Debugging

- Debugging
- Debug environments
- Debug via serial

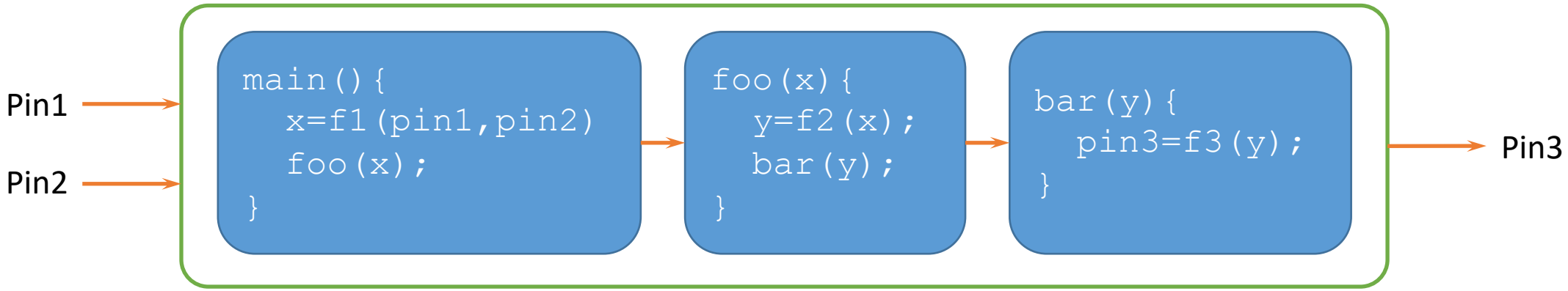
Debugging

Debug and trace

- Controllability and observability are required
- Controllability
  - Ability to control sources of data used by the system
  - Input pins, input interfaces (serial, Ethernet, etc.)
  - Registers and internal memory
- Observability
  - Ability to observe intermediate and final results
  - Output pins, output interfaces
  - Registers and internal memory

I/O access is insufficient

- Observation of I/O is not enough to debug



- If Pin2 is incorrect, how do we locate the bug?

Properties of a debugging environment

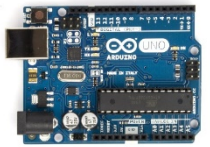
- Run control of the target
  - Start and stop the program execution
  - Observe data at stop points
- Real-time monitoring of target execution
  - Non-intrusive in terms of performance
- Timing and functional accuracy
  - Debugged system should ac like the real system

Debug environments

Remote debugger

- Frontend running on the host
- Debug monitor hidden on target
  - Typically triggered when debug events occur
  - Hitting a breakpoint, receiving request from host, etc.
- Debug monitor maintains communication link

Communication link



Host

Target

Remote debug tradeoffs

- Advantages
  - Good run control using breakpoints to stop execution
  - Debug monitor can alter memory and registers
  - Perfect functional accuracy
- Disadvantages
  - Debug interrupts alter timing so real-time monitoring is not possible
  - Need a spare communication channel
  - Need program in RAM to add breakpoints

Embedded debug interfaces

- Many modern processors include embedded debug logic
  - Embedded trace macrocell (ARM)
  - Background debug mode (Freescale)
- Debug logic permanently build into the processor
- A few dedicated debug pins are added

Debug and trace features

- Breakpoints, stopping points in the code
- Watchpoints, memory locations which trigger stop
- On-the-fly memory access
- Examine/change internal processor values
- Single-step through the code
- Export softeware-generated data (printf)
- Timestamp information for each event
- Instruction trace (special purpose HW needed)

Debug via serial

Serial protocols

- Data is transmitted serially
  - Only 1 bit needed (plus common ground)
- Parallel data transmitted serially
- Original bytes/words regrouped by the receiver
- Many protocols are serial to reduce pin usage
  - Pins are precious

UART

- Universal asynchronous receiver/transmitter
- Used for serial communication between devices
- UART is asynchronous; no shared clock
- Asynchronous allows longer distance communication
  - Clock skew is not a problem

UART applications

- Used by modems to communicate with network
- Computers used to have an RS232 port, standard
- Not well used anymore, outside of embedded systems
  - Replaced by USB, Ethernet, etc.
- Simple, low HW overhead
- Build into most microcontrollers

Serial on Arduino

Arduino serial communication

- UART protocol used over the USB cable
- Initialize by using `Serial.begin()`
- `Serial.begin(speed)` or `Serial.begin(speed,config)`
  - `speed` is the baud rate
  - `config` sets the data bits, parity, and stop bits
  - `Serial.begin(9600)`
  - `Serial.begin(9600, SERIAL_8N1)`
    - 8 data bits, no parity bit, and 1 stop bit
- Usually call `Serial.begin()` in the setup function

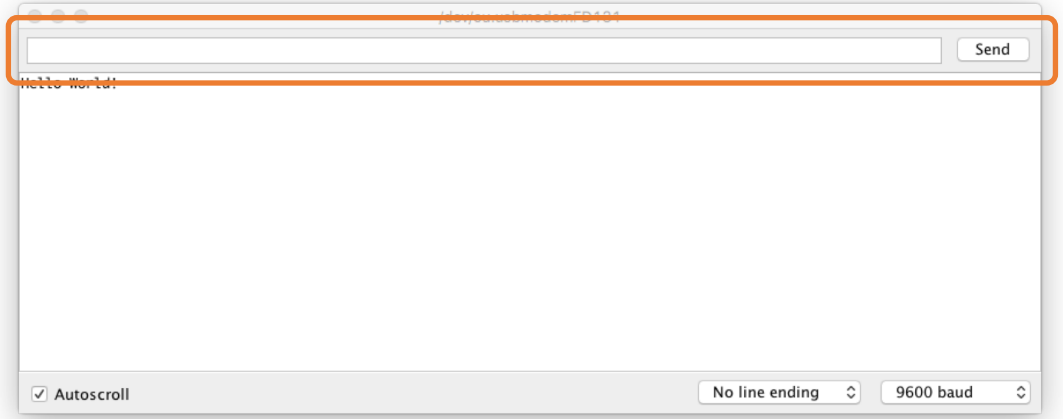
Sending text over serial

- Use `Serial.print()` or `Serial.println()` to print text in the serial monitor
  - Strings are converted to ASCII and sent using UART

```
Serial.println("Hello World!");
```

Reading data over serial

- Data can be sent to the Arduino via the serial monitor



- When data is sent, it goes into a buffer in the Arduino until it is read
- `Serial.available()` is used to see how many bytes are waiting in the buffer

```
int byteNum = Serial.available();
```

Serial.read()

- Returns 1 byte from the serial buffer

```
int bval = Serial.read();
```

- Returns −1 if no data is available
- Serial.readBytes() writes several bytes into a buffer

```
char buff[10];
Serial.readBytes(buff,10);
```