**Embedded systems** are product sub-systems controlled by a special-purpose computer. The computer provides software-control by using transducers called actuators and sensors.

- **Sensors** are transducers that convert physical energy to electrical energy. In general, sensors ***meter*** the environment by measuring some change in a physical parameter.
- **Actuators** are transducers that convert electrical energy to physical energy. In general, actuators ***change*** the environment by using electricity to change a physical parameter.
- **Microcontrollers** are single-chip computers that are the brain of ***most*** embedded systems.
- **Figure 1** is a basic model used to diagram embedded systems that use microcontrollers.

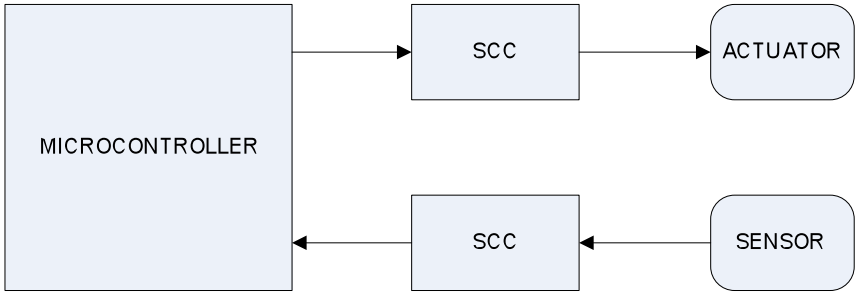


**Figure 1**: The Basic Embedded System Model

**Pins** interface sensors and actuators to the microcontroller.

- **Ports** are named collections of pins for easy programmatic reference.
- **Input-only** port pins are only used for sensor signals.
- **Output-only** port pins are only used for actuator signals.
- **Bidirectional** port pins can be used for either sensor or actuator signals.
- **Pins** can be shared by a number of I/O devices to keep the size of the chip small.

**I/O control registers** hold binary numbers that determine port pin behavior. Each program will use different types of control registers based on the system functionality and the I/O devices used.

- **Configuration control registers** configure I/O devices that use port pins. Examples include clock timing, interrupt masks, and pin edge response.
- **Direction control registers** control the direction of bidirectional port pins.
- **Input registers** sample sensor signals on command.
- **Output registers** control the voltage output to actuator signals.
- **Parameter registers** hold parameters needed by I/O device functions. Example include time values for timer functions and voltage values for analog comparators,
- **Result registers** hold results produced by some I/O devices.
- **Status registers** flag events for software notice.

The **Atmel ATmega32** 8-bit microcontroller has a pin interface summarized in Table 1.

- **Discrete digital I/O** is available on all port pins.
- **On-chip I/O devices** share the port pins.

| TABLE 1: ATmega32 PORT PINS | | |
|---|---|---|
| **PORT NAME** | **PINS AND DIRECTIONS** | **SHARED BY THESE I/O DEVICES** |
| PORTA | 8 bidirectional pins | Analog-to-digital converter |
| PORTB | 8 bidirectional pins | SPI serial, analog comparator, timer, external interrupts |
| PORTC | 8 bidirectional pins | Timer, JTAG in-system programming, two-wire serial |
| PORTD | 8 bidirectional pins | Timer, external interrupts, UART serial |

The **Atmel ATmega32** serial communications devices include a standard universal synchronous/asynchronous receiver transmitter (USART) that is ready for interfacing to RS232 serial devices. Additionally, the microcontroller provides two other standard serial interfaces: the synchronous peripheral interface (SPI) and a two-wire interface (TWI). All of these protocols are widely used in embedded systems. This document focuses on the USART.

A **USART** is a standardized device that can:

- transmit a binary number as a square waveform to a receiver
- receive a square waveform and converting the information to a binary number
- use a synchronous communications model where a master clock it transmitted to the receiver to coordinate data bit sampling through time
- use an asynchronous communications model where no clock is transmitted to the receiver
- use built-in hardware to synchronize to an incoming signal in asynchronous mode
- transmit and receive at the same time (full-duplex)
- use interrupts to signal transmit or receive complete

An **RS232** interface IC is often connected to the digital USART signal to convert the USART 0V and 5V signals to RS232 serial levels used by many peripheral devices and standard personal computers.

- RS232 logic 0 is any voltage between 3V and 15V
- RS232 logic 1 is any voltage between -3V and -15V
- The actual voltage depends on the power supply. Common voltages are ±5, and ±12V.
- The large voltage swing of RS232 makes it not attractive in low-power design. The result is that RS232 serial ports have largely disappeared from laptops and handheld devices. Instead, low-power alternatives such as USB or Firewire.
- **Research** your laboratory board. Does it contain an RS232 interface IC? If so, what is the chip called?

The **ATmega32 USART** is controlled through

- **UCSRA** is mainly a **status register** where the USART announces the completion of events for polled interaction
- **UCSRB** is a **control register** where interrupts are enabled and the data size is set
- **UCSRC** is a **control register** where the communications model is set: synchronicity, start/stop bit control, parity control, and data size. USARTs only communicate correctly if both sides use the same synchronicity, start/stop bits, parity, baud, and data size. **Note** that the default value of this register implements a common model – and thus often engineers will not even program a new value into UCSRC.
- **UDR** is the name given to both the transmit register and the receive register. It is a convenience for the programmer because it allows the programmer to simply think of **USART data register** rather than **USART transmit register** or **USART receive register**. A read from UDR retrieves the last received binary number. A write to the UDR places a binary number into the transmit register and initiates transmission.
- **UBRRH** and **UBRRL** are parameter registers that hold the desired baud rate. Baud is the unit for signal elements per second. The value of the parameter is calculated using the equation:

  $$UBRR = [ fosc / (16*BAUD) ] - 1$$

  where *fosc* is the system clock frequency and BAUD is the desired baud rate. The calculated value is truncated (integer division). **For example,** UBRR should have the number 51 stored in order to set a 9600 BAUD data rate on an ATmega32 that uses an 8MHz crystal.

**Operation** of the USART is straightforward regardless of the selected communication model (synchronous or asynchronous) and the service mode (polled or interrupts). **Refer** to the ATmega32 data sheets for a thorough discussion of the control registers.

- **Initialize** the control registers for the desired frame format (start/stop, parity, data size) and baud rate.
- **Turn on** the receiver and transmitter using the RXEN and TXEN bits.
- **Interrupts** can be used to service arrival of data and completion of transmission. Interrupts are enabled in the control registers using the RXCIE, TXCIE, and UDRIE control bits. **However**, the ATMON monitor OS used by MSOE computer engineering students uses USART interrupts to communicate through the serial cable. Thus, interrupts should not be used when ATMON is operating in the background. ***Use polled mode with ATMON***. This does not apply to biomedical engineering or electrical engineering student laboratory boards.
- **Polled mode** can be used to monitor the receive complete (RXC) status flag to determine if new data has arrived.
- **Polled mode** can be used to monitor the UDR empty (UDRE) status flag to know when a transmission is complete. This step is **vital** because a user program cannot place a new number in the UDR until the last number has been fully transmitted. **Remember** that 9600 baud is quite a bit slower than a program looping at 16MHz!

- **Asynchronous mode** requires minimal configuration and operates correctly with the terminal applications on personal computers.

**C Programming Example**

```
// ECHO.C
//
// this program configures asynchronous
// data transfer at 2400 baud
// and echos any received character
// back down to the cable

#include <avr\io.h>

// using defines at the top of the file to
// make a one-stop place to quickly change
// oscillator frequency or BAUD_RATE rather
// that hard-coding it in main
//
// let a configuration subroutine in a device
// driver or let main calculate the UBRR value
//
// for most baud rates the equation produces
// and 8-bit value so UBRRH should be zeroed.

#define FOSC 8E6
#define BAUD 24E2

void main(void)
{
  // declare auto local variables
  char USARTdata = 0;

  // configure the baud rate
  UBRRH = 0;
  UBRRL = ( FOSC / 16*BAUD ) - 1;

  // configure asynchronous, even parity,
  // 1 stop bit, 8-bit characters
  // careful with UCSRC - you must put a 1 in
  // the MSB when writing to the UCSRC

  UCSRC = 0b10100011;

  // turn on the transmitter and the receiver
  // using predefined bit names and OR to make a
  // highly readable program (to some people)
  // just to illustrate another style

  UCSRB = (1 << TXEN) | (1 << RXEN);


  // infinite loop:  wait for character then retransmit it
  while(1)
  {
```

 

```
    // check RXC for receive-complete
    // this example uses an AND mask where
    // a 1 has been shifted into the RXC bit position
    // all other bits in the MASK are 0 because 1 is 0b00000001
    // and after the shift 0b10000000

    if(UCSRA & (1<<RXC))
    {
      // must read the data out of UDR to extract it and clear RXC

      USARTdata = UDR;
    }

    // if something has been received send it back
    if(USARTdata)
    {
      // make sure the data register is not transmitting currently
      // so that we don't overwhelm it
      if(UCSRA & (1<<UDRE))
      {
        UDR = USARTdata; // echo back
        USARTdata = 0;   // clear the local variable so that no character left
      }
    }

// note that this example illustrates using if-else but while could also have
// been used if the code is allowed to block the main loop

}
```

Assembly language programmers should note that the C code given above reads well if you consider assignment, if-then-else, and while programmatically and disregard the other parts that may be less clear if you have not learned C. Note that assembly language programmers will likely implement the flag checks using sbic or sbis instructions.