# 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
- Serial signals (what you are using for the heli)
- Analog

# I/O Systems

Many devices are operating independently of the processor – except when communication happens

- We say that these devices are acting **asynchronously** of the processor
- The processor must have some way of knowing that something has changed with the device (e.g., that it is ready to send or receive information)

# An Example:
# SICK Laser Range Finder

- Laser is scanned horizontally
- Using phase information, can infer the distance to the nearest obstacle (within a very narrow region)
- Spatial resolution: ~.5 degrees, 1 cm
- Can handle full 180 degrees at 20 Hz

# I/O By Polling

One possible approach: the processor continually checks the state of the device:

```
do {
  x = PINB & 0x10;
}while(x == 0);
y = PINC …
```

# I/O By Polling

What is wrong with this approach?

# I/O By Polling

What is wrong with this approach?

- In embedded systems, we are typically managing many devices at once

# I/O By Polling

- We can potentially be waiting for a long time before the state changes
  - We call this **busy waiting**
- The processor is wasting time that could be used to do other tasks

What is one way to solve this?

# I/O By Polling: An Alternative

Alternative: do something while we are waiting

```
do {
   x = PINB & 0x10;
  <go do something else>
}while(x == 0);
y = PINC …
```

# I/O By Polling: An Alternative

Polling works great … but:

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

# 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 typically measured in nano- to micro- seconds)

# Administrivia

- Down to one functional heli
- Some replacement parts arrive today

- By deadline: demo at least parts 1-4
  - Hand in other components
- Demo part 5 as soon as feasible
  - Once other helis are up: two day time limit
  - No more than a week

# Last Time

Counter/Timers

- Counting events: external events or clock ticks
- Prescalar divides the clock frequency (implemented as yet another counter)

I/O by Polling

# Today

An alternative to polling: interrupts

- Processor is **interrupted** from what it is doing to perform some other task
- Once done with the task, returns to what it was previously doing

# I/O By Polling: An Alternative

Alternative: do something while we are waiting

```
do {
   x = PINB & 0x10;
  <go do something else>
}while(x == 0);
y = PINC …
```

# I/O By Polling: An Alternative

Polling works great … but:

- We have to guarantee that our "something else" does not take too long (otherwise, we may miss the event)
- Depending on the device, "too long" may be very short

# An Alternative: Interrupts

- Hardware mechanism that allows some event to temporarily interrupt an ongoing task
- The processor then executes an **interrupt handler** (a small piece of code)
- Execution then continues with the original program

# Some Sources of Interrupts (Mega8)

External:

- An input pin changes state
- The UART receives a byte on a serial input

Internal:

- A clock
- Processor reset
- The on-board analog-to-digital converter completes its conversion

# Interrupts

There are many possible interrupts

- How do we know which one has occurred?
- How does the processor respond to a specific interrupt?

# Interrupts

How do we know which interrupt has occurred?

- The mega8 hardware identifies each interrupt with a unique signal

# Interrupts

The mega8 hardware identifies each interrupt with a unique signal

How does the processor respond to a specific interrupt?

- The processor stores an **interrupt table** in program memory
- Each unique signal has its own table entry

# Mega8 Interrupt Table Implementation

| addressLabels | Code | | Comments |
|---|---|---|---|
| $000 | rjmp | RESET | ; Reset Handler |
| $001 | rjmp | EXT_INT0 | ; IRQ0 Handler |
| $002 | rjmp | EXT_INT1 | ; IRQ1 Handler |
| $003 | rjmp | TIM2_COMP | ; Timer2 Compare Handler |
| $004 | rjmp | TIM2_OVF | ; Timer2 Overflow Handler |
| $005 | rjmp | TIM1_CAPT | ; Timer1 Capture Handler |
| $006 | rjmp | TIM1_COMPA | ; Timer1 CompareA Handler |
| $007 | rjmp | TIM1_COMPB | ; Timer1 CompareB Handler |
| $008 | rjmp | TIM1_OVF | ; Timer1 Overflow Handler |
| $009 | rjmp | TIM0_OVF | ; Timer0 Overflow Handler |
| $00a | rjmp | SPI_STC | ; SPI Transfer Complete Handler |
| $00b | rjmp | USART_RXC | ; USART RX Complete Handler |
| $00c | rjmp | USART_UDRE | ; UDR Empty Handler |
| $00d | rjmp | USART_TXC | ; USART TX Complete Handler |
| $00e | rjmp | ADC | ; ADC Conversion Complete Handler |
| $00f | rjmp | EE_RDY | ; EEPROM Ready Handler |
| $010 | rjmp | ANA_COMP | ; Analog Comparator Handler |
| $011 | rjmp | TWSI | ; Two-wire Serial Interface |

# Mega8 Interrupt Table Implementation

Address in the program memory

| address | Labels | Code | | Comments |
|---|---|---|---|---|
| $000 | | rjmp | RESET | ; Reset Handler |
| $001 | | rjmp | EXT_INT0 | ; IRQ0 Handler |
| $002 | | rjmp | EXT_INT1 | ; IRQ1 Handler |
| $003 | | rjmp | TIM2_COMP | ; Timer2 Compare Handler |
| $004 | | rjmp | TIM2_OVF | ; Timer2 Overflow Handler |
| $005 | | rjmp | TIM1_CAPT | ; Timer1 Capture Handler |
| $006 | | rjmp | TIM1_COMPA | ; Timer1 CompareA Handler |
| $007 | | rjmp | TIM1_COMPB | ; Timer1 CompareB Handler |
| $008 | | rjmp | TIM1_OVF | ; Timer1 Overflow Handler |
| $009 | | rjmp | TIM0_OVF | ; Timer0 Overflow Handler |
| $00a | | rjmp | SPI_STC | ; SPI Transfer Complete Handler |
| $00b | | rjmp | USART_RXC | ; USART RX Complete Handler |
| $00c | | rjmp | USART_UDRE | ; UDR Empty Handler |
| $00d | | rjmp | USART_TXC | ; USART TX Complete Handler |
| $00e | | rjmp | ADC | ; ADC Conversion Complete Handler |
| $00f | | rjmp | EE_RDY | ; EEPROM Ready Handler |
| $010 | | rjmp | ANA_COMP | ; Analog Comparator Handler |
| $011 | | rjmp | TWSI | ; Two-wire Serial Interface |

# Mega8 Interrupt Table Implementation

Change program counter to the location identified by "EXT_INT1"

```
addressLabels Code                   Comments
$000          rjmp   RESET           ; Reset Handler
$001          rjmp   EXT_INT0        ; IRQ0 Handler
$002          rjmp   EXT_INT1        ; IRQ1 Handler
$003          rjmp   TIM2_COMP       ; Timer2 Compare Handler
$004          rjmp   TIM2_OVF        ; Timer2 Overflow Handler
$005          rjmp   TIM1_CAPT       ; Timer1 Capture Handler
$006          rjmp   TIM1_COMPA      ; Timer1 CompareA Handler
$007          rjmp   TIM1_COMPB      ; Timer1 CompareB Handler
$008          rjmp   TIM1_OVF        ; Timer1 Overflow Handler
$009          rjmp   TIM0_OVF        ; Timer0 Overflow Handler
$00a          rjmp   SPI_STC         ; SPI Transfer Complete Handler
$00b          rjmp   USART_RXC       ; USART RX Complete Handler
$00c          rjmp   USART_UDRE      ; UDR Empty Handler
$00d          rjmp   USART_TXC       ; USART TX Complete Handler
$00e          rjmp   ADC             ; ADC Conversion Complete Handler
$00f          rjmp   EE_RDY          ; EEPROM Ready Handler
$010          rjmp   ANA_COMP        ; Analog Comparator Handler
$011          rjmp   TWSI            ; Two-wire Serial Interface
```

# Interrupt Example

Suppose we are executing the "something else" code:

LDS R1 (A) ← **PC**

LDS R2 (B)

CP R2, R1

BRGE 3

LDS R3 (D)

ADD R3, R1

STS (D), R3

# An Example

Suppose we are executing the "something else" code:

```
LDS R1 (A)
LDS R2 (B)  <- PC
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3
```

# An Example

Suppose we are executing the "something else" code:

LDS R1 (A)

LDS R2 (B)

CP R2, R1 ← **PC**

BRGE 3

LDS R3 (D)

ADD R3, R1

STS (D), R3

# An Example

An interrupt occurs (EXT_INT1):

LDS R1 (A)

LDS R2 (B)

CP R2, R1 ← **PC**

BRGE 3

LDS R3 (D)

ADD R3, R1

STS (D), R3

# An Example

An interrupt occurs (EXT_INT1):

LDS R1 (A)

LDS R2 (B)

CP R2, R1 → rjmp EXT_INT1 ← **PC**

BRGE 3

LDS R3 (D)

ADD R3, R1

STS (D), R3

# An Example

An interrupt occurs (EXT_INT1):

LDS R1 (A)

LDS R2 (B)

CP R2, R1

rjmp EXT_INT1 ← PC

BRGE 3

remember this location

LDS R3 (D)

ADD R3, R1

STS (D), R3

# An Example

Execute the interrupt handler

LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3

rjmp EXT_INT1

PC

EXT_INT1:

LDS R1 (G)
LDS R5 (L)
ADD R1, R2
:
RETI

# An Example

Execute the interrupt handler

LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3

EXT_INT1:

LDS R1 (G)
PC → LDS R5 (L)
ADD R1, R2
:
RETI

# An Example

Execute the interrupt handler

LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3

EXT_INT1:

LDS R1 (G)
LDS R5 (L)
PC → ADD R1, R2
:
RETI

# An Example

Execute the interrupt handler

LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3

EXT_INT1:

LDS R1 (G)
LDS R5 (L)
PC → ADD R1, R2
:
RETI

# An Example

Return from interrupt

LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3

EXT_INT1:

LDS R1 (G)
LDS R5 (L)
ADD R1, R2
:
PC → RETI

# An Example

Return from interrupt

LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3 PC
LDS R3 (D)
ADD R3, R1
STS (D), R3

EXT_INT1:

LDS R1 (G)
LDS R5 (L)
ADD R1, R2
:
RETI

# An Example

Continue execution with original

LDS R1 (A)

LDS R2 (B)

CP R2, R1

BRGE 3

LDS R3 (D) ← **PC**

ADD R3, R1

STS (D), R3

EXT_INT1:

LDS R1 (G)

LDS R5 (L)

ADD R1, R2

:

RETI

# An Example

Continue execution with original

LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1 ← **PC**
STS (D), R3

EXT_INT1:

LDS R1 (G)
LDS R5 (L)
ADD R1, R2
:
RETI

# Interrupt Routines

- Generally a very small number of instructions
  - We want a quick response so the processor can return to what it was originally doing
- Register use
  - If the interrupt routine makes use of registers, then it must restore their state before returning
  - We accomplish this through the use of a special data structure called a **stack**

# Divert to interrupts and timer/counters

# Last Time

- Interrupts
  - Response to external or internal event
  - Temporarily halt the execution of the main program to execute an event-related section of code
- Timer/counters and interrupts
  - Interrupts at regular intervals
  - Useful for control and timing
  - Pulse width modulation

# Configuring Timer/Counter Interrupts

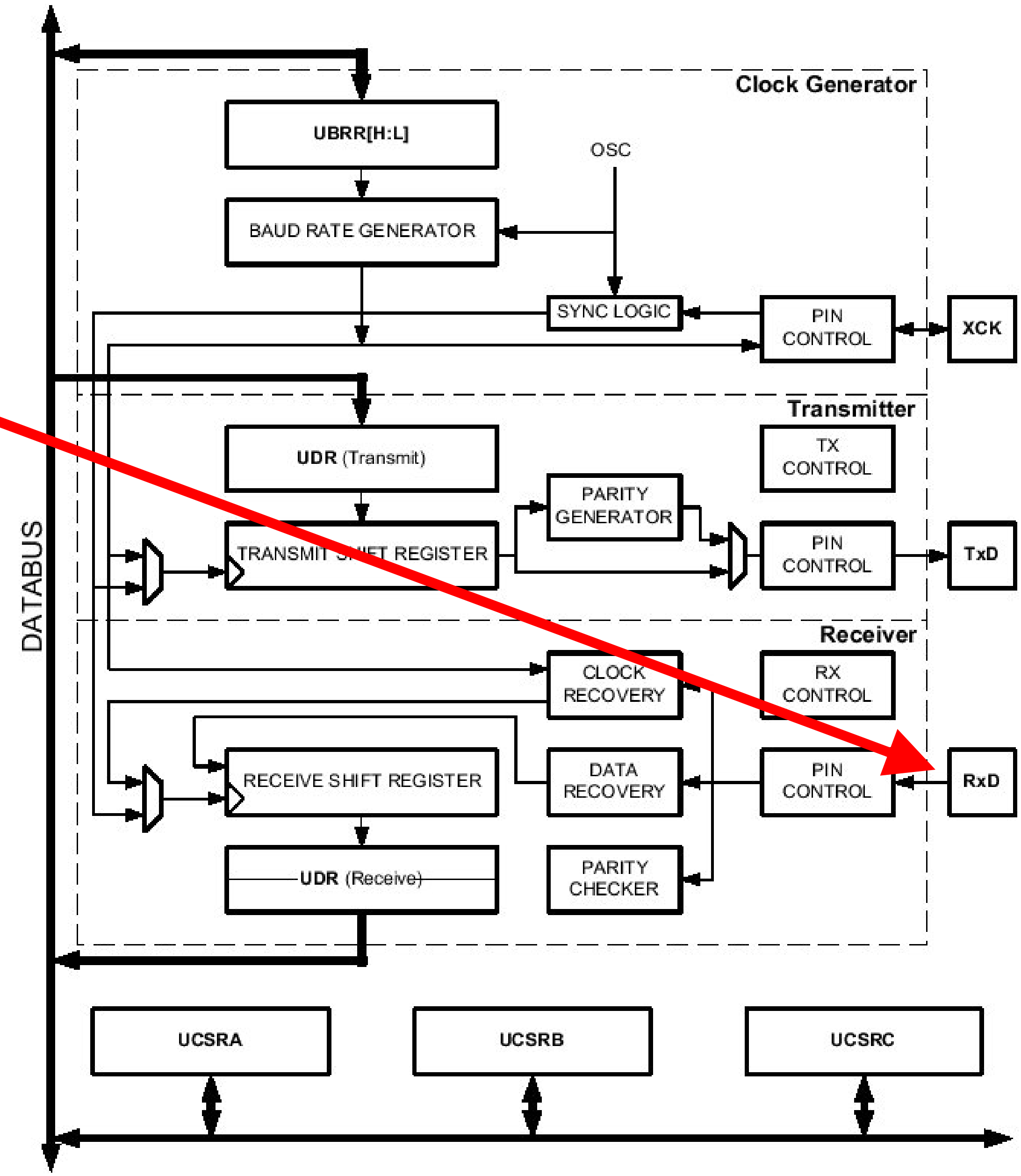
- Set prescalar: divide main clock frequency
- Provide an interrupt service routine (ISR)
- Enable the specific interrupt
  - In our examples, we use timer0_enable()
- Enable global interrupts
  - sei()
  - Turns on all interrupts

# Today

- ISR example:
  - Processing serial input
- Finite state machines (FSMs)
  - Representing temporal behavior

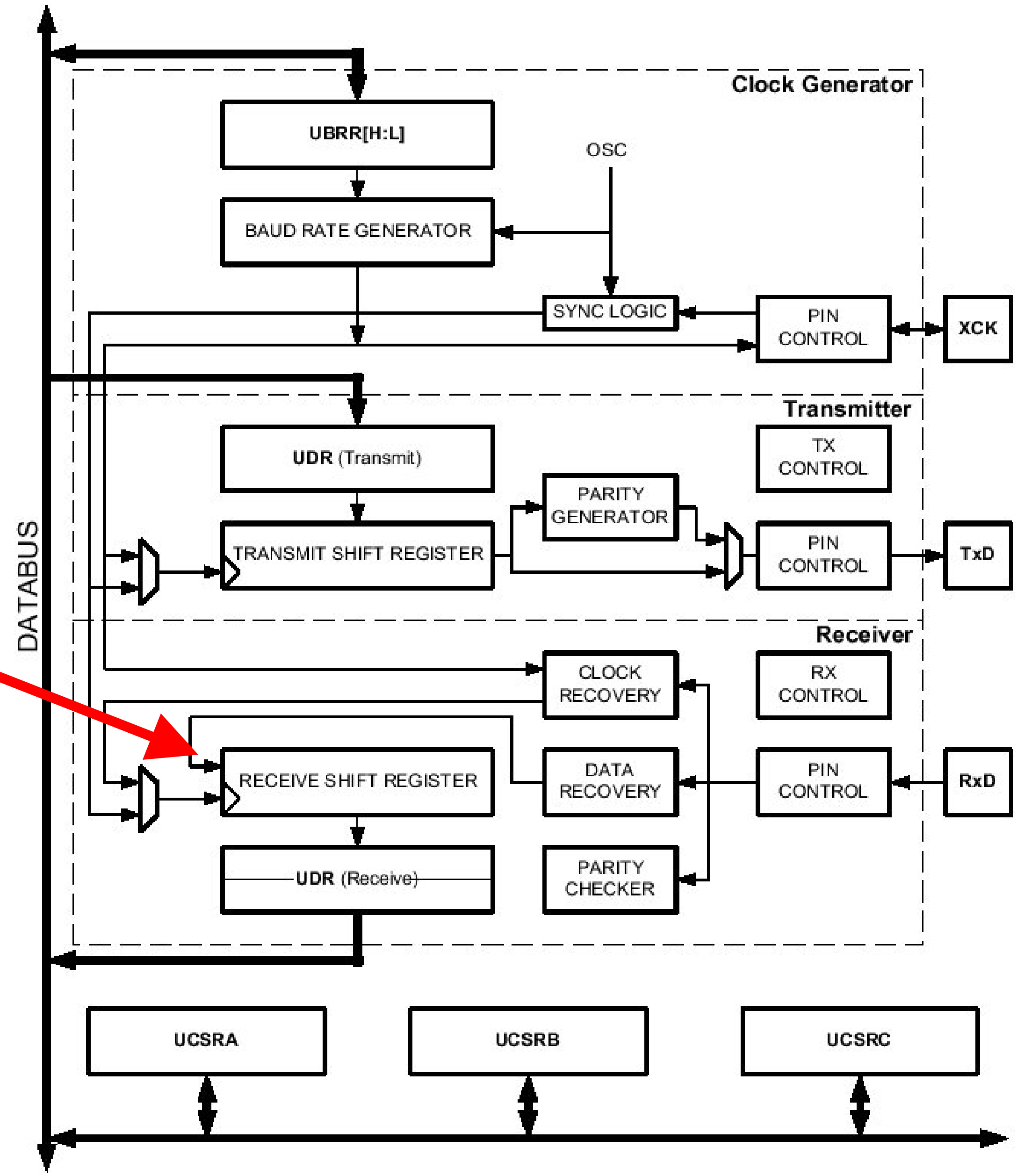
# Receive

- Receive pin (PD0)



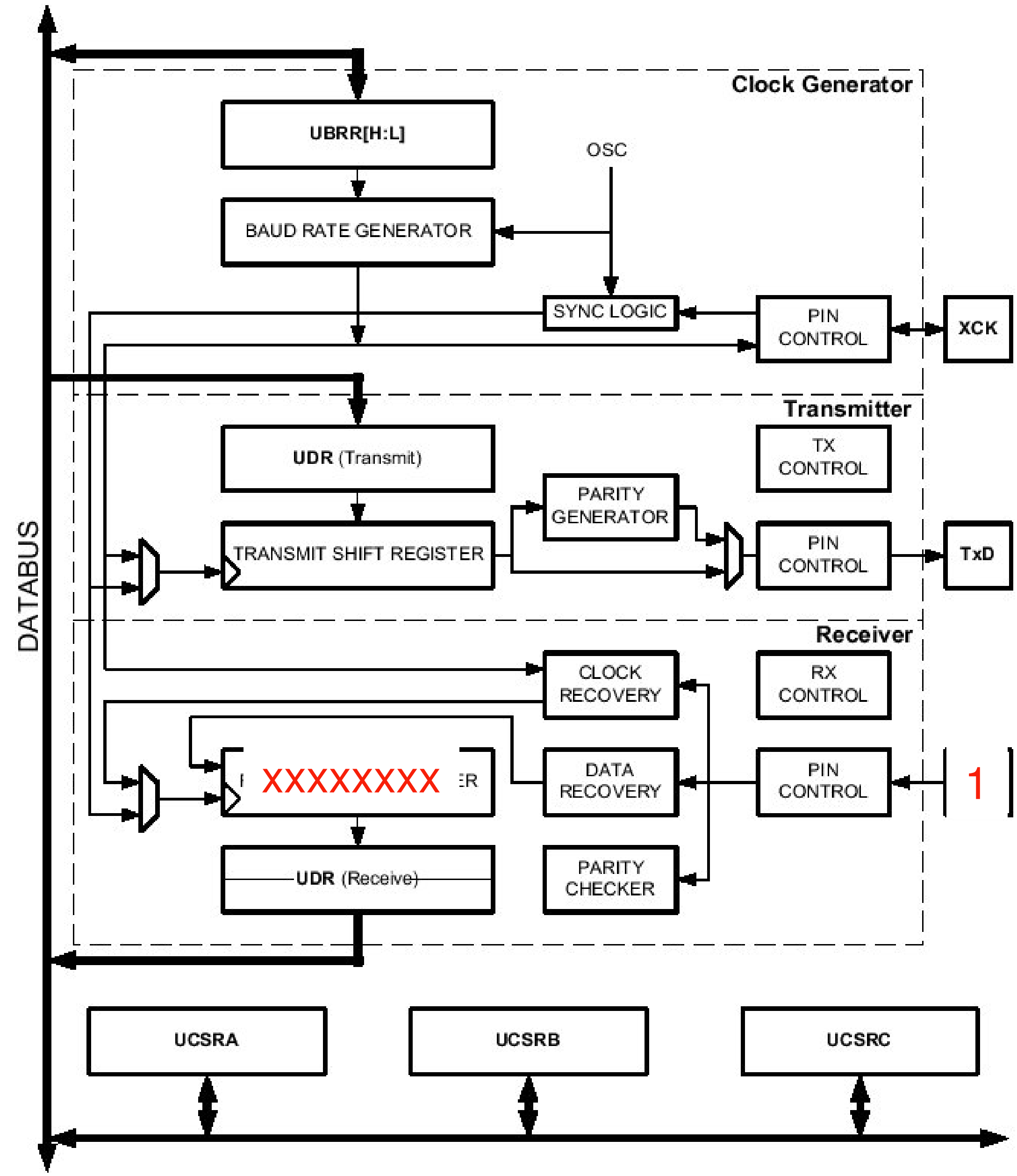
# Receive

- Receive pin (PD0)
- Receive shift register



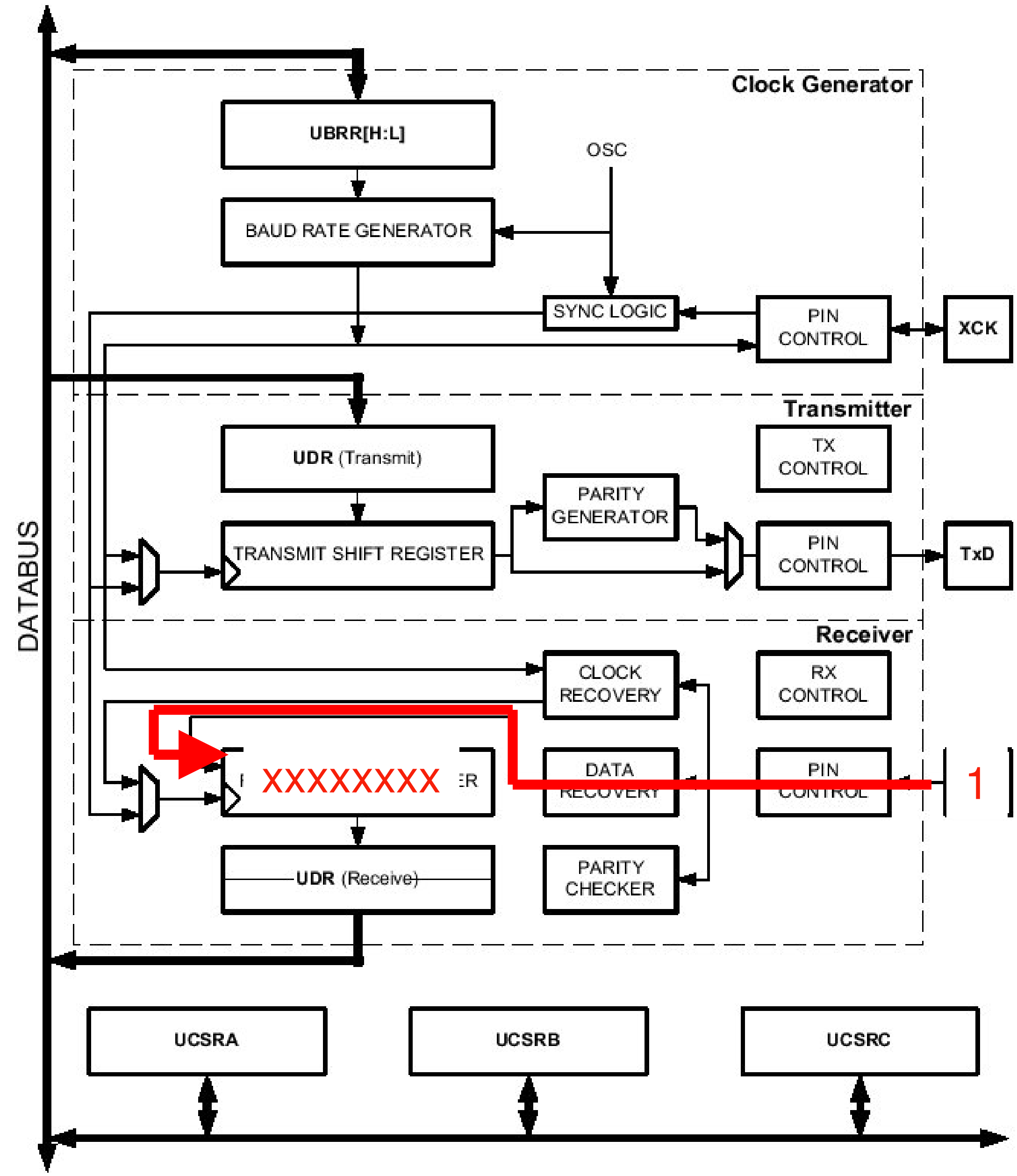
# Receive

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



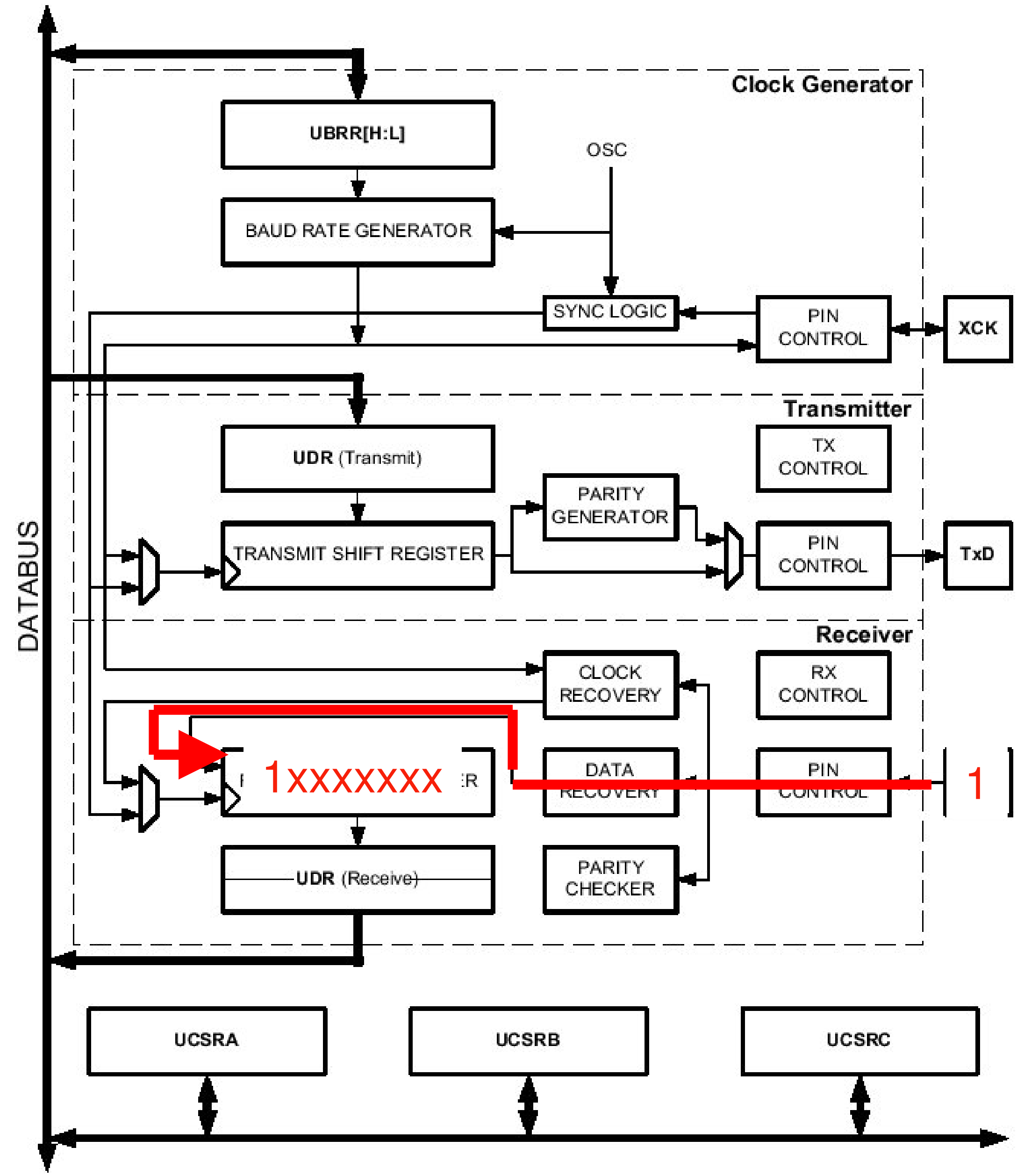
# Receive

- "1" is presented to the shift register



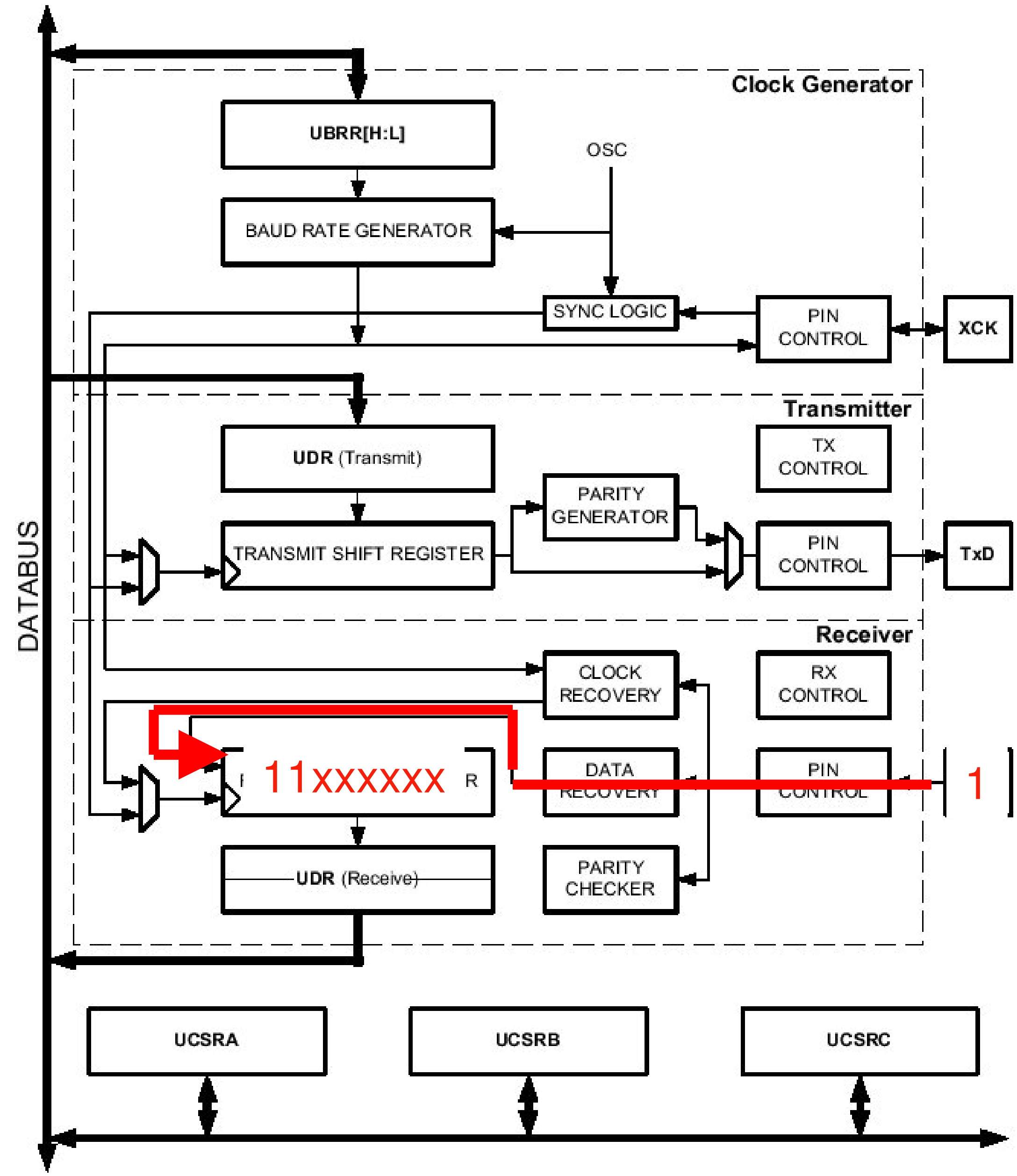
# Receive

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



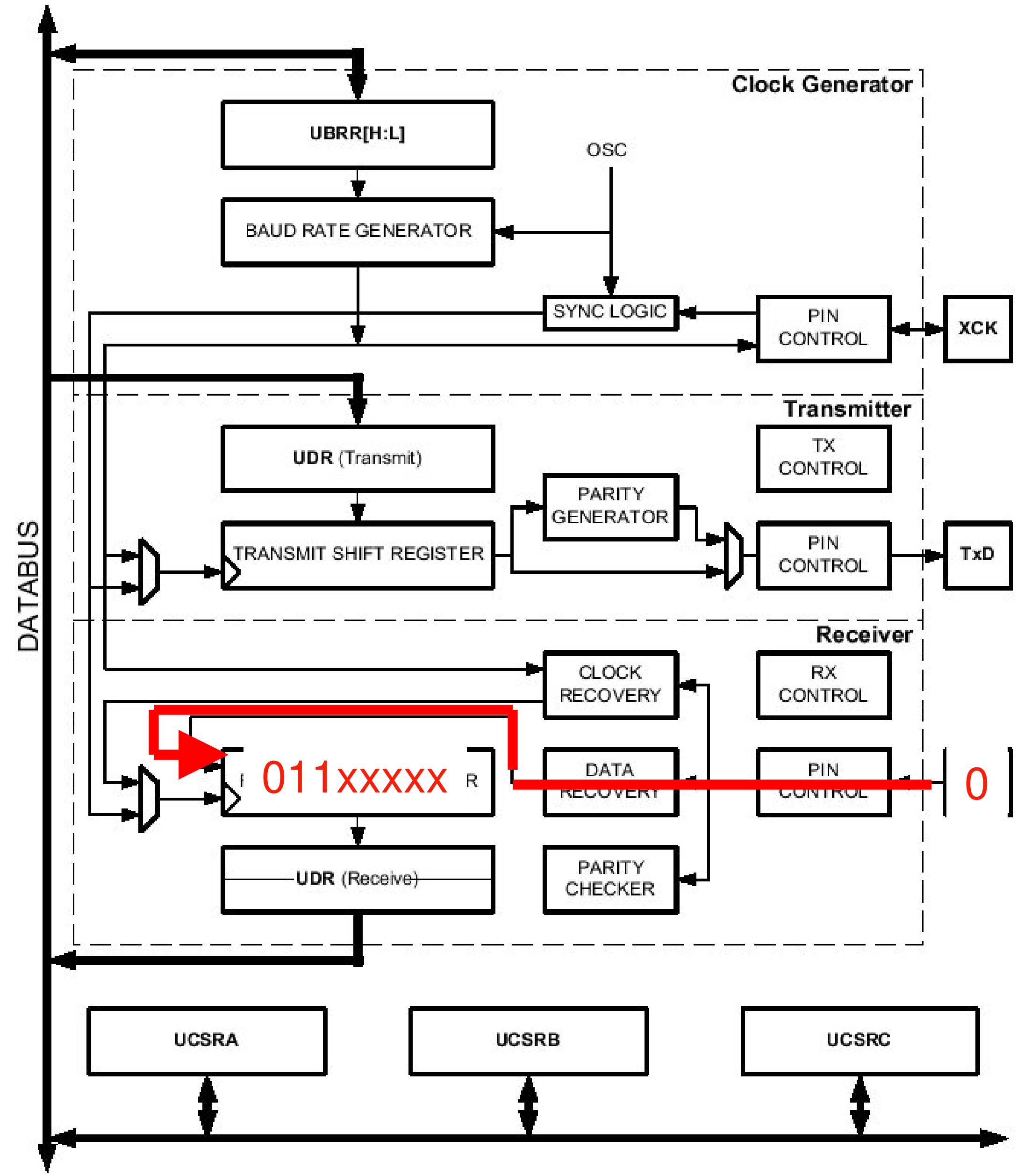
# Receive

Next bit is shifted in



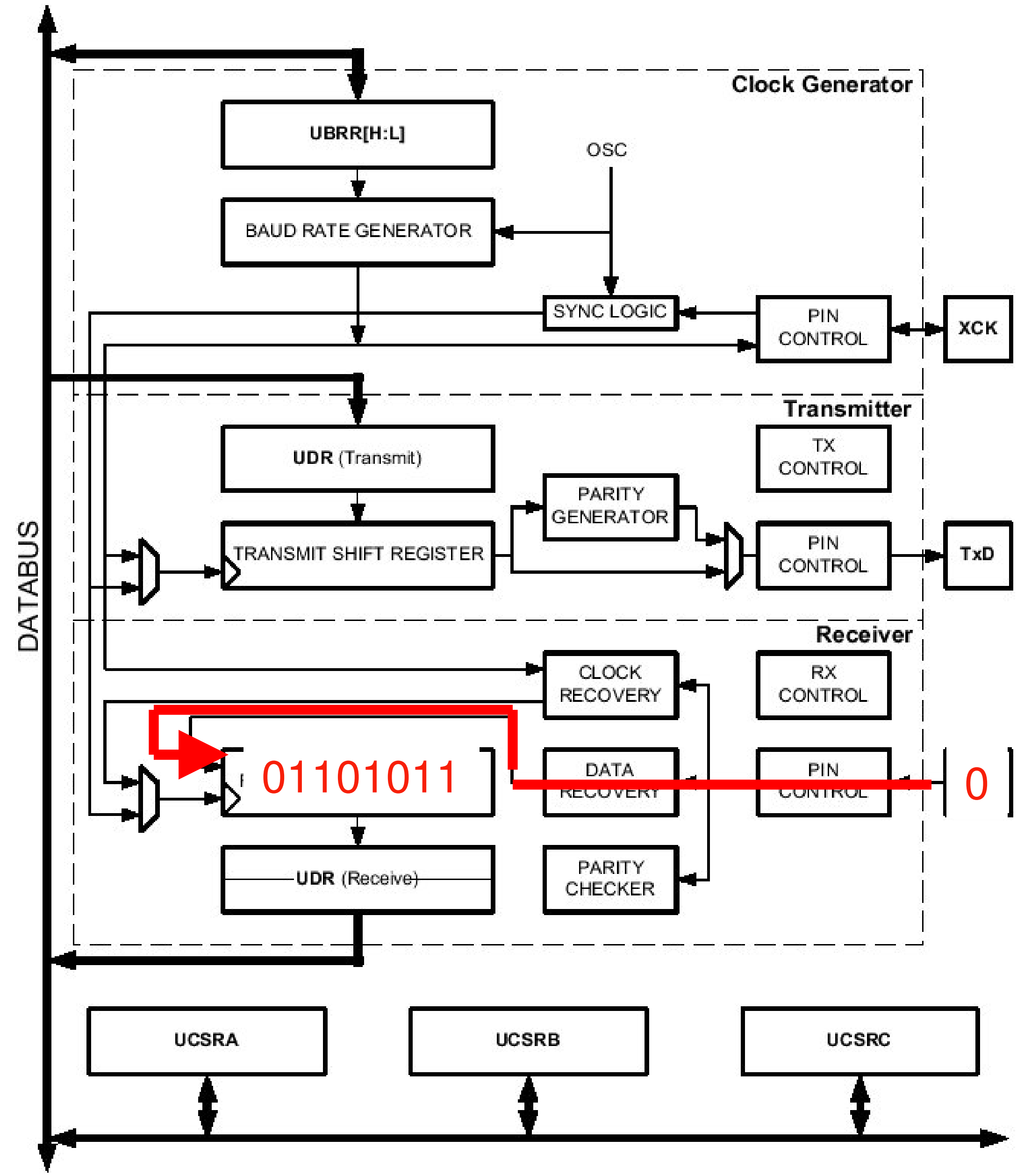
# Receive

And the next bit…



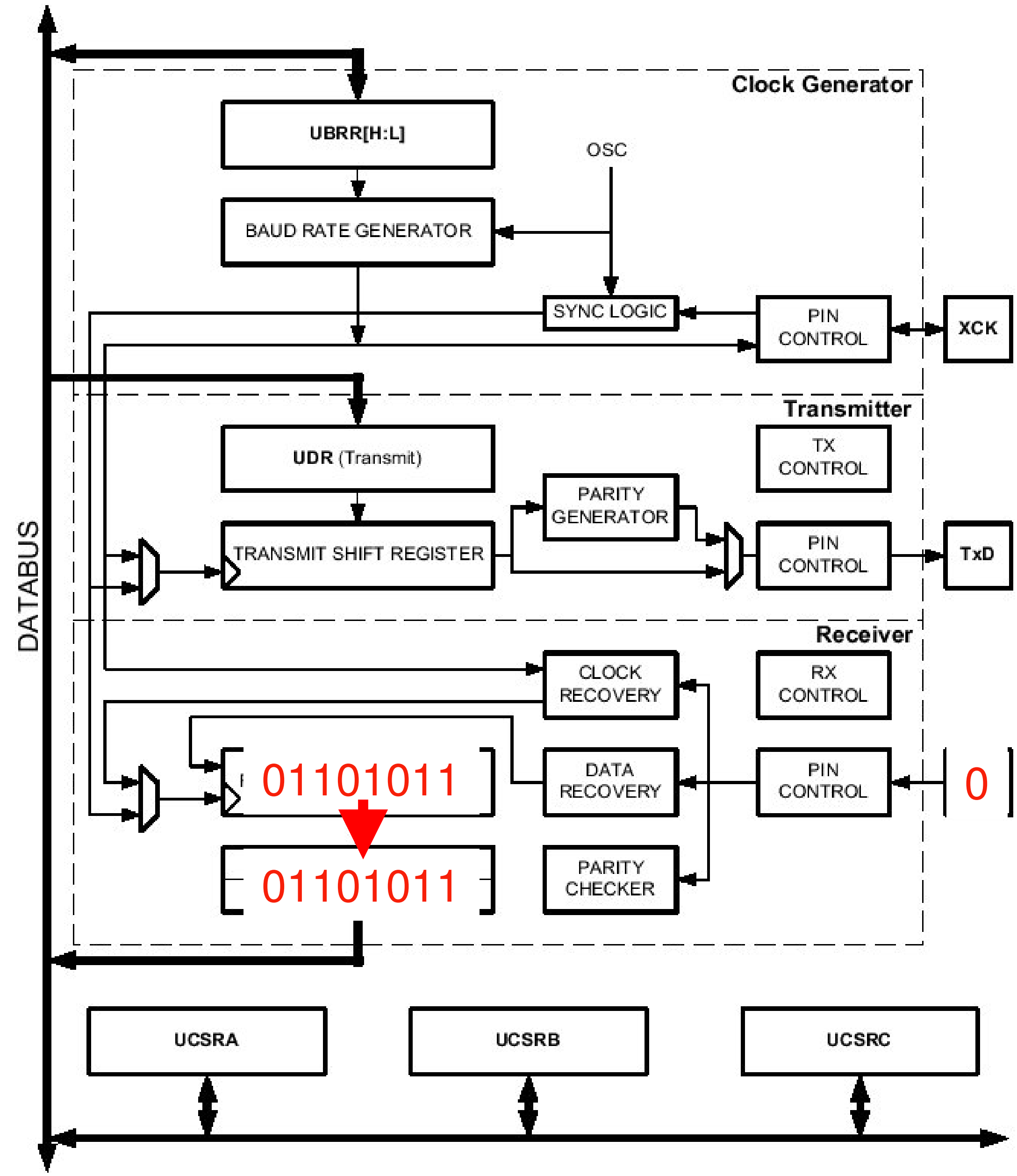
# Receive

And the 8th bit



# Receive

Completed byte is stored in the UART buffer



# Back to Receiving Serial Data…

```
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?

# 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 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:

```
int 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(UART_RECV_vect) {




}
```

# Interrupt Handler

```
volatile char buffer[BUF_SIZE];
volatile uint8_t front;
volatile uint8_t nchars;

// Called when the UART receives a byte
ISR(UART_RECV_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() {




}
```

# 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
- We 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
- Disabling interrupts
- … and others

# Disabling Interrupts

- How can we modify get_next_character()?

- The 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 interrups
  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

# Enabling/Disabling Interrupts

Depending on the problem you are solving, you can either:

- Enable/disable global interrupts
- Enable/disable just one of the interrupts
  - Typical if only one will do

# Timer 0 Interrupt

- Enable: `timer0_enable`
- Disable: `timer0_disable`

Similar functions for timers 1 and 2