Last Time

Bit manipulation

- Determining pin configuration: input/output
- Determining the output pin state
- Reading the input pin

Today

- Communicating between devices
  - Serial communication
  - Communication in code

- Project 1 is due on Tuesday: don't delay on getting started
- Next Thursday's class: in the lab

Input/Output Systems

Processor needs to communicate with other devices:

- Receive signals from sensors
- Send commands to actuators
- Or both (e.g., disks, audio, video devices)

I/O Systems

Communication can happen in a variety of ways:

- Binary parallel signal (e.g., project 1)
- Analog
- Serial signals

An Example: SICK Laser Range Finder

- Laser is scanned horizontally
- Using phase information, can infer the distance to the nearest obstacle
- Resolution: ~.5 degrees, 1 cm
- Can handle full 180 degrees at 20 Hz

Serial Communication

- Communicate a set of bytes using a single signal line
- We do this by sending one bit at a time:
  - The value of the first bit determines the state of a signal line for a specified period of time
  - Then, the value of the 2nd bit is used
  - Etc.

Serial Communication

The sender and receiver must have some way of agreeing on when a specific bit is being sent

- Typically, each side has a clock to tell it when to write/read a bit
- In some cases, the sender will also send a clock signal (on a separate line)
- In other cases, the sender/receiver will first synchronize their clocks before transfer begins

Asynchronous Serial Communication

- The sender and receiver have their own clocks, which they do not share
- This reduces the number of signal lines
- Bidirectional transmission, but the two halves do not need to be synchronized in time

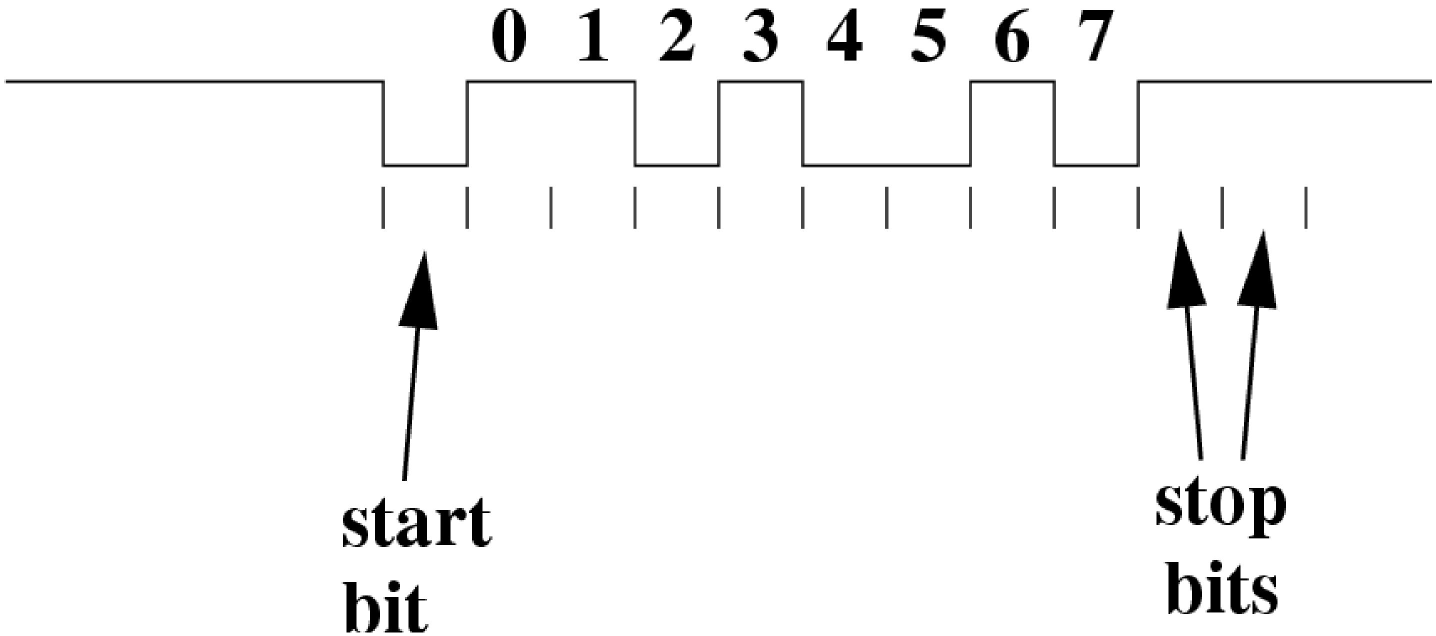
But: we still need some way to agree that data is valid. How?

Asynchronous Serial Communication

How can the two sides agree that the data is valid?

- Must both be operating at essentially the same transmit/receive frequency
- A data byte is prefaced with a bit of information that tells the receiver that data is coming
- The receiver uses the arrival time of this **start bit** to synchronize its clock

A Typical Data Frame



The stop bits allow the receiver to immediately check whether this is a valid frame

- If not, the byte is thrown away

Data Frame Handling

Most of the time, we do not personally deal with the data frame level. Instead, we rely on:

- Hardware solutions: Universal Asynchronous Receiver Transmitter (UART)
  - Very common in computing devices
- Software solutions in libraries

One Standard: RS232-C

Defines a logic encoding standard:

- “High” is encoded with a voltage of -5 to -15 (-12 to -13V is typical)
- “Low” is encoded with a voltage of 5 to 15 (12 to 13V is typical)

RS232-C

Originally intended to connect:

- Data Terminal Equipment (DTE)
  - Teletypes
- to Data Communication Equipment (DCE)
  - Modems

Now that we are connecting a computer to some peripheral, it is not always clear which is the DTE and which is the DCE

RS232-C

Defines a pin assignment standard. For example, with the DB-9 connectors:

- Pin 2: receive (to DTE from DCE)
- Pin 3: transmit (from DTE to DCE)
- Pin 5: common (ground)

Also common to have DB-25 connectors (older standard)

RS232 on the Mega8

Our mega 8 has a Universal, Asynchronous serial Receiver/Transmitter (UART)

- Handles all of the bit-level manipulation
- You only have to interact with it on the byte level

Mega8 UART C Interface

OUlib support:

`serial0_init(9600)`: initialize the port @9600 bits per second

`getchar()`: receive a character

`kbhit()`: is there a character in the buffer?

`putchar(0x45)`: put a character out to the port

See the Atmel HOWTO

Character Representation

- A “char” is just an 8-bit number
- In some cases, we just interpret it differently.
- But: we can still perform mathematical operations on it

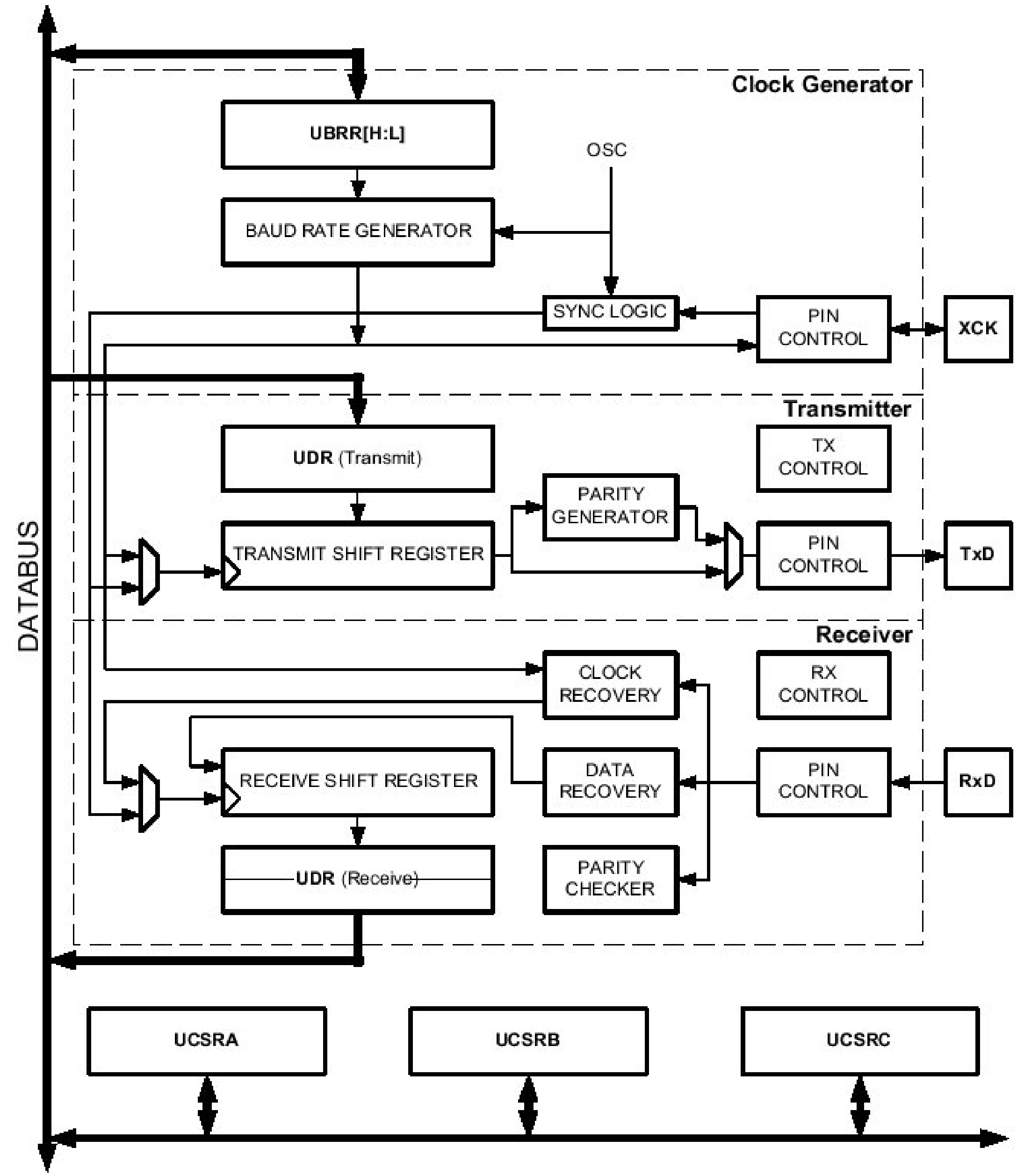
Character Representation: ASCII

Binary	Dec	Hex	Glyph
010 0000	32	20	SP
010 0001	33	21	!
010 0010	34	22	"
010 0011	35	23	#
010 0100	36	24	$
010 0101	37	25	%
010 0110	38	26	&
010 0111	39	27	'
010 1000	40	28	(
010 1001	41	29	)
010 1010	42	2A	*
010 1011	43	2B	+
010 1100	44	2C	,
010 1101	45	2D	-
010 1110	46	2E	.
010 1111	47	2F	/
011 0000	48	30	0
011 0001	49	31	1
011 0010	50	32	2
011 0011	51	33	3
011 0100	52	34	4
011 0101	53	35	5
011 0110	54	36	6
011 0111	55	37	7
011 1000	56	38	8
011 1001	57	39	9
011 1010	58	3A	:
011 1011	59	3B	;
011 1100	60	3C	<
011 1101	61	3D	=
011 1110	62	3E	>
011 1111	63	3F	?

Binary	Dec	Hex	Glyph
100 0000	64	40	@
100 0001	65	41	A
100 0010	66	42	B
100 0011	67	43	C
100 0100	68	44	D
100 0101	69	45	E
100 0110	70	46	F
100 0111	71	47	G
100 1000	72	48	H
100 1001	73	49	I
100 1010	74	4A	J
100 1011	75	4B	K
100 1100	76	4C	L
100 1101	77	4D	M
100 1110	78	4E	N
100 1111	79	4F	O
101 0000	80	50	P
101 0001	81	51	Q
101 0010	82	52	R
101 0011	83	53	S
101 0100	84	54	T
101 0101	85	55	U
101 0110	86	56	V
101 0111	87	57	W
101 1000	88	58	X
101 1001	89	59	Y
101 1010	90	5A	Z
101 1011	91	5B	[
101 1100	92	5C	\
101 1101	93	5D	]
101 1110	94	5E	^
101 1111	95	5F	_

Binary	Dec	Hex	Glyph
110 0000	96	60	`
110 0001	97	61	a
110 0010	98	62	b
110 0011	99	63	c
110 0100	100	64	d
110 0101	101	65	e
110 0110	102	66	f
110 0111	103	67	g
110 1000	104	68	h
110 1001	105	69	i
110 1010	106	6A	j
110 1011	107	6B	k
110 1100	108	6C	l
110 1101	109	6D	m
110 1110	110	6E	n
110 1111	111	6F	o
111 0000	112	70	p
111 0001	113	71	q
111 0010	114	72	r
111 0011	115	73	s
111 0100	116	74	t
111 0101	117	75	u
111 0110	118	76	v
111 0111	119	77	w
111 1000	120	78	x
111 1001	121	79	y
111 1010	122	7A	z
111 1011	123	7B	{
111 1100	124	7C	|
111 1101	125	7D	}
111 1110	126	7E	~

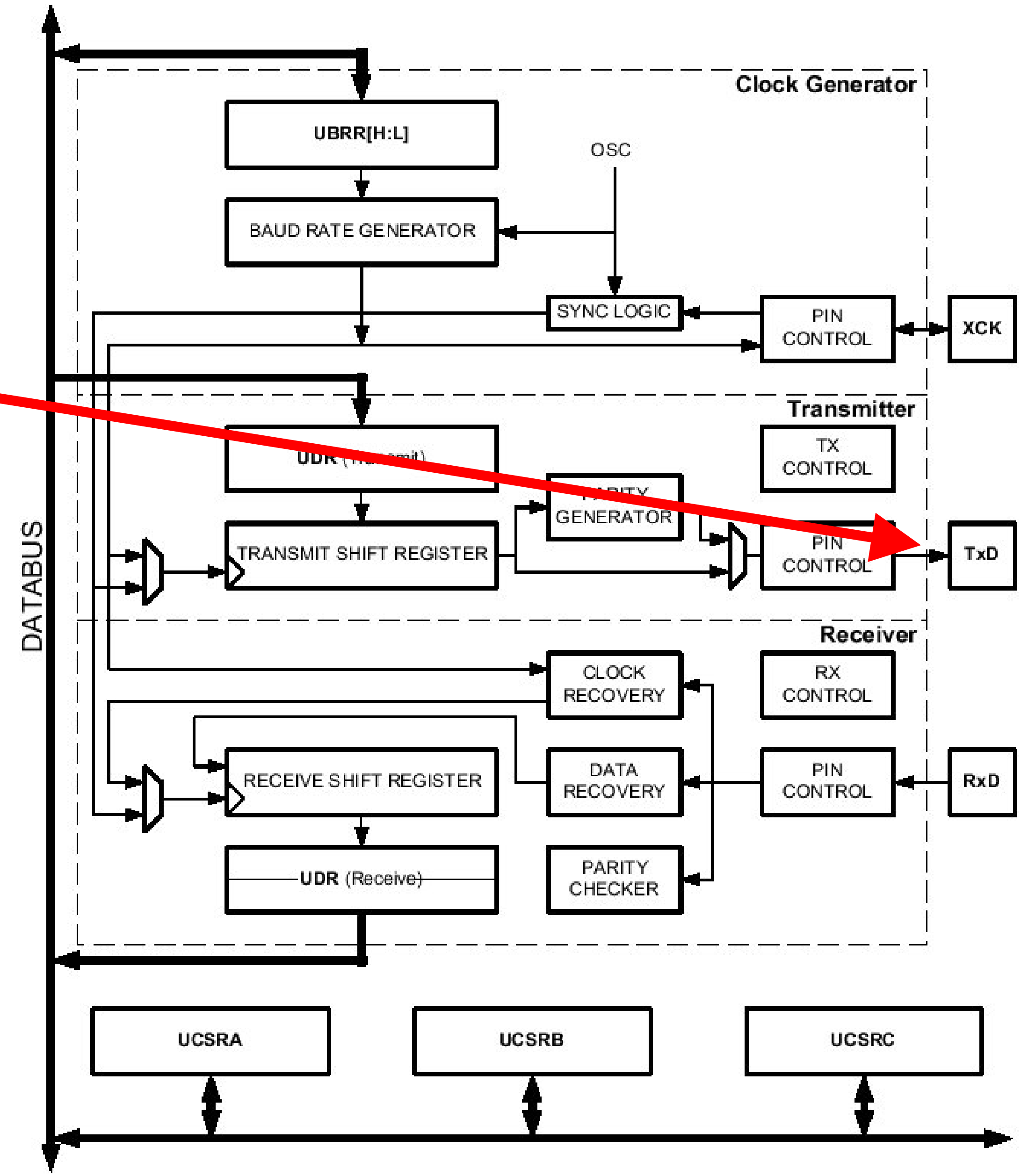
Mega8 UART



Clock Generator
UBRR[H:L]
OSC
BAUD RATE GENERATOR
SYNC LOGIC
PIN CONTROL
XCK
Transmitter
UDR (Transmit)
TX CONTROL
PARITY GENERATOR
TRANSMIT SHIFT REGISTER
PIN CONTROL
TxD
DATABUS
Receiver
CLOCK RECOVERY
RX CONTROL
RECEIVE SHIFT REGISTER
DATA RECOVERY
PIN CONTROL
RxD
UDR (Receive)
PARITY CHECKER
UCSRA
UCSRB
UCSRC

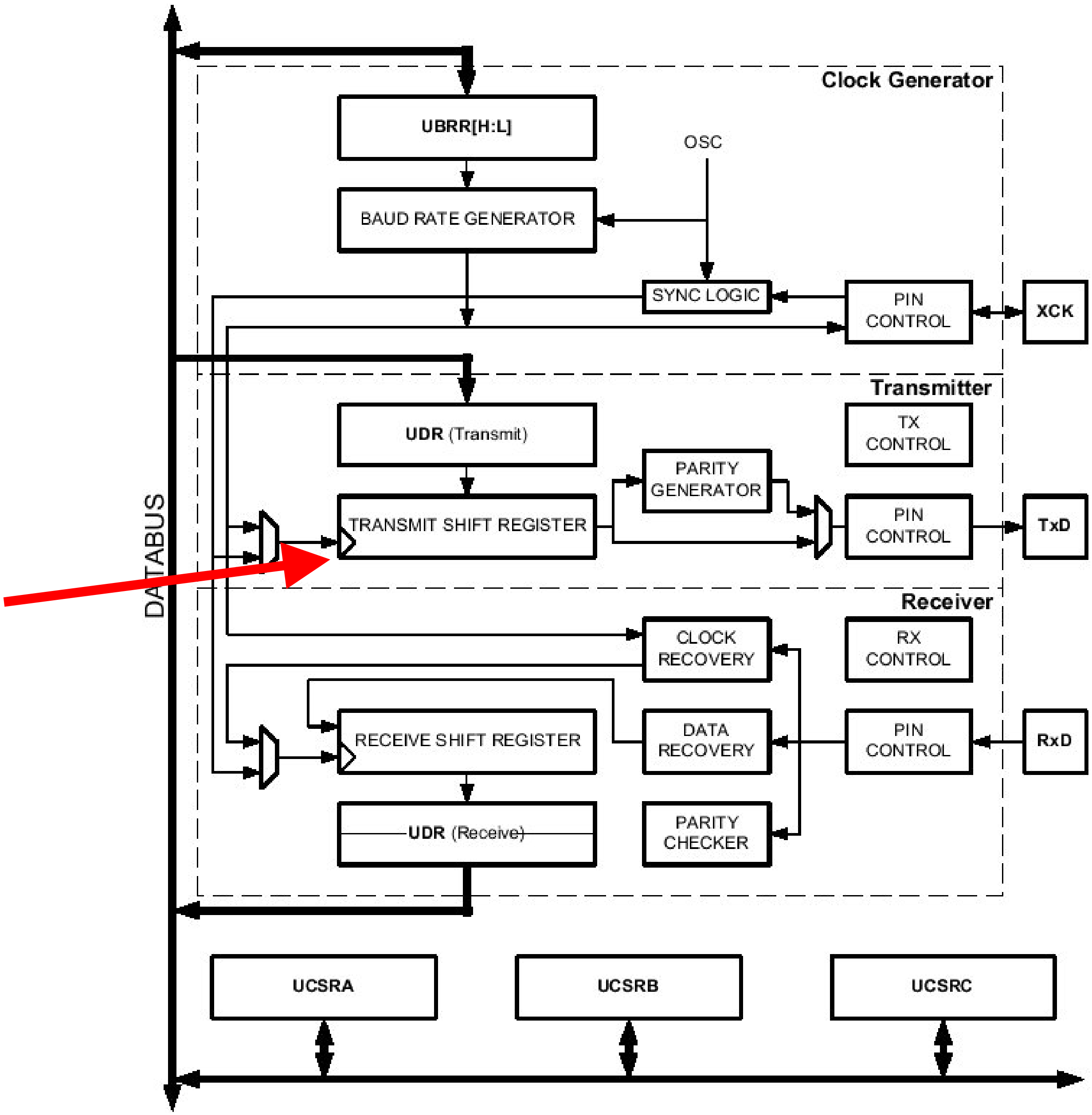
Mega8 UART

- Transmit pin (PD1)



Mega8 UART

- Transmit pin (PD1)
- Transmit shift register

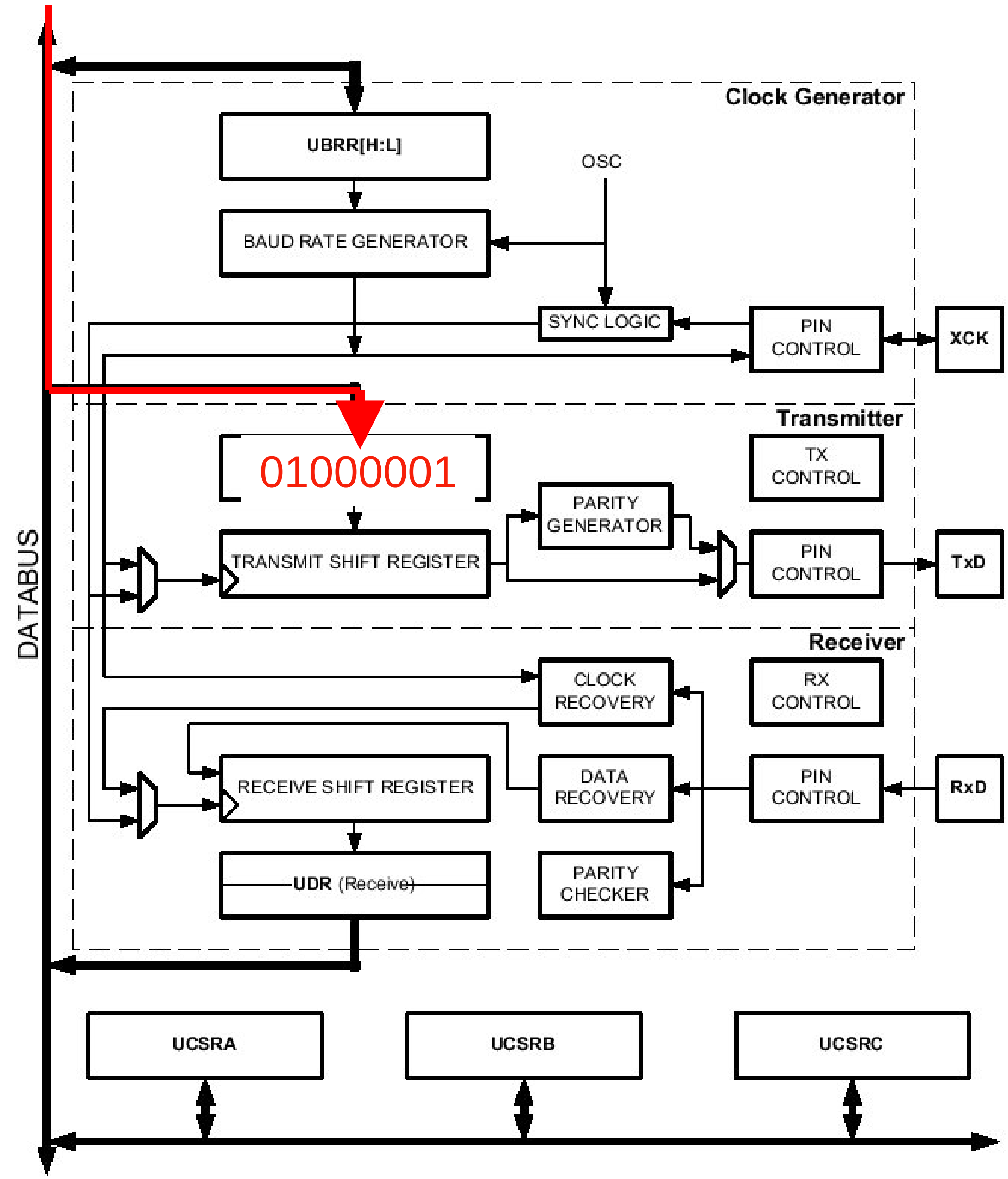


Writing a Byte to the Serial Port

```
putchar('A');
```

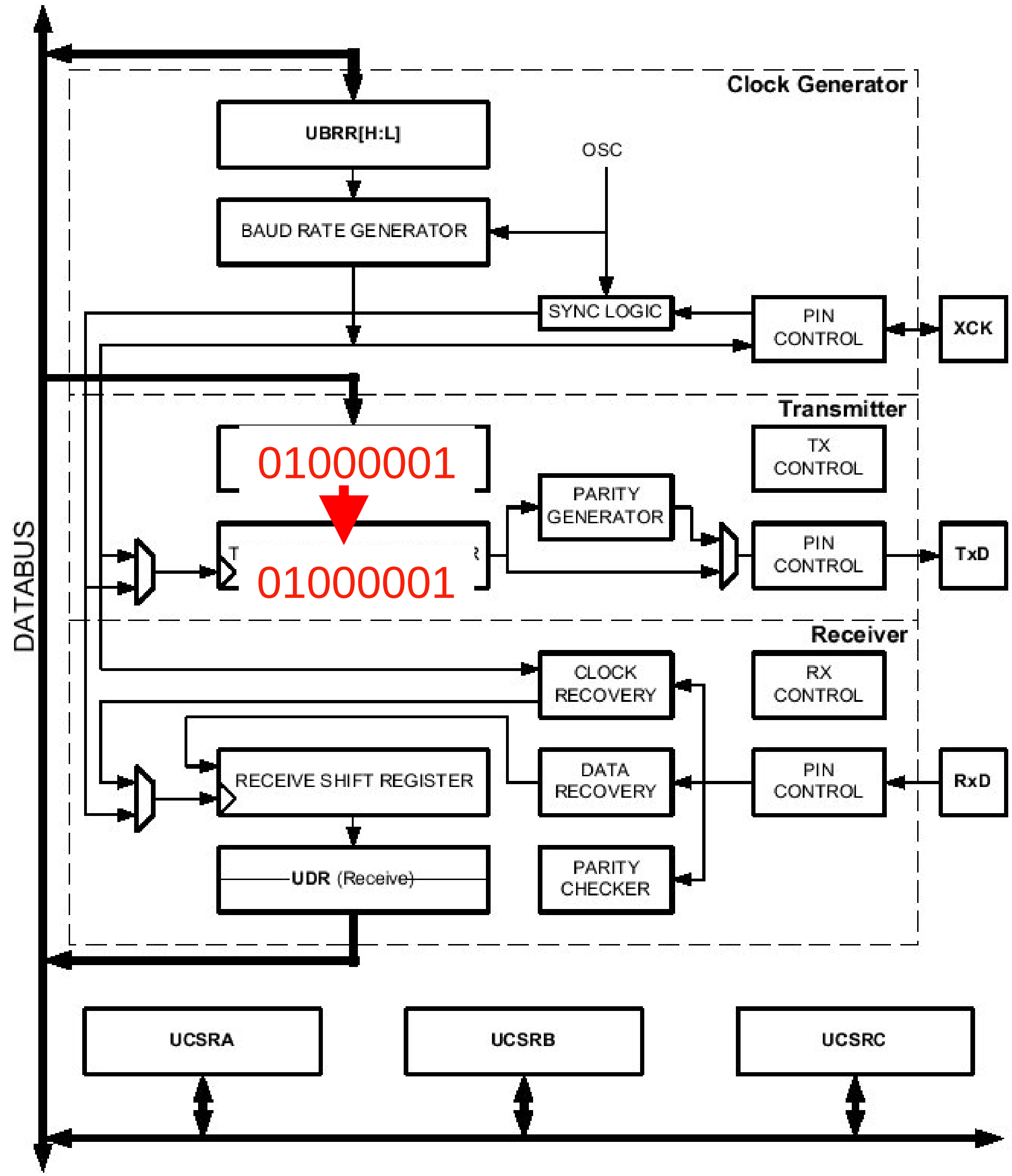
Transmit

```
putchar('A');
```



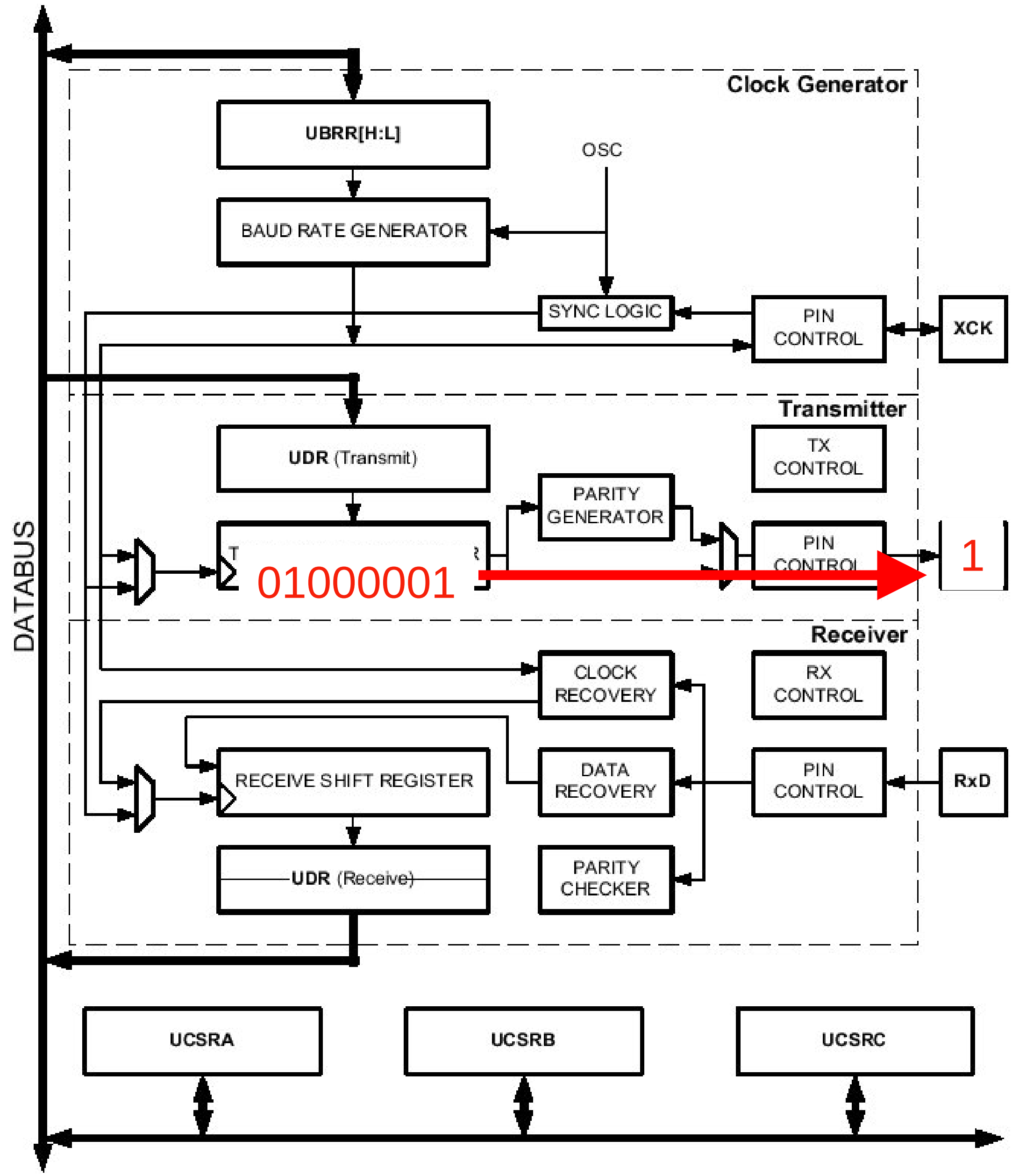
Transmit

When UART is ready, the buffer contents are copied to the shift register



Transmit

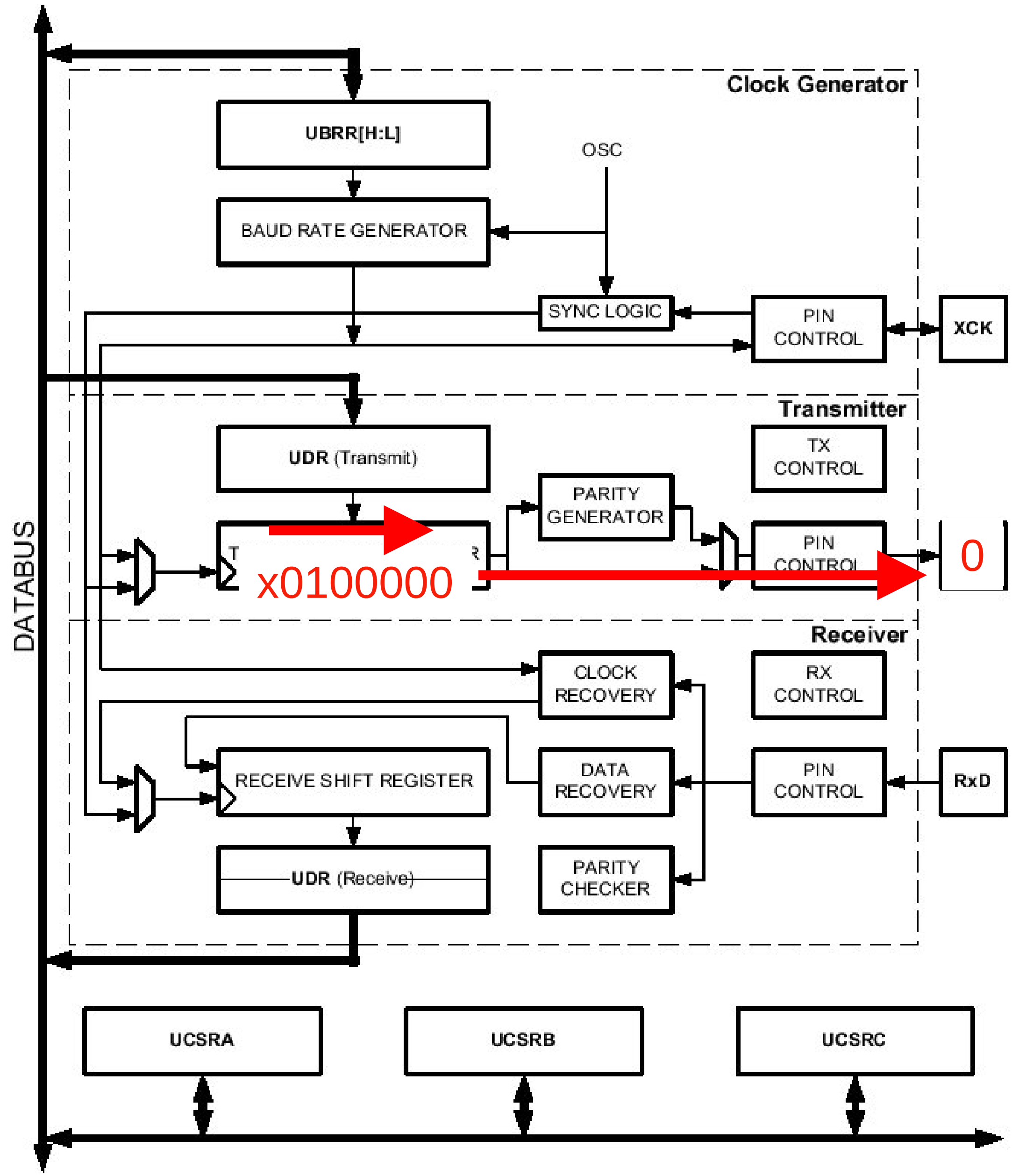
The least significant bit (LSB) of the shift register determines the state of the pin



Transmit

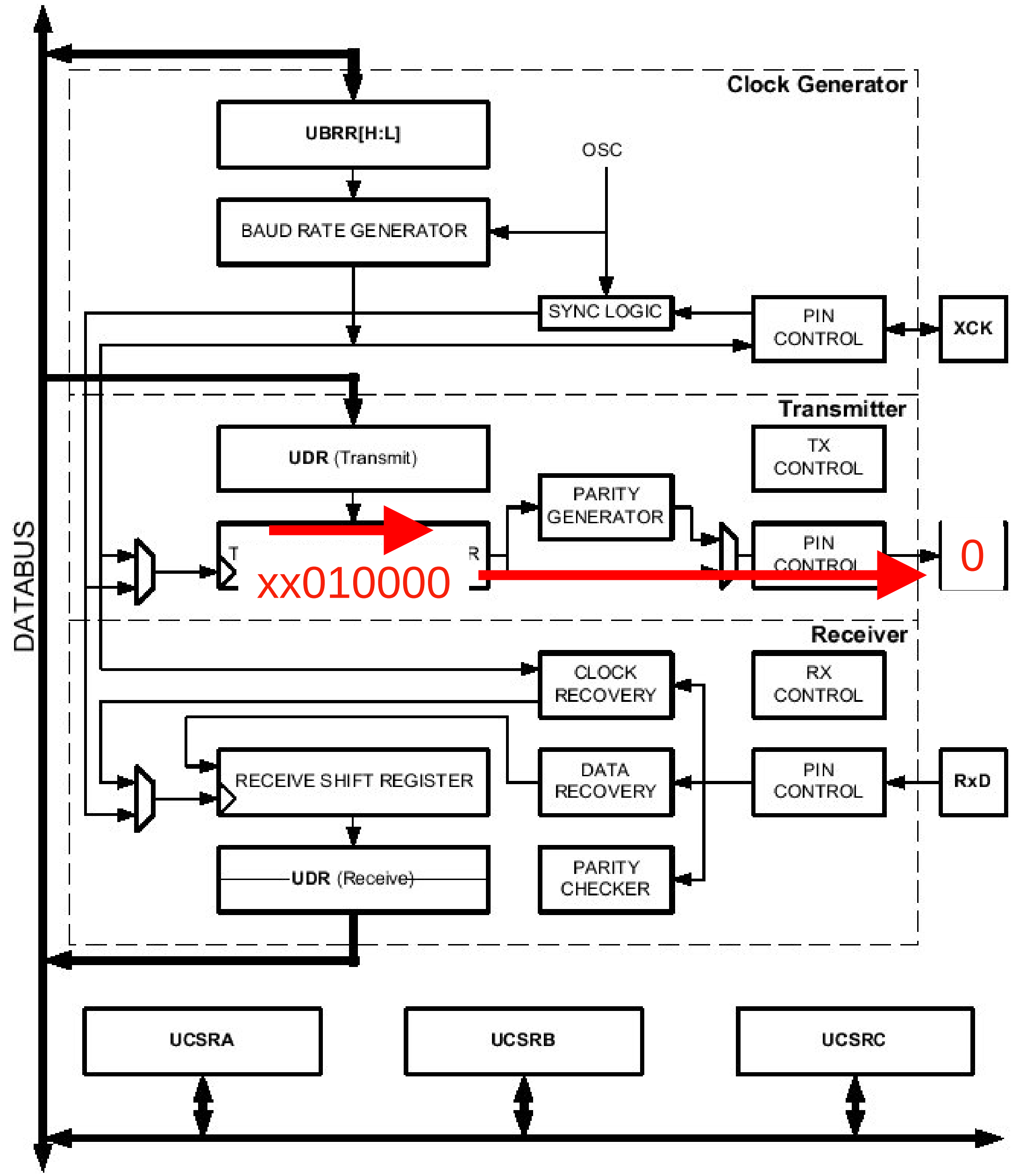
After a delay, the UART shifts the values to the right

x = value doesn't matter



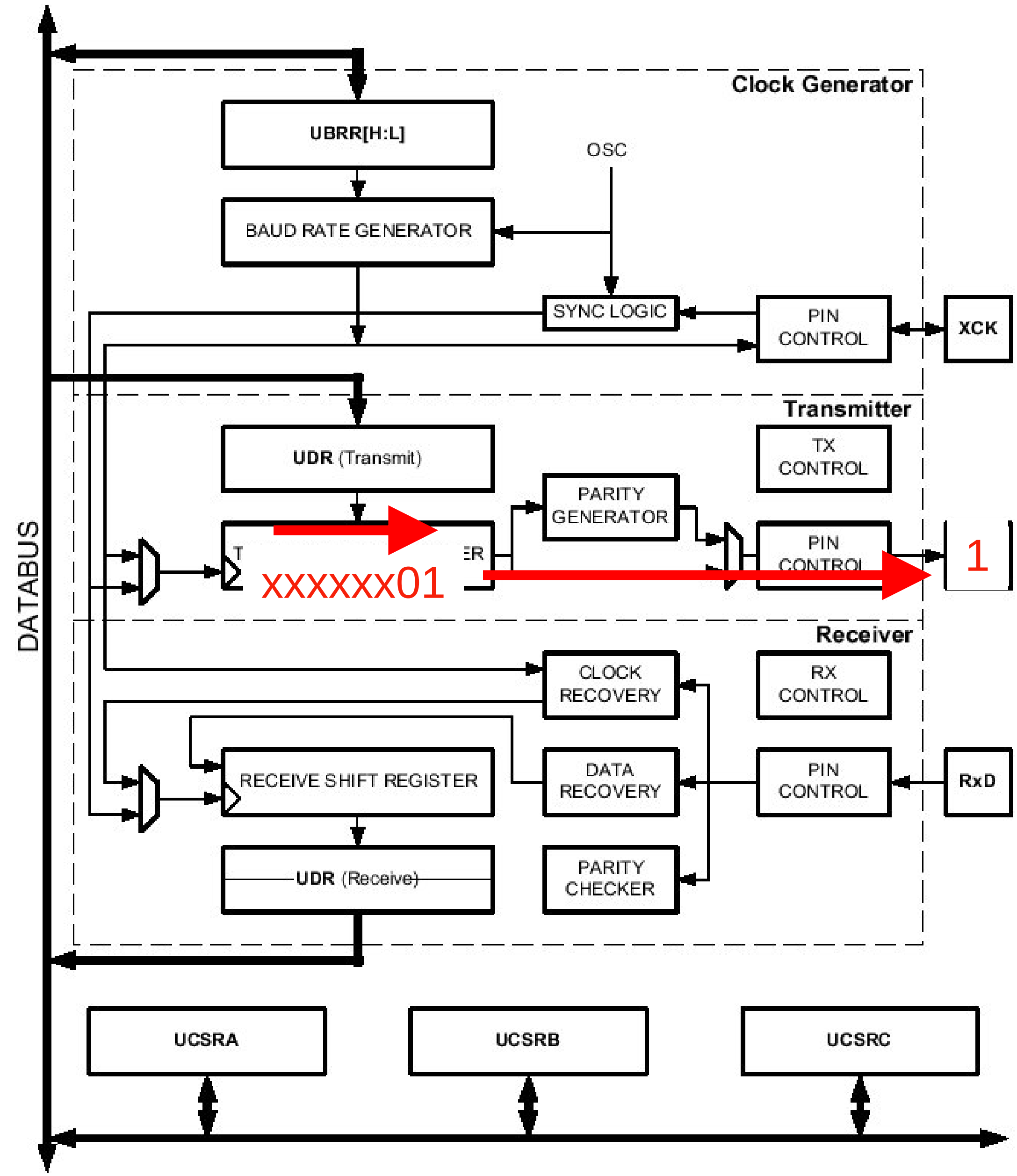
Transmit

Next shift



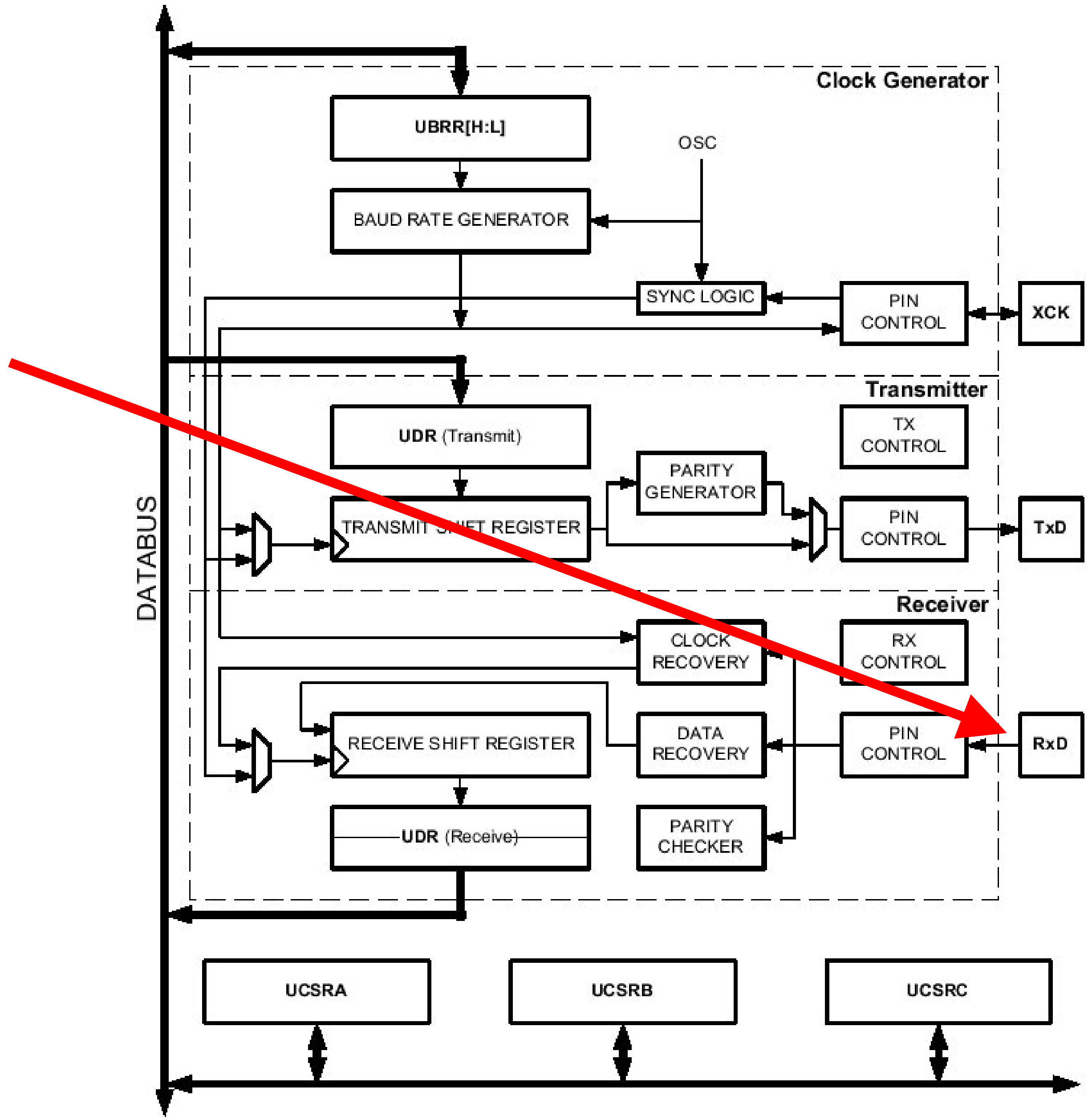
Transmit

Several shifts later…



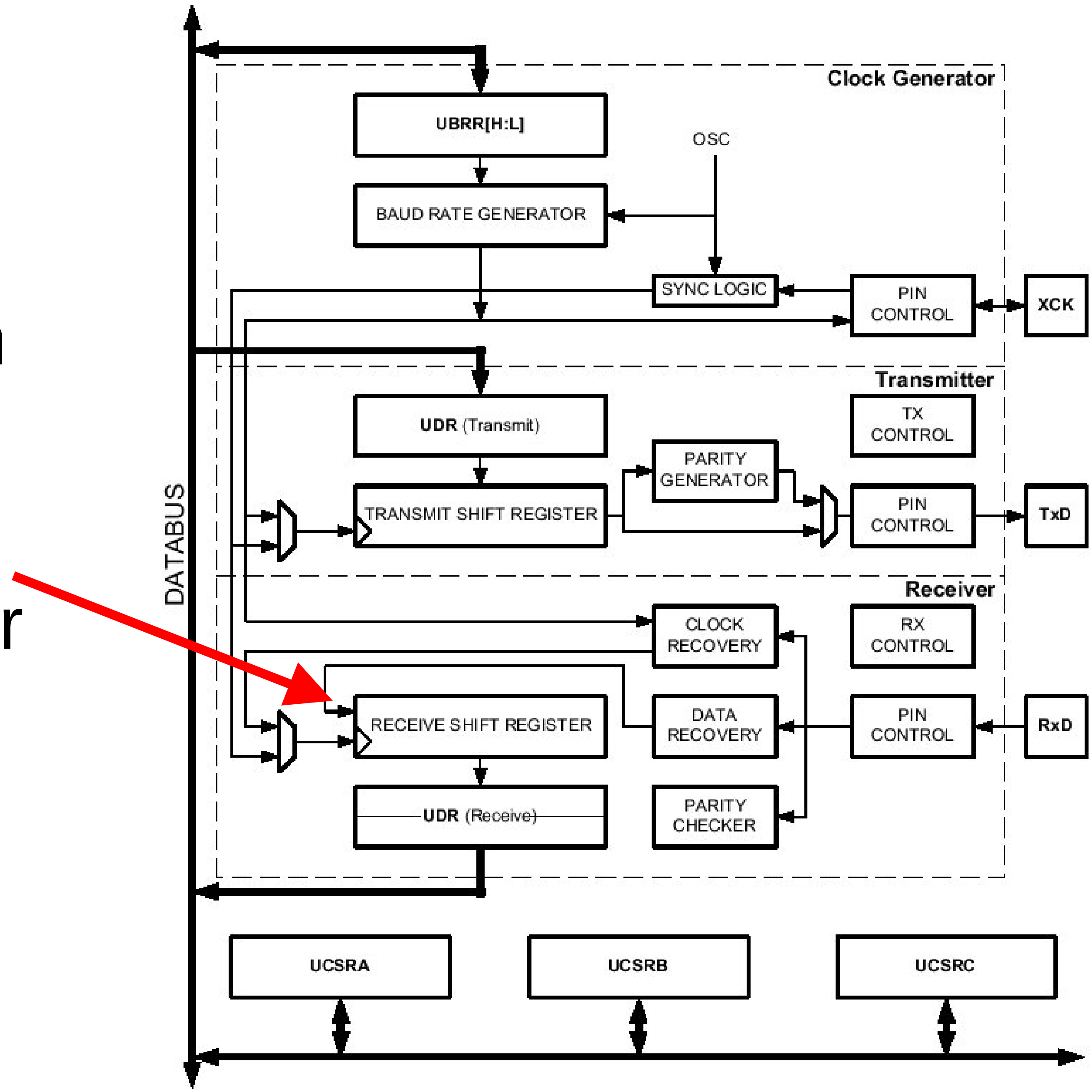
Receive

- Receive pin (PD0)



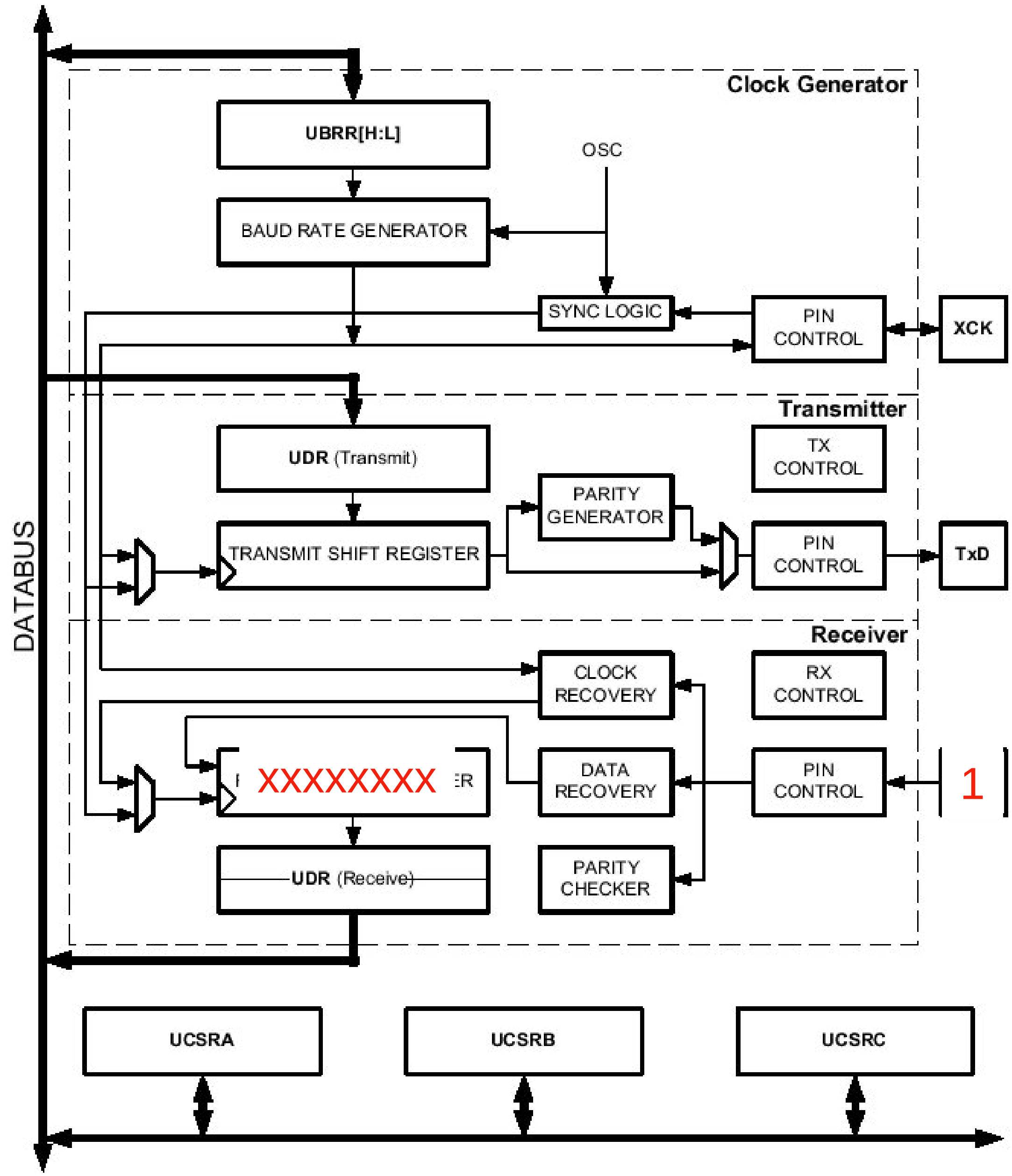
Receive

- Receive pin (PD0)
- Receive shift register



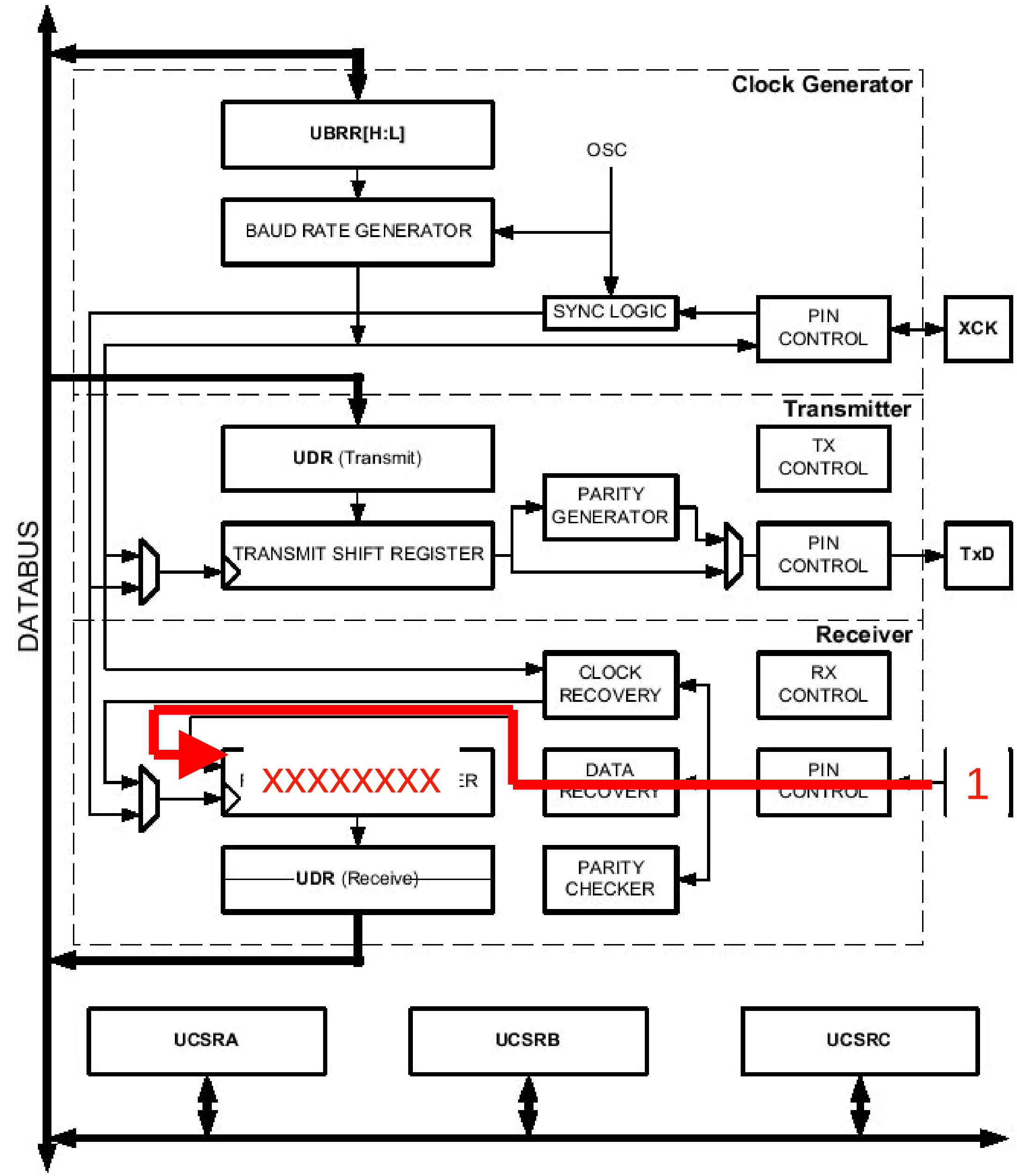
Receive

- "1" on the pin
- Shift register initially in an unknown state



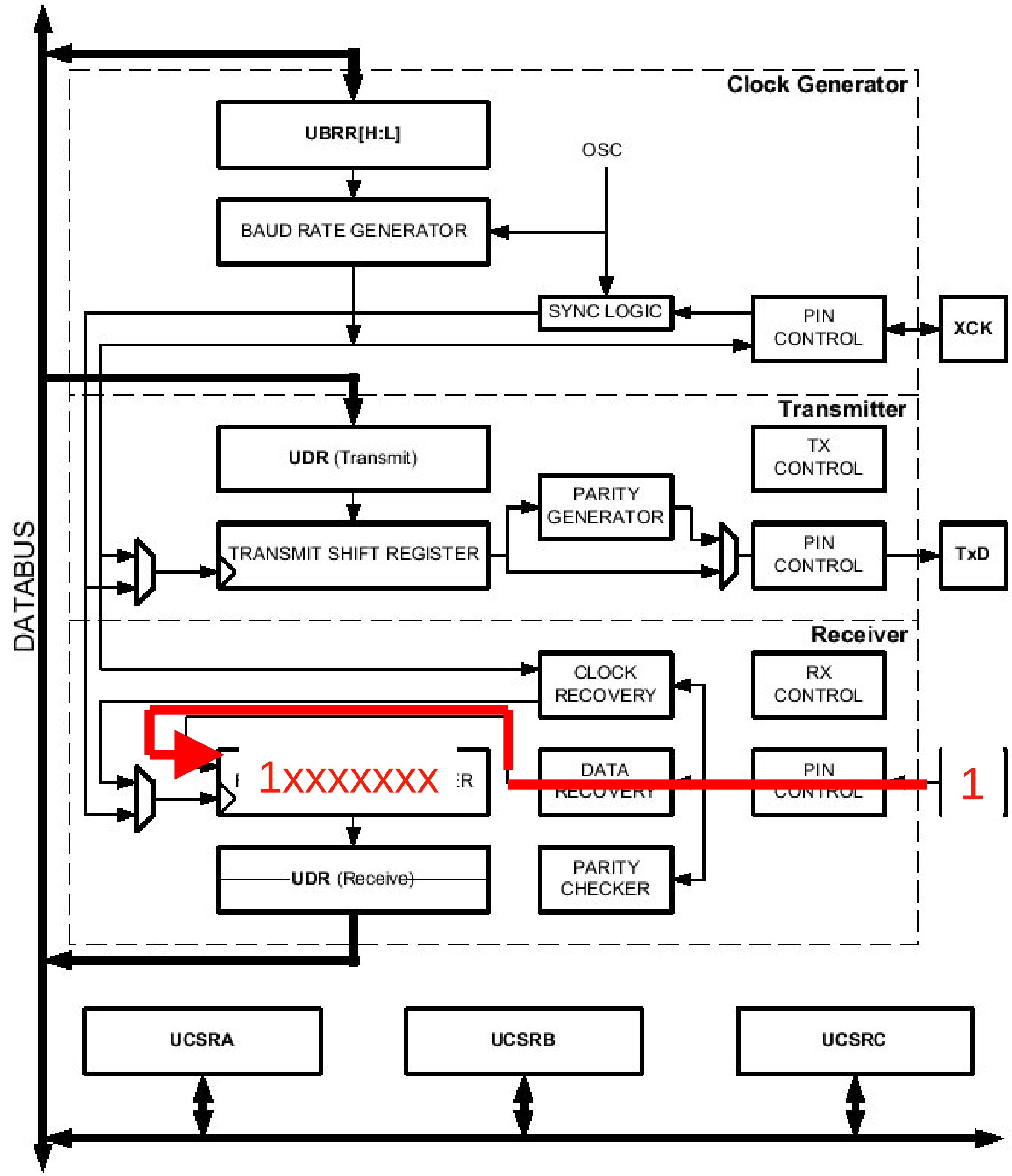
Receive

- "1" is presented to the shift register



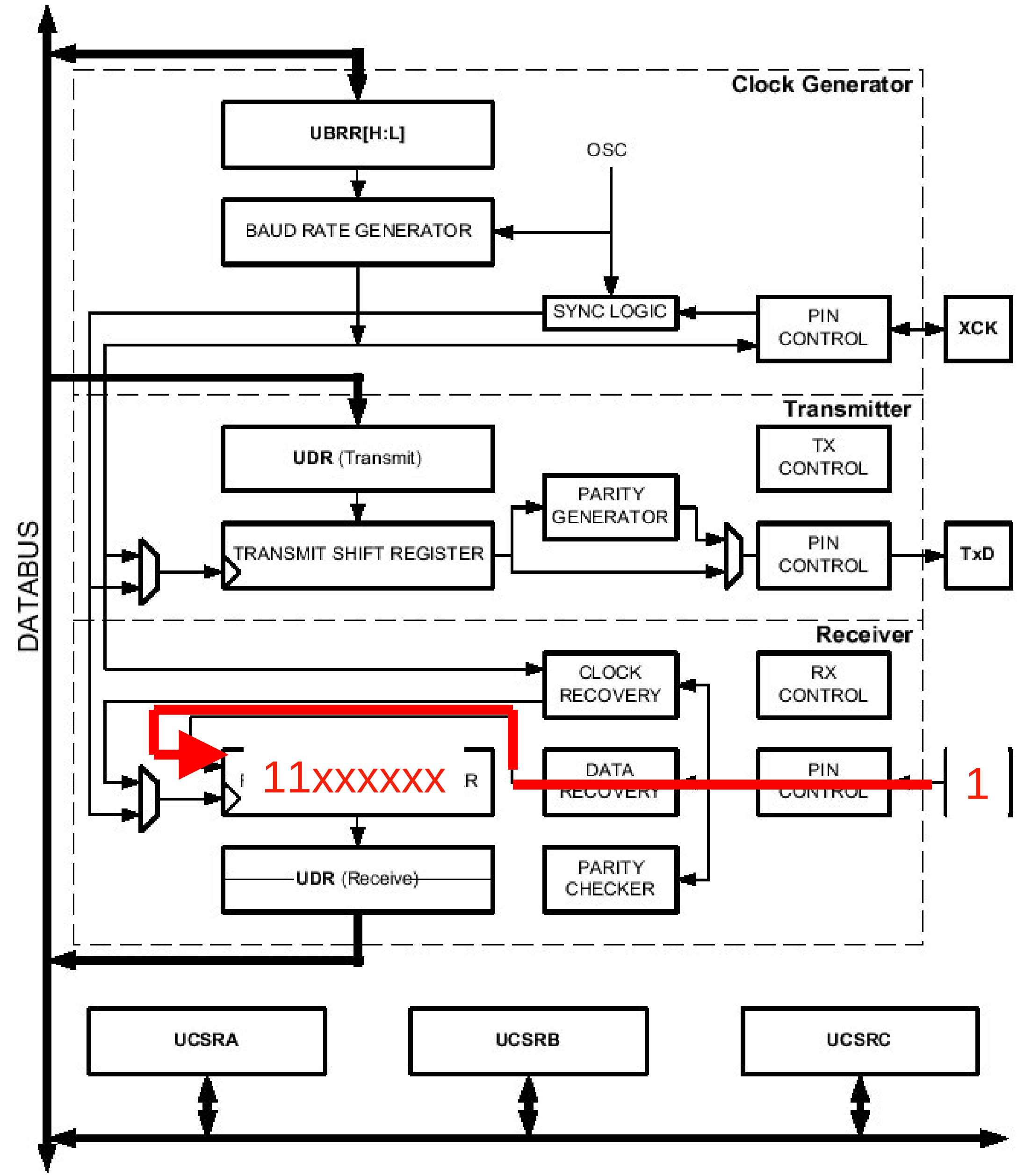
Receive

"1" is shifted into the most significant bit (msb) of the shift register



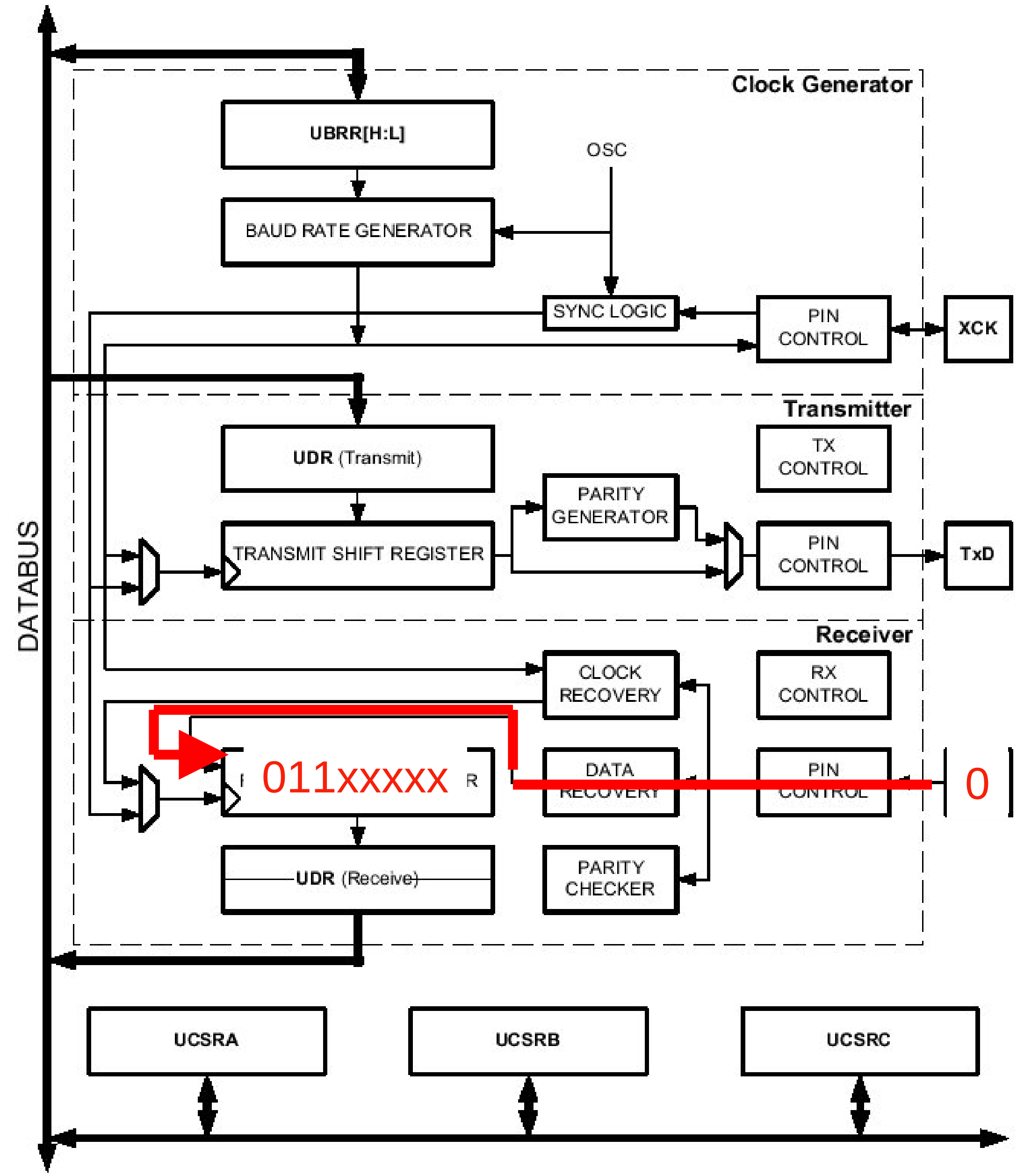
Receive

Next bit is shifted in



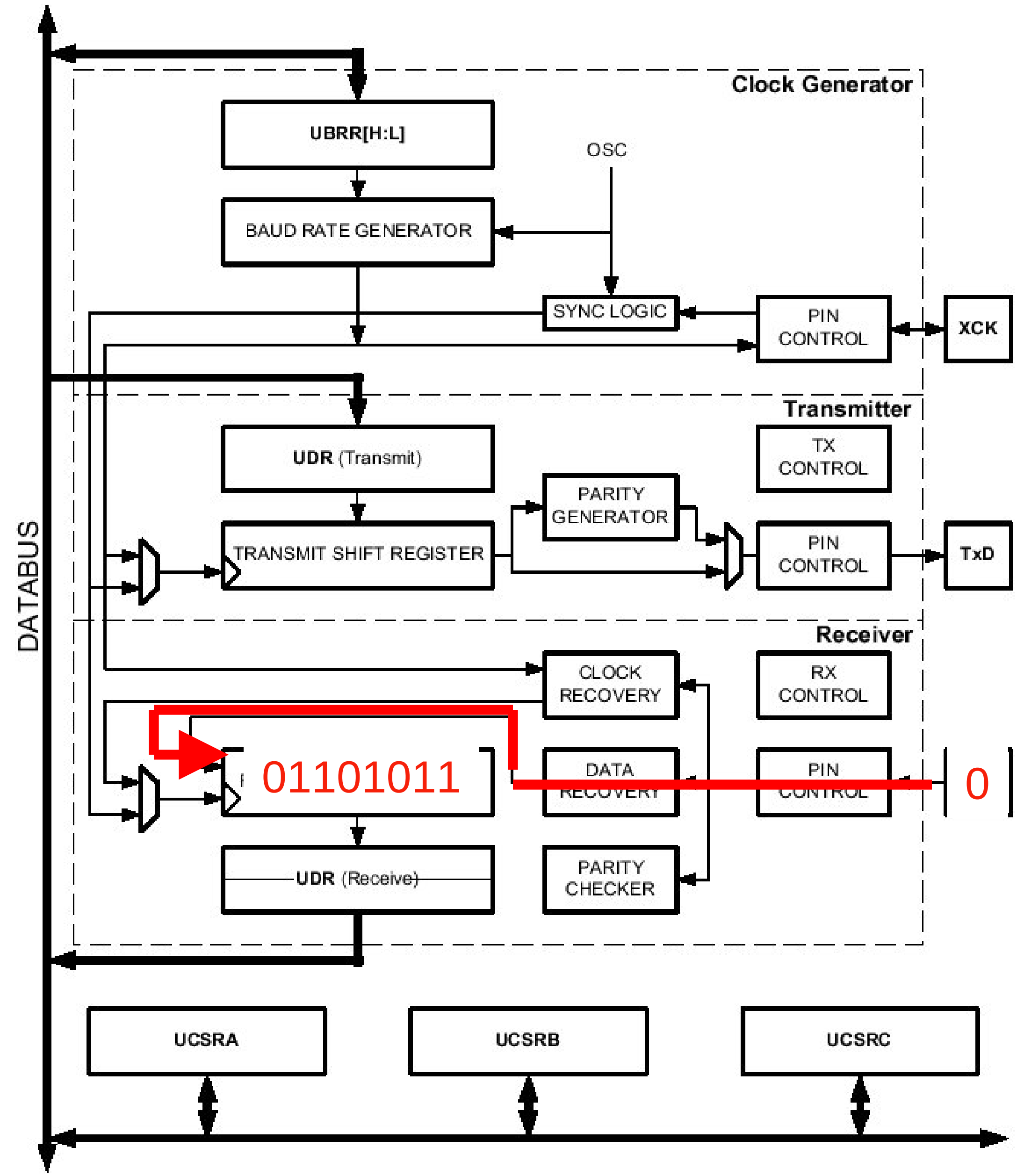
Receive

And the next bit…



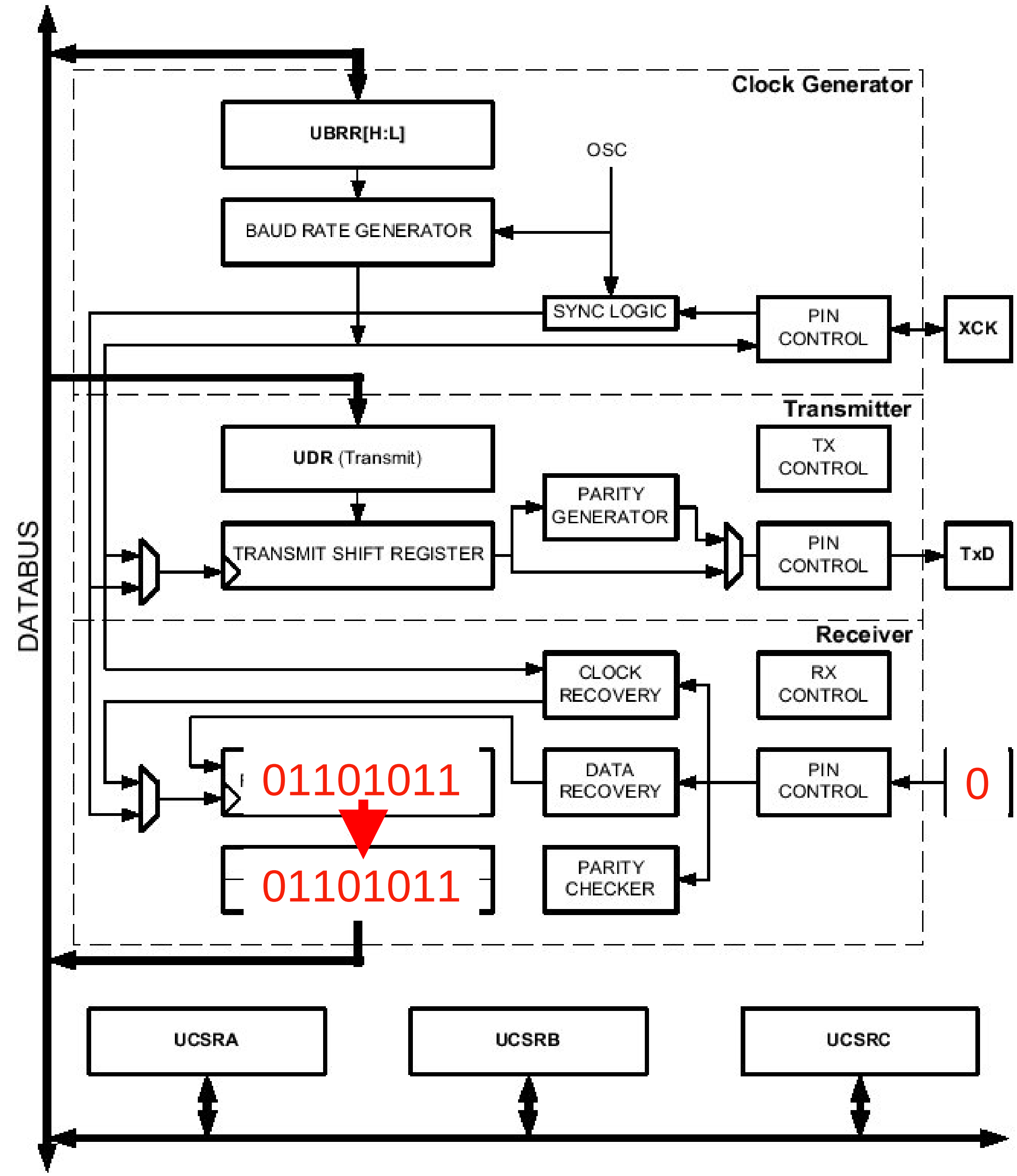
Receive

And the 8th bit



Receive

Completed byte is stored in the UART buffer



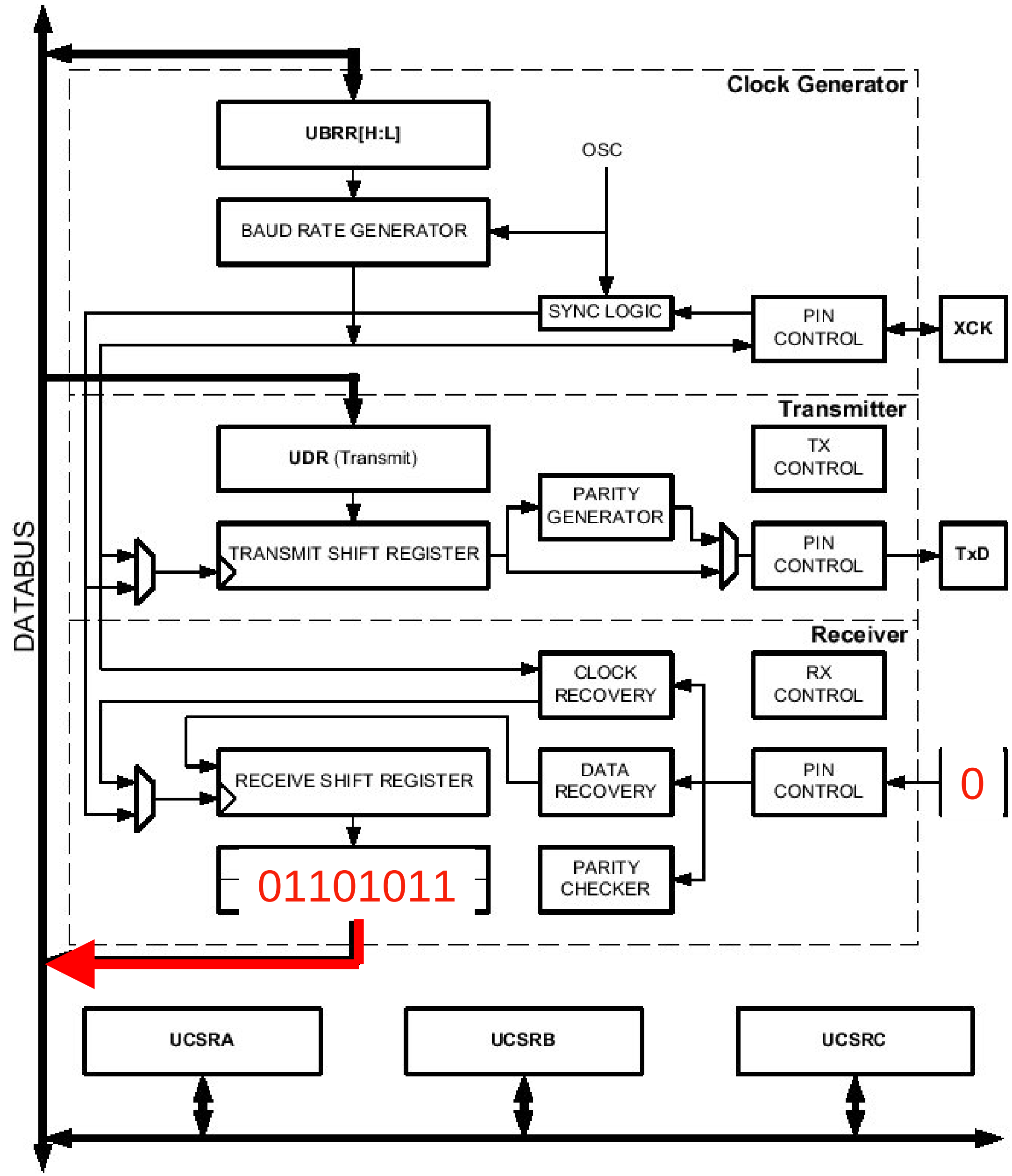
Reading a Byte from the Serial Port

```
int c;

c=getchar();
```

Receive

getchar() retrieves this byte from the buffer



Reading a Byte from the Serial Port

```
int c;

c=getchar();
```

Note: getchar() “blocks” until a byte is available

- Will only return with a value once one is available to be returned

Processing Serial Input

```
int c;
while(1) {
  if(kbhit()) {
     // A character is available for reading
     c = getchar();
     <do something with the character>
  }
  <do something else while waiting>
}
```

kbhit() tells us whether a byte is ready to be read

Mega8 UART C Interface

`printf()`: formatted output

`scanf()`: formatted input

See the LibC documentation or the AVR C textbook

Note: scanf() does not work properly with serial0_init() (more on this later)

Serial I/O by Polling

```
int c;
while(1) {
  if(kbhit()) {
     // A character is available for reading
     c = getchar();
     <do something with the character>
  }
  <do something else while waiting>
}
```

Next Time

- Building circuits with Atmel mega8s
- Getting ready for project 2

Last Time

- Interrupts in general
- External interrupt request
  – The mega8 has 2 pins
- Serial protocols
- RS232-C

Today

- Serial processing: from polling to interrupts

I/O By Polling

Polling works great … but:

I/O By Polling

Polling works great … but:

- We have to guarantee that our other tasks do not take too long (otherwise, we may miss the event)
- Depending on the device, “too long” may be very short

Serial I/O by Polling

```
int c;
while(1) {
  if(kbhit()) {
     // A character is available for reading
     c = getchar();
     <do something with the character>
  }
  <do something else while waiting>
}
```

With this solution, how long can "something else" take?

I/O by Polling

In practice, we typically reserve this polling approach for situations in which:

- We know the event is coming very soon
- We must respond to the event very quickly

(both are measured in nano- to micro-seconds)

Receiving Serial Data

How can we allow the “something else” to take a longer period of time?

Receiving Serial Data

How can we allow the “something else” to take a longer period of time?

- The UART implements a 1-byte buffer
- Let’s create a larger buffer…

Receiving Serial Data

Creating a larger (circular) buffer. This will be a globally-defined data structure composed of:

- N-byte memory space:

```
char buffer[BUF_SIZE];
```

- Integers that indicate the first element in the buffer and the number of elements:

```
uint8_t front, nchars;
```

Buffered Serial Data

Implementation:

- We will use an interrupt routine to transfer characters from the UART to the buffer as they become available

- Then, our main() function can remove the characters from the buffer

Interrupt Handler

```
// Called when the UART receives a byte
ISR(USART_RXC_vect) {
  // Handle the character in the UART buffer
  if(nchars == BUF_SIZE) {
     getchar();
  }else{
     uint8_t i = (front + nchars)%BUF_SIZE;
     buffer[i] = getchar();
     ++nchars;
  }
}
```

Interrupt Handler

```
// Called when the UART receives a byte
ISR(USART_RXC_vect) {
  // Handle the character in the UART buffer
  int c = getchar();

  if(nchars < BUF_SIZE) {
     buffer[(front+nchars)%BUF_SIZE] = c;
     nchars += 1;
  }
}
```

Reading Out Characters

```
// Called by a “main” program
// Get the next character from the circular buffer
int get_next_character() {
  if(nchars == 0)
      return(-1);  // No characters
  else{
      // Return the next character
      int tmp = buffer[front];
      front = (front + 1)%BUF_SIZE;
      --nchars;
      return(tmp);
  }
}
```

Last Time

- Interrupt Service Routines
- Circular buffers
  - Also known as “First In-First Out” queues
  - ISR filled the buffer as soon as serial data came in
  - Main program removed characters as needed

Today

- The shared data problem
  – Can occur when an ISR and the main program access and modify the same data structures
- Finite state machines
  – Expressing sequential behavior

Reading Out Characters

```
// Called by a "main" program
// Get the next character from the circular buffer
int get_next_character() {
   int c;
   if(nchars == 0)
       return(-1); // Error
   else {
       // Pull out the next character
       c = buffer[front];

       // Update the state of the buffer
       --nchars;
       front = (front + 1)%BUF_SIZE;
       return(c);
   }
}
```

An Updated main()

```
int c;
while(1) {
  do {
     ????

  }while(???);
  <do something else while waiting>
}
```

An Updated main()

```
int c;
while(1) {
  do {
     c = get_next_character();
     if(c != -1)
          <do something with the character>
  }while(c != -1);

  <do something else while waiting>
}
```

Buffered Serial Data

This implementation captures the essence of what we want, but there are some subtle things that we must handle ….

Buffered Serial Data

Subtle issues:

- The reading side of the code must make sure that it does not allow the buffer to overflow
  - But at least we have BUF_SIZE times more time before this happens
- We also have a shared data problem …

The Shared Data Problem

- Two independent segments of code that could access the same data structure at arbitrary times
- In our case, get_next_character() could be interrupted while it is manipulating the buffer
  - This can be very bad

Solving the Shared Data Problem

- There are segments of code that we want to execute without being interrupted

- We call these code segments **critical sections**

Solving the Shared Data Problem

There are a variety of techniques that are available:

- Clever coding
- Hardware: test-and-set instruction
- Semaphores: software layer above test-and-set
- Disabling interrupts

Solving the Shared Data Problem

There are a variety of techniques that are available:

- Clever coding ⟵
- Hardware: test-and-set instruction
- Semaphores: software layer above test-and-set
- Disabling interrupts ⟵

Disabling Interrupts

- How can we modify get_next_character()?

- It is important that the critical section be as short as possible

Assume:

- serial_receive_enable(): enable interrupt flag
- serial_receive_disable(): clear (disable) interrupt flag

Modified get_next_character()

```
int get_next_character() {
  int c;
  serial_receive_disable();
  if(nchars == 0)
      serial_receive_enable();
      return(-1); // Error
  else {
      // Pull out the next character
      c = buffer[front];
      --nchars;
      front = (front + 1)%BUF_SIZE;
      serial_receive_enable();
      return(c);
  }
}
```

Initialization Details

```
main()
{
  nchars = 0;
  front = 0;

  // Enable UART receive interrupt
  serial_receive_enable();

  // Enable global interrupts
  sei();
      :
```

Enabling/Disabling Interrupts

- Enabling/disabling interrupts allows us to ensure that a specific section of code (the critical section) cannot be interrupted
  - This allows for safe access to shared variables

- But: must not disable interrupts for a very long time

Final Note

For what we are doing:

- ISRs are not interrupted
- This means that the ISR is already a critical section