ROWLAND TECHNOLOGY

MULTIUART

SPI to 4-Channel UART Bridge

Product Datasheet & Developer Guide

Key Features

1 SPI Interface → 4 Independent UART Channels

Buffered Rx/Tx • Per-Channel Baud Rate • 1200–115200 Baud

Arduino Library • Flowcode Component • GPL-3.0

Compatible with Arduino and Flowcode environments

Table of Contents

1. Product Overview

The Rowland Technology MULTIUART is a compact hardware bridge board that converts a single SPI bus connection into four independent, buffered hardware UART channels. It allows a microcontroller or single-board computer with only one SPI peripheral to communicate simultaneously with up to four separate UART devices, dramatically expanding serial port capacity without consuming additional microcontroller resources.

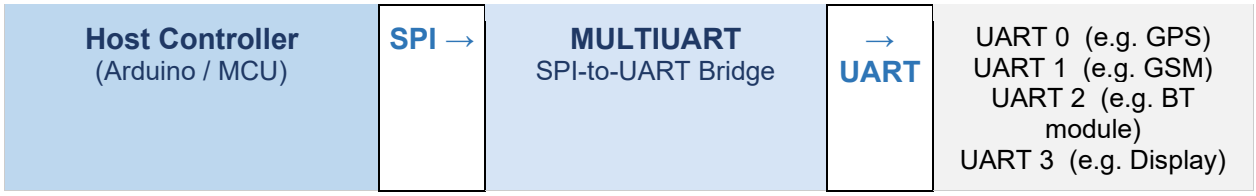
The board is designed for use in embedded systems, robotics, IoT gateways, and industrial control applications where multiple serial devices must be managed from a single master. An Arduino library and Flowcode component are provided to simplify integration.

1.1 Key Specifications

Specification	Value
Interface (host side)	SPI (MOSI, MISO, SCK, CS)
UART channels	4 independent full-duplex channels (UART 0–3)
Supported baud rates	1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200 baud
Baud rate storage	Stored in non-volatile flash (survives power cycle)
Rx/Tx buffering	Hardware-buffered on both transmit and receive paths
SPI bit order	MSB first
SPI clock dividers	DIV2, DIV4, DIV8, DIV16, DIV32, DIV64, DIV128
Programming language	Firmware in C (Microchip); Host library in C++ (Arduino)
Arduino library	Yes — MULTIUART class (MULTIUART.h / MULTIUART.cpp)
Flowcode support	Flowcode 6, 7, and 8 (.fcpx component included)
License	GPL-3.0

1.2 Block Diagram (Conceptual)

The MULTIUART sits between your host microcontroller's SPI bus and up to four target UART devices:



2. Getting Started

2.1 What You Need

- MULTIUART board
- Arduino (Uno, Mega, or compatible) or any SPI-capable microcontroller
- SPI wiring: MOSI, MISO, SCK, and one CS (chip select) pin
- MULTIUART Arduino library (MULTIUART.h and MULTIUART.cpp from the repository)
- Arduino IDE 1.x or later
- USB cable for programming your Arduino

2.2 Wiring the MULTIUART

Connect the MULTIUART board to your Arduino using the SPI bus. The default example uses pin D10 as the Chip Select (CS):

MULTIUART Pin	Arduino Pin	Notes
MOSI	D11 (MOSI)	SPI data from Arduino to MULTIUART
MISO	D12 (MISO)	SPI data from MULTIUART to Arduino
SCK	D13 (SCK)	SPI clock
CS	D10 (configurable)	Chip Select — active LOW. Any digital output pin can be used.
GND	GND	Common ground
VCC	3.3V or 5V	Power supply — check board variant for correct voltage

NOTE: *If using a CS pin other than D10, update the MULTIUART constructor call in your sketch: MULTIUART multiuart(YOUR_PIN);*

2.3 Installing the Library

- Download MULTIUART.h and MULTIUART.cpp from the repository root.
- Place both files in a folder named MULTIUART inside your Arduino libraries directory (e.g. Documents/Arduino/libraries/MULTIUART/).
- Restart the Arduino IDE.
- The library will appear under Sketch > Include Library > MULTIUART.
- Open the Demo sketch from File > Examples > MULTIUART > Demo to verify.

2.4 Quick-Start Sketch

The minimum sketch to initialise the board and send a message on all four UART channels:

```
#include <SPI.h>
#include <MULTIUART.h>

MULTIUART multiuart(10);   // CS on pin D10

void setup() {
```

```
  multiuart.initialise(SPI_CLOCK_DIV64);
  multiuart.SetBaud(0, 3);  // UART 0 = 9600 baud
  multiuart.SetBaud(1, 3);  // UART 1 = 9600 baud
  multiuart.SetBaud(2, 7);  // UART 2 = 115200 baud
  multiuart.SetBaud(3, 7);  // UART 3 = 115200 baud
}

void loop() {
  multiuart.TransmitString(0, "Hello from UART 0", 18);
  delay(1000);
}
```

NOTE: *SetBaud() writes to flash memory and takes ~20 ms per call (flash erase and write cycle). Only call SetBaud() during setup(), not in the main loop.*

2.5 SPI Clock Speed Selection

Choose the SPI clock divider based on your host microcontroller speed and wiring quality. On a 16 MHz Arduino Uno:

Divider Constant	SPI Clock (16 MHz Arduino)	Notes
`SPI_CLOCK_DIV2`	8 MHz	Maximum speed — short wiring only
`SPI_CLOCK_DIV4`	4 MHz	Fast, generally reliable
`SPI_CLOCK_DIV8`	2 MHz	Good balance of speed and reliability
`SPI_CLOCK_DIV16`	1 MHz	Reliable for longer wires
`SPI_CLOCK_DIV32`	500 kHz	Conservative
`SPI_CLOCK_DIV64`	250 kHz	Used in the demo sketch — safe default
`SPI_CLOCK_DIV128`	125 kHz	Maximum reliability / slowest

3. Arduino Library API Reference

Include the library in your sketch and create a MULTIUART object by passing the Arduino pin number used for Chip Select:

```
#include <SPI.h>
#include <MULTIUART.h>
MULTIUART multiuart(10);   // 10 = CS pin number
```

All UART channel arguments are zero-indexed integers in the range 0–3.

3.1 Constructor

Method	Parameters	Returns	Description
`MULTIUART(ss)`	ss: int — Arduino pin number for CS	Object	Constructor. Configures the CS pin as an output and stores the pin number.

3.2 Initialisation

Method	Parameters	Returns	Description
`initialise(SPIDivider)`	SPIDivider: SPI_CLOCK_DIVx constant	void	Starts the SPI bus, sets bit order to MSB first, and applies the specified clock divider. Must be called in setup() before any other method.

3.3 Baud Rate Configuration

Method	Parameters	Returns	Description
`SetBaud(UART, BAUD)`	UART: char (0–3) BAUD: char (0–7)	void	Sets the baud rate for the specified UART channel. The value is stored in non-volatile flash — persists across power cycles. Waits 20 ms for flash write. Call only in setup().

Baud Rate Code Table

Code (BAUD)	Baud Rate
0	1200
1	2400
2	4800
3	9600
4	19200
5	38400
6	57600

7	115200

3.4 Status / Buffer Check

Method	Parameters	Returns	Description
`CheckRx(UART)`	UART: char (0–3)	char (byte count)	Returns the number of bytes currently waiting in the receive buffer for the specified UART channel. Returns 0 if no data is available.
`CheckTx(UART)`	UART: char (0–3)	char (byte count)	Returns the number of bytes currently waiting in the transmit buffer for the specified UART channel. Useful to check if a previous transmission has completed.

3.5 Receive Methods

Method	Parameters	Returns	Description
`ReceiveByte(UART)`	UART: char (0–3)	char	Reads a single byte from the receive buffer of the specified UART channel.
`ReceiveString(RETVAL, UART, NUMBYTES)`	RETVAL: char* UART: char (0–3) NUMBYTES: char	void (via pointer)	Reads NUMBYTES bytes from the receive buffer into the RETVAL buffer. Appends a null terminator. Ensure RETVAL is large enough to hold NUMBYTES + 1 characters.

NOTE: *Always call CheckRx() before ReceiveString() and pass the returned count as NUMBYTES to avoid reading stale or uninitialised data.*

3.6 Transmit Methods

Method	Parameters	Returns	Description
`TransmitByte(UART, DATA)`	UART: char (0–3) DATA: char	void	Sends a single byte to the specified UART channel.
`TransmitString(UART, DATA, NUMBYTES)`	UART: char (0–3) DATA: char* NUMBYTES: char	void	Sends NUMBYTES bytes from the DATA buffer to the specified UART channel. Does not transmit a null terminator automatically.

4. Usage Examples

4.1 Transmit on All Four UART Channels

Send a test string on each channel simultaneously:

```
#include <SPI.h>
#include <MULTIUART.h>

MULTIUART multiuart(10);

void setup() {
  multiuart.initialise(SPI_CLOCK_DIV64);
  multiuart.SetBaud(0, 3);  // 9600 baud
  multiuart.SetBaud(1, 3);
  multiuart.SetBaud(2, 7);  // 115200 baud
  multiuart.SetBaud(3, 7);
}

void loop() {
  multiuart.TransmitString(0, "UART 0 Test", 12);
  multiuart.TransmitString(1, "UART 1 Test", 12);
  multiuart.TransmitString(2, "UART 2 Test", 12);
  multiuart.TransmitString(3, "UART 3 Test", 12);
  delay(2000);
}
```

4.2 Receive Data with Buffer Check

Safely read any incoming data from all four channels and print it to the Arduino Serial Monitor:

```
#include <SPI.h>
#include <MULTIUART.h>

MULTIUART multiuart(10);
char buf[64];

void setup() {
  Serial.begin(9600);
  multiuart.initialise(SPI_CLOCK_DIV64);
  multiuart.SetBaud(0, 3);
  multiuart.SetBaud(1, 3);
  multiuart.SetBaud(2, 3);
  multiuart.SetBaud(3, 3);
}

void loop() {
  for (char ch = 0; ch < 4; ch++) {
    char count = multiuart.CheckRx(ch);
    if (count > 0) {
      multiuart.ReceiveString(buf, ch, count);
      Serial.print("UART "); Serial.print(ch);
      Serial.print(": "); Serial.println(buf);
    }
  }
```

```
  delay(500);
}
```

4.3 Send a Single Byte

Use TransmitByte() when sending control characters or single-byte commands:

```
multiuart.TransmitByte(0, 0x0D);  // Send carriage return on UART 0
multiuart.TransmitByte(2, 0xA5);  // Send command byte 0xA5 on UART 2
```

4.4 Read a Single Byte

Use ReceiveByte() when you expect exactly one byte in response to a command:

```
multiuart.TransmitByte(1, 0x55);          // Send request
delay(10);                                  // Wait for response
char response = multiuart.ReceiveByte(1); // Read single byte back
```

4.5 Full Demo Sketch

The complete demo sketch from the repository transmits on all channels then reads back any responses:

```
#include <SPI.h>
#include <MULTIUART.h>

MULTIUART multiuart(10);
char LENGTH;
char Str1[20];

void setup() {
  multiuart.initialise(SPI_CLOCK_DIV64);
  Serial.begin(9600);
  multiuart.SetBaud(0, 3);  // 9600 baud
  multiuart.SetBaud(1, 3);
  multiuart.SetBaud(2, 7);  // 115200 baud
  multiuart.SetBaud(3, 7);
}

void loop() {
  delay(2000);
  multiuart.TransmitString(0, "UART 0 Test", 12);
  multiuart.TransmitString(1, "UART 1 Test", 12);
  multiuart.TransmitString(2, "UART 2 Test", 12);
  multiuart.TransmitString(3, "UART 3 Test", 12);
  delay(2000);
  for (char ch = 0; ch < 4; ch++) {
    Serial.print("UART "); Serial.print((int)ch); Serial.print(": ");
    LENGTH = multiuart.CheckRx(ch);
    if (LENGTH > 0) {
      multiuart.ReceiveString(Str1, ch, LENGTH);
      Serial.println(Str1);
    }
  }
}
```

5. Deployment Guide

5.1 Initialisation Sequence

Always follow this sequence when starting your application:

- Call initialise() with a suitable SPI clock divider.
- Call SetBaud() for each UART channel you intend to use — do this only once in setup(), not in loop().
- Wait briefly (~100 ms) after SetBaud() calls before beginning communication, to allow the board to stabilise.
- Communicate using TransmitByte / TransmitString and CheckRx / ReceiveByte / ReceiveString.

5.2 Sharing the SPI Bus

The MULTIUART uses standard SPI with a dedicated CS pin, so it can share the SPI bus with other SPI devices (SD cards, displays, sensors, etc.). Ensure:

- Each SPI device has its own CS pin.
- Only one device's CS is LOW at any time.
- The Arduino SPI library manages CS automatically when using the MULTIUART library.
- If other SPI devices use different SPI modes or bit orders, call SPI.setBitOrder(MSBFIRST) before each MULTIUART transaction as needed.

5.3 Buffer Management

The MULTIUART board maintains hardware receive buffers for each UART channel. Follow these guidelines:

- Poll CheckRx() regularly to avoid buffer overflow — if the buffer fills up, incoming bytes may be lost.
- In high-throughput applications, reduce the main loop delay or use interrupts to poll more frequently.
- Use CheckTx() to confirm that a previous TransmitString() has completed before sending more data on the same channel.
- Each TransmitString() call must not exceed the transmit buffer size. For large payloads, break them into smaller chunks.

5.4 Multi-Channel Concurrent Use

All four UART channels operate independently and concurrently in hardware. The SPI bus is time-shared between channels at the software level. For best results:

- Service all channels in a round-robin polling loop, not sequentially with long delays.
- Keep individual SPI transaction times short to avoid starving busy channels.
- When channels have very different baud rates, the faster channels will fill their Rx buffers more quickly — poll them proportionally more often.

5.5 Baud Rate Persistence

Baud rate settings are stored in non-volatile flash on the MULTIUART board. This means:

- After power-on, the board retains the last baud rates set — you do not need to call SetBaud() on every boot if the rates have not changed.
- However, it is good practice to always call SetBaud() during initialisation to ensure the firmware is in the expected state, especially during development.
- Each SetBaud() call causes a flash erase-write cycle (~20 ms) — avoid calling it in loops or repeatedly.

5.6 Flowcode Integration

A Flowcode component (.fcpx) is provided in the flowcode/ folder, compatible with Flowcode 6, 7, and 8. Import the component into Flowcode to use drag-and-drop graphical programming blocks for all MULTIUART functions, including transmit, receive, and baud rate configuration.

5.7 Best Practices

- Always call initialise() before any other method.
- Call SetBaud() in setup() only — never in loop().
- Always check CheckRx() before receiving to avoid reading stale data.
- Allocate receive buffers large enough for the expected payload plus a null terminator.
- Use SPI_CLOCK_DIV64 as a safe default; increase speed only if throughput is insufficient.
- Test each UART channel individually before combining them in a single application.

6. Troubleshooting

Problem	Solution
No communication on any UART channel	Check SPI wiring (MOSI, MISO, SCK, CS). Verify the CS pin number matches the constructor argument. Confirm initialise() is called in setup().
Garbage data received on UART	Check that the baud rate set with SetBaud() matches the connected device's baud rate. Verify the SPI clock is not too fast for your wiring — try SPI_CLOCK_DIV64.
SetBaud() appears to have no effect	SetBaud() writes to flash and takes ~20 ms. Ensure you are calling it in setup() and waiting briefly before transmitting. Power-cycle the board after changing rates.
ReceiveString() returns empty or corrupt data	Always call CheckRx() first and pass the returned count to ReceiveString(). Do not call ReceiveString() when CheckRx() returns 0.
Only UART 0 works; others do not	Confirm the UART argument is correct (0–3). Verify the connected device is wired to the correct UART header on the board.
SPI conflicts with other devices	Ensure each SPI device has a unique CS pin. Do not share CS lines. Check that no other code pulls the MULTIUART CS pin LOW unexpectedly.
Buffer overflow / data loss at high baud rates	Poll CheckRx() more frequently. Reduce delay() calls in your loop. Consider processing received data via a faster SPI clock (lower divider value).
Flowcode component not found	Import the .fcpx file from the flowcode/ folder in the repository manually via Flowcode's component manager.

7. Technical Reference

7.1 SPI Command Protocol

The MULTIUART uses a binary SPI command protocol. Each transaction begins by asserting CS LOW, sending a command byte, then transferring data bytes, before releasing CS HIGH. A small delay (50–250 microseconds) is applied between transfers to allow the firmware to process each byte.

Command Byte	UART Channel	Additional Bytes	Operation
`0x10 | ch`	`0–3`	Read 1 byte (Rx count)	CheckRx — returns number of bytes in receive buffer
`0x20 | ch`	`0–3`	Send count; Read N bytes	Receive — transfers NUMBYTES bytes from Rx buffer
`0x30 | ch`	`0–3`	Read 1 byte (Tx count)	CheckTx — returns number of bytes in transmit buffer
`0x40 | ch`	`0–3`	Send count + N data bytes	Transmit — sends NUMBYTES bytes to Tx buffer
`0x80 | ch`	`0–3`	Send baud rate code (0–7)	SetBaud — stores baud rate in flash, waits 20 ms

NOTE: *The UART channel number (0–3) is ORed with the command base byte. For example, CheckRx on UART 2 sends 0x10 | 0x02 = 0x12.*

7.2 Timing

Timing Parameter	Value
Post-command delay (CheckRx/Tx)	250 microseconds
Post-transfer delay (general)	50 microseconds
Post-CS-release delay	50 microseconds
SetBaud() flash write time	~20 milliseconds (enforced by delay(20) in firmware)
SPI bit order	MSB first

7.3 Repository Structure

File / Folder	Contents
MULTIUART.h	Arduino library header — class declaration and API
MULTIUART.cpp	Arduino library implementation — all method bodies
keywords.txt	Arduino IDE keyword highlighting definitions
examples/Demo/	Demo.ino — complete example sketch
firmware/	Stable firmware release and pre-compiled hex file

flowcode/	Flowcode component (.fcpx) for Flowcode 6, 7, and 8
README.md	Project overview

7.4 Links

GitHub Repository: https://github.com/RowlandTechnology/MULTIUART
License: GNU General Public License v3.0 (GPL-3.0)
Author: Ben Rowland, Rowland Technology (created 19 August 2016)