Introduction

Serial Peripheral Interface (SPI) is a hardware communication standard that allows two-way communication between a master node and a slave node. The standard is flexible enough to allow multiple slaves, forming a Bluetooth-like star network. Unique chip select signals identify which slave is being addressed by the master. Two data lines (MOSI & MISO), and a clock line (SCLK) are shared by all participating nodes.

One of the main drawbacks of SPI is that it only supports one master. For most single-microcontroller embedded systems this is not a problem. But more sophisticated electronics may feature multiple microcontrollers that require access to shared resources. One example is in real-time measurement systems, where one microcontroller controls a process while another records data to a memory unit. The following example demonstrates an implementation of a *multiple master SPI scheme*.

Hardware

A secondary microcontroller (ATtiny167, or Tiny) is tasked with control over a high-speed ADC and recording its data to a memory unit. In order to transfer that information back to a PC via USB, the main microcontroller (ATmega32U4, or Mega) also needs access to that memory unit. Since the memory unit can only act as a slave, a careful negotiation process between both microcontrollers must be in place in order to ensure reliable data management.

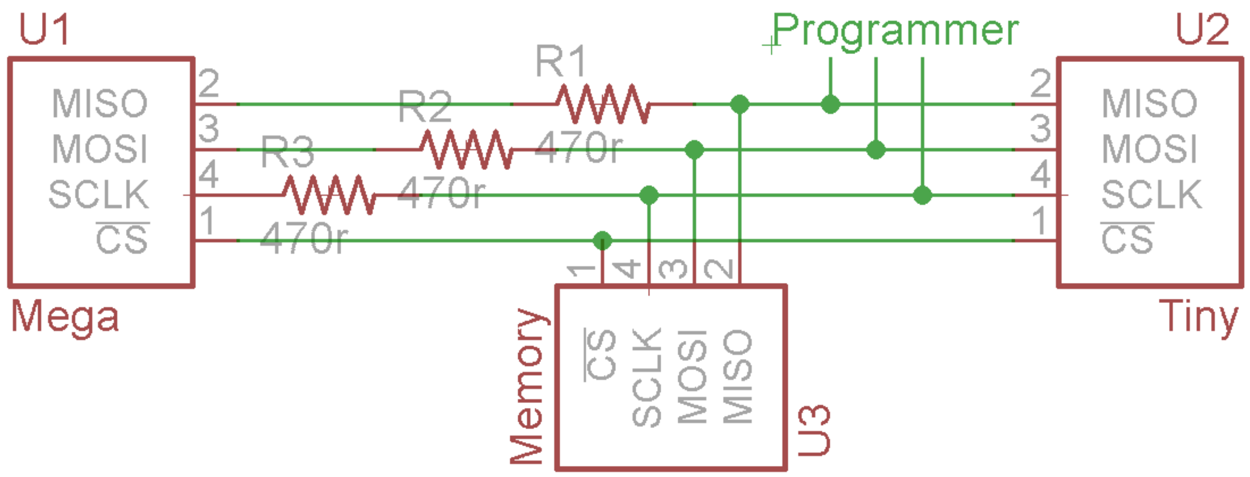

Figure 1: System schematic of multiple master SPI scheme.

The primary issue to avoid is *contention*, where one driver (i.e. Mega) tries to force an SPI line (say MOSI) in one direction, while another driver (i.e. Tiny) tries to force it in the other direction. Both microcontrollers have tri-stated pins, which means at any time the driver can be disabled. By default both micros have their SPI ports disabled, which allows the bus to freely float undisturbed. The UMAST signal is used by Mega to indicate it has precedence to access the memory. When Mega is ready to transfer control, it will drop UMAST low. Tiny will then indicate it has control by raising its TMAST signal high. Both micros can see each others' *flags* and can keep tabs on each other in case something goes wrong (TERR is a general purpose error flag for Tiny to use in case it experiences an unknown issue).

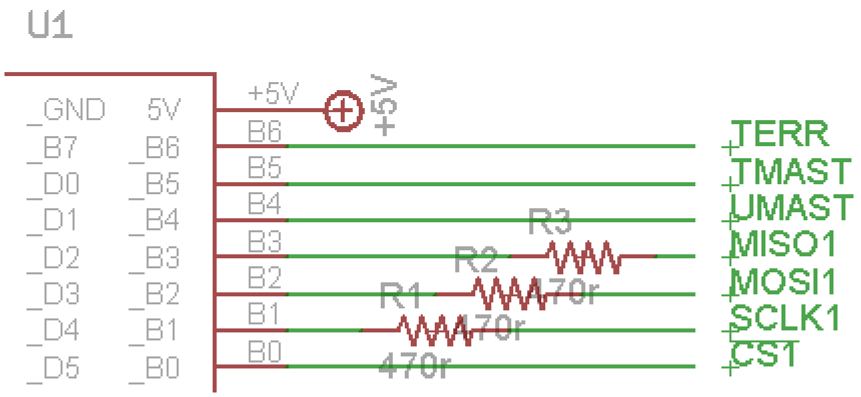

Figure 2: Master side SPI, related flags, and resistors.

One final issue to address is the programming of the Tiny microcontroller, which also utilizes the SPI port (except for the !CS signal). In order to protect Mega from programming attempts by an external programmer, we place series resistors on its SPI port.

Firmware

The following pseudocode for each microcontroller utilizes the following shared functions. Register specific instructions (such as the assignment of Port A) is specific to the AtTiny167 microcontroller. These functions can be easily adapted for use in the Atmega32U4.

```
void SPI_SHUTDOWN()
{
	// 1. disable SPI
	SPCR = 0x00; SPSR = 0x00;
	// 2. set SPI to HiZ input
	// 3. set T1MAST to output low
	DDRA = 0x8A; PORTA = 0x88;
}

void SPI_SETUP()
{
	DDRA = 0xFA; // all out except MISO and UMAST1
	PORTA = 0xFF;
	SPCR = 0x50; // enable SPI Master, latch data on rising clock edge
	SPSR = 0x01; // SPCR sets f = fosc/4, here we set double speed so fSPI = fosc/2
}
```

The core of the Tiny firmware routine is an infinitely repeating loop. Its default state is waiting for the Atmega32U4 to relinquish control over the memory unit (by waiting on UMAST to drop low). Also by default its SPI port is disabled and the pins are set to high impedance inputs, to avoid contention.

After UMAST drops low, Tiny raises its own flag, sets up its SPI port, and begins recording data from an ADC to memory. The implementation of these functions varies with respect to the ADC and memory modules used (of which there are thousands), so only the pseudocode has been provided. After the memory has been filled (to MAX), Tiny lowers its own flag, and the while loop reiterates, shutting down SPI and waiting for UMAST to be lowered again.

```
while (1)
{
	SPI_SHUTDOWN();      // disables SPI after each write session
	while (UMAST) {};    // wait if UMAST is high
	TMAST = 1;           // send TMAST high
	SPI_SETUP();         // enable SPI
	ADC_CONVERT(1);      // start ADC conversion
	for (int i = 0; i < MAX; i++)
	{
		ADC_CONVERT(0); // start ADC data transfer
		getADC(data);   // receive ADC data
		ADC_CONVERT(1); // start ADC conversion
		memWrite(data); // write ADC to memory
	}
	TMAST = 0;           // drop TMAST low
}
```

The relevant portion of the Mega firmware is itself a subfunction of the main firmware. It is left to the user to decide when this process occurs within the workflow of the main processor. Like Tiny, the default state of Mega is to have SPI disabled and have high impedance input pins. The UMAST signal is default high (to keep Tiny from acting when it is not needed). Upon initiating this process, Mega releases the UMAST signal and Tiny begins its process.

Meanwhile Mega does its own thing. This is represented by the delay function. This delay should not be longer than Tiny's data recording process – when Tiny checks for UMAST next it must be high to prevent an unnecessary round of data recording. For this reason it is best to raise UMAST immediately after lowering it, perhaps with a maximum delay of 10 milliseconds.

```
void process(long delay)
{
	UMAST = 0;    // signal Tiny to start recording
	while (delay--) _delay_us(1); // process timing control
	UMAST = 1;           // signal Tiny to end process
	while (TMAST) {};    // wait for Tiny to be done
	SPI_SETUP();         // enable SPI
	for (int p = 0; p < MAX; p++) // retrieve data and return via USB

	{
		for (int i = 0; i < 64; i++) txBuffer[i] = SPI1_RX();
		usb_rawhid_send(txBuffer,10);
	}
	SPI_SHUTDOWN();      // disable SPI
}
```

After Tiny is done with the memory unit and lowers its TMAST flag, Mega sets up its SPI module and retrieves the data from the memory unit, and transfers it over USB to a computer. This step can be replaced by some other external communication method (UART, virtual serial, etc).