Application Note

Local Interconnect Network (LIN) Automatic Baud Rate Detection Mechanism for MSPM0 Devices

Kalyan Gajjala, Gaurang Gupta, Ashwini Gopinath *EP-MSP*

ABSTRACT

This application note provides a detailed overview of the automatic baud rate detection mechanism provided by MSPM0 UART and UNICOMM UART receiver LIN nodes. The information presented is specifically relevant to devices that achieve the required clock frequency tolerance through synchronization methods. In MSPM0 UART and UNICOMM UART modules, automatic baud rate detection is accomplished using a combination of hardware and software solutions.

This document begins by outlining the tolerance requirements specified by the LIN (Local Interconnect Network) protocol. It then describes how MSPM0 UART/UNICOMM UART modules are designed to meet these tolerance limits while reliably detecting variable baud rates. The application note breaks down the synchronization process, enabling accurate data reception even in systems with potential clock inaccuracies. Integration details, practical implementation methodologies, and application-specific considerations are covered to aid embedded system designers in leveraging automatic baud rate detection effectively within their projects.

Table of Contents

Trademarks

All trademarks are the property of their respective owners.

1 Introduction to LIN Protocol

LIN (Local Interconnect Network) is a serial communication protocol specifically designed for automotive applications. It utilizes the low-cost standard UART interface, enabling deployment through software solution. LIN protocol allows speeds from 1 to 20Kb/S.

A standard LIN frame is divided into two parts: Header and Response as shown in Figure 1-1. Header consists of break field, Sync field and protected identifier (PID) field. Response consists of data and checksum.

An inter-byte space is simply the time between the end of one byte's stop bit and the start bit of the next byte. The response space is the time between the PID field and the first data byte.

Every byte field (except the break field) is sent in a way that the least significant bit (LSB) is sent first, and the most significant bit (MSB) is sent last.

Some nomenclature which will be used for rest of this Application Note (used as is from LIN specification for ease of use):

- Nominal bit rate : F_{Nom}
- Deviation of Responder node from nominal bit node before synchronization: F_{TOL_UNSYNC} (<+/-14%).
- Deviation of Responder node after synchronization: F_{TOL_SYNC} (<+/- 1.5%).

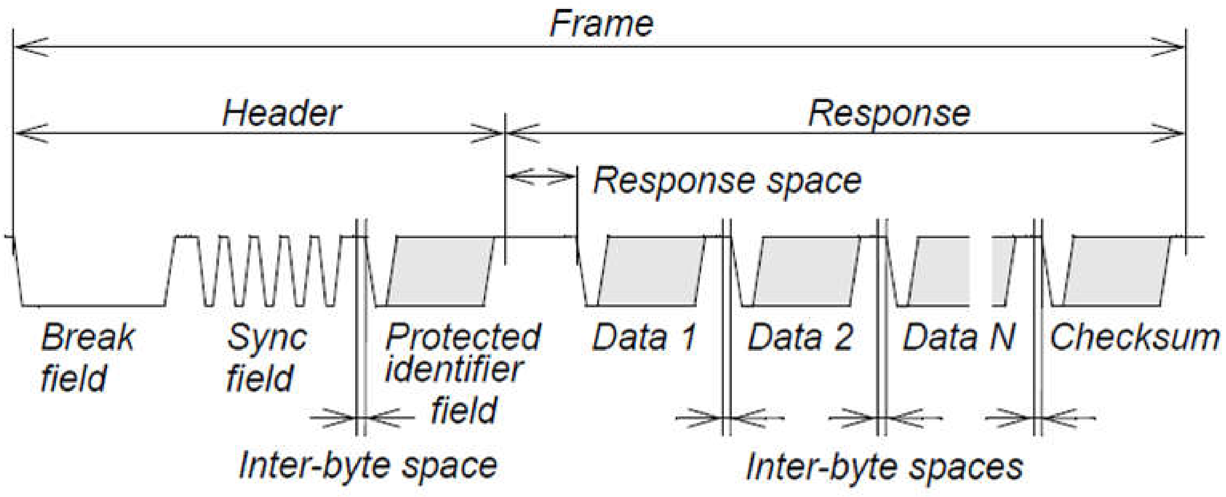


Figure 1-1. LIN Frame

1.1 Break Field

The break field signals the start of a LIN frame and is always sent by the commander node with at least 13 nominal bit times in the dominant state(zeros), followed by a break delimiter, which must be at least one nominal bit time as shown in the following figure. Responder nodes can detect the break field when they see at least 9.5 consecutive dominant bits and don't need to verify that the break delimiter is actually 1-bit time long.

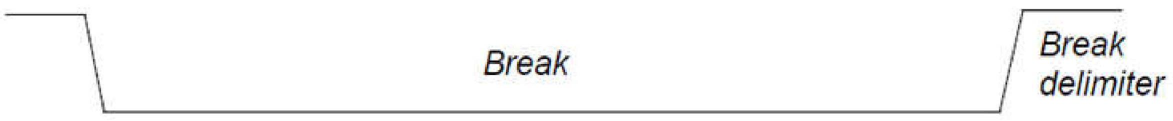


Figure 1-2. Break Field

1.2 SYNC Byte Field

The **Sync Field**, a byte with the fixed hexadecimal value **0x55**, enables devices on a LIN bus to synchronize their clocks with the master node. The value 0x55 represents a binary pattern of alternating 1s and 0s, creating predictable signal transitions (edges).

Here is how this works:

- The LIN commander transmits this alternating pattern.
- The Responder nodes measure the time between these signal transitions to determine the master's precise bit timing.
- The Responder nodes then use this measurement to adjust their baud rate, ensuring they are perfectly synchronized with the Responder's baud rate.

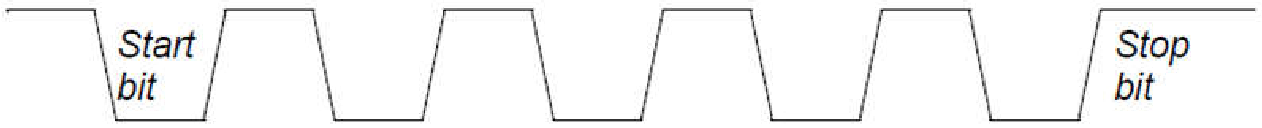


Figure 1-3. Sync Field

1.3 PID Field

The PID (Protected Identifier) in LIN is an 8-bit field within each LIN frame header that uniquely identifies the message is shown in the following figure.

It contains a 6-bit identifier (ID), allowing for 64 unique message types and remaining 2 bits are parity bits (called P0 and P1), calculated from the ID bits to help detect errors in the transmission.

P0 = ID0 XOR ID1 XOR ID2 XOR ID4

P1 = NOT(ID1 XOR ID3 XOR ID4 XOR ID5)

The PID is always sent by the commander node, and all responder nodes use it to decide if they should respond to the message, ignore it, or just listen.

The parity bits help ensure the reliability of the data by allowing nodes to check if the ID was received correctly.

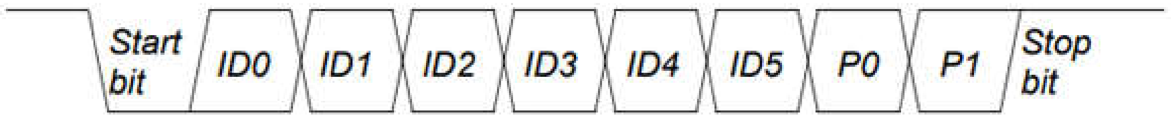


Figure 1-4. PID Field

1.4 Data

The data field carries the actual information exchanged between commander and responder nodes, such as sensor readings or control commands.

The data field typically contains between 1 and 8 bytes which represents the real payload. The number of data bytes contained in a frame defined by the frame's PID and agreed between commander and responder nodes.

When the commander node requests data by sending a header, the responder node replies by populating the data field with relevant data and sending it back.

The data is always followed by a checksum byte, which helps all nodes confirm that the data was received correctly and not corrupted during transmission.

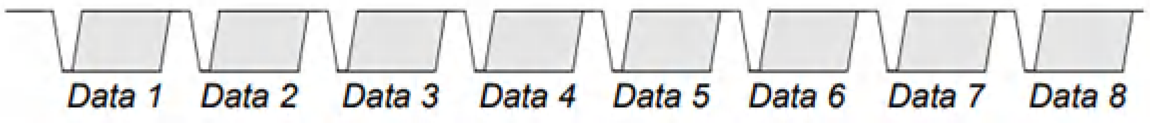


Figure 1-5. Data

1.5 Checksum

The checksum is always the last byte in the response part of the LIN frame and is sent by the node sending the data. Checksum calculation is adding up all the data bytes or all data bytes and the protected identifier (using modulo-256 addition, so the sum "wraps around" if it's higher than 255), then invert the total so that adding the checksum to all the bytes equals 0xFF.

2 Initial Baud Rate Setting

Refer to the MSPM0 device specific user guide for the UART Functional Block Diagram. The incoming clock to the MSPM0 UART/UNICOMM UART can be further divided using CLKDIV register bitfield; this divided clock is referred to as the functional clock or UART clock. Refer to the respective device datasheet for the clock speed of the IP.

The baud-rate divisor is a 22-bit number consisting of a 16-bit integer (IBRD) and a 6-bit fractional part (FBRD). The number formed by these two values is used by the baud-rate generator to determine the bit sample period. Having a fractional baud-rate divisor allows UART to generate all standard baud-rates very accurately.

The 16-bit integer is loaded into UART Integer Baud-Rate Divisor IBRD register and the 6-bit fractional part is loaded into UART Fractional Baud-Rate Divisor FBRD register.

The baud-rate divisor (BRD) can be calculated by using the following formula:

BRD =functional clock / (Oversampling x Baud rate)

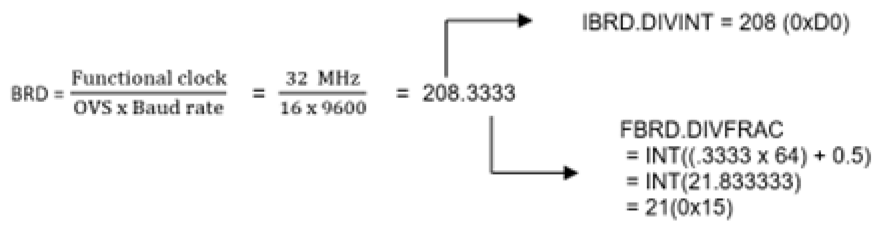
Functional clock is the clock output of the UART clock control logic, configured by CLKSEL and CLKDIV. Oversampling is selected by high-speed oversampling enable (HSE) bit in CTL0 register and the selected oversampling can be 16, 8, or 3.

- IBRD = INT(BRD): contains integer part of baud rate divisor
- FBRD = INT ((BRD – INT(BRD)*64+0.5): contains residual fractional part of baud rate divisor

The integer part of BRD is loaded into IBRD register. The 6-bit fractional number must be loaded into the FBRD register.

The following example shows a simple method to calculate IBRD.DIVINT and FBRD.DIVFRAC for a baud rate of 9600 bit/s:

- Functional clock = 32 MHz
- Oversampling = 16
- Baud rate = 9600 bit/s


IBRD.DIVINT = 208 (0xD0)
BRD = Functional clock / OVS x Baud rate = 32 MHz / 16 x 9600 = 208.3333
FBRD.DIVFRAC
= INT((.3333 x 64) + 0.5)
= INT(21.833333)
= 21(0x15)

3 Realization of LIN Protocol MSPM0 UART/ UNICOMM UART

The UART module in MSPM0/UNICOMM incorporates dedicated hardware features to support Local Interconnect Network (LIN) protocol implementation. These hardware enhancements are specifically designed to reduce software overhead and ensure precise timing control required for LIN communications. For supporting local interconnect network (LIN) protocol, the following hardware enhancements are implemented in the UART module:

- 16 bit up-counter (LINCNT) clocked by the UART clock (for more details on how the UART clock is derived refer to the TRM).
- Interrupt capability on counter overflow (CPU_INT.IMASK.LINOVF).
- 16 bit capture register (LINC0) with two configurable modes:
  - Capture of LINCNT value on RXD falling edge. Interrupt capability on capture
  - Compare of LINCNT with interrupt capability on match
- 16 bit capture register (LINC1) can be configured:
- Capture LINCNT value on RXD rising edge. Interrupt capability on capture.
- Interrupt capability on Rx Rising edge(RXPE) and Rx Falling edge(RXNE).

The UART module in MSP devices can be used as a LIN Commander and a LIN Responder. The sections below describe in detail about the enhancements done in order to support the LIN protocol.

Note

If both LINC0_CAP and LINC0_MATCH are set to 1 at the same time, the LINC0 register will behave in MATCH mode as the MATCH behavior will override the CAP behavior.

In UNICOMM based devices, LINC0 MMR should be configured before configuring CLKDIV if CLKDIV needs to be configured to a non-zero value. (Refer to device errata for more details).

3.1 LIN Transmission

Commander Mode: MSPM0 UART/UNICOMM UART generates the break, SYNC, and PID, and then sends or receives data after decoding the PID.

3.1.1 Break Field

The break field is a key part of LIN communication that marks the start of a LIN frame. To generate the break field in MSP devices, the LCRH.BRK bit must be used. Setting the BRK bit to in UART.LCRH register generates the break field by forcing UART to output continuous low signal.

The duration is controlled by the software/application.

The LCRH.BRK bit must be set BEFORE putting any data in TXDATA/FIFO. The duration of the break field is at least 13 dominant bit times, followed by one-bit timer for break delimiter.

Once break field has been sent, the LCRH.BRK bit should be cleared.

Implementation Sequence:

- Assert LCRH.BRK bit
- Maintain dominant state for required duration using software
- Clear LCRH.BRK bit for delimiter

Note

LCRH.BRK bit configuration must precede any TXDATA/FIFO operations to ensure proper Break field generation.

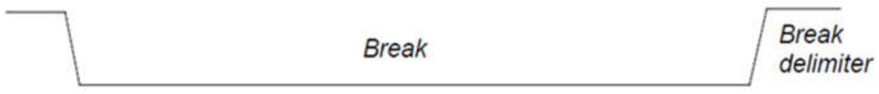


Figure 3-1. Break Field Depiction

Alternatively, the LIN break field can be precisely timed using LINCNT and LINC0 in Compare Mode. LINC0 can be configured to match the break field duration, LCRH.BRK can be set to initiate break field transmission, then LINCNT can be enabled. The break field ends when LCRH.BRK is cleared in the LINC0 ISR. Moreover, interrupts from other IPs like GPTIMER can be used for this purpose.

3.1.2 Sync Field

Following Break field transmission and LCRH.BRK bit de-assertion, the synchronization field (0x55) must be loaded into the UART TXFIFO to initiate the SYNC field transmission phase.

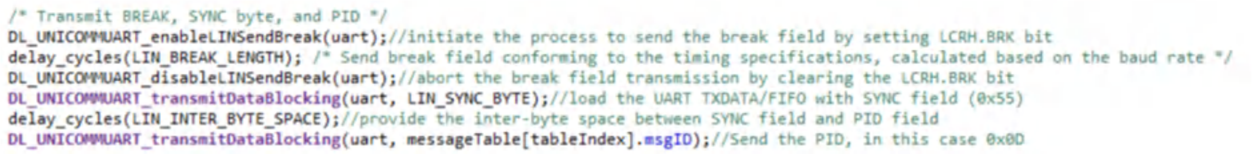
3.1.3 PID Field

Following SYNC field transmission completion, the Protected Identifier (PID) must be loaded into TXDATA/FIFO, maintaining requisite inter-byte spacing between SYNC field and PID transmission sequences.

3.1.4 Data Field

Following Protected Identifier (PID) transmission completion, the TXDATA/FIFO can be populated with the requisite data bytes designated for transmission in the LIN frame.

The following code sequence demonstrates the implementation protocol for sequential transmission of Break field, Synchronization field, and Protected Identifier (PID) in the LIN frame header:



```
/* Transmit BREAK, SYNC byte, and PID */
DL_UNICOMMUART_enableLINSendBreak(uart);//initiate the process to send the break field by setting LCRH.BRK bit
delay_cycles(LIN_BREAK_LENGTH); /* Send break field conforming to the timing specifications, calculated based on the baud rate */
DL_UNICOMMUART_disableLINSendBreak(uart);//abort the break field transmission by clearing the LCRH.BRK bit
DL_UNICOMMUART_transmitDataBlocking(uart, LIN_SYNC_BYTE);//load the UART TXDATA/FIFO with SYNC field (0x55)
delay_cycles(LIN_INTER_BYTE_SPACE);//provide the inter-byte space between SYNC field and PID field
DL_UNICOMMUART_transmitDataBlocking(uart, messageTable[tableIndex].msgID);//Send the PID, in this case 0x0D
```

Figure 3-2. Software Sequence to Transmit Break Field, Sync Field and Data Bytes

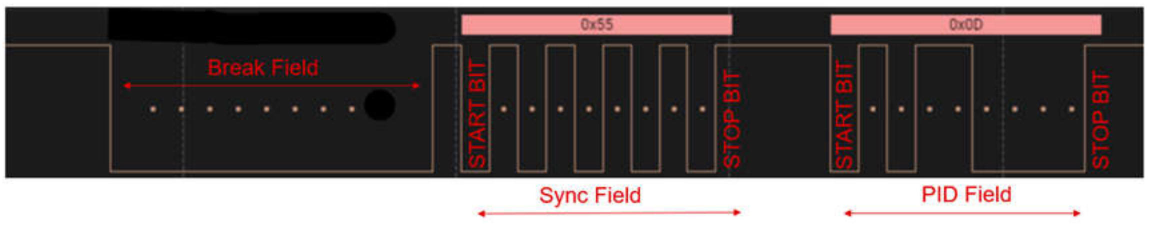


Figure 3-3. Break, Sync, PID Fields Transmitted by the LIN Commander

3.1.5 Checksum

The LIN protocol implements an 8-bit error detection mechanism utilizing an inverted checksum calculation. The checksum computation incorporates sequential byte summation with carry management for values exceeding 0xFF, followed by one's complement operation on the final sum.

Two checksum methodologies are specified:

1. Classic Checksum: Computation encompasses data bytes exclusively
2. Enhanced Checksum: Computation encompasses Protected Identifier (PID) concatenated with data bytes

Checksum Algorithm:

- Sequential byte accumulation
- Carry addition for sums exceeding 0xFF
- Result inversion (one's complement)

Implementation Example:

Protected Identifier: 0x0D

Data Field: [0xAB, 0xBC, 0xCD, 0xDE, 0xEF]

Enhanced Checksum Computation Sequence:

Step 1: 0x0D

Initial value = 0x0D

Step 2: Add 0xAB

0x0D + 0xAB = 0xB8

(No need to subtract 255 as sum < 256)

Step 3: Add 0xBC

0xB8 + 0xBC = 0x174

When sum ≥ 256 (0x100), subtract 255 (0xFF)

0x174 - 0xFF = 0x75

Step 4: Add 0xCD

0x75 + 0xCD = 0x142

When sum ≥ 256, subtract 255

0x142 - 0xFF = 0x43

Step 5: Add 0xDE

0x43 + 0xDE = 0x121

When sum ≥ 256, subtract 255

0x121 - 0xFF = 0x22

Step 6: Add 0xEF

0x22 + 0xEF = 0x111

When sum ≥ 256, subtract 255

0x111 - 0xFF = 0x12

Final Step: Invert the result

Checksum = 0xFF - 0x12 = 0xED

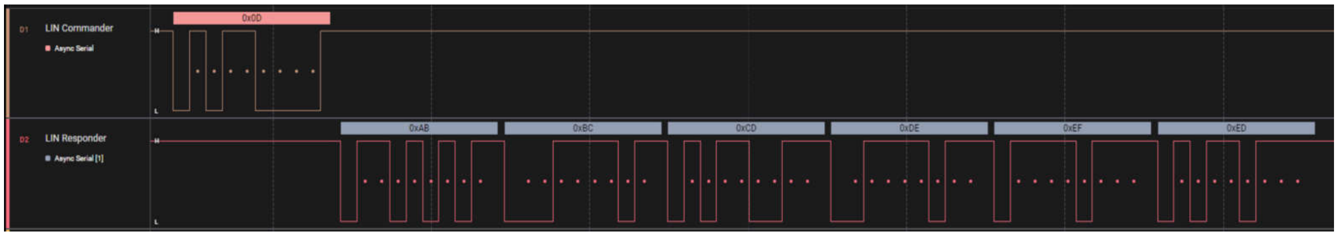
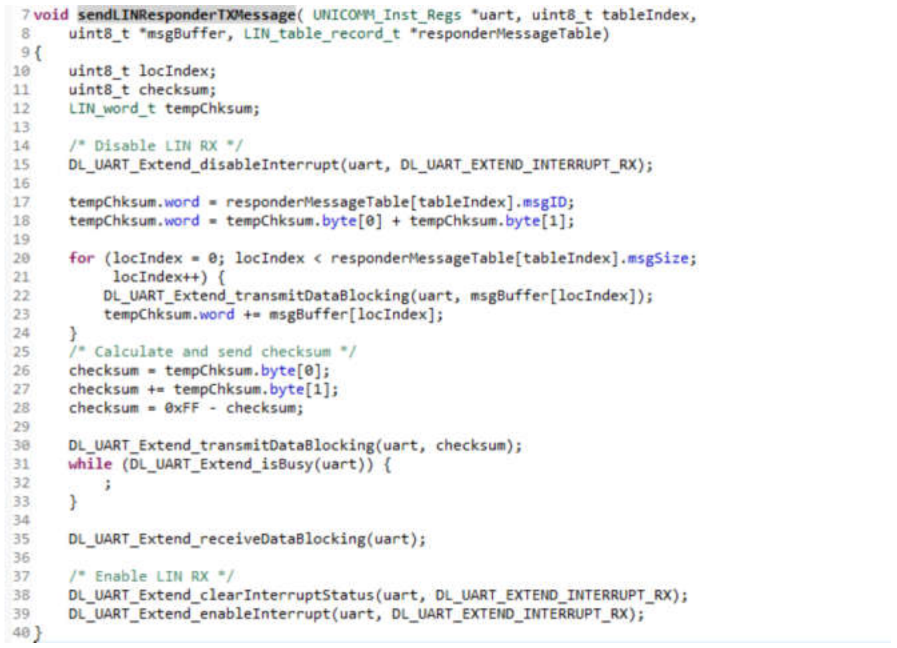


Figure 3-4. Transmission of Checksum

The code section attached below demonstrates the process to calculate and send the LIN Checksum.



```
void sendLINResponderTXMessage( UNICOMM_Inst_Regs *uart, uint8_t tableIndex,
    uint8_t *msgBuffer, LIN_table_record_t *responderMessageTable)
{
    uint8_t locIndex;
    uint8_t checksum;
    LIN_word_t tempChksum;

    /* Disable LIN RX */
    DL_UART_Extend_disableInterrupt(uart, DL_UART_EXTEND_INTERRUPT_RX);

    tempChksum.word = responderMessageTable[tableIndex].msgID;
    tempChksum.word = tempChksum.byte[0] + tempChksum.byte[1];

    for (locIndex = 0; locIndex < responderMessageTable[tableIndex].msgSize;
         locIndex++) {
        DL_UART_Extend_transmitDataBlocking(uart, msgBuffer[locIndex]);
        tempChksum.word += msgBuffer[locIndex];
    }
    /* Calculate and send checksum */
    checksum = tempChksum.byte[0];
    checksum += tempChksum.byte[1];
    checksum = 0xFF - checksum;

    DL_UART_Extend_transmitDataBlocking(uart, checksum);
    while (DL_UART_Extend_isBusy(uart)) {
        ;
    }

    DL_UART_Extend_receiveDataBlocking(uart);

    /* Enable LIN RX */
    DL_UART_Extend_clearInterruptStatus(uart, DL_UART_EXTEND_INTERRUPT_RX);
    DL_UART_Extend_enableInterrupt(uart, DL_UART_EXTEND_INTERRUPT_RX);
}
```

Figure 3-5. Software Sequence to Transmit Checksum

3.2 LIN Reception

Responder Mode: MSPM0 UART/UNICOMM UART waits for break detection and then sends or receives data after decoding the PID.

LIN commander issues a break field and sync field at the start of every frame. Hardware has been added such that the LIN responder software driver can reasonably detect BREAK-SYNC and measure the necessary timing parameters to adjust the baud rate or determine an error.

3.2.1 Break Field Detection

The reception of LIN frames requires precise Break field detection utilizing counter and compare mode functionalities.

Configuration sequence:

1. Counter Initialization
   - Reset LIN counter (UARTx.LINCNT = 0)
2. Compare Mode Configuration
   - Assert counter compare match mode (UARTx.LINCTL.LINC0_MATCH = 1)
   - Configure UARTx.LINC0 with 9.5 x Tbit threshold value
   - Enable LINC0 match interrupt (CPU_INT.IMASK.LINC0 = 1)
3. Counter Control Parameters (UARTx.LINCTL)
   - Enable RXD low-state counting (LINCTL.CNTRXLOW = 1)
   - Configure falling-edge counter reset (LINCTL.ZERONE = 1)
   - Activate counter operation (LINCTL.CTRENA = 1)
4. Detection Features
   - RX rising edge interrupt capability (CPU_INT.IMASK.RXPE = 1)
   - When the RXPE interrupt fires, the software can read the LINCNT directly to see the BREAK field timing.

Optional: User can enable the LIN counter overflow interrupt (CPU_INT.IMASK.LINOVF = 1) to detect the BREAK field is too long and overflows 16bit counter. The timeout can be calculated as **$t_{Timeout}$= 216/ UART clock**

3.2.2 SYNC Field Validation

Synchronization field validation is critical for ensuring precise timing accuracy of the LIN frame header and determining the commander's baud rate parameters. A successful validation sequence confirms multiple criteria: proper Break field detection, precise communication timing, accurate Protected Identifier (PID) reception, and overall frame synchronization integrity.

The synchronization field comprises a predefined 0x55 byte pattern (01010101), specifically designed to facilitate:

- Precise timing references through alternating bit patterns
- Four distinct bit-time measurement opportunities
- Deterministic transition intervals for validation

This structured pattern enables receiving nodes to:

- Determine actual communication parameters
- Implement baud rate adjustments as necessary
- Achieve synchronization with the master timing reference

Upon validation of the Break field, the system initiates SYNC field measurements utilizing the LINC1 capture register that captures the LINCNT value on RX rising edges. The LINCNT counter is configured to reset on falling edges and increment during RX LOW states. LINC1 capture operations and RX rising edge interrupts are triggered at each rising edge transition. During interrupt service routines, the software analyzes individual bit timing parameters through LINC1 register values, validating timing specifications and implementing baud rate adjustments when required.

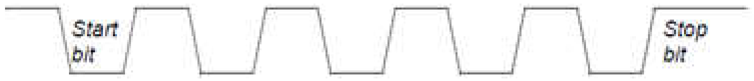


Figure 3-6. Sync Field- 0x55

3.3 LIN Transceiver

This application note uses the TLIN2029A-Q1 evaluation module (EVM) as the external LIN transceiver. The following figure shows the block diagram of how MSPM0 Commander and Responder interfaced with the TLIN2029A-Q1 transceiver.

For more details about schematic connections between transceiver and LIN commander/responder, refer to TLIN2029-Q1 EVM user’s guide.

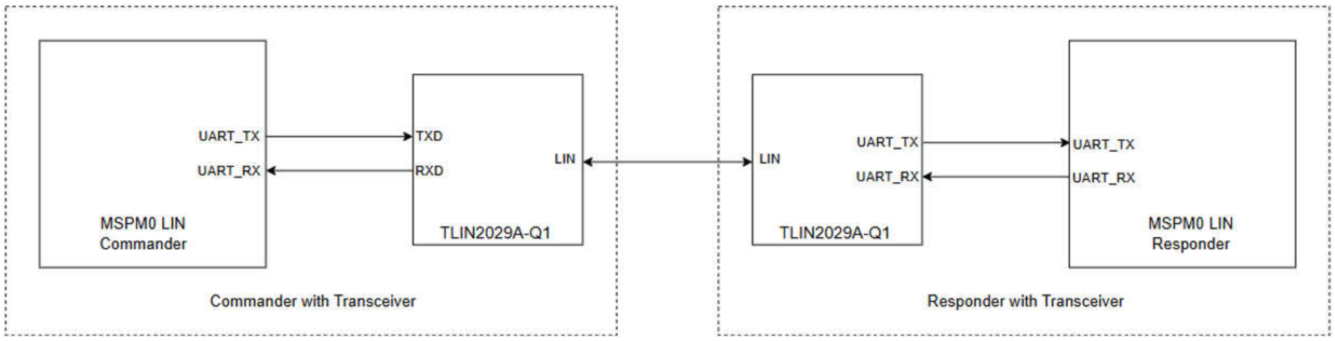


Figure 3-7. Block Diagram of MSPM0 Commander and Responder Interfaced with TLIN2029A-Q1 Transceiver

4 Automatic Baud Rate Detection

Auto baud rate detection in LIN is the process by which Responder nodes automatically recognize and adjust to the Commander's baud rate each time a LIN frame starts.

The commander node sends a special sync field at the start of every frame, which is always the byte 0x55 (binary 01010101).

Responder nodes can calculate the current baud rate, by measuring the bit time(Tbit) in the sync byte utilizing the MSPM0/UNICOMM registers (refer to Section 3.2.2).

4.1 Procedure to Measure Bit-width using MSPM0 UART / UNICOMM UART

Responder nodes can calculate the current baud rate, by measuring the bit time(Tbit) at each rising edge in the sync byte, rising edges are available in distances of 1, 3, 5, 7 & STOP bit times as shown in the following figure.

Forcalculating the SYNC field bit timings in Automatic baud rate detection, the following registers can be used on MSPM0/UNICOMM:

1. Initialize LIN counter to 0 (LINCNT = 0) after detecting a valid break field.
2. Enable interrupt on RX rising edge (CPU_INT.IMASK.RXPE = 1)
3. Setup LIN count control (LINCTL) registers
    a. Enable LIN counter clearing on RX falling edge (LINCTL.ZERONE = 1)
    b. Enable count while low signal on RX (LINCTL.CNTRXLOW = 1)
    c. Enable LIN counter capture on rising RX edge (LINCTL.LINC1CAP = 1)
    d. Enable LIN counter (LINCTL.CTRENA = 1)

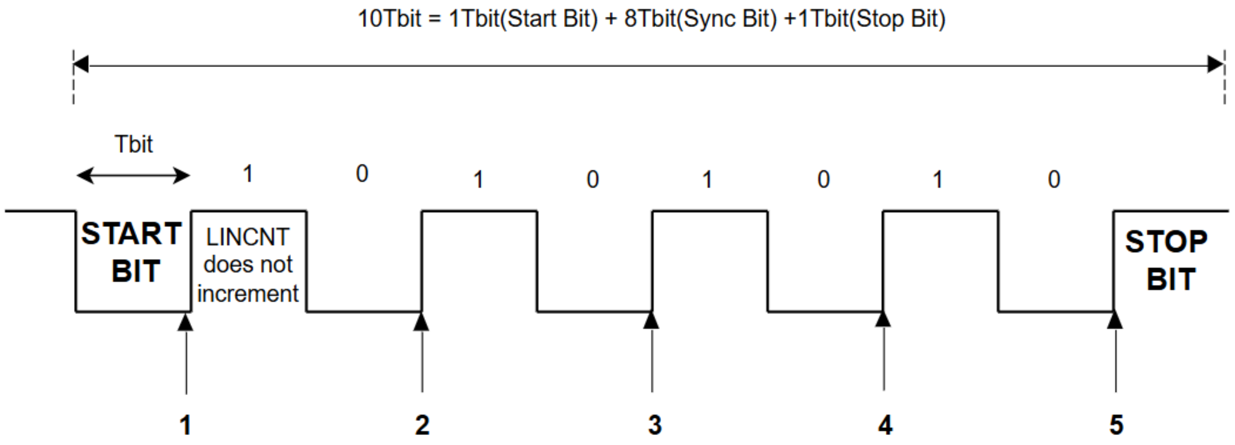

10Tbit = 1Tbit(Start Bit) + 8Tbit(Sync Bit) +1Tbit(Stop Bit)
Tbit
1
0
1
0
1
0
1
0
START BIT
LINCNT does not increment
STOP BIT
1
2
3
4
5

Figure 4-1. LIN SYNC Field Validation

Actions performed at each rising edge of the RX line during the sync field are as follows.

1. LIN counter reset to 0 on each falling RX edge and starts counting when RX is low for Tbit duration.
2. The LINCNT value will be captured by LINC1 register at each RX rising edge.
3. RX rising edge interrupt is triggered (RXPE) for five iterations.
    - In each RX rising edge interrupt service routine (ISR) iteration, the LINC1 capture register is read. The captured value represents the Tbit time, since LIN counter is configured to count only when RX is low.

4.2 Calculation of Correct Baud Rate

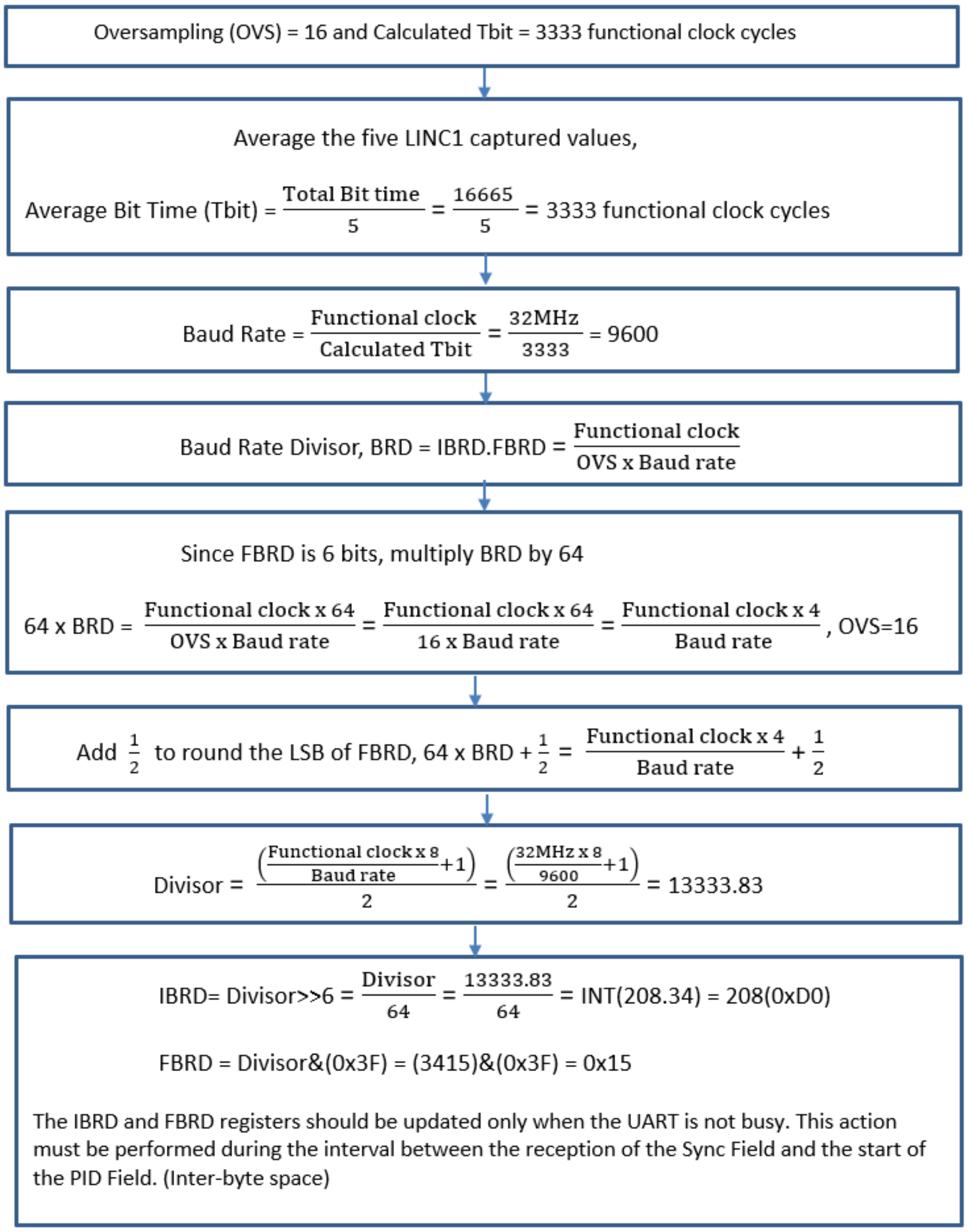
If Responder's clock is running at 32MHz (nominal frequency, F_{Nom}) before synchronization and Commander node sent the SYNC field at 9600 baud rate. Software can calculate the correct baud rate divisors by the following steps:

Average the five LINC1 captured values.

$$\text{Average Bit Time (Tbit)} = \frac{\text{Total Bit time}}{5} = \frac{16665}{5} = 3333 \text{ functional clock cycles} \tag{1}$$

The calculated Tbit indicates the number of functional clock cycles in each bit time. User must update the IBRD/FBRD registers with the calculated values to maintain baud rate tolerance after synchronization (F_{TOL_SYNC}).

The following flowchart example shows a straightforward approach to determine the values for IBRD (Integer Baud Rate Divisor) and FBRD (Fractional Baud Rate Divisor) when the Tbit duration has been calculated.



Oversampling (OVS) = 16 and Calculated Tbit = 3333 functional clock cycles

Average the five LINC1 captured values,

$$\text{Average Bit Time (Tbit)} = \frac{\text{Total Bit time}}{5} = \frac{16665}{5} = 3333 \text{ functional clock cycles}$$

$$\text{Baud Rate} = \frac{\text{Functional clock}}{\text{Calculated Tbit}} = \frac{32\text{MHz}}{3333} = 9600$$

$$\text{Baud Rate Divisor, BRD} = \text{IBRD.FBRD} = \frac{\text{Functional clock}}{\text{OVS x Baud rate}}$$

Since FBRD is 6 bits, multiply BRD by 64

$$64 \text{ x BRD} = \frac{\text{Functional clock x 64}}{\text{OVS x Baud rate}} = \frac{\text{Functional clock x 64}}{\text{16 x Baud rate}} = \frac{\text{Functional clock x 4}}{\text{Baud rate}}, \text{OVS=16}$$

$$\text{Add } \frac{1}{2} \text{ to round the LSB of FBRD, } 64 \text{ x BRD} + \frac{1}{2} = \frac{\text{Functional clock x 4}}{\text{Baud rate}} + \frac{1}{2}$$

$$\text{Divisor} = \frac{\left(\frac{\text{Functional clock x 8}}{\text{Baud rate}} + 1\right)}{2} = \frac{\left(\frac{\text{32MHz x 8}}{9600} + 1\right)}{2} = 13333.83$$

$$\text{IBRD= Divisor>>6} = \frac{\text{Divisor}}{64} = \frac{13333.83}{64} = \text{INT(208.34)} = \text{208(0xD0)}$$

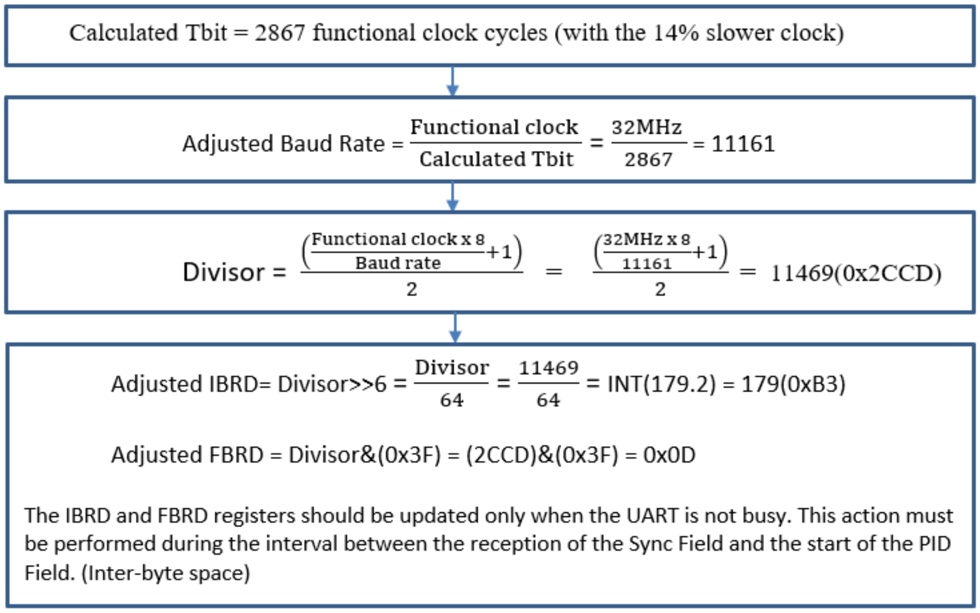
FBRD = Divisor&(0x3F) = (3415)&(0x3F) = 0x15

The IBRD and FBRD registers should be updated only when the UART is not busy. This action must be performed during the interval between the reception of the Sync Field and the start of the PID Field. (Inter-byte space)

Figure 4-2. Flow Chart of LIN Baud Rate Divisors Calculation at Nominal Frequency (32MHz)

4.2.1 Crystal Error at Responder Node

If the responder's clock is running 14% slower than the nominal rate before synchronization, specifically, operating at 27.52 MHz instead of the intended 32 MHz at a baud rate of 9600, then the calculated Tbit will be 2867 clock cycles (with the erroneous clock), rather than the expected 3333 cycles with the actual clock.



Calculated Tbit = 2867 functional clock cycles (with the 14% slower clock)

$$\text{Adjusted Baud Rate} = \frac{\text{Functional clock}}{\text{Calculated Tbit}} = \frac{32\text{MHz}}{2867} = 11161$$

$$\text{Divisor} = \frac{\left(\frac{\text{Functional clock x 8}}{\text{Baud rate}}+1\right)}{2} = \frac{\left(\frac{32\text{MHz x 8}}{11161}+1\right)}{2} = 11469(0\text{x}2\text{CCD})$$

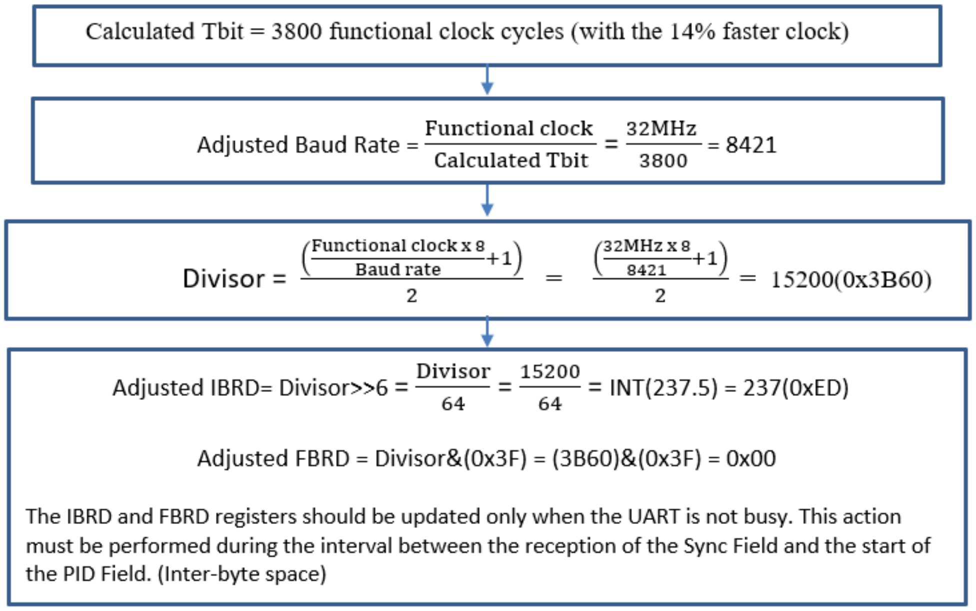
$$\text{Adjusted IBRD= Divisor>>6} = \frac{\text{Divisor}}{64} = \frac{11469}{64} = \text{INT}(179.2) = 179(0\text{xB}3)$$

Adjusted FBRD = Divisor&(0x3F) = (2CCD)&(0x3F) = 0x0D

The IBRD and FBRD registers should be updated only when the UART is not busy. This action must be performed during the interval between the reception of the Sync Field and the start of the PID Field. (Inter-byte space)

Figure 4-3. Flowchart of LIN Baud Rate Divisors Calculation at 14% Slower Clock (27.52MHz)

If the responder's clock is running 14% faster than the nominal rate before synchronization, specifically, operating at 36.48 MHz instead of the intended 32 MHz at a baud rate of 9600, then the calculated Tbit will be 3800 clock cycles (with the erroneous clock), rather than the expected 3333 cycles with the actual clock.



Calculated Tbit = 3800 functional clock cycles (with the 14% faster clock)

$$\text{Adjusted Baud Rate} = \frac{\text{Functional clock}}{\text{Calculated Tbit}} = \frac{32\text{MHz}}{3800} = 8421$$

$$\text{Divisor} = \frac{\left(\frac{\text{Functional clock x 8}}{\text{Baud rate}}+1\right)}{2} = \frac{\left(\frac{32\text{MHz x 8}}{8421}+1\right)}{2} = 15200(0\text{x}3\text{B}60)$$

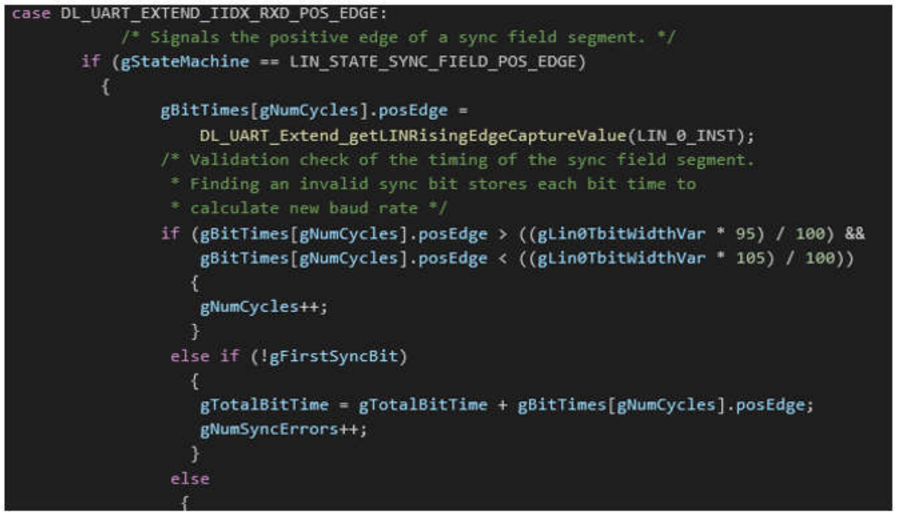
$$\text{Adjusted IBRD= Divisor>>6} = \frac{\text{Divisor}}{64} = \frac{15200}{64} = \text{INT}(237.5) = 237(0\text{xED})$$

Adjusted FBRD = Divisor&(0x3F) = (3B60)&(0x3F) = 0x00

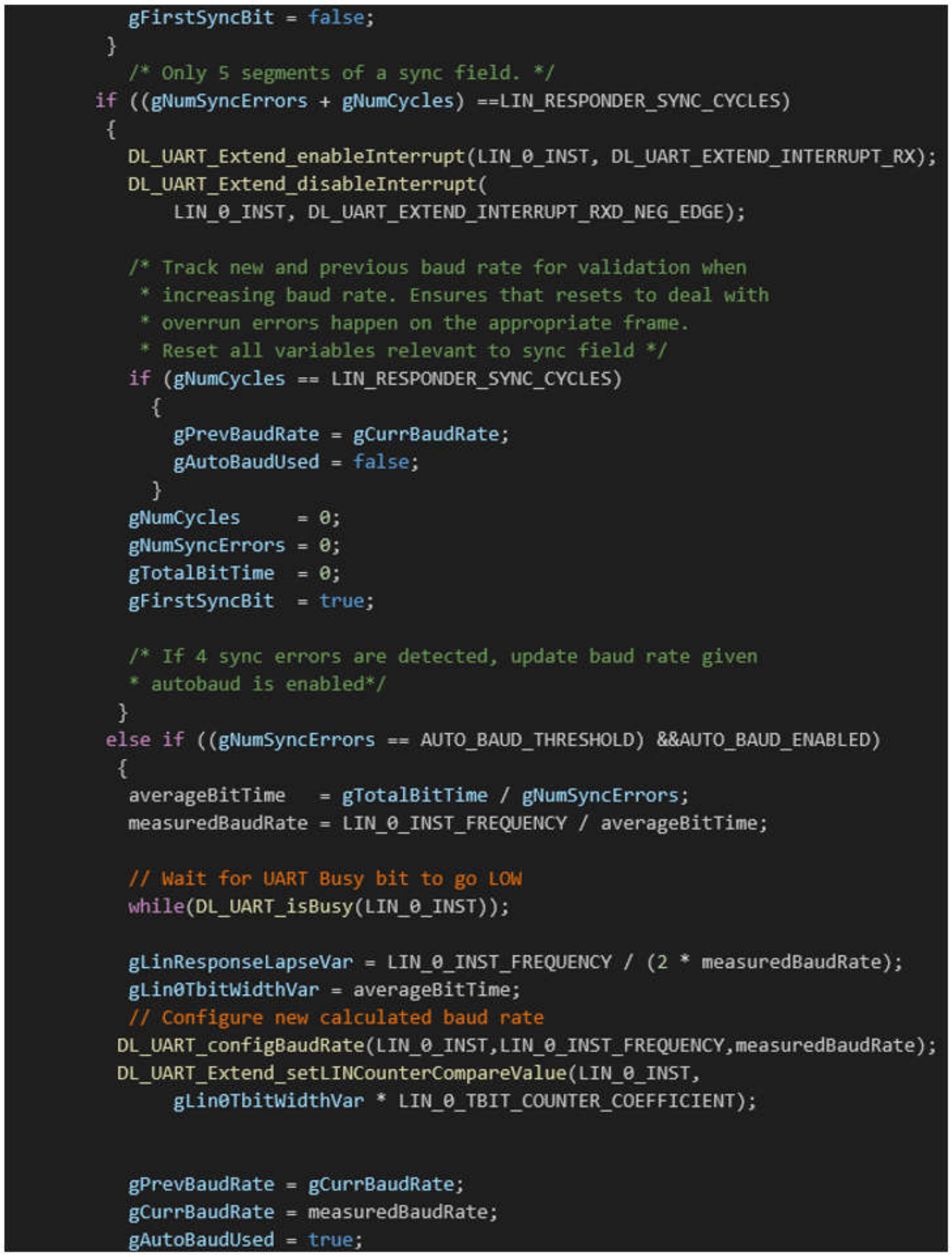
The IBRD and FBRD registers should be updated only when the UART is not busy. This action must be performed during the interval between the reception of the Sync Field and the start of the PID Field. (Inter-byte space)

Figure 4-4. Flowchart of LIN Baud Rate Divisors Calculation at 14% Faster Clock (36.48MHz)

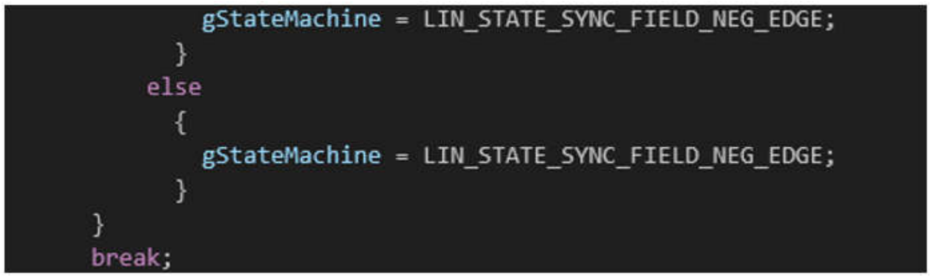
The following software code explains the baud rate detection and configuration within the rising edge ISR.



```
case DL_UART_EXTEND_IIDX_RXD_POS_EDGE:
              /* Signals the positive edge of a sync field segment. */
          if (gStateMachine == LIN_STATE_SYNC_FIELD_POS_EDGE)
            {
                  gBitTimes[gNumCycles].posEdge =
                      DL_UART_Extend_getLINRisingEdgeCaptureValue(LIN_0_INST);
                  /* Validation check of the timing of the sync field segment.
                   * Finding an invalid sync bit stores each bit time to
                   * calculate new baud rate */
                  if (gBitTimes[gNumCycles].posEdge > ((gLin0TbitWidthVar * 95) / 100) &&
                      gBitTimes[gNumCycles].posEdge < ((gLin0TbitWidthVar * 105) / 100))
                     {
                      gNumCycles++;
                     }
                   else if (!gFirstSyncBit)
                     {
                      gTotalBitTime = gTotalBitTime + gBitTimes[gNumCycles].posEdge;
                      gNumSyncErrors++;
                     }
                   else
                    {
```



```
          gFirstSyncBit = false;
        }
          /* Only 5 segments of a sync field. */
       if ((gNumSyncErrors + gNumCycles) ==LIN_RESPONDER_SYNC_CYCLES)
        {
          DL_UART_Extend_enableInterrupt(LIN_0_INST, DL_UART_EXTEND_INTERRUPT_RX);
          DL_UART_Extend_disableInterrupt(
              LIN_0_INST, DL_UART_EXTEND_INTERRUPT_RXD_NEG_EDGE);

          /* Track new and previous baud rate for validation when
           * increasing baud rate. Ensures that resets to deal with
           * overrun errors happen on the appropriate frame.
           * Reset all variables relevant to sync field */
          if (gNumCycles == LIN_RESPONDER_SYNC_CYCLES)
            {
              gPrevBaudRate = gCurrBaudRate;
              gAutoBaudUsed = false;
            }
          gNumCycles     = 0;
          gNumSyncErrors = 0;
          gTotalBitTime  = 0;
          gFirstSyncBit  = true;

          /* If 4 sync errors are detected, update baud rate given
          * autobaud is enabled*/
         }
        else if ((gNumSyncErrors == AUTO_BAUD_THRESHOLD) &&AUTO_BAUD_ENABLED)
         {
          averageBitTime    = gTotalBitTime / gNumSyncErrors;
          measuredBaudRate = LIN_0_INST_FREQUENCY / averageBitTime;

          // Wait for UART Busy bit to go LOW
          while(DL_UART_isBusy(LIN_0_INST));

          gLinResponseLapseVar = LIN_0_INST_FREQUENCY / (2 * measuredBaudRate);
          gLin0TbitWidthVar = averageBitTime;
          // Configure new calculated baud rate
         DL_UART_configBaudRate(LIN_0_INST,LIN_0_INST_FREQUENCY,measuredBaudRate);
         DL_UART_Extend_setLINCounterCompareValue(LIN_0_INST,
              gLin0TbitWidthVar * LIN_0_TBIT_COUNTER_COEFFICIENT);


          gPrevBaudRate = gCurrBaudRate;
          gCurrBaudRate = measuredBaudRate;
          gAutoBaudUsed = true;
```



```
                gStateMachine = LIN_STATE_SYNC_FIELD_NEG_EDGE;
            }
            else
            {
                gStateMachine = LIN_STATE_SYNC_FIELD_NEG_EDGE;
            }
        }
        break;
```

5 Deviation in Baud Rate after Synchronization

If the responder's clock operates 14% slower than the nominal rate before synchronization, the functional clock becomes 27.52 MHz rather than the intended 32 MHz. After synchronization, the IBRD and FBRD register values are 0xB3 and 0x0D, respectively.

$$\text{Adjusted Baud Rate} = \frac{\text{Functional clock}}{\text{OVS x IBRD.FBRD}} = \frac{27.52\text{MHz}}{16 \text{ x } 179.13} = 9601.1$$

The expected baud rate is 9600, corresponding expected bit time (Tbit) is 104.16 µs.

The actual calculated baud rate after synchronization is 9601.1, resulting in an actual Tbit is 104.15 µs.

$$\text{Percentage of error in Baud Rate post synchronization} = \frac{(\text{Expected Tbit} - \text{Actual Tbit})}{\text{Actual Tbit}} \times 100$$

$$= \frac{(104.16 - 104.15)}{104.16} \times 100 = 0.01\%$$

Following synchronization, the responder node's baud rate deviation is 0.01%, which is well within the LIN Specification's allowed tolerance limit (FTOL_SYNC < ±1.5%).

6 References

1. MSPM0 G-Series 80-MHz Microcontrollers Technical Reference Manual (Rev. C)
2. LIN-2.2 Specification Document
3. TLIN2029-Q1 EVM user's guide
4. MSPM0Gx51x Mixed-Signal Microcontrollers With CAN-FD Interface datasheet (Rev. B)