UART Architecture

Young Won Lim
10/5/22

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

Based on

ARM System-on-Chip Architecture, 2nd ed, Steve Furber

Introduction to ARM Cortex-M Microcontrollers
– Embedded Systems, Jonathan W. Valvano

Digital Design and Computer Architecture,
D. M. Harris and S. L. Harris

ARM assembler in Raspberry Pi
Roger Ferrer Ibáñez

https://thinkingeek.com/arm-assembler-raspberry-pi/

UART Background

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

Data Frame (1)

For UART to work the following settings need to be the same
on both the transmitting and receiving side:

Baud Rate
a common unit of measurement of symbol rate, which is one of the components that determine the speed of communication over a data channel.

Parity bit
a bit added to a string of binary code. Parity bits are a simple form of error detecting code.

Data bits size

Stop bits size

Flow Control
the process of managing the rate of data transmission between two nodes to prevent a fast sender from overwhelming a slow receiver. It provides a mechanism for the receiver to control the transmission speed, so that the receiving node is not overwhelmed with data from transmitting node.

In the most common settings of 8 data bits, no parity and 1 stop bit (aka 8N1), the protocol efficiency is 80%. Ethernet by comparison is up to 97%.

https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

Data Frame (2)

A UART frame consists of 5 elements:

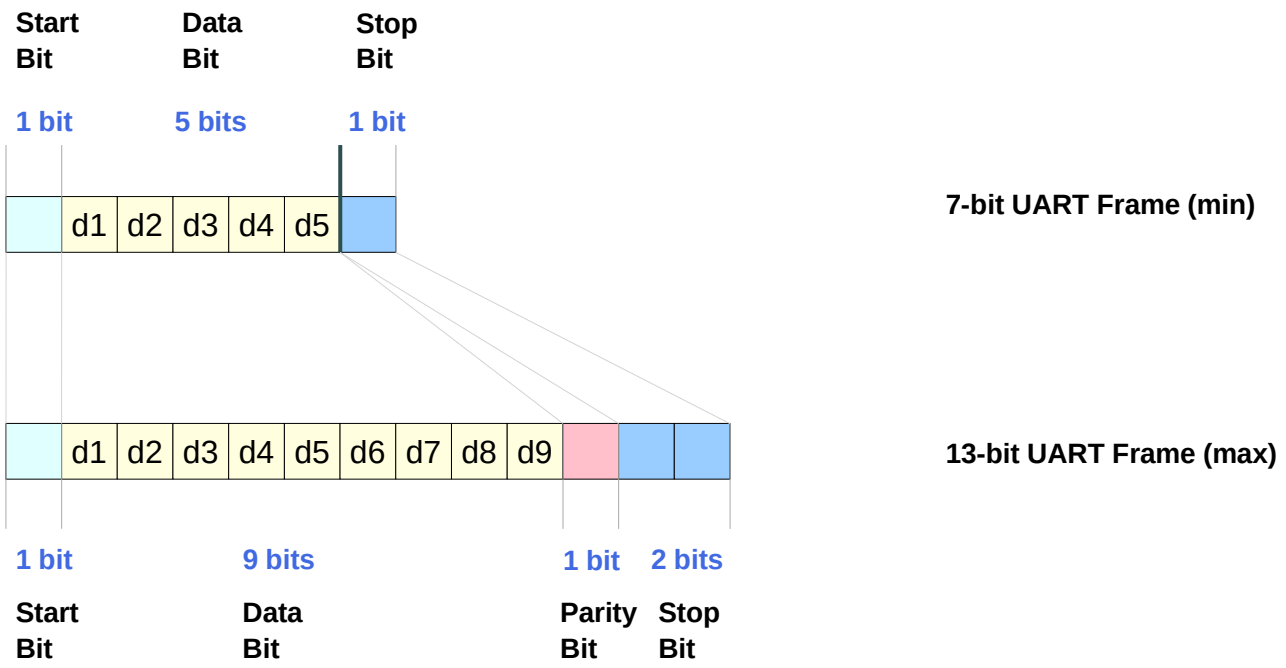
Idle (logic high (1))
Start bit (logic low (0))
Data bits
Parity bit
Stop (logic high (1))

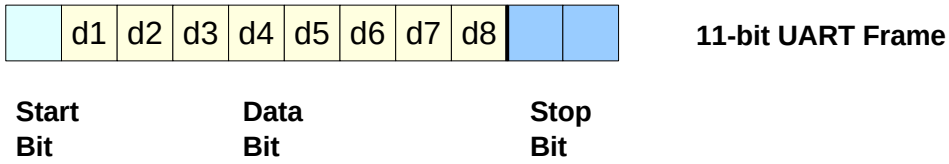
The idle, no data state is high-voltage, or powered.
This is a historic legacy from telegraphy,
in which the line is held high to show
that the line and transmitter are not damaged.

Each character is framed as a logic low start bit, data bits, possibly a parity bit and one or more stop bits. In most applications the least significant data bit (the one on the left in this diagram) is transmitted first, but there are exceptions (such as the IBM 2741 printing terminal).

https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

Data Frame (2)





https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

Data Frame (3)

Start bit
The start bit signals the receiver that a new character is coming.

Data bit
The next five to nine bits, depending on the code set employed, represent the character.

Parity bit
If a parity bit is used, it would be placed after all of the data bits.
It describes the odd or eveness of the number.

Stop bit
The next one or two bits are always in the mark (logic high, i.e., '1') condition and called the stop bit(s). They signal to the receiver that the character is complete. Since the start bit is logic low (0) and the stop bit is logic high (1) there are always at least two guaranteed signal changes between characters.

If the line is held in the logic low condition for longer than a character time, this is a **break condition** that can be detected by the UART.

https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

Data Frame (4)

The number of data and formatting bits,
the presence or absence of a parity bit,
the form of parity (even or odd) and
the transmission speed must be pre-agreed
by the communicating parties.

The "stop bit" is actually a "stop period";
the stop period of the transmitter may be arbitrarily long.
It cannot be shorter than a specified amount, usually 1 to 2 bit times.

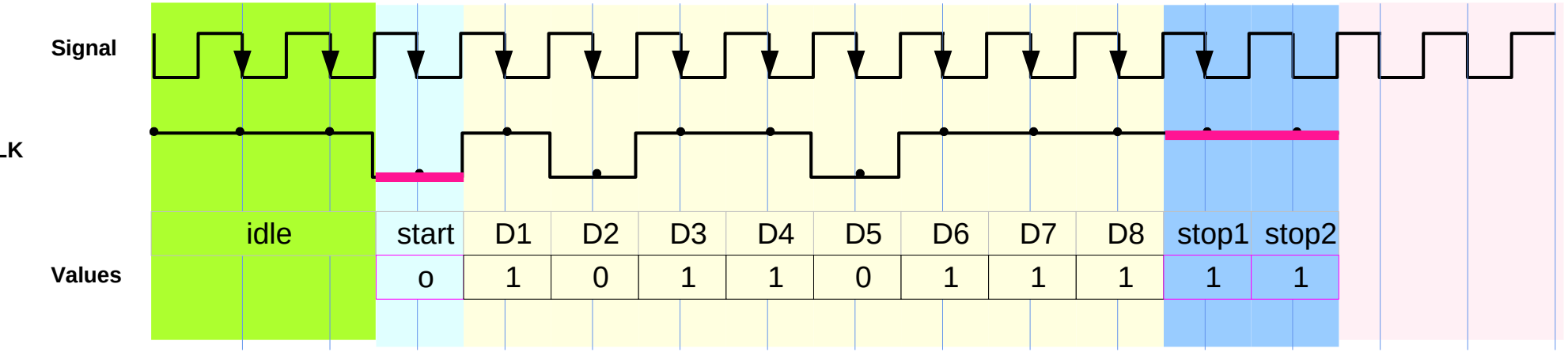
The receiver requires a shorter stop period than the transmitter.

At the end of each data frame, the receiver stops briefly to wait for the next start bit.
It is this difference which keeps the transmitter and receiver synchronized.

BCLK = Base Clock

https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

Examples of UART Data Frame

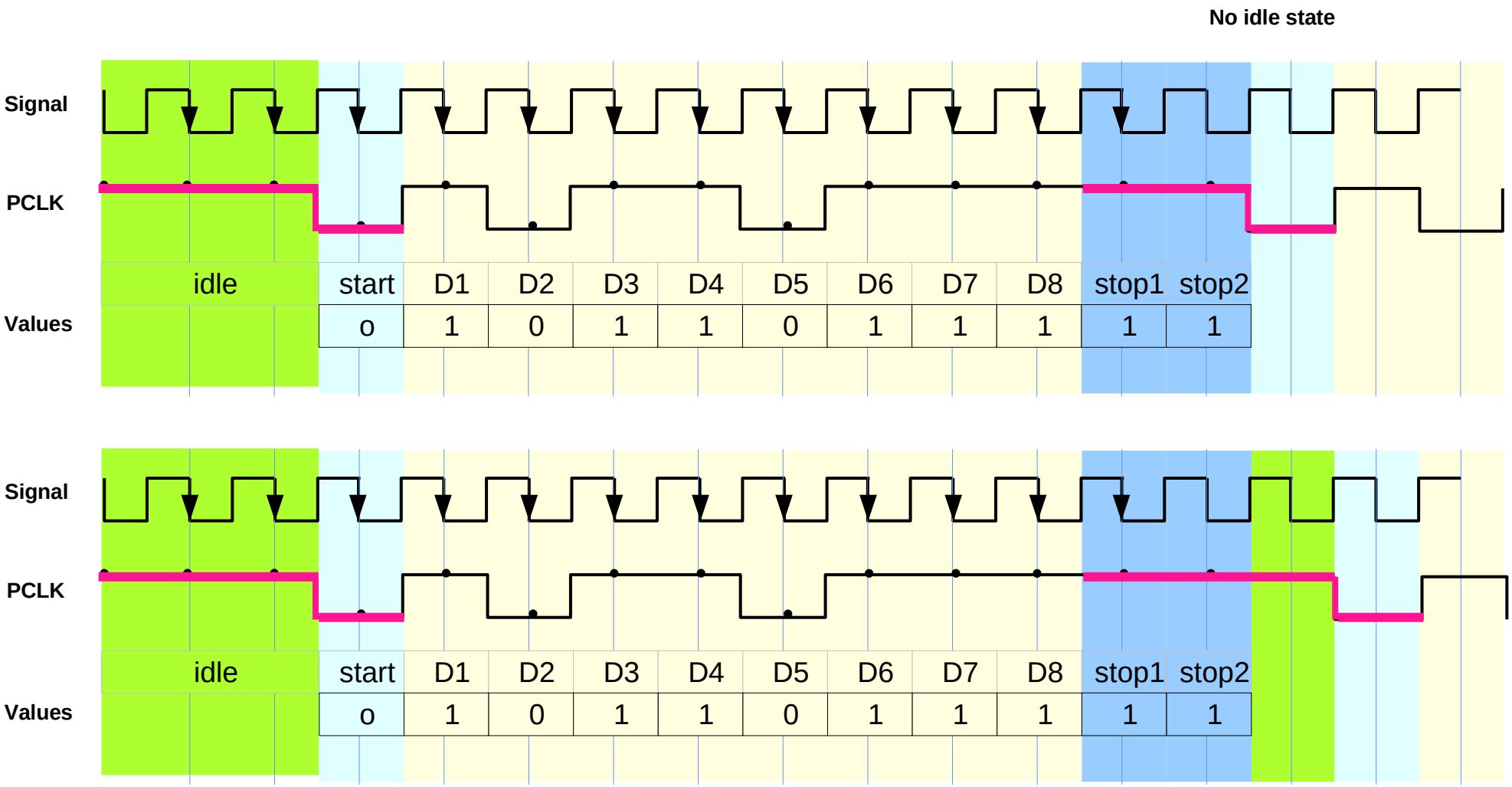


11-bit UART frame

In this diagram, one byte is sent,
consisting of a start bit,
followed by *eight* data bits (D0-7),
and *two* stop bits

https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

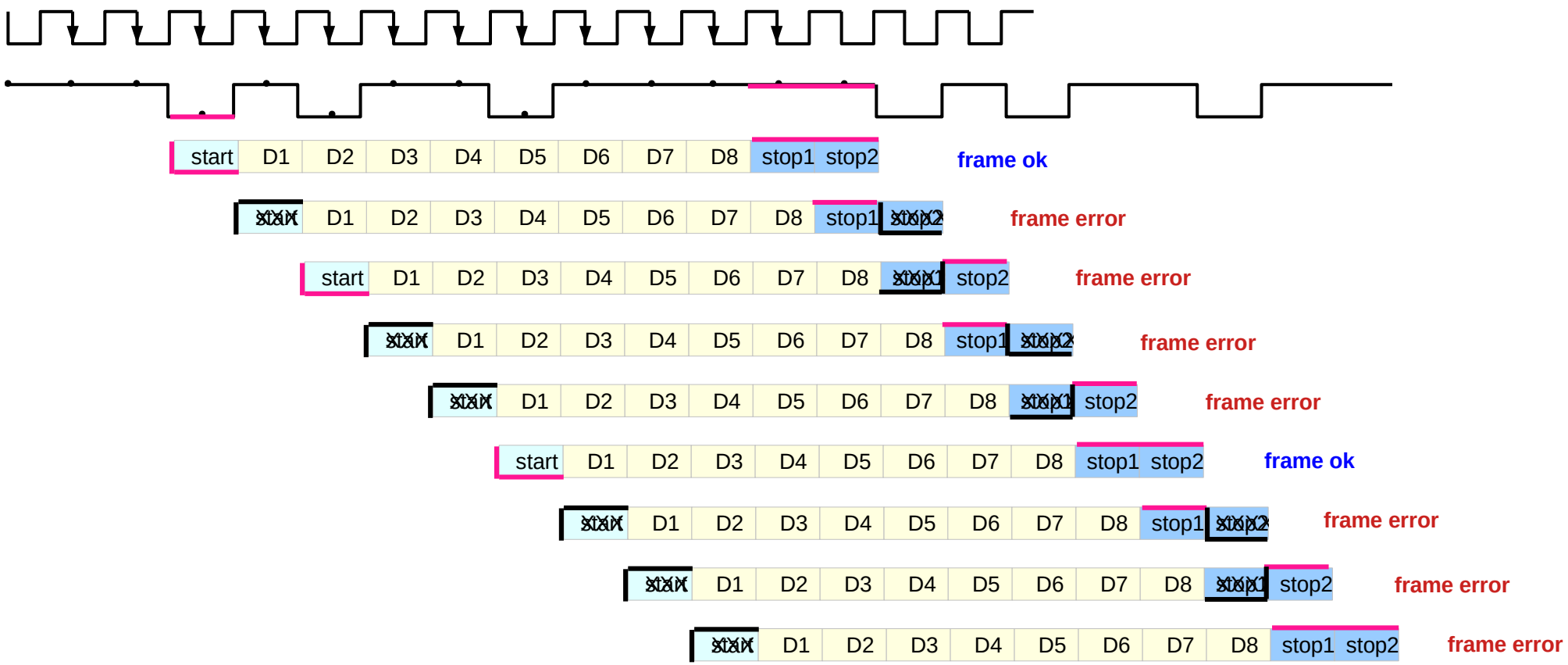
Examples of UART Data Frame



https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

1-bit idle state

Examples of UART Data Frame



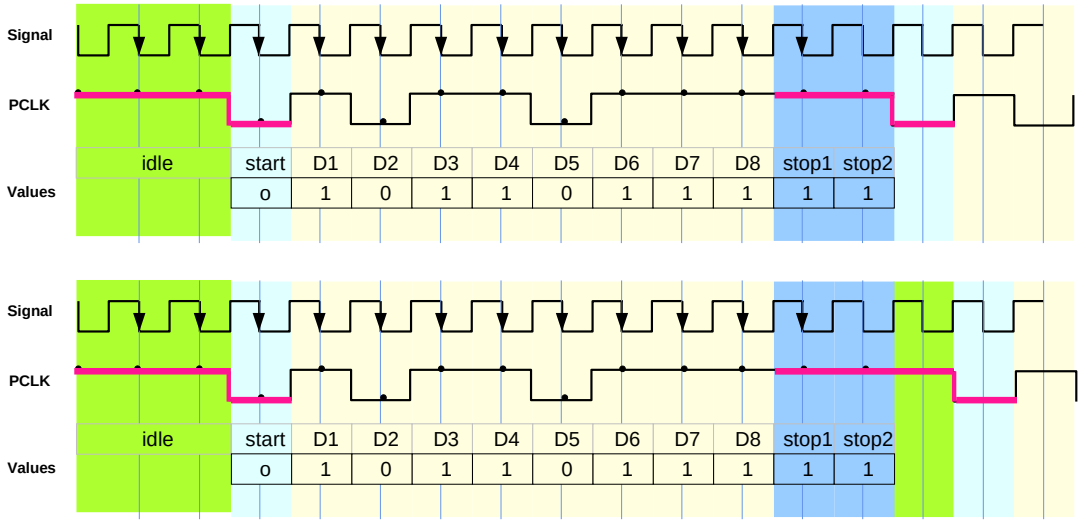
https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

Examples of UART Data Frame

the "stop bit" is actually a "stop period";
the stop period of the transmitter may be arbitrarily long.
It cannot be shorter than a specified amount, usually 1 to 2 bit times.

The receiver requires a shorter stop period than the transmitter.

At the end of each data frame,
the receiver stops briefly to wait for the next start bit.
It is this difference which keeps the transmitter and receiver synchronized.



https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

Overrun error

An **overrun error** occurs
when the UART receiver cannot process the character
that just came in before the next one arrives.

Various devices have different amounts of buffer space
to hold received characters.

The CPU or DMA controller must service the UART
in order to remove characters from the input buffer.

If the CPU or DMA controller does not service
the UART quickly enough and
the receiver buffer becomes full,
an overrun error will occur, and
incoming characters will be lost.

https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

Underrun error

An **underrun error** occurs
when the UART transmitter has completed
sending a character and
the transmit buffer is empty.

In asynchronous modes this is treated
as an indication that no data remains to be transmitted,
rather than an error, since additional stop bits can be appended.

This error indication is commonly found in USARTs,
since an underrun is more serious in synchronous systems.

https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

Framing error

A UART will detect a **framing error**
when it does not see a "stop" bit at the expected "stop" bit time.

As the "start" bit is used to identify the beginning of an incoming character,
its timing is a reference for the remaining bits.

If the data line is not in the expected state (high)
when the "stop" bit is expected
(according to the number of data and parity bits
for which the UART is set),
the UART will signal a framing error.

A "break" condition on the line is also signaled
as a framing error.

https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

Framing errors (2)

Framing errors can be caused by clock skew

If the transmitter clock and receiver clock
are not derived from the same source
(which is the case most of the time),
then one will run faster than the other.

When the timing error is too large,
you may occasionally read a wrong bit.

the receiver has detected the start bit and
where it expects the stop bit the data is inverted.

This can also be due to data corruption
caused by line interference impinging on the stop bit.

You always need to check this for each byte received.

https://electronics.stackexchange.com/questions/83379/what-causes-uart-errors

Parity error

A **parity error** occurs
when the parity of the number of one-bits
disagrees with that specified by the parity bit.

Parity checking is often used
for the detection of transmission errors.

Use of a parity bit is optional,
so this error will only occur
if parity-checking has been enabled.

https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

Parity Errors (2)

Parity errors occur
when parity is implemented on the data link
and there is a corruption that causes a parity mismatch in the received data.

You always need to check this for each byte received.

https://electronics.stackexchange.com/questions/83379/what-causes-uart-errors

Break Condition (1)

A **break condition** occurs
when the receiver input is at the "space" (logic low, i.e., '0') level
for longer than some duration of time, typically,
for more than a character time.

This is not necessarily an error,
but appears to the receiver as a character of all zero-bits
with a **framing error**.

https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

Break Condition (2)

The term "break" derives from current loop signaling,
which was the traditional signaling used for teletypewriters.

The "spacing" condition of a current loop line
is indicated by no current flowing, and
a very long period of no current flowing
is often caused by a break or other fault in the line.

Some equipment will deliberately transmit the "space" level
for longer than a character as an attention signal.

When signaling rates are mismatched,
no meaningful characters can be sent,
but a long "break" signal can be a useful way
to get the attention of a mismatched receiver
to do something (such as resetting itself).

Computer systems can use the long "break" level
as a request to change the signaling rate,
to support dial-in access at multiple signaling rates.
The DMX512 protocol uses the break condition
to signal the start of a new packet.

https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

Synchronizing (7)

However, straight up USART signals
are not appropriate for RF.

Consider looking into Manchester coding.
Rather than sending data as highs/lows,
you send it as transitions from high to low or low to high.
It makes clock recovery much easier,
as it encodes the clock into every bit,
and it operates on transitions which is RF's natural friend.
You can't send it as quickly, but it will be much more reliable.

Also consider error detection and if possible, error correction.
With simple error detection you will be able
to verify whether adjustments to your algorithm
are improving the signal or not objectively.

https://en.wikipedia.org/wiki/Universal_asynchronous_https://electronics.stackexchange.com/questions/541108/how-to-determine-the-end-of-transmission-uart-transmitter

Data Frame (4)

The UART actually does know it.
When you start using the UART,
you must configure all parameters,
like at what bit rate the bits are sent,
and the number of data,
parity (if any) and stop bits
it is expected to receive,
and so the stop bit always is
the last bit of an asynchronous start-stop frame.

So the start bit starts a frame,
which includes the all the bits
and stop bit is always last.

https://en.wikipedia.org/wiki/Universal_asynchronous_https://electronics.stackexchange.com/questions/541108/how-to-determine-the-end-of-transmission-uart-transmitter

Data Frame (4)

If the receiver is expecting
a high stop bit after 8 data bits,
but if the stop bit is not high,
then most UARTS will signal a framing error,
so the user can know
that there was an error receiving this byte,
and it can then be ignored or reacted upon,
like trying to resync after an error.

Which actually is how sending a "break" signal works.
The transmitter sends logic 0 for very long time,
so that the receiver sees it as start bit,
and the data (and parity) bits will be 0 too,
and then stop bit is also 0,
so receiving a full 0x00 byte with framing error
is a sign of receiving a break condition.

https://en.wikipedia.org/wiki/Universal_asynchronous_https://electronics.stackexchange.com/questions/541108/how-to-determine-the-end-of-transmission-uart-transmitter

Data Frame (4)

When you configure your UART
you need to specify not only the baudrate
but also the number of bits.
Your example has configured it to 7 data bits
and one parity bit, thus the receiver knows
that the first 8 bits can not be stop bits.

A UART receiver has to be "told" beforehand
how many data bits there are and
whether parity is used or not and,
if used, whether it's even parity or odd parity or
forced parity.
It also has to be "told" beforehand the data rate.
Sometimes, it's even necessary to inform a UART
that there might be two stop bits before the next byte is received.

So, on this basis, it will know when the end of the transmission is.

https://en.wikipedia.org/wiki/Universal_asynchronous_https://electronics.stackexchange.com/questions/541108/how-to-determine-the-end-of-transmission-uart-transmitter

Break (3)

Receive break is also regarded as an error
although it's really an indication that the incoming data
has fallen to logical zero for longer than 1 byte of data.

Normally logical 1 is the "ambient" state
between successive data bytes
and it remains this way.

https://electronics.stackexchange.com/questions/83379/what-causes-uart-errors

Synchronizing (1)

UART timing for asynchronous data relies on
knowledge of the data rate and
having a clock that is typically 16 x faster.

In the absense of any data edges,
the correctly timed clock can sample the data
pretty much at the middle of the symbol.

https://en.wikipedia.org/wiki/Universal_asynchronous_https://electronics.stackexchange.com/questions/541108/how-to-determine-the-end-of-transmission-uart-transmitter

Synchronizing (2)

UARTs (rs232) have a start bit (0) and a stop bit(1)
But they use wire, and the noise is very low -basically none.

On a noisy link, this works very badly.
If the start bit is wrong, everything after it is wrong

The codes are chosen to always have
enough 1/0 transitions to keep in bit sync.
ie. you can't get a run of 32 1s,
there will always be a transition every N bits, worst case.

https://en.wikipedia.org/wiki/Universal_asynchronous_https://electronics.stackexchange.com/questions/541108/how-to-determine-the-end-of-transmission-uart-transmitter

Synchronizing (3)

If you do use a UART over radio,
the preamble should be chars that have
a single 0/1 transition in them,
so that the UART can get back in sync.

As should be obvious, if you data was 1/0/1/0
then the uart would never know which edge was the start bit.

it wants to have roughly equal 1/0 balance.
So 0xF0 is ideal, the sequence will be start=0 0000 1111 stop=1

https://en.wikipedia.org/wiki/Universal_asynchronous_https://electronics.stackexchange.com/questions/541108/how-to-determine-the-end-of-transmission-uart-transmitter

Synchronizing (4)

In asynchronous communication
you have a defined speed, the baudrate.

The receiver knows how long a bit time is.
It waits for an edge and then starts counting
till it is in the middle of the bit-time.
Then it samples the input.

Waiting for en edge is done using 'oversampling'.
You read the input status much faster then the bit rate.
Common is to use 16x oversampling, but 8x also works.

There is free software that implements a UART.
If you want to see how it is done in hardware find Verilog source code.

https://en.wikipedia.org/wiki/Universal_asynchronous_https://electronics.stackexchange.com/questions/541108/how-to-determine-the-end-of-transmission-uart-transmitter

Synchronizing (4')

the 'start and end of their bytes' part.

A UART starts with a start bit which is always low.
It ends with a stop bit which is always high.

Thus you wait for a 0 on the line
and you know that is the start bit.

You then count e.g. 10 bits (start, 8 data, stop)
and the tenth bit should be high.
It is very well possible that a continuous bit stream
is sampled at the wrong point and
still honors the 'start is low stop is high' protocol.
I therefore try to have gasp between bytes to prevent this.

https://en.wikipedia.org/wiki/Universal_asynchronous_https://electronics.stackexchange.com/questions/541108/how-to-determine-the-end-of-transmission-uart-transmitter

Synchronizing (5)

It might be instructive to look at a simpler algorithm
for finding the stop and start bits.

First, sample the input at 4 times the bit rate.

If you data rate is 9,600bits per second,
then you'll want to sample it at 38,400 Hz.

When there's no transmission,
you'll get a great deal of noise,
and the sample input may see random highs and lows.

When a start bit is sent, you'll sense it
on at least 3 consecutive samples.
Thereafter, you'll sample every 4th sample
as the actual bit received,
offset by one sample so it's close to the middle of each bit.

You will eventually receive the stop bit
- if it's not *correct*, then you can discard
all the data and try again, waiting for a start bit.

https://en.wikipedia.org/wiki/Universal_asynchronous_https://electronics.stackexchange.com/questions/541108/how-to-determine-the-end-of-transmission-uart-transmitter

Synchronizing (6)

That's the simple case.
Once you have that working, you'll find you're still getting bad data,
and that's where you employ more techniques to recover the data:

Instead of using only one sample near the middle of each bit,
look at all three samples that should occur inside the bit time,
and use majority vote to determine the actual bit value

Increase the sample rate to 8x or 16x,
which will give you a much larger number of samples per bit to use,
and get you closer to collecting information throughout the whole bit
rather than just 3/4 in the middle of it.

Store the data stream while receiving,
and use a correlater that moves along the stream
to find the start and stop bits.
This way you're not throwing out a possible byte
because you got a bad start bit right before the real data.

https://en.wikipedia.org/wiki/Universal_asynchronous_https://electronics.stackexchange.com/questions/541108/how-to-determine-the-end-of-transmission-uart-transmitter

Synchronizing (6')

Rather than sampling in the middle of the bytes,
look for the transitions - RF sends transitions better than static levels.
Find the beginning of the start bit,
then using a small window around every bit transition,
look to see if there's a transition,
and then you'll have some information about the previous bit and the next bit.

https://en.wikipedia.org/wiki/Universal_asynchronous_https://electronics.stackexchange.com/questions/541108/how-to-determine-the-end-of-transmission-uart-transmitter

NXP LPC214x UARTs

http://www.ocfreaks.com/lpc2148-interrupt-tutorial/

U0IIR

U0RBR	Receiver Buffer Register
U0THR	Transmit Holding Register
U0DLL	Divisor Latch LSB
U0DLM	Divisor Latch MSB
U0IER	Interrupt Enable Register
U0IIR	Interrupt ID Register
U0FCR	FIFO Control Register
U0LCR	Line Control Register
U0LSR	Line Status Register
U0SCR	Scratch Pad Register
U0ACR	Auto-baud Control Register
U0FDR	Fractional Divider Register
U0TER	Tx Enable Register

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

UART (Universal Asynchronous Receiver / Transmitter)

- 16 byte Receive and Transmit FIFOs
- Register locations conform to ‘550 industry standard.
- Receiver FIFO trigger points at 1, 4, 8, and 14 bytes.
- Built-in fractional baud rate generator with autobauding capabilities.
- Mechanism that enables software and hardware flow control implementation.

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

1. Receiver Buffer Register (U0RBR)

The U0RBR is the top byte of the UART0 Rx FIFO.

The top byte of the Rx FIFO
contains the oldest character received and
can be read via the bus interface.

The LSB (bit 0) represents the “oldest” received data bit.

If the character received is less than 8 bits,
The unused MSBs are padded with zeroes.

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

2. Transmit Holding Register (U0THR)

The U0THR is the top byte of the UART0 TX FIFO.

The top byte is the newest character in the TX FIFO and can be written via the bus interface. T

he LSB represents the first bit to transmit.

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

3. Divisor Latch Register (U0DLL)

The UART0 Divisor Latch is
part of the UART0 Fractional Baud Rate Generator and
holds the value used to divide the clock
supplied by the fractional prescaler
in order to produce the baud rate clock,
which must be 16x the desired baud rate (Equation 1).

The U0DLL and U0DLM registers together form 16 bit divisor
where U0DLL contains the lower 8 bits of the divisor and
U0DLM contains the higher 8 bits of the divisor.

A 0x0000 value is treated like a 0x0001 value
as division by zero is not allowed.

The Divisor Latch Access Bit (DLAB) in U0LCR must be one
in order to access the UART0 Divisor Latches

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

4. Fractional Divider Register (U0FDR)

The UART0 Fractional Divider Register (U0FDR)
controls the clock pre-scaler for the baud rate generation and
can be read and written at user's discretion.

This pre-scaler takes the VPB clock and
generates an output clock per specified fractional requirements.

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

baudrate

UART0 baudrate can be calculated as:

$$UART\,0_{baudrate} = \frac{PCLK}{16 \times [U\,0\,DLM : U\,0\,DLL] \times \left(1 + \frac{DivAddVal}{MulVal}\right)}$$

Where PCLK is the peripheral clock, U0DLM and U0DLL are the standard UART0 baud rate divider registers, and DIVADDVAL and MULVAL are UART0 fractional baudrate generator specific parameters

The value of MULVAL and DIVADDVAL should comply to the following conditions:
1. 0 < MULVAL ≤ 15
2. 0 ≤ DIVADDVAL ≤ 15

If the U0FDR register value does not comply to these two requests then the fractional divider output is undefined. If DIVADDVAL is zero then the fractional divider is disabled and the clock will not be divided.

The value of the U0FDR should not be modified while transmitting/receiving data or data may be lost or corrupted.

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

baudrate

Usage Note: For practical purposes, UART0 baudrate formula can be written in a way that identifies the part of a UART baudrate generated without the fractional baudrate generator, and the correction factor that this module adds

$$UART\,0_{baudrate} = \frac{PCLK}{16 \times [U\,0\,DLM : U\,0\,DLL]} \times \left(\frac{MulVal}{(MulVal + DivAddVal)} \right)$$

Based on this representation, fractional baudrate generator contribution can also be described as a prescaling with a factor of MULVAL / (MULVAL + DIVADDVAL).

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

baudrate

Example 1: Using UART0baudrate formula from above,
it can be determined that system with

PCLK = 20 MHz,
U0DL = 130 (U0DLM = 0x00 and U0DLL = 0x82),
DIVADDVAL = 0 and
MULVAL = 1

will enable UART0 with UART0baudrate = 9615 bauds.

Example 2: Using UART0baudrate formula from above,
it can be determined that system with

PCLK = 20 MHz,
U0DL = 93 (U0DLM = 0x00 and U0DLL = 0x5D),
DIVADDVAL = 2 and
MULVAL = 5

will enable UART0 with UART0baudrate = 9600 bauds.

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

baudrate

	MULVAL=0, DIVADDVAL = 0			Optimal MULVAL, DIVADDVAL		
Desired baudrate	U0DLM:U0DLL Hex	Dec	% error	U0DLM:U0DLL Dec	Fractional pre-scaler value	% error
50	61A8	25000	0.0000	25000	1/(1+0)	0.0000
75	411B	16667	0.0020	12500	3/(3+1)	0.0000
110	2C64	11364	0.0032	6250	11/(11+9)	0.0000
134.5	244E	9294	0.0034	3983	3/(3+4)	0.0001
150	208D	8333	0.0040	6250	3/(3+1)	0.0000
300	1047	4167	0.0080	3125	3/(3+1)	0.0000
600	0823	2083	0.0160	1250	3/(3+2)	0.0000
1200	0412	1042	0.0320	625	3/(3+2)	0.0000
1800	02B6	694	0.0640	625	9/(9+1)	0.0000
2000	0271	625	0.0000	625	1/(1+0)	0.0000
2400	0209	521	0.0320	250	12/(12+13)	0.0000
3600	015B	347	0.0640	248	5/(5+2)	0.0064
4800	0104	260	0.1600	125	12/(12+13)	0.0000

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

baudrate

	MULVAL=0, DIVADDVAL = 0			Optimal MULVAL, DIVADDVAL		
Desired baudrate	U0DLM:U0DLL Hex	Dec	% error	U0DLM:U0DLL Dec	Fractional pre-scaler value	% error
7200	00AE	174	0.2240	124	5/(5+2)	0.0064
9600	0082	130	0.1600	93	5/(5+2)	0.0064
19200	0041	65	0.1600	31	10/(10+11)	0.0064
38400	0021	33	1.3760	12	7/(7+12)	0.0594
56000	0021	22	1.4400	13	7/(7+5)	0.0160
57600	0016	22	1.3760	19	7/(7+1)	0.0594
112000	000B	11	1.4400	6	7/(7+6)	0.1600
115200	000B	11	1.3760	4	7/(7+12)	0.0594
224000	0006	6	7.5200	3	7/(7+6)	0.1600
448000	0003	3	7.5200	2	5/(5+2)	0.3520

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

6. Interrupt Enable Register (U0IER)

The U0IER is used to enable UART0 interrupt sources.

Bit0 U0IER[0] RBR Interruupt Enable
enables the Receive Data Available interrupt for UART0
also controls the Character Receive Time-out interrupt

Bit1 U0IER[1] THRE Interrupt Enable
enables the Transmitter Holding Register Empty interrupt for UART0
the status of this can be read from U0LSR[5]

Bit2 U0IER[2] RX Line Status Interrupt Enable
enables the UART0 RX line status interrupts
the status of this interrupt can be read from U0LSR[4:1]

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

6. Interrupt Enable Register (U0IER)

The U0IER is used to enable UART0 interrupt sources.

Bit8 U0IER[8] ABTOIntEn
enables the Auto-Baud Time-Out interrupt for UART0

Bit1 U0IER[1] ABEOIntEn
enables the End-Of Auto-Baud interrupt

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

7. Interrupt Identification Register (U0IIR)

The U0IIR provides a status code
that denotes the priority and source of a pending interrupt.

The interrupts are frozen during an U0IIR access.

If an interrupt occurs during an U0IIR access,
the interrupt is recorded for the next U0IIR access.

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

7. Interrupt Identification Register (U0IIR)

Bit0 Interrupt Pending
the U0IIR[0] is active low
the pending interrupt can be determined by evaluating U0IIR[3:1]

Bit3:1 Interrupt Identification
U0IIR[3:1] identifies an interrupt corresponding to the UART0 Rx FIFO
All other combinations of U0IIR[3:1] not list are reserved
(000, 100, 101, 111)

011	1	RLS (Receive Line Status)
010	2a	RDA (Receive Data Available)
110	2b	CTI (Character Time-Out Indicator
001	3	THRE (Transmitter Holding Register Empty)

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

7. Interrupt Identification Register (U0IIR)

Bit7:6 FIFO Enable
These bits are equivalent to U0FCR[0]

Bit8 ABEOInt End of auto-baud interrupt
true if auto-baud has finished successfully and interrupt is enabled

Bit9 ABTOInt Auto-baud time-out interrupt
true if auto-baud has timed out and interrupt is enabled

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

UART0 interrupt handling

U0IIR[3:0] = 0001
- Priority : -
- Interrupt Type : None
- Interrupt Source : None
- Interrupt Reset : -

OE (Overrun Error)
PE (Parity Error)
FE (Framing Error)
BI (Break Interrupt)

U0IIR[3:0] = 0110 RLS (Receive Line Status)
- Priority : Highest
- Interrupt Type : RX Line Status / Error
- Interrupt Source : OE or PE or FE or BI
- Interrupt Reset : U0LSR Read

U0IIR[3:0] = 0100 RLS (Receive Line Status)
- Priority : Second
- Interrupt Type : RX Data Available
- Interrupt Source : Rx data available or trigger level reached in FIFO (U0FCR0=1)
- Interrupt Reset : U0RBR Read or UART0 FIFO drops below trigger level

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

011	1	RLS (Receive Line Status)
010	2a	RDA (Receive Data Available)
110	2b	CTI (Character Time-Out Indicator
001	3	THRE (Transmitter Holding Register Empty)

UART0 interrupt handling

U0IIR[3:0] = 1100 CTI (Character Time-Out Indicator

Priority : Second

Interrupt Type : Character Time-out indication

Interrupt Source : Minimum of one character in the Rx FIFO and
no character input or removed during a time period
depending on how many characters are in FIFO
and what the trigger level is set at (3.5 to 4.5 character times).

The exact time will be:
[(word length) × 7 − 2] × 8 + [(trigger level − number of characters) × 8 + 1] RCLKs

Interrupt Reset : U0RBR Read

U0IIR[3:0] = 0010 THRE (Transmitter Holding Register Empty)

Priority : Third

Interrupt Type : THRE

Interrupt Source : THRE

Interrupt Reset : U0IIR Read (if source of interrupt) or THR write

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

011	1	RLS (Receive Line Status)
010	2a	RDA (Receive Data Available)
110	2b	CTI (Character Time-Out Indicator
001	3	THRE (Transmitter Holding Register Empty)

UART0 interrupt handling

Interrupts are handled as described in Table 105.
Given the status of U0IIR[3:0], an interrupt handler routine
can determine the cause of the interrupt
and how to clear the active interrupt.

The U0IIR must be read in order to clear
the interrupt prior to exiting the Interrupt Service Routine.

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

UART0 interrupt handling

The UART0 **RLS** interrupt (U0IIR[3:1] = 011)
is the highest priority interrupt and
is set whenever any one of four error conditions
occur on the UART0 Rx input:

overrun error (OE),
parity error (PE),
framing error (FE) and
break interrupt (BI).

The UART0 Rx error condition that set the interrupt
can be observed via U0LSR[4:1].

The interrupt is cleared upon an U0LSR read.

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

UART0 interrupt handling

Overrun error is when a new byte is received
before the previous byte was read by a CPU.

Slightly different when a FIFO is involved
but amounts to the same thing
- valid received data is lost due to CPU slowness.

Always check this before reading a byte and
if the byte is part of a longer message (or command)
throw the whole message/command away
and somehow request the transmitter
to resend the whole message/command.

Under run is not really an error but indicates
to the sending UART that it's transmit buffer is empty
i.e. it is requesting a new byte to transmit.
You don't need to check this.

https://electronics.stackexchange.com/questions/83379/what-causes-uart-errors

UART0 interrupt handling

The UART0 RDA interrupt (U0IIR[3:1] = 010) shares the second level priority with the CTI interrupt (U0IIR[3:1] = 110). The RDA is activated when the UART0 Rx FIFO reaches the trigger level defined in U0FCR[7:6] and is reset when the UART0 Rx FIFO depth falls below the trigger level. When the RDA interrupt goes active, the CPU can read a block of data defined by the trigger level.

The CTI interrupt (U0IIR[3:1] = 110) is a second level interrupt and is set when the UART0 Rx FIFO contains at least one character and no UART0 Rx FIFO activity has occurred in 3.5 to 4.5 character times. Any UART0 Rx FIFO activity (read or write of UART0 RSR) will clear the interrupt. This interrupt is intended to flush the UART0 RBR after a message has been received that is not a multiple of the trigger level size. For example, if a peripheral wished to send a 105 character message and the trigger level was 10 characters, the CPU would receive 10 RDA interrupts resulting in the transfer of 100 characters and 1 to 5 CTI interrupts (depending on the service routine) resulting in the transfer of the remaining 5 characters.

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

UART0 interrupt handling

The UART0 THRE interrupt (U0IIR[3:1] = 001) is a third level interrupt and is activated when the UART0 THR FIFO is empty provided certain initialization conditions have been met. These initialization conditions are intended to give the UART0 THR FIFO a chance to fill up with data to eliminate many THRE interrupts from occurring at system start-up. The initialization conditions implement a one character delay minus the stop bit whenever THRE=1 and there have not been at least two characters in the U0THR at one time since the last THRE = 1 event. This delay is provided to give the CPU time to write data to U0THR without a THRE interrupt to decode and service. A THRE interrupt is set immediately if the UART0 THR FIFO has held two or more characters at one time and currently, the U0THR is empty. The THRE interrupt is reset when a U0THR write occurs or a read of the U0IIR occurs and the THRE is the highest interrupt (U0IIR[3:1] = 001).

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

8. FIFO Control Register (U0FCR)

The U0FCR controls the operation of the UART0 Rx and TX FIFOs.

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

9. Line Control Register (U0LCR)

The U0LCR determines the format of the data character that is to be transmitted or received.

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

10. Line Status Register (U0LSR)

The U0LSR is a read-only register
that provides status information on the UART0 TX and RX blocks.

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

11. Scratch pad register

The U0SCR has no effect on the UART0 operation.

This register can be written and/or read at user's discretion.

There is no provision in the interrupt interface
that would indicate to the host
that a read or write of the U0SCR has occurred.

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

12. Auto-baud Control Register

The UART0 Auto-baud Control Register (U0ACR)
controls the process of measuring the incoming clock/data rate
for the baud rate generation and
can be read and written at user's discretion.

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

14. Transmit Enable Register

LPC2141/2/4/6/8's U0TER enables implementation of software flow control.

When TXEn=1, UART0 transmitter will keep sending data
as long as they are available.

As soon as TXEn becomes 0,
UART0 transmission will stop.

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

U0IIR

Bit0	Interrupt Pending	Note that U0IIR[0] is active low. The pending interrupt can be determined by evaluating U0IIR[3:1].
Bit3:1	Interrupt Identification	U0IER[3:1] identifies an interrupt corresponding to the UART0 Rx FIFO. All other combinations of U0IER[3:1] not listed above are reserved (000,100,101,111).
Bit5:4	-	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.
Bit7:6	FIFO Enable	These bits are equivalent to U0FCR[0].
Bit8	ABEOInt	End of auto-baud interrupt. True if auto-baud has finished successfully and interrupt is enabled.
Bit9	ABTOInt	Auto-baud time-out interrupt. True if auto-baud has timed out and interrupt is enabled.
Bit31:10	-	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.

https://www.keil.com/dd/docs/datashts/philips/user_manual_lpc214x.pdf

References

[1] http://wiki.osdev.org/ARM_RaspberryPi_Tutorial_C
[2] http://blog.bobuhiro11.net/2014/01-13-baremetal.html
[3] http://www.valvers.com/open-software/raspberry-pi/
[4] https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/downloads.html