UART Communications Utility

ECE 3005 Spring 2021
GT Offroad
Georgia Institute of Technology
Akash Harapanahalli

Contents

Overview

What is the Communications Utility?

A new set of libraries built to facilitate communication over UART between a mesh network of directly wired (UART) or wireless (XBEE) units.

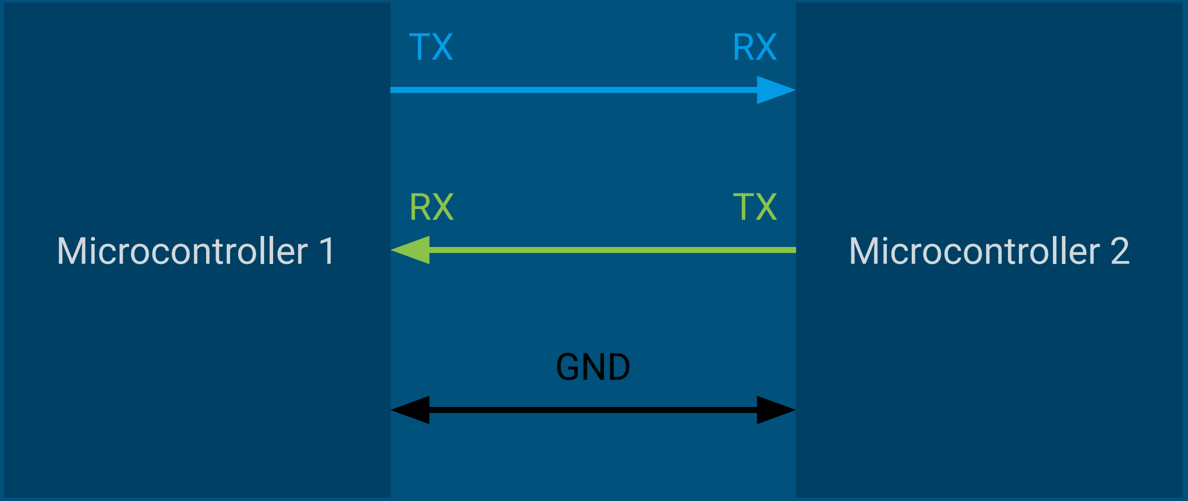

Figure 1:
UART Wiring Setup

Why use the Comms Utility?

- Send and receive desired sensors
- Dynamic allocation at runtime
- Unified sensor approach
- Easy to integrate
- Can write to SD card

Code Comparison for SD Writing Program

350 lines → 50 lines

Terminology

- **PACK** → byte array representation of a sensor
- **PACKET** → full set of bytes sent back and forth over UART
- **ACTIVE SENSOR** → sensor that is being recorded by current microcontroller
- **PASSIVE SENSOR** → sensor that is being received over UART

Example Usage

Example Usage: Example's Goals

Code Goals:

Over Serial1 (TX1/RX1),

1. Write an LDS sensor.
2. Write an HE speed sensor.
3. Receive one HE speed sensor.

```
#include <UARTComms.h>
#include <Sensor.h>

void setup() {

}

void loop() {

}
```

Example Usage: Construct Objects

```
#include <UARTComms.h>
#include <Sensor.h>

// <-- CODE GOES HERE

void setup() {

}

void loop() {

}
```

```
// Create UARTComms object on Serial1
// 115200 baud
UARTComms uart1(115200, Serial1);

// Create Active LDS Sensor (on port 1)
LinearDisplacementSensor lds(1);

// Create Active HE Speed Sensor (on port 6)
HallEffectSpeedSensor he1(6, 200);

// Create Passive HE Speed Sensor (no port)
HallEffectSpeedSensor he2;
```

Example Usage: Begin UARTComms Object

```
#include <UARTComms.h>
#include <Sensor.h>

// <-- STEP 1 CODE

void setup() {
    // <-- CODE GOES HERE
}

void loop() {

}
```

```
// Sets up the UART port.
uart1.begin();
```

Example Usage: Attach All Sensors

```
#include <UARTComms.h>
#include <Sensor.h>

// <-- STEP 1 CODE

void setup() {
    // <-- STEP 2 CODE
    // <-- CODE GOES HERE
}

void loop() {

}
```

```
// Attach LDS Sensor as an output of uart1
uart1.attach_output_sensor(lds, LDS_TEST1);

// Attach HE Sensor 1 as an output of uart1
uart1.attach_output_sensor(he1, HE_TEST1);

// Attach HE Sensor 2 as an input from uart1
uart1.attach_input_sensor (he2, HE_TEST2);
```

Example Usage: Begin Active Sensors

```
#include <UARTComms.h>
#include <Sensor.h>

// <-- STEP 1 CODE

void setup() {
    // <-- STEP 2 CODE
    // <-- STEP 3 CODE
    // <-- CODE GOES HERE
}

void loop() {

}
```

```
// Sets up the I/O ports
// No setup for he2 because it is passive
lds.begin();
he1.begin();
```

Example Usage: Update UARTComms Object

```
#include <UARTComms.h>
#include <Sensor.h>

// <-- STEP 1 CODE

void setup() {
    // <-- STEP 2 CODE
    // <-- STEP 3 CODE
    // <-- STEP 4 CODE
}

void loop() {
    // <-- CODE GOES HERE
}
```

```
// Automatically receives and sends data
// Use lds, he1, and he2 as normal
uart1.update();
```

Steps for Using the Communications Utility

1. Construct Objects
2. Begin *UARTComms* Object
3. Attach All Sensors
4. Begin Active Sensors
5. Update *UARTComms* Object

```
#include <UARTComms.h>
#include <Sensor.h>

// <-- STEP 1 CODE

void setup() {
    // <-- STEP 2 CODE
    // <-- STEP 3 CODE
    // <-- STEP 4 CODE
}

void loop() {
    // <-- STEP 5 CODE
}
```

Adding Sensors and Functionality

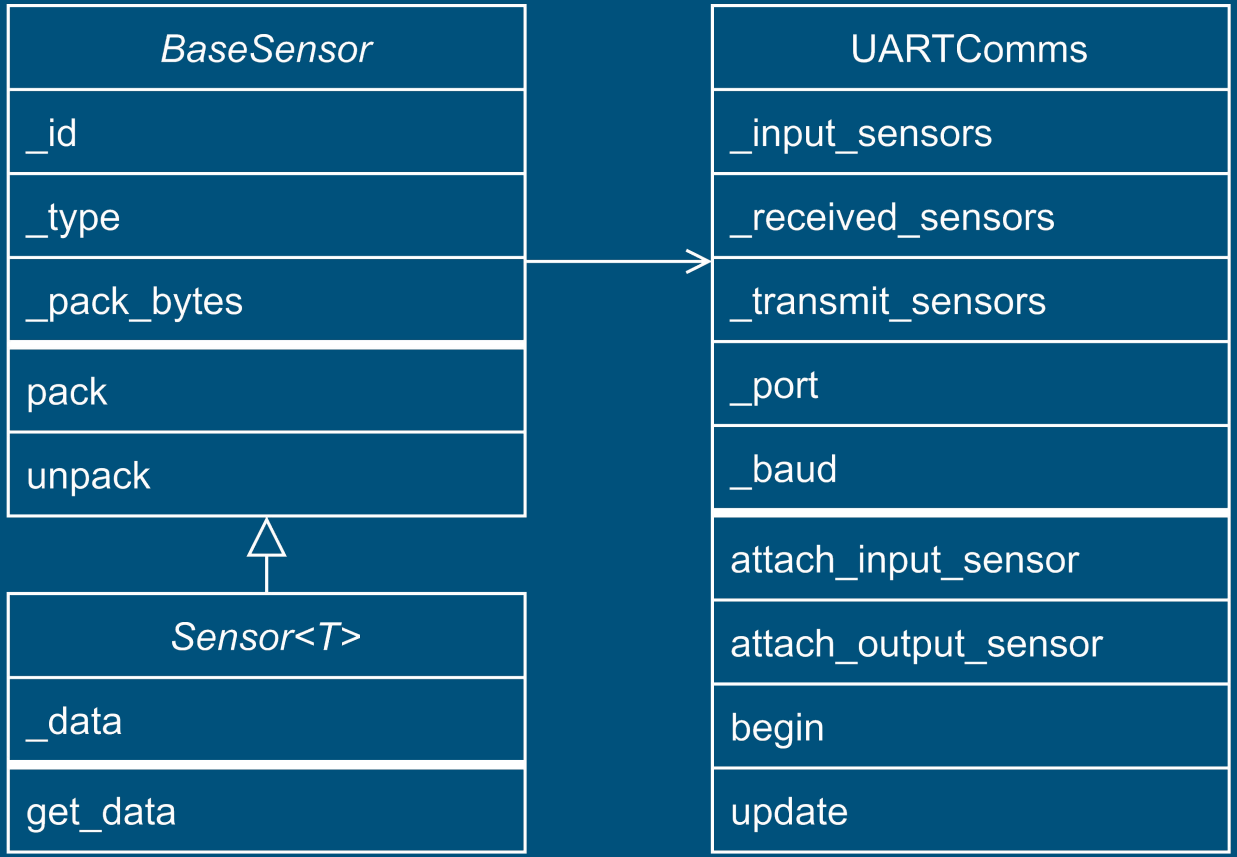

Figure 2:
Class Hierarchy

Adding New Sensors: Identify Data Type

- Sensor template expansion
- Choose/create convenient data type
  - `uint32_t`, `int16_t`, `float`, etc.
  - `std::vector`
  - Custom structs for multiple return variables

```
class NewSensor : public Sensor<DataType> {

};
```

Adding New Sensors: Create Constructors

- Active sensors likely have an input pin in constructor.
- Passive sensors need a default constructor
- *_pack_bytes* needs to be defined.
  - The number of bytes the pack for this sensor will occupy.
  - `uint32_t` --> 4 bytes
  - six `uint16_t` --> 12 bytes

```
NewSensor(uint8_t pin) : _pin(pin) {
    _pack_bytes = ???;
}
NewSensor(){
    _pack_bytes = ???;
}
```

Adding New Sensors: Define get_data

- Unified get_data can be used by all sensors
- Update _data using inputs iff the sensor is ACTIVE
  - Sensors are ACTIVE by default until attached as an input sensor.
  - PASSIVE sensors will be updated by unpack instead.
- Any helper functions should call get_data

```
const DataType& get_data(){
    if(_type == ACTIVE){
        // Update _data using input pins
    }
    return _data;
}
```

Adding New Sensors: Define pack

- Input is a pointer to a byte array of _pack_bytes size
- Package your _data into pack
  - Pointer tricks might be helpful

```
void pack(byte* pack){
   // Set byte array using _data
}
```

Adding New Sensors: Define unpack

- Input is a pointer to a const byte array of _pack_bytes size
- Exactly the opposite of pack
- Take pack and set _data
  - Pointer tricks might be helpful, often can flip commands from pack

```
void unpack(const byte* pack){
    // Set _data using byte array
}
```

Adding New Sensors: Include New Sensor

- Include your new .h file at the bottom of Sensor.h
- Add some specific sensor IDs for usage in SensorId.h

```
#include “./DerivedSensors/NewSensor.h”
```

Steps for Adding New Sensors

1. Identify Data Type
2. Create Constructor(s)
3. Define *get_data* Function
4. Define *pack* Function
5. Define *unpack* Function
6. Include New Sensor and Add ID's

Refer to Libraries/Sensor/DerivedSensors for examples.

Note: All functions using _data (e.g. pack, unpack) need to be defined directly in the header. Since sensors are relatively simple anyways, to avoid complication, try to keep the whole implementation in a single header.

Conclusion

The End

Thanks for listening!