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
• Analog
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
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 2\textsuperscript{nd} 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 sides 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
Frame-Level Error Detection

- Due to timing and noise problems, the receiver may not receive the correct data
- We would like to catch these errors as early in the process as possible
- The first line of defense: include extra bits in the data frame that can be used for error detection and/or correction
  - This can also be done by our UART
Frame-Level Error Detection

Parity bit: indicates whether there is an odd or even number of 0s in the byte

- Transmitter computes the parity bit and includes it in the data frame
- Receiver also computes parity of the received byte
- If the two do not match, then an error is raised
  - How the error is dealt with is determined by the meta-level protocol
Frame-Level Error Correction

• When we use a single parity bit, we assume that in the worst case, a single bit is corrupted
• But: we can be more sophisticated about catching errors if we transmit more bits (at the cost of a larger data frame)
• Instead, we tend to do these types of checks on a set of bytes: “checksum”s
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 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

But: “Low” is 0v and “High” is +5V
Mega8 UART
Mega8 UART

- Transmit pin (PD1)
Mega8 UART

- Transmit pin (PD1)
- Transmit shift register
Mega8 UART

- Receive pin (PD0)
Mega8 UART

- Receive pin (PD0)
- Receive shift register
Mega8 UART C Interface

Lib C support (standard C):
getchar(): receive a character
putchar(‘a’): put a character out to the port
puts(“foobar”): put a string out to the port
printf(“foobar %d %s”, 45, “baz”): put a formatted string out to the port (not recommended for the atmels)
Mega8 UART C Interface

OUlib support:

```c
fp = serial_init_buffered(0, 9600, 10, 10)
    Initialize the port @9600 bits per second (input and output
    buffers are both 10 characters long)

serial_buffered_input_waiting(fp)
    Is there a character in the buffer?
```

See the Atmel HOWTO: examples/serial
Summary: Using OUIlib + LibC

• At the top of your source file:

```c
#include "oulib_serial_buffered.h"
```

• Initialization (in your main() function):

```c
fp = serial_init_buffered(0, 9600, 10, 10)
sei();
```

• Getting a character:

```c
char c;
C = getchar();
```

• Sending a character:

```c
putchar(’f’);
```
Mega8 UART

putchar('c')
putchar('c')
- 'c' placed onto the data bus and written to UDR
Mega8 UART

putchar(‘c’)

• ‘c’ placed onto the data bus and written to UDR

• When TSR is ready, ‘c’ is copied from UDR to TSR
Mega8 UART

`putchar('c')`

- ‘c’ placed onto the data bus and written to UDR
- When TSR is ready, ‘c’ is copied from UDR to TSR
- TSR shifts bits out to pin sequentially
Mega8 UART C Interface

`printf()`: formatted output
`scanf()`: formatted input

See the LibC documentation or the AVR C textbook
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

```c
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:
  ```c
  volatile uint8_t buffer[BUF_SIZE];
  ```
- Integers that indicate the first element in the buffer and the number of elements:
  ```c
  volatile 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
c    c = getchar();
}
}
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
int16_t get_next_character() {
    int16_t c;

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

    <do something else while waiting>
}

<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
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()
{
    serial0_init(9600);
    nchars = 0;
    front = 0;

    // Enable UART receive interrupt
    serial_receive_enable();

    // Enable global interrupts
    sei();
};
Enable/Disable Serial Interrupt

One bit of UCSRB determines whether the serial receive interrupt is enabled or disabled. Here is the code:

```c
inline void serial_receive_enable(void) {
    UCSRB |= _BV(RXCIE);  // Enable serial receive interrupt
}

inline void serial_receive_disable(void) {
    UCSRB &= ~_BV(RXCIE); // Disable receive interrupt
}
```
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
Alternative Solutions

• There is a clever change to the data structure that does not require enabling/disabling interrupts

• Also possible to put this buffer “behind” getchar() (and hence all other input functions)
  – See serial0_init() in OUlib to see how the hardware specific implementation is hooked into the general LIBC functions