ES0012
Errata sheet

AT32A403A device limitations

Chip identification

This errata sheet applies to the AT32A403A series. This series features an ARM™ 32-bit Cortex®-M4 core.

Table 1. Device summary

Device	Flash memory	Part numbers
AT32A403A	1024 KB	AT32A403AACGU7, AT32A403AACGT7, AT32A403AARGT7, AT32A403AAVGT7,
	512 KB	AT32A403AACEU7, AT32A403AACET7, AT32A403AARET7, AT32A403AAVET7,
	256 KB	AT32A403AACCU7, AT32A403AACCT7, AT32A403AARCT7, AT32A403AAVCT7,

Contents

List of tables

1 AT32A403A limitations

Table 2 summarizes the limitations on AT32A403A device that have been identified so far.

Table 2. Summary of device limitations

Sections	Description
1.1 CAN	1.1.1. Bit stuffing error causes data error during CAN communication
	1.1.2. Failed to filter RTR bit of standard frame in 32-bit identifier mask mode
	1.1.3. CAN sends unexpected messages in case of narrow pulse disturbance on BS2
	1.1.4 Fail to cancel mailbox transmit command when CAN bus disconnected
1.2 I2C	1.2.1. I2C slave communication failed when APB equals or less than 4MHz
	1.2.2. BUSERR is detected by I2C before start of communication
1.3 I2S	1.3.1. UDR flag is set in I2S slave transmission mode combined with discontinuous communication state
	1.3.2. Data reception error when I2S 24-bit data is packed into 32-bit format
1.4 PWC	1.4.1. Unable to wake up Deepsleep mode after AHB frequency division
	1.4.2. Systick interrupt wakes up Deepsleep mode
1.5 CRM	1.5.1. CLKOUT output exception after entering Deepsleep mode
1.6 TMR	1.6.1. TMR overflow event in encoder mode counter
	1.6.2. Break input failed when TMREN=0 (TMR disabled)
1.7 RTC	1.7.1. Actual RTC counter value is the programmed value plus 1
1.8 FLASH	1.8.1. Program error may occur when sLib is placed in NZW area
	1.8.2. Erasing NZW during code execution causes program exception
	1.8.3. CPU read Flash causes program exception during SPIM erase
1.9 SPI	1.9.1. CS failing edge not synchronized in slave SPI hardware CS mode
1.10 EXINT	1.10.1. Software triggers twice EXINT line interrupt responses

1.1 CAN

1.1.1 Bit stuffing error causes data error during CAN communication

- Description:

  If a bit stuffing error occurs in the data filed during CAN communication due to external disturbance, CAN will stop receiving the current data frame and send an error to the bus, but the next data frame will be out of order while the subsequent messages are able to return to normal automatically.

- Workaround:

  Method 1:

  Enable the error interrupt (its priority must be set very high) corresponding to the interrupt number in the Error Type Record (ETR bit). Once a bit stuffing error is detected, reset CAN (only need reset CAN registers and relevant GPIOs, without resetting NVIC), and re-initialize CAN in the CAN error interrupt function.

  This method applies to the scenario where a quick CAN initialization is required in order to ensure a quick resume of CAN communication, avoiding excess CAN data loss.

  Take a CAN1 as an example, its typical code as follows:

```
/* Enable CAN error interrupt corresponding to the last CAN error interrupt number and give very high
        priority */
  nvic_irq_enable(CAN1_SE_IRQn, 0x00, 0x00);
  can_interrupt_enable(CAN1, CAN_ETRIEN_INT, TRUE);
  can_interrupt_enable(CAN1, CAN_EOIEN_INT, TRUE);
/* Interrupt service functions */
void CAN1_SE_IRQHandler(void)
{
  __IO uint32_t err_index = 0;
  if(can_flag_get(CAN1,CAN_ETR_FLAG) != RESET)
  {
    err_index = CAN1->ests & 0x70;
    can_flag_clear(CAN1, CAN_ETR_FLAG);
    if(err_index == 0x00000010)
    {
      can_reset(CAN1);
      /* Call CAN initialization function*/
    }
  }
}
```

Notes:

a) The interrupt in CAN Error Type Record (ETR) should be given with very high priority

b) As it takes some time to finish CAN initialization, CAN’s inability to resume communication immediately when an error occurs may cause loss of data.

Method 2:

Enable the error interrupt (its priority must be set as very high) corresponding to the CAN error interrupt number in the Error Type Record (ETR bit). Once a bit stuffing error is detected, reset CAN (only need reset CAN registers and relevant GPIOs, without resetting NVIC), record the reset event, and re-initialize CAN in other low-priority interrupts or main functions.

This method applies to the scenario where the CAN communication is unable to resume in time, but the CAN re-initialization must be performed in order not to affect operations of other applications.

Take a CAN1 as an example, its typical code as follows:

```
/* Enable CAN error interrupt corresponding to the last CAN error interrupt number and give very high
        priority */
  nvic_irq_enable(CAN1_SE_IRQn, 0x00, 0x00);
  can_interrupt_enable(CAN1, CAN_ETRIEN_INT, TRUE);
  can_interrupt_enable(CAN1, CAN_EOIEN_INT, TRUE);
/* Interrupt service functions */
__IO uint32_t can_reset_index = 0;
void CAN1_SE_IRQHandler(void)
{
  __IO uint32_t err_index = 0;
  if(can_flag_get(CAN1,CAN_ETR_FLAG) != RESET)
  {
    err_index = CAN1->ests & 0x70;
    can_flag_clear(CAN1, CAN_ETR_FLAG);
    if(err_index == 0x00000010)
    {
      can_reset(CAN1);
      can_reset_index = 1;
    }
  }
}
```

Then the application polls whether “can_reset_index” is set or not at the desired place (in main functions, say). If set, call the CAN initialization function.

Notes:

a) The interrupt in CAN Error Type Record should be given very high priority

b) As it takes some time to finish CAN initialization, CAN’s inability to resume communication immediately when an error occurs may cause loss of data.

Method 3:

Enable CAN error interrupt (its priority must be set as very high) corresponding to the CAN error interrupt number in the Error Type Record (ETR bit). Once a bit stuffing error is detected, send an invalid message with a very-high-priority identifier.

This method applies to the scenario in which one doesn't want to spend time on resetting CAN , all message identifiers on CAN bus are known, and each CAN node receives messages in accordance with the identifier filtering conditions.

Take a CAN1 as an example, its typical code as follows:

```
/* Send a frame of invalid message with a very-high-priority identifier */
static void can_transmit_data(void)
{
        uint8_t transmit_mailbox;
        can_tx_message_type tx_message_struct;
        tx_message_struct.standard_id = 0x0;
        tx_message_struct.extended_id = 0x0;
        tx_message_struct.id_type = CAN_ID_STANDARD;
        tx_message_struct.frame_type = CAN_TFT_DATA;
        tx_message_struct.dlc = 8;
        tx_message_struct.data[0] = 0x00;
        tx_message_struct.data[1] = 0x00;
        tx_message_struct.data[2] = 0x00;
        tx_message_struct.data[3] = 0x00;
        tx_message_struct.data[4] = 0x00;
        tx_message_struct.data[5] = 0x00;
        tx_message_struct.data[6] = 0x00;
        tx_message_struct.data[7] = 0x00;
        can_message_transmit(CAN1, &tx_message_struct);
}
/* Enable the error interrupt corresponding to the last CAN error interrupt number and set very high interrupt
            priority*/
    nvic_irq_enable(CAN1_SE_IRQn, 0x00, 0x00);
    can_interrupt_enable(CAN1, CAN_ETRIEN_INT, TRUE);
    can_interrupt_enable(CAN1, CAN_EOIEN_INT, TRUE);
/* Interrupt service function*/
void CAN1_SE_IRQHandler(void)
{
    __IO uint32_t err_index = 0;
    if(can_flag_get(CAN1,CAN_ETR_FLAG) != RESET)
    {
        err_index = CAN1->ests & 0x70;
        can_flag_clear(CAN1, CAN_ETR_FLAG);
        if(err_index == 0x00000010)
        {
           can_transmit_data();
        }
    }
}
```

Notes:

a) The interrupt in CAN Error Type Record should be configured as very high priority

b) This method is only applicable to the scenario where the transmit FIFO priority is determined by the message identifier.

c) The identifier of the invalid message of this method is changeable. But its priority must be the highest among the CAN bus, and it cannot be received as a normal message by other nodes.

1.1.2 Failed to filter RTR bit of standard frame in 32-bit identifier mask mode

- Description:

  When the CAN filter mode is configured in 32-bit identifier mask mode, the RTR bit (remote frame identifier) is unable to be filtered effectively during a standard frame filtering.

  When the following conditions are met, follow the “Workaround” below to solve this problem:

  1. 32-bit wide identifier mask mode is used
  2. A standard frame is being filtered but the remote frame passing through filter is unwanted

- Workaround:

  Method 1: By software. When filtering a standard frame in 32-bit wide identifier mask mode, the software is used to get the status of the RTR bit (remote frame identifier) and decide if whether this frame is of interest. For example,

```
void CAN1_RX0_IRQHandler(void)
{
    can_rx_message_type rx_message_struct;
    if(can_flag_get(CAN1,CAN_RF0MN_FLAG) != RESET)
    {
        can_message_receive(CAN1, CAN_RX_FIFO0, &rx_message_struct);
        /* only store the data frame,discard the remote frame */
        if((rx_message_struct.id_type == CAN_ID_STANDARD) && (rx_message_struct.frame_type ==
CAN_TFT_DATA))
        {
           /* user store the receive data */
        }
    }
}
```

  Method 2: Use other filtering mode according to the needs, such as, 32-bit wide identifier list mode, 16-bit wide identifier mask mode or 16-bit wide identifier list mode.

1.1.3 CAN sends unexpected messages in case of narrow pulse disturbance on BS2

- Description:

  In case of a large amount of narrow pulses (pulse width less than 1tp) on CAN bus, the CAN nodes are likely to send unexpected messages, for instance, a data frame is sent as a remote frame, a standard frame as an extended one, or data phase error occurs.

- Workaround:

  Set synchronization width RSAW = BTS2 segment width in order to avoid unexpected errors.

  It should be noted that after RSAW =BTS2 is asserted, the CAN bus communication speed is reduced when there is a lot of disturbance on CAN bus.

```
static void can_configuration(void)
{
  …

  /* can baudrate, set baudrate = pclk/(baudrate_div *(3 + bts1_size + bts2_size)) */
  can_baudrate_struct.baudrate_div = 12;
  can_baudrate_struct.rsaw_size = CAN_RSAW_3TQ;
  can_baudrate_struct.bts1_size = CAN_BTS1_8TQ;
  can_baudrate_struct.bts2_size = CAN_BTS2_3TQ;

  …
}
```

1.1.4 Fail to cancel mailbox transmit command when CAN bus disconnected

- Description:

  As a node for data transmission, if the following two conditions are both present for CAN, it is not possible to clear or cancel a transmit command in a mailbox within CAN error passive interrupt, causing that the to-be-sent message command has not been canceled during the period of CAN bus being disconnected, and such message would be retransmitted after CAN bus communication resumes.

  1. CAN bus (CANH/L) is disconnected intentially or accidentally
  2. Auotmaitc retransmission feature is enabled

- Workaround:

  Enable CAN error passive interrupt and disable its automatic retransmission before re-enabling automatic retransmission in the message transmit function, as shown below:

  1) Enable error passive interrupt during CAN initialization

```
nvic_irq_enable(CAN1_SE_IRQn, 0x00, 0x00);
can_interrupt_enable(CAN1, CAN_EPIEN_INT, TRUE);
can_interrupt_enable(CAN1, CAN_EOIEN_INT, TRUE);
```

  2) Disable automatic transmission feature in CAN error passive interrupt function

```
void CAN1_SE_IRQHandler(void)
{
  if(can_flag_get(CAN1,CAN_EPF_FLAG) != RESET)
  {
    CAN1->mctrl |= (uint32_t)(1<<4);
    can_flag_clear(CAN1, CAN_EPF_FLAG);
  }
}
```

3) Re-enable automatic transmission feature in CAN message transmit function

```
CAN1->mctrl &= (uint32_t)~(1<<4);
```

1.2 I2C

1.2.1 I2C slave communication failed when APB equals or less than 4MHz

- Description:

  I2C is unable to communicate at 400kHz in slave mode when the APB clock is equal to or less than 4MHz.

- Workaround:

  Increase the APB clock to 8 MHz, or reduce the I2C speed to 100kHz.

1.2.2 BUSERR is detected by I2C before start of communication

- Description:

  When all the following conditions are met, BUSERR condition would be detected by I2C, causing communication error.

  Condition 1: I2C is enabled

  Condition 2: Before the start of communication

  Condition 3: BUSERR timing takes place on the bus

- Workaround:

  Check if the BUSERR flag is set or not before the start of communication. If it is set, just need clear this flag to enable communication. Optionally, enable error interrupt, and clear it in the interrupt after the BUSERR flag is set.

1.3 I2S

1.3.1 UDR flag is set in I2S slave transmission mode combined with discontinuous communication state

- Description:

  The UDR flag is set incorrectly, when I2S is in slave transmission mode and in discontinuous communication state, even if data have been written before the start of communication.

- Workaround:

  According to the protocol, it is recommended to use DMA or interrupts for fast data transfer in I2S slave transmission mode to ensure smooth communication.

1.3.2 Data reception error when I2S 24-bit data is packed into 32-bit format

- Description:

  When I2S 24-bit data is packed into 32-bit frame format, the remaining 8 invalid CLK data would be received as normal data by the receiver.

- Workaround:

  Method 1: Both the receiver and transmitter use the same way of packing 24-bit data into 32-bit format.
  Method 2: Discard these 8 invalid CLK data in the frame format using software.

1.4 PWC

1.4.1 Unable to wake up Deepsleep mode after AHB frequency division

- Description:
  After the AHB frequency is divided, no wakeup sources can wake up the device from Deepsleep mode.
- Workaround:
  Do not perform AHB frequency division in Deepsleep mode.
  Remove AHB frequency division before entering Deepsleep mode, and then configure the desired AHB frequency division after waking up from Deepsleep mode.

1.4.2 Systick interrupt wakes up Deepsleep mode

- Description:
  If the Systick or Systick interrupt is not disabled before entering Deepsleep mode, the Systick would keep running after Deepsleep mode entry, and the subsequent Systick interrupt would wake up Deepsleep mode.
- Workaround:
  Disable Systick or Systick interrupt before entering Deepsleep mode.

1.5 CRM

1.5.1 CLKOUT output exception after entering Deepsleep mode

- Description:
  If the DEEPSLEEP_DEBUG bit is set 0, and the CLKOUT is used as a system clock output, there would still be clock output (at LICK clock frequency) on the CLKOUT pin after the Deepsleep mode is entered.
- Workaround:
  Configure CLKOUT as NOCLK before entering Deepsleep mode, and then configure it as system clock output after leaving Deepsleep mode.

1.6 TMR

1.6.1 TMR overflow event in encoder mode counter

- Description:

  In encoder mode counter, if the counter counts back and forth between 0 and PR, the OVFIF bit of the TMR would not be set either at an overflow or underflow event.
- Workaround 1:

  Configure the C3IF and C4IF channels of the TMR which is using the encoder as output mode, set C3DT = AR, and C4DT = 0, and enable C3DT and C4IF interrupts.
  For C3IF event & downcounting interrupt, it indicates an underrun
  For C4IF event & upcounting interrupt, it indicates an overrun
  This method has its limitation: If the input frequency of the encoder mode counter is so fast, interrupts would occur frequently and have to be handled by software, causing not enough time for software to deal with interrupts. Thus this method applies to the scenario where the external input frequency of the encoder is not so fast.
- Workaround 2:

  Turn to a TMR with enhanced mode (the counter can be extended from 16-bit to 32-bit width) in order to expand the encoder's counting range that detects forward and reverse rotation, and set the initial value of the counter to PR/2 so as to avoid timer overflow
  Note: The forward and reverse rotation of the encoder must be limited to a certain range. An overflow still occurs if the encoder were always rotated in one direction. Thus This method applies to the scenario where the rotation of the encoder is controlled at certain range.

1.6.2 Break input failed when TMREN=0 (TMR disabled)

- Description:

  When TMREN=0 (Timer is not enbled), break input failed to work, causing it unable to trigger break event or interrupt.

  Example: in single-pulse mode, TMREN is cleared (0) automatically at the end of one-cycle counting. But due to above-mentioned reason relating to break input, output enable bit (OEN) cannot be cleared, nor can a break flag be set.

- Workaround:

  None.

1.7 RTC

1.7.1 Actual RTC counter value is the programmed value plus 1

- Description:

  After the completion of RTC value setting, the actual RTC value turns out to be the programmed value plus 1.

- Workaround:

  Set a frequency division value before programming RTC counter value.

```
rtc_wait_config_finish();
rtc_divider_set(32767);

rtc_wait_config_finish();
rtc_counter_set(100);
```

1.8 Flash

1.8.1 Program error may occur when sLib is placed in NZW area

- Description:

  When the sLib (security library) is placed in the non-zero-wait area, if the CPU accesses I-code and D-code back and forth quickly during program runtime, it is likely that the mismatched bus is mistakenly used to access sLib given the fact that accessing internal Flash memory takes some time during bus ID switching. This mismatching operation sends an erroneous instruction (or data) to CPU, causing program error.

- Workaround:

  Place sLib in the zero-wait area.

1.8.2 Erasing NZW during code execution causes program exception

- Description:

  The erase operation in the zero-wait (ZW) area does not affect program running. However, if the program contains instructions from both zero-wait (ZW) and non-zero-wait (NZW) area, the program exception may occur because of reading data in NZW area.

  For instance, an interrupt can be handled during ZW area erase operation, but if the interrupt handler functions involve both ZW and ZW areas, the program exception may occur.

- Workaround:

  The reason behind this issue is that all execution codes must be placed in the same area (all in ZW or NZW area) during ZW erase operation.

  To slove this problem, you need disable interrupt enable bits before erase, and then enable them after the completion of erase. Meanwhile, the code related to erase functions must be placed in the same area.

1.8.3 CPU read Flash causes program exception during SPIM erase

- Description:

  If CPU reads Flash memory during SPIM erase operation, the read Flash command would be treated as reading SPIM mistakenly, causing data error and then program exception.

  For instance, SPIM erase functions are compiled in NZW area, so they fetch instructions from NZW area during erase operation. During this period, reading Flash by CPU will cause program error.

- Workaround:

  Do not read Flash during the process of SPIM erase operation.

  To make this happen, disable interrupt enable bits before starting erase, and then enable interrupt enable bits after the completion of erase. And the codes related to erase functions must be compiled in ZW area or RAM.

1.9 SPI

1.9.1 CS failing edge not synchronized in slave SPI hardware CS mode

- Description:

  In SPI slave hardware CS mode, the initial CLK synchronization for data transfer is not performed at each CS falling edge.

- Workaround:

  Solution A: Strictly control the slave CS line, and pull high the CS line as soon as the communication is complete.

  Solution B: Enable CRC check. Once a CRC error is detected, reset SPI and restart handshake communication.

1.10 EXINT

1.10.1 Software triggers twice EXINT line interrupt responses

- Description:

  When the software trigger function is used for EXINT line, twice interrupt responses are generated on the same EXINT line at each software trigger command.

- Workaround:

  Enter the interrupt function associated with software trigger, and perform the EXINT flag clear command twice.

  Alternatively, use the flag clear function defined in the latest version of BSP (described below), or modify code based on the following method.

```
void exint_flag_clear(uint32_t exint_line)
{
        if((EXINT->swtrg & exint_line) == exint_line)
        {
            EXINT->intsts = exint_line;
            EXINT->intsts = exint_line;
        }
        else
        {
            EXINT->intsts = exint_line;
        }
}
```

2 Document revision history

Table 3. Document revision history

Date	Version	Changes
2023.08.03	2.0.0	Initial release