of the high processing speed and flexible Timer Counter (TC), an effective software implementation can easily be performed.

AT91
ARM Thumb® Microcontrollers

Application Note

Rev. 1154A–08/98

Theory of Operation

The interface defined by the 7816-3 specification consists of the following signals:

- GND: common reference
- VCC: power supply for the Smart Card
- CLK: system clock of the Smart Card
- RST: system reset of the Smart Card
- I/O: data signal

The data transfer is performed on the ISO 7816-3 I/O line. This signal is an asynchronous half-duplex serial interface on which the baud rate closely depends on the clock signal (CLK), as it systematically is a ratio of this clock. Two parameters, FI and DI, define the clock divisor to obtain the bit duration:

$(Fi / Di) \times (1 / f)$.

where:

- f is the clock (CLK) frequency,
- Fi is the clock rate conversion factor (defined by FI) and
- Di is the bit rate adjustment factor (defined by DI)

The following tables show the relationship between FI and Fi, DI and Di:

FI	0000	0001	0010	0011	0100	0101	0110	0111
Fi	372	372	558	744	1116	1488	1860	RFU

FI	1000	1001	1010	1011	1100	1101	1110	1111
Fi	RFU	512	768	1024	1536	2048	RFU	RFU

Notes: 1. Fi, indicated values of the clock rate conversion factor
2. RFU = Reserved for Future Use

DI	0000	0001	0010	0011	0100	0101	0110	0111
Di	RFU	1	2	4	8	16	32	RFU

DI	1000	1001	1010	1011	1100	1101	1110	1111
Di	12	20	RFU	RFU	RFU	RFU	RFU	RFU

Notes: 1. Di, indicated values of the bit rate adjustment factor
2. RFU = Reserved for Future Use

In order to allow the management of the I/O line in half duplex mode by both sides, this line is connected to the supply voltage via a pull-up resistor. The output stages of the devices connected to the bus must have an open-drain or open-collector to perform a wired-AND function.

A character consists of:

- one start bit at low level
- 8 data bits of information (from Least Significant Bit to Most Significant Bit)
- one parity bit
- two (if no error) or three (if error) stop bits at high level

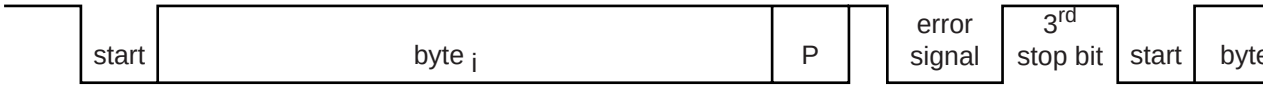
The ISO 7816-3 specification implements two different conventions for data transfer: direct or reverse convention. For more convenience, this application note refers to the direct convention, which is to say:

- not inverted bits (logical “1” = HIGH state)
- Least Significant Bit (LSB) first
- even parity (number of “1’s” in data + parity fields is even)

The beginning of the byte is indicated by a start bit at LOW level. Then, the 8 data bits are transmitted, followed by the parity bit. Afterwards, 2 stop bits, at HIGH level, are transmitted. If the receiver does not detect a parity error, it waits for the next start bit and the sender transmits the next byte after the 2 stop bits. If the receiver does detect an error, it indicates this to the sender by setting the line at LOW level somewhere between the half of the first stop bit and the half of the second stop bit. Therefore, the sender checks the I/O line at the end of the first stop bit and, if it is LOW, transmits a third stop bit at HIGH level, and then re-sends the same byte from the beginning.

Figure 1. Communication on the I/O Line without Parity Error

Figure 2. Communication on the I/O Line with Parity Error



For more information concerning the I/O Line, please refer to the ISO 7816-3 Specification.

AT91M40400 Implementation

This application note describes how to implement an ISO 7816-3 I/O line with an AT91M40400. It describes only the byte transfer feature. It does not refer to the type of protocol used (T = 0 or T = 1), or the type of frame. These points are relevant to a higher protocol layer and to the application. Nor does it describe the clock and reset (which can be generated with another timer channel) or the V_{CC} generation.

One of the three channels from the AT91M40400 Timer Counter block can be chosen to implement the I/O line. The received bytes are stored in a reception buffer. The bytes to transmit come from a transmission buffer. By default, the ISO line is in reception mode, except when a transmission is requested by a higher application level.

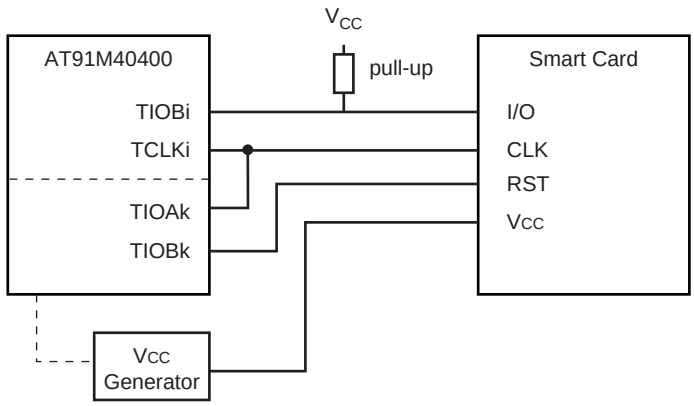
The timer is clocked from its external clock input (TCLKi) linked to the ISO clock and the corresponding TIOB signal is used as the I/O line. It is used in wave form mode in order to drive TIOB as input or output. Note that when driven for transmission, the TIOB line is physically held for HIGH level as well as for LOW. Bit duration and sampling are performed by using the TC_RC (Register C) and TC_RA (Register A):

- TC_RC is set with the bit time value (the clock divisor Fi / Di).
  In reception and transmission mode, the counter is reset at each RC Compare event.
  In transmission mode, each RC Compare event generates an interrupt in order to set each bit at the beginning of a bit period.
- TC_RA is set with the half bit time value (the clock divisor divided by 2).
  In reception mode, each RA Compare event generates an interrupt in order to get the I/O line state at the middle of each bit.

Connection

At reset, all PIOs of the AT91M40400 are programmed as input. Therefore, the I/O line must be connected to the supply voltage via a pull-up resistor (refer to the AT91M40400 datasheet for more details concerning the value of the pull-up resistors).

Figure 3. Connection of the AT91M40400 with a Smart Card



Notes: 1. i = TC channel used as an I/O line
2. k = TC channel used for clock generation

Source Files

Files	Contents
std_c.h	*Standard C definitions*
tc.h	*Timer types and constants*
aic.h	*Interrupt controller types and constants*
pio.h	*Peripheral Input/Output types and constants*
iso_line.h	*ISO 7816 I/O line types and constants*
iso_line.c	*User interface functions*
iso_tx.c	*Transmission interrupt handlers*
iso_rx.c	*Reception interrupts handlers*
iso_irq.s	*Interrupt handlers written in assembler*

Peripheral Treatments

For each peripheral used (Timer Counter, Interrupt Controller, etc.) a structure is defined to provide easy access to the peripheral registers. Constants are also defined to provide easy access to the register fields. These types and constants are defined in *tc.h*, *aic.h*, *pio.h*.

Global Variables

Global variables relative to a timer used as the ISO line are grouped into a structure (**ISO_LINE_VAR** type defined in *iso_line.h*).

This structure groups the timer characteristics (base address, PIO pins, etc), buffers pointers, working variables, and interrupt handler addresses (AIC handler and C handler).

The *iso_line.c* file defines a structure instance for each timer (**isoLineVar0**, **IsoLineVar1**, **isoLineVar2**) and an array of pointers to these structures (**isoLineVar**) to allow access to these variables with the timer number.

Upper Layer Interface

The functions to manage the ISO line are defined in *iso_line.c*

u_int IsoLineOpen (u_int timer_id)	Initialize a timer to be used as an ISO line
u_int IsoLineClose (u_int timer_id)	Close the current ISO line
u_int IsoLineBaudRate (u_int timer_id, u_char ta1)	Modify the clock divisor of the timer input clock
*u_int IsoLineRxEnable (u_int timer_id, u_char *buf, u_int sz)*	Enable the reception buffer
u_int IsoLineRxCount (u_int timer_id, u_int cnt)	Get the number of received bytes or wait for a number of bytes to be received
*u_int IsoLineTxFrame (u_int timer_id, u_char *buf, u_int sz, u_int tryNb)*	Send a frame on the ISO line
u_int IsoLineTxStatus (u_int timer_id)	Get timer status in transmission mode (in progress, ended, or number of try overflow)

The **IsoLineOpen** function configures a timer channel to be used as an ISO I/O line manager (i.e. fill RA, RB registers, initialize variable structures and interrupt management).

```
Begin
| Disable interrupts
| Initialize the default handler address
| Enable the Timer interrupts on the Interrupt controller
| Set TIOB internal output state
| Define as peripheral the pins reserved for the Channel
| Initialize the mode of the timer
| Initialize the RA ( 0.5 bit) and RC Register (1 bit)
| Enable timer clock
End
```

The **IsoLineClose** function disables interrupts on the ISO line timer, allowing it to be used for other tasks after disconnecting from the ISO lines.

```
Begin
| Disable all interrupts of the current timer
End
```

The **IsoLineBaudRate** function allows the user to modify the Fi and Di (clock divisor) parameters.

```
Begin
| Set the RA (0.5 bit) and RC (1 bit) Register values
| Update Rx-Tx switch time (switchTime = (Fi / 64) *Di)
End
```

The **IsoLineRxEnable** function sets the TC channel in reception mode on the I/O line and enables the user to set the reception buffer characteristics.

```
Begin
| Set timer buffer variables
| Start byte reception (IsoRxReceiveByte)
| Endif
End
```

The **IsoLineRxCount** function allows the user to count the number of bytes received since the last **IsoLineRxEnable**, or to expect the reception of a defined number.

```
Begin
| if cnt not NULL
| | Loop while number of received bytes < cnt
| Endif
| return number of received bytes
End
```

The **IsoLineTxFrame** function allows the user to send a frame under interrupt. The user selects a maximum number of erroneous attempts to send a byte. If the maximum number of erroneous attempts is reached, the transmission is aborted.

```
Begin
| Save number of tries (error retry)
| Save buffer variables
| Set current byte with first byte to send
| Prepare the first byte sending (IsoTxSendByte)
End
```

The **IsoLineTxStatus** function allows the user to obtain the current transmission status (in progress, ended, or try overflow).

```
Begin
| If too much tries return TX_OVERFLOW
| If all bytes transmitted return TX_ENDED
| Else return TX_IN_PROGRESS
End
```

Interrupt Management

Transmission and reception are performed by an interrupt. The initialization relative to these interrupts (stack, etc.) is not described in this document but must be programmed before using the functions defined in this application note.

Source Vector Register (SVR)

In the AIC (Advanced Interrupt Controller), the Source Vector Register (SVR) of the Timer Counter used to manage the ISO line is filled with an assembler interrupt handler.

This assembler handler (coded in *iso_irq.s*) executes the basic interrupt treatments (register saving and restoring, stack management, end of interrupt acknowledgment, etc.). These operations are performed using the **IRQ_ENTRY** and **IRQ_EXIT** macros provided by the AT91 library. Between these macros, the handler loads the variable structure corresponding to the timer which caused the interrupt (see Global variables section) and calls the **IsoLineHandlerInt** C handler with this structure as parameter.

IsoLineHandlerInt

IsoLineHandlerInt performs the basic operations to do for all handlers:

- Increase bit counter for the state automate management
- Read the timer status to acknowledge interrupt.
- Call the C handler corresponding to the current state of the automate (read parity, send acknowledgment, etc.). This handler is preset by the field **isoTCHandler** of the variables structure.

```
Begin
| Increment bit count for automate management
| Acknowledge interrupt status
| Call handler following the automate state
End
```

Reception

The file “iso_rx.c” manages the reception. It consists of:

- one Device Service Routine (DSR) which initializes the reception when requested by the application
- a set of Interrupt Service Routines (ISR) which activate a state machine. These routines are called by **IsoLineHandlerInt**

The reception is initialized by the function **IsoRxReceiveByte** which sets TC_CMR (Channel Mode Register) by configuring TIOB as an external trigger event on the falling edge. It also enables the interrupt on RA compare event. This way, once the first falling edge is detected, an interrupt is generated at the middle of each bit period after the first falling edge (start bit). Then, the transfer itself is managed under interrupts. The state machine is initialized by setting the interrupt handler to **IsoRxGoMiddle**.

```
Begin
| Disable timer interrupts
| Validate trig on falling edge of I/O line (TIOB)
| If buffer available
| | Set the mode register : trig on falling edge of TIOB
| | Initialize received byte and parity
| | Initialize state machine to 'IsoRxGoMiddle'
| | Enable interrupts on RA Compare event
| | Start timer counter
| Endif
End
```

Figure 4. States and the Corresponding C Interrupt Handlers

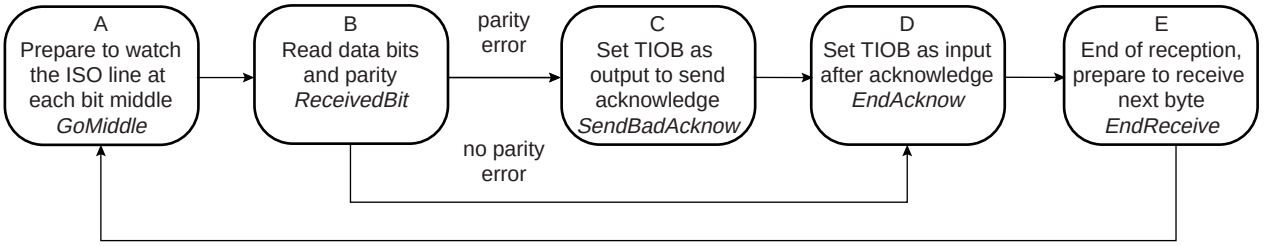


Figure 5. Reception without Parity Error

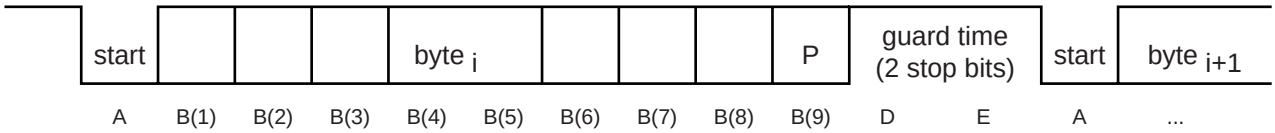


Figure 6. Reception with Parity Error

Note: 1. () = bit counter value

The **IsoRxGoMiddle** function disables the external trigger, but maintains TIOB as an external trigger event (in order to leave it as an input). It also initializes the bit counter and the calculated parity value. The interrupt handler is changed to **IsoRxReceivedBit**.

```
Begin
| Enable trigger on RC Compare event and disable external trigger
| bit counter <- 0
| Change IT handler to ‘IsoRxReceiveBits’
End
```

The **IsoRxReceivedBit** function gets the bit value on the I/O line from the PIO_PDSR (Pin Data Status Register) to get the data or parity bit value. On each bit, the parity calculation is updated. When the bit counter reaches 9, the calculated parity must be 0. If it is not, the interrupt handler is changed to **IsoRxSendBadAcknow**, otherwise the byte is saved in the reception buffer and the interrupt handler is changed to **IsoRxEndAcknow**.

```
Begin
| Read the received bit
| Calculate the parity
| If bit counter < 9
| | Update the current byte
| Else
| | If parity error
| | | Change interrupt handler to 'IsoRxSendBadAcknow'
| | Else
| | | Change interrupt handler to 'IsoRxEndAcknow'
| | Endif
| Endif
End
```

The **IsoRxSendBadAcknow** function configures the TIOB pin as a PIO output and clears the line (error acknowledgment). The interrupt handler is changed to **IsoRxEndAcknow**.

```
Begin
| Set TIOB as PIO output and set output at low level
| Change interrupt handler to IsoRxEndAcknow
End
```

The **IsoRxEndAcknow** function resets TIOB as the timer input. The interrupt handler is changed to **IsoRxEndReceive**.

```
Begin
| Set TIOB as timer pin (not PIO pin)
| Change interrupt handler to IsoRxEndReceive
End
```

The **IsoRxEndReceive** function saves the received byte in the buffer if the parity was correct and calls the **IsoRxReceiveByte** function to initialize a new byte reception.

```
Begin
| If no parity error
| | Save byte in the reception buffer
| Endif
| Initialize new byte reception (IsoRxReceiveByte)
End
```

Transmission

The file “iso_tx.c” manages the transmission. It consists of:

- one Device Service Routine (DSR) which initializes the reception when requested by the application
- a set of Interrupt Service Routines (ISR) which activate a state machine. These routines are called by **IsoLineHandlerInt**

The transmission is initialized by the function **IsoTxSendByte** which sets the TC_CMR (Channel Mode Register) by configuring TIOB as an external trigger event, in order to not change the line state before the start of the transmission. It also enables the interrupt on an RC compare event, and launches the timer with a software trigger. Then, the transfer itself is managed by interrupts. The transmission is delayed in order to allow the other equipment to switch its line interface. This delay is calculated following the current baud rate. Then, for each bit, the line is set/cleared by using the “RC Compare effect on TIOB” feature of the TC. Therefore, for each interrupt, the state automate manages the bit to be sent at the next interrupt. The state machine is initialized by setting the interrupt handler to **IsoTxWaitByte**.

```
Begin
| Disable timer clock and interrupts
| Initialize sent byte count and parity
| Validate start bit on I/O line (TIOB)
| Initialize interrupt handler to 'IsoTxWaitByte'
| Enable interrupt on RC Compare event
| Enable clock and launch timer
End
```

Figure 7. Machine States and the Corresponding C Interrupt Handlers

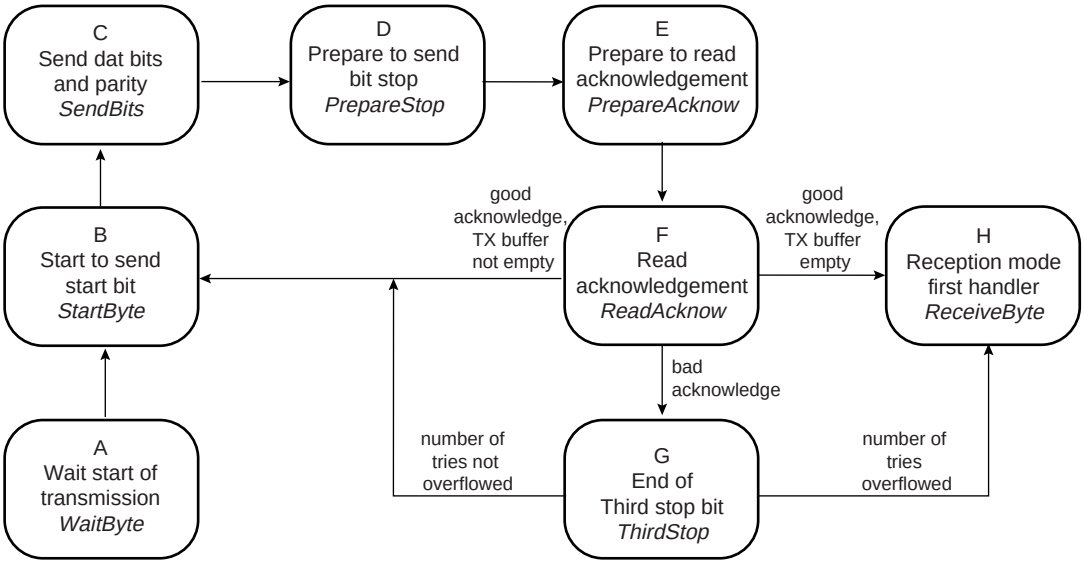


Figure 8. Transmission without Parity Error

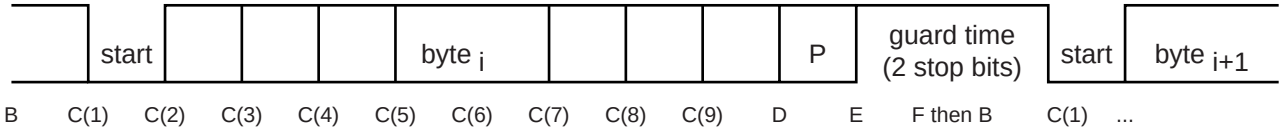
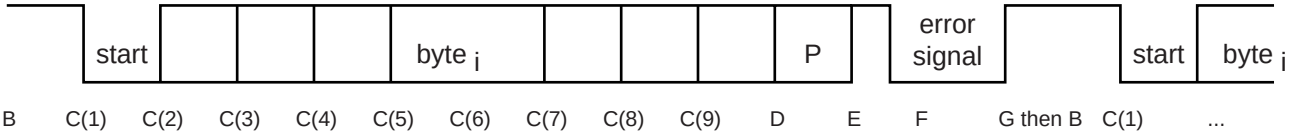


Figure 9. Transmission with Parity Error



Note: 1. () = bit counter value

The **IsoTxWaitByte** function expects that the delay for the line driver switch has expired, then validates the management of the start bit at the next interrupt. The interrupt handler is changed to **IsoTxStartByte**.

```
Begin
| If bit counter is switchTime
| | Change interrupt handler to 'IsoTxStartByte'
| Endif
End
```

The **IsoTxStartByte** function gets the byte to send, initializes the calculated parity and the bit counter, and sets TIOB as an output by selecting XC0 as an external event (disabled). At the next RC Compare, TIOB is cleared in order to send the start bit. The interrupt handler is changed to **IsoTxSendBits**. This function is called to start the transmission of each byte.

```
Begin
| Validate start bit on I/O line (TIOB)
| Get byte to transmit
| Initialize sent byte count and parity
| Change interrupt handler to 'IsoTxSendBits'
End
```

The **IsoTxSendBits** function programs the state of TIOB for the bit to send *at the next bit period.* This is done by using TC_CMR to set or clear TIOB at the next RC Compare event. Once all data bits are sent, the TC is prepared to set TIOB with the parity value, then the interrupt handler is changed to **IsoTxPrepareStop**.

```
Begin
| If bit counter < 9
| | Get bit to transfer and Prepare next one
| | Calculate the parity
| Else
| | Bit to transfer is the calculated parity
| | Change interrupt handler to IsoTxPrepareStop
| Endif
| Set output = bit at next RC Compare event
End
```

The **IsoTxPrepareStop** function handles the interrupt when the parity is being sent. Therefore, TIOB is maintained as an output. The interrupt handler is changed to **IsoTxPrepareAcknow**.

```
Begin
| End of parity => Set output for stop bit
| Change interrupt handler to IsoTxPrepareAcknow
End
```

The **IsoTxPrepareAcknow** function handles the interrupt when the parity is ended. Therefore, TIOB is set as an input (by setting an external trigger on TIOB, without enabling it) in order to read the acknowledgment at the next bit period. The interrupt handler is changed to **IsoTxReadAcknow**.

```
Begin
| Set TIOB as input
| Change interrupt handler to IsoTxReadAcknow
End
```

The **IsoTxReadAcknow** function checks the acknowledgment sent by the receiver (Using PIO Pin Data Status Register (PDSR)). If the byte is not acknowledged (low level), the interrupt handler is changed to **IsoTxThirdStop** in order to send a third stop bit before re-transmitting the byte. If the acknowledgment is correct, the buffer pointer is updated, then either there is another byte to send (buffer not empty) and the cycle is repeated by calling **IsoTxStartByte**, or there are no more bytes to send (buffer empty), and the channel is prepared for a reception by calling the function **IsoRxReceiveByte**.

```
Begin
| If byte acknowledged (High level on TIOB pin)
| | Update tx buffer and clear number of try
| | If tx buffer is empty
| | | start reception (IsoRxReceiveByte)
| | Else (tx buffer not empty)
| | | Send next byte (IsoTxSendByte)
| | Endif
| Else (byte not acknowledged)
| | Change interrupt handler to IsoTxThirdStop
| Endif (byte not acknowledged)
End
```

The **IsoTxThirdStop** function handles the interrupt of the third stop bit. If the limiting number of attempts is not reached then a new attempt is repeated by calling **IsoTxStartByte**. If the limiting number of attempts is reached, the channel is prepared for reception by calling the function **IsoRxReceiveByte**.

```
Begin
| Increase try number
| If try number reached
| | start reception (IsoRxReceiveByte)
| Else
| | Start retransmission (IsoTxSendByte)
| Endif
End
```

Tips and Warnings

Delay Before Transmission

In this application note, a delay has been inserted before sending the first byte. This delay can be removed if the application (pear equipment) does not need it or it can be modified. It is calculated in the function **IsoLineBaudRate**, and saved in the field '*switchTime*' of the variable structure. Note that the real value of this delay is equal to:

('*switchTime*' + 2) * bit time value

There is a first bit time when the timer is software triggered in the function **IsoTxSendByte**, and a second bit time before managing the start bit by the function **IsoTxStartByte**.

Clock and Reset Generation

This application note does not describe how to generate the clock (CLK) and reset (RST) signals. This can easily be done by using an other TC channel configured as a double waveform generator:

- TIOA can be used to generate the clock. It is:
  - cleared on a software trigger
  - toggled on RA and RC Compare events
- TIOB can be used to generate the reset. It is:
  - cleared on a software trigger
  - set on RC Compare after the number of cycles defined by the application

Transfer Convention

This application note describes only the direct convention (LSB first, bits not complemented). The reverse convention can easily be implemented by doing the following:

- Complement the byte after getting it from the Tx buffer (function **IsoTxStartByte**) for transmission, and before saving it in the Rx buffer (function **IsoRxEndReceive**) for reception
- Invert the "shift" methods for transmission (function **IsoTxSendBits**) and reception (function **IsoRxReceiveBits**)
- Initialize the calculated parity at '1' in place of '0' for transmission (function **IsoTxStartByte**) as well as for reception (function **IsoRxReceiveByte**)

Time-out on Reception

The reception described in this application note does not implement the time-out feature. This can easily be done by software triggering the timer at the beginning of a byte reception (function **IsoRxReceiveByte**), and by testing the state of the line in the interrupt management (function **IsoRxGoMiddle**):

- either the line is high and the count bit can be compared to the time-out value
- or the line is low, and the treatment of the start bit detection can be executed

Atmel Headquarters

Corporate Headquarters
2325 Orchard Parkway
San Jose, CA 95131
TEL (408) 441-0311
FAX (408) 487-2600

Europe
Atmel U.K., Ltd.
Coliseum Business Centre
Riverside Way
Camberley, Surrey GU15 3YL
England
TEL (44) 1276-686677
FAX (44) 1276-686697

Asia
Atmel Asia, Ltd.
Room 1219
Chinachem Golden Plaza
77 Mody Road
Tsimshatsui East
Kowloon, Hong Kong
TEL (852) 27219778
FAX (852) 27221369

Japan
Atmel Japan K.K.
Tonetsu Shinkawa Bldg., 9F
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
TEL (81) 3-3523-3551
FAX (81) 3-3523-7581

Atmel Operations

Atmel Colorado Springs
1150 E. Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906
TEL (719) 576-3300
FAX (719) 540-1759

Atmel Rousset
Zone Industrielle
13106 Rousset Cedex, France
TEL (33) 4 42 53 60 00
FAX (33) 4 42 53 60 01

Fax-on-Demand
North America:
1-(800) 292-8635
International:
1-(408) 441-0732

e-mail
literature@atmel.com

Web Site
http://www.atmel.com

BBS
1-(408) 436-4309

1154A–08/98/xM