Using ATSAMD21 SERCOM for more SPI, I2C and Serial ports

Created by lady ada


Last updated on 2023-08-29 03:04:37 PM EDT
Table of Contents

Overview

Muxing it up
- Multiplex Table
- Arduino's MUX Table
- Predefined SERCOMs
- Available SERCOM & Pins
- SERCOM 1
- SERCOM 2
- SERCOM 5
- Freeing up SERCOM5

Creating a new SPI
- How SPI is Created Now
- SAMD51 SPI Pin Pads
- OK so let's make a new SPI SERCOM already

Creating a new Serial
- How Serial is Created Now
- SAMD51 Serial SERCOM Pads
- OK so let's make a new Serial SERCOM already
- SAMD51 Interrupts

Creating a new Wire
- How Wire is Created Now
- OK so let's make a new I2C SERCOM already
Overview

If you've been using AVR's for a bit, particularly the Arduino UNO's ATmega328, you have likely bumped into the limits of having a single SPI, Wire (I2C) and UART (Serial) port. You can update to the Arduino MEGA, that'll give you some more UARTs, but you're still stuck with a single SPI and I2C port.
When we first checked out the ATSAMD21 chip (the processor used in the Arduino Zero and Adafruit Feather M0), we were very happy to see that the chip has 6 "SERCOM"s, a.k.a SERial COMmunication modules. Each one of these modules can be used for I2C, SPI or Serial. That means you can have 3 UART's & 3 SPI's, or maybe 2 SPI's and 4 I2C's. Basically, you have a ton of options for adding more hardware support for the most common 3 chip-to-chip protocols.

- Up to six Serial Communication Interfaces (SERCOM), each configurable to operate as either:
  - USART with full-duplex and single-wire half-duplex configuration
  - I2C up to 3.4MHz
  - SPI
  - LIN slave

Now, the default Arduino IDE setup for the Zero/ATSAMD21 still only comes with one UART, one SPI and one I2C device, in the 'classic' locations
...but that still leaves you with 3 full SERCOM's to do whatever you like!

## Muxing it up

So, wow, 6 SERCOMs, totally configurable - best thing ever! So what's the catch?

It's just a little one really - when setting up the SERCOM you have to configure what pins it will use (you have many options) and it takes a little time to look up the exact multiplexing setup and make sure you have it right. Not a huge deal, but has to be done right.

### Multiplex Table

If you look in the ATSAMD21 datasheet, there's a big table in section 6 called Multiplexing and Considerations, and it says...

Each pin is by default controlled by the PORT as a general purpose I/O and alternatively it can be assigned to one of the peripheral functions A, B, C, D, E, F, G or H. To enable a peripheral function on a pin, the Peripheral Multiplexer Enable bit in the Pin Configuration register corresponding to that pin (PINCFGn.PMUXEN, n = 0-31) in the PORT must be written to one.

The selection of peripheral function A to H is done by writing to the Peripheral Multiplexing Odd and Even bits in the Peripheral Multiplexing register (PMUXn.PMUXE/O) in the PORT
Which basically means - each pin has different capabilities, such as digital I/O, analog input, timer/pwm output, and sercom connectivity. On most chips, there's only one pin that can act as the I2C clock pin, but on this chip, there's quite a few pins that can be selected. So the trick is figuring out which pin you want to have that capability and 'MUXing it'.

Here's how to read the table, here we've got the first page:

<table>
<thead>
<tr>
<th>Pin</th>
<th>Type</th>
<th>RFC</th>
<th>I2C</th>
<th>ADC</th>
<th>AC</th>
<th>PTC</th>
<th>DAC</th>
<th>SERCOM</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The chip we're using is the ATSAMD21G note that the J series does have more pins so for example, pins 5-10 on the ATSAMD21J (PB04 thru PB07) don't apply to our chips.

In columns C and D, you can see the SERCOM pads available. I've also circled in blue the matching I/O Pins.

Each SERCOM has 4 possible pads: SERCOM#/PAD[x] where # can be 0 thru 5 (6 total SERCOMs), and x can be 0 thru 3 (4 total pads).

Some pins can only act as a single SERCOM's pads - for example PA00 is the first pin in the table and can only act as SERCOM1.PAD[0]

On the other hand, some pins can do dual duty: PA08 lower on the list can act as either SERCOM #0's PAD 0 or it can act as SERCOM #2's PAD 0.

**Arduino's MUX Table**

To map the raw pin name to the Arduino Zero 'board pin' you can read the table in `variants.cpp` (in the board package)
<table>
<thead>
<tr>
<th>Pin number</th>
<th>ZERO Board pin</th>
<th>PIN</th>
<th>Label/Name</th>
<th>Comments (* is for default peripheral in use)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Digital Low</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0 -&gt; RX</td>
<td>PA11</td>
<td></td>
<td>EIC/EXTINT[11]</td>
</tr>
<tr>
<td>1</td>
<td>1 &lt; TX</td>
<td>PA10</td>
<td></td>
<td>EIC/EXTINT[10]</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>PA14</td>
<td></td>
<td>EIC/EXTINT[14]</td>
</tr>
<tr>
<td>3</td>
<td>-3</td>
<td>PA09</td>
<td></td>
<td>EIC/EXTINT[9] ADC/</td>
</tr>
<tr>
<td>4</td>
<td>-4</td>
<td>PA08</td>
<td></td>
<td>EIC/NMI ADC/</td>
</tr>
<tr>
<td>5</td>
<td>-5</td>
<td>PA15</td>
<td></td>
<td>EIC/EXTINT[15]</td>
</tr>
<tr>
<td>8</td>
<td>-8</td>
<td>PA06</td>
<td></td>
<td>EIC/EXTINT[6] ADC/</td>
</tr>
<tr>
<td>9</td>
<td>-9</td>
<td>PA07</td>
<td></td>
<td>EIC/EXTINT[7] ADC/</td>
</tr>
<tr>
<td>14</td>
<td>-14</td>
<td>PA02</td>
<td></td>
<td>EIC/EXTINT[2] ADC/</td>
</tr>
<tr>
<td>ADC/AIN[18]</td>
<td>PTC/Y[0]</td>
<td>PA02</td>
<td>A0</td>
<td>EIC/EXTINT[2] ADC/</td>
</tr>
<tr>
<td>16</td>
<td>A2</td>
<td>PB09</td>
<td>A2</td>
<td>EIC/EXTINT[9] ADC/</td>
</tr>
<tr>
<td>17</td>
<td>A3</td>
<td>PA04</td>
<td>A3</td>
<td>EIC/EXTINT[4] ADC/</td>
</tr>
<tr>
<td>ADC/AIN[18]</td>
<td>PTC/Y[2]</td>
<td>SERCOM0/PAD[0] TCC0/WO[0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>A4</td>
<td>PA05</td>
<td>A4</td>
<td>EIC/EXTINT[5] ADC/</td>
</tr>
</tbody>
</table>
To save me from having to constantly look it up, here's a table of all the ATSAMD21G's SERCOM MUXs:

<table>
<thead>
<tr>
<th>Pin</th>
<th>Arduino 'Pin'</th>
<th>SERCOM</th>
<th>SERCOM alt</th>
</tr>
</thead>
<tbody>
<tr>
<td>PA00Used by xtal</td>
<td>SERCOM1.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PA01Used by xtal</td>
<td>SERCOM1.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PB08A1</td>
<td>SERCOM4.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PB09A2</td>
<td>SERCOM4.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PA04A3</td>
<td>SERCOM0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PA06D8</td>
<td>SERCOM0.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PA07D9</td>
<td>SERCOM0.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PA08D4</td>
<td>SERCOM0.0</td>
<td>SERCOM2.0</td>
<td></td>
</tr>
<tr>
<td>PA09D3</td>
<td>SERCOM0.1</td>
<td>SERCOM2.1</td>
<td></td>
</tr>
<tr>
<td>PA10D1</td>
<td>SERCOM0.2</td>
<td>SERCOM2.2</td>
<td></td>
</tr>
<tr>
<td>PA11D0</td>
<td>SERCOM0.3</td>
<td>SERCOM2.3</td>
<td></td>
</tr>
<tr>
<td>PB10D23 / MOSI</td>
<td>SERCOM4.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PB11D24 / SCK</td>
<td>SERCOM4.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PA12D22 / MISO</td>
<td>SERCOM2.0</td>
<td>SERCOM4.0</td>
<td></td>
</tr>
<tr>
<td>PA13Used by EDBCSERCOM2.1</td>
<td>SERCOM4.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PA14D2</td>
<td>SERCOM2.2</td>
<td>SERCOM4.2</td>
<td></td>
</tr>
<tr>
<td>PA15D5</td>
<td>SERCOM2.3</td>
<td>SERCOM4.3</td>
<td></td>
</tr>
<tr>
<td>PA16D11</td>
<td>SERCOM1.0</td>
<td>SERCOM3.0</td>
<td></td>
</tr>
<tr>
<td>PA17D13</td>
<td>SERCOM1.1</td>
<td>SERCOM3.1</td>
<td></td>
</tr>
<tr>
<td>PA18D10</td>
<td>SERCOM1.2</td>
<td>SERCOM3.2</td>
<td></td>
</tr>
<tr>
<td>PA19D12</td>
<td>SERCOM1.3</td>
<td>SERCOM3.3</td>
<td></td>
</tr>
<tr>
<td>PA20D6</td>
<td>SERCOM5.2</td>
<td>SERCOM3.2</td>
<td></td>
</tr>
<tr>
<td>PA21D7</td>
<td>SERCOM5.3</td>
<td>SERCOM3.3</td>
<td></td>
</tr>
<tr>
<td>PA22D20 / SDA</td>
<td>SERCOM3.0</td>
<td>SERCOM5.0</td>
<td></td>
</tr>
<tr>
<td>PA23D21 / SCL</td>
<td>SERCOM3.1</td>
<td>SERCOM5.1</td>
<td></td>
</tr>
<tr>
<td>PA24Used by USB</td>
<td>SERCOM3.2</td>
<td>SERCOM5.2</td>
<td></td>
</tr>
<tr>
<td>PA25Used by USB</td>
<td>SERCOM3.3</td>
<td>SERCOM5.3</td>
<td></td>
</tr>
<tr>
<td>PB22D30 / EDBG TX</td>
<td>SERCOM5.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PB23D31 / EDBG RX</td>
<td>SERCOM5.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PA30Used by SWCLK</td>
<td>SERCOM1.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PA31Used by SWDIO</td>
<td>SERCOM1.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PB02A5</td>
<td>SERCOM5.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PB03D25 / RX LED</td>
<td>SERCOM5.1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that a bunch of these pins are simply *not available* if you've got a Zero or Feather M0, because they are used for programming or LEDs, or not brought out to headers in some other way, so lets remove them & sort by pin

<table>
<thead>
<tr>
<th>Pin Arduino 'Pin'</th>
<th>SERCOM</th>
<th>SERCOM alt</th>
</tr>
</thead>
<tbody>
<tr>
<td>PA11D0</td>
<td>SERCOM0.3</td>
<td>SERCOM2.3</td>
</tr>
<tr>
<td>PA10D1</td>
<td>SERCOM0.2</td>
<td>SERCOM2.2</td>
</tr>
<tr>
<td>PA14D2</td>
<td>SERCOM2.2</td>
<td>SERCOM4.2</td>
</tr>
<tr>
<td>PA09D3</td>
<td>SERCOM0.1</td>
<td>SERCOM2.1</td>
</tr>
<tr>
<td>PA08D4</td>
<td>SERCOM0.0</td>
<td>SERCOM2.0</td>
</tr>
<tr>
<td>PA15D5</td>
<td>SERCOM2.3</td>
<td>SERCOM4.3</td>
</tr>
<tr>
<td>PA16D11</td>
<td>SERCOM1.0</td>
<td>SERCOM3.0</td>
</tr>
<tr>
<td>PA17D13</td>
<td>SERCOM1.1</td>
<td>SERCOM3.1</td>
</tr>
<tr>
<td>PA18D10</td>
<td>SERCOM1.2</td>
<td>SERCOM3.2</td>
</tr>
<tr>
<td>PA19D12</td>
<td>SERCOM1.3</td>
<td>SERCOM3.3</td>
</tr>
<tr>
<td>PA20D6</td>
<td>SERCOM5.2</td>
<td>SERCOM3.2</td>
</tr>
<tr>
<td>PA21D7</td>
<td>SERCOM5.3</td>
<td>SERCOM3.3</td>
</tr>
<tr>
<td>PA06D8</td>
<td>SERCOM0.2</td>
<td></td>
</tr>
<tr>
<td>PA07D9</td>
<td>SERCOM0.3</td>
<td></td>
</tr>
<tr>
<td>PA18D10</td>
<td>SERCOM1.2</td>
<td>SERCOM3.2</td>
</tr>
<tr>
<td>PA16D11</td>
<td>SERCOM1.0</td>
<td>SERCOM3.0</td>
</tr>
<tr>
<td>PA19D12</td>
<td>SERCOM1.3</td>
<td>SERCOM3.3</td>
</tr>
<tr>
<td>PA17D13</td>
<td>SERCOM1.1</td>
<td>SERCOM3.1</td>
</tr>
<tr>
<td>PB08A1</td>
<td>SERCOM4.0</td>
<td></td>
</tr>
<tr>
<td>PB09A2</td>
<td>SERCOM4.1</td>
<td></td>
</tr>
</tbody>
</table>
Predefined SERCOMs

OK so first things we want to sort out is what SERCOMs are already taken, this tutorial will assume you won't 'bash' existing SERCOMs (altho you are free to, if you want to give up the hardware serial UART for I2C, say).

The debug hardware serial port

which is used when connecting to the Programming/Debug Port on an Arduino Zero is on SERCOM 5, pads 2 & 3. It's not used on an Adafruit Feather because there's no debug port

I2C (SDA & SCL)

use SERCOM 3 (pads 0 & 1)

The SPI port

is on SERCOM 4, and uses pads 0, 2, and 3

And finally, the

hardware Serial interface (e.g. Serial1)

is on SERCOM 0
That leaves you with:

SERCOMs 1 & 2 on an Arduino Zero

and

SERCOMs 1, 2, and 5 on a Feather M0

Available SERCOM & Pins

OK we’re nearly done paring down that table to the SERCOMs that we can use and pins that are available.

**SERCOM 1**

Available on both Zero’s and Feathers, you can use pins 10, 11, 12 and 13 for SERCOM 1

<table>
<thead>
<tr>
<th>Pin Arduino 'Pin'</th>
<th>SERCOM</th>
<th>SERCOM alt</th>
</tr>
</thead>
<tbody>
<tr>
<td>PA18D10</td>
<td>SERCOM1.2</td>
<td>SERCOM3.2</td>
</tr>
<tr>
<td>PA16D11</td>
<td>SERCOM1.0</td>
<td>SERCOM3.0</td>
</tr>
<tr>
<td>PA19D12</td>
<td>SERCOM1.3</td>
<td>SERCOM3.3</td>
</tr>
<tr>
<td>PA17D13</td>
<td>SERCOM1.1</td>
<td>SERCOM3.1</td>
</tr>
</tbody>
</table>

**SERCOM 2**

This one has more options

<table>
<thead>
<tr>
<th>Pin Arduino 'Pin'</th>
<th>SERCOM</th>
<th>SERCOM alt</th>
</tr>
</thead>
<tbody>
<tr>
<td>PA11D0</td>
<td>SERCOM0.3</td>
<td>SERCOM2.3</td>
</tr>
<tr>
<td>PA10D1</td>
<td>SERCOM0.2</td>
<td>SERCOM2.2</td>
</tr>
<tr>
<td>PA14D2</td>
<td>SERCOM2.2</td>
<td>SERCOM4.2</td>
</tr>
<tr>
<td>PA09D3</td>
<td>SERCOM0.1</td>
<td>SERCOM2.1</td>
</tr>
<tr>
<td>PA08D4</td>
<td>SERCOM0.0</td>
<td>SERCOM2.0</td>
</tr>
<tr>
<td>PA15D5</td>
<td>SERCOM2.3</td>
<td>SERCOM4.3</td>
</tr>
<tr>
<td>PA12D22 / MISO</td>
<td>SERCOM2.0</td>
<td>SERCOM4.0</td>
</tr>
</tbody>
</table>

but some are already used, so while you could theoretically use D0 & D1 those are already used for the Hardware Serial UART, and D22/MISO is used for SPI. So that leaves D2 thru D5
SERCOM 5

This one is used for programming on the Zero with EDBG interface but you can use it on the Feather M0 since only native USB is used.

Two of those are used by SDA/SCL which, unless you want to reuse for a different SERCOM, you'll only have:

Note that you don't get access to SERCOM5.1!

Freeing up SERCOM5

If you'd like to free up SERCOM5 on a Zero, and you won't be using the programming port (because that's how Serial data is passed back and forth) you can comment out

```c
Uart_Serial( &sercom5, PIN_SERIAL_RX, PIN_SERIAL_TX, PAD_SERIAL_RX, PAD_SERIAL_TX );
```

and

```c
void SERCOM5_Handler()
{
    Serial.IrqHandler();
}
```
At the end of the variants file () (you'll need to dig for where this is downloaded after you've added SAMD support, and you'll also have to re-remove after each board manager update of that package.

Note that you'll definitely not be able to use Serial anymore, you can only use USBSerial. Like I said, its a bit annoying, you may want to grab the Adafruit SAMD package, and 'pretend' like your Zero is a Feather M0, it'll work just fine, and then Serial will point to USBSerial. whee!

Creating a new SPI

SPI is a high speed, 3-wire protocol that can be clocked at up to 12MHz on the ATSAMD21 and can also take advantage of DMA (that's for another tutorial)

Each SPI SERCOM has 3 pins - Serial Clock line (SCK), Microcontroller Out/Serial In (MOSI) and Microcontroller In/Serial Out (MISO). In these examples we'll assume the ATSAMD21 is acting as the Microcontroller, since that's the by-far-most-common example. In this way the chip can control wifi modules, SD cards, DotStar LEDs, various sensors and actuators, etc.

So you might be wondering "Why does she keep bringing up that SPI uses only 3 pins?" It's cuz SERCOM's have 4 pads, but we only need three. There is one unused one that can be used as a normal GPIO (or whatever) pin

How SPI is Created Now

Luckily, Atmel & Arduino did a really great job with structuring SERCOMs so you can create and assign new ones. You can find the actual code that is used to create the "SPI" SERCOM in SPI.cpp ()

```
SPIClass SPI (&PERIPH_SPI, PIN_SPI_MISO, PIN_SPI_SCK, PIN_SPI_MOSI, PAD_SPI_TX, PAD_SPI_RX);
```

The macros are see in variants.h ()

```
#define PIN_SPI_MISO         (22u)
#define PIN_SPI_MOSI         (23u)
#define PIN_SPI_SCK          (24u)
#define PERIPH_SPI           sercom4
#define PAD_SPI_TX           SPI_PAD_2_SCK_3
#define PAD_SPI_RX           SERCOM_RX_PAD_0
```

You can see here that SPI is of type SPIClass and if you expand the macros:
SPIClass SPI (&sercom4, 22, 24, 23, SPI_PAD_2_SCK_3, SERCOM_RX_PAD_0);

We can match this up with the pins we know are used for SPI:

PA12D22 / MISO   SERCOM2.0   SERCOM4.0
PB10D23 / MOSI   SERCOM4.2
PB11D24 / SCK    SERCOM4.3

You can see that the first argument (&sercom4) does in fact match our observation that SPI is on SERCOM4. The second, third and fourth arguments match with the MISO/SCK/MOSI pins.

The last two args are what actually define the pin muxing. We can see these definitions in SERCOM.h

```c
typedef enum
{
    SPI_PAD_0_SCK_1 = 0,
    SPI_PAD_2_SCK_3,
    SPI_PAD_3_SCK_1,
    SPI_PAD_0_SCK_3
} SercomSpiTXPad;
```

and

```c
typedef enum
{
    SERCOM_RX_PAD_0 = 0,
    SERCOM_RX_PAD_1,
    SERCOM_RX_PAD_2,
    SERCOM_RX_PAD_3
} SercomRXPad;
```

The two together let us define the muxing. For example:

- SPI_PAD_0_SCK_1 means MOSI is on SERCOMn.0 and SCK is on SERCOMn.1
- SPI_PAD_2_SCK_3 means MOSI is on SERCOMn.2 and SCK is on SERCOMn.3
- SPI_PAD_3_SCK_1 means MOSI is on SERCOMn.3 and SCK is on SERCOMn.1
- SPI_PAD_0_SCK_3 means MOSI is on SERCOMn.0 and SCK is on SERCOMn.3

Then we can define which pin is for MISO

- SERCOM_RX_PAD_0 means MISO on SERCOMn.0
- SERCOM_RX_PAD_1 means MISO on SERCOMn.1
- SERCOM_RX_PAD_2 means MISO on SERCOMn.2
- SERCOM_RX_PAD_3 means MISO on SERCOMn.3
As you can tell, you can't quite have every possibility, for example you cannot have SCK on pad 0 or pad 2. And you cant have MOSI on pad 1. But you do have a lot of options!

**SAMD51 SPI Pin Pads**

The SAMD51 is more restrictive than the SAMD21. While MISO can still be on any pad, there's only two possibilities for SCK & MOSI:

<table>
<thead>
<tr>
<th>DIPO[1:0]</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>PAD[0]</td>
<td>SERCOM PAD[0] is used as data input</td>
</tr>
<tr>
<td>0x1</td>
<td>PAD[1]</td>
<td>SERCOM PAD[1] is used as data input</td>
</tr>
<tr>
<td>0x2</td>
<td>PAD[2]</td>
<td>SERCOM PAD[2] is used as data input</td>
</tr>
<tr>
<td>0x3</td>
<td>PAD[3]</td>
<td>SERCOM PAD[3] is used as data input</td>
</tr>
</tbody>
</table>

SCK can only be on PAD 1 and MOSI can only be on PAD 0 or PAD 3!

**OK so let's make a new SPI SERCOM already**

ok ok so you are getting impatient! Lets try creating a new SERCOM and testing it out

Let's make a SPI device that 'mimics' the ATmega328 SPI pin numbering on Digital 11, 12, and 13:

- SCK on Digital 13
- MISO on Digital 12
- MOSI on Digital 11

Looking at our SERCOMs from the Mux table, we should go for SERCOM1
To put SCK signal on D13 (SERCOM1.1) and MOSI on D11 (SERCOM1.0) we want to use SPI_PAD_0_SCK_1 for the TX mux. That leaves on MISO to do, on D12 (SERCOM1.3), which is SERCOM_RX_PAD_3

So something like this:

```cpp
SPIClass SPI1 (&sercom1, 12, 13, 11, SPI_PAD_0_SCK_1, SERCOM_RX_PAD_3);
```

Let's test it out!

```cpp
#include <SPI.h>
SPIClass mySPI (&sercom1, 12, 13, 11, SPI_PAD_0_SCK_1, SERCOM_RX_PAD_3);

void setup() {
  Serial.begin(115200);
  mySPI.begin();
}

uint8_t i=0;
void loop() {
  Serial.println(i);
  mySPI.beginTransaction(SPISettings(8000000, MSBFIRST, SPI_MODE0));
  mySPI.transfer(i++);
  mySPI.endTransaction();
}
```

OK ... that won't work. Why? Good question! Check out those pin definitions in the variants.cpp file:

```cpp
const PinDescription g_APinDescription[] =
  ...{
      PORTA, 16, PIO TIMER, (PIN_ATTR_DIGITAL|PIN_ATTR_PWM|PIN_ATTR_TIMER),
      No_ADC_Channel, TCC2 CH0, TCC2 CH0, EXTERNAL INT 0 }, // TCC2/VO[0]
      PORTA, 19, PIO TIMER ALT, (PIN_ATTR_DIGITAL|PIN_ATTR_PWM|PIN_ATTR_TIMER ALT),
      No_ADC_Channel, PW0 CH3, TCC0 CH3, EXTERNAL INT 3 }, // TCC0/WO[3]

      // 13 (LED)
      PORTA, 17, PIO PWM, (PIN_ATTR_DIGITAL|PIN_ATTR_PWM), No_ADC_Channel, PW2 CH1,
      NOT ON TIMER, EXTERNAL INT 1 }, // TCC2/VO[1]
```

The first argument is the port (e.g. PORTA or PORTB), the second is the port's pin number, e.g. PORTA, 16 -> PA16. The third argument 'sets' the type of pinmux we're going to use.

In this case, these are set to be used as PIO_TIMER, PIO_TIMER_ALT and PIO_PWM
You can do that by calling `pinPeripheral(pinnumber, function)` which is an internal function but hey we're being hardk0re

```cpp
#include <SPI.h>
#include "wiring_private.h" // pinPeripheral() function

SPIClass mySPI (&sercom1, 12, 13, 11, SPI_PAD_0_SCK_1, SERCOM_RX_PAD_3);

void setup() {
  Serial.begin(115200);

  // do this first, for Reasons
  mySPI.begin();

  // Assign pins 11, 12, 13 to SERCOM functionality
  pinPeripheral(11, PIO_SERCOM);
  pinPeripheral(12, PIO_SERCOM);
  pinPeripheral(13, PIO_SERCOM);
}

uint8_t i=0;
void loop() {
  Serial.println(i);
  mySPI.beginTransaction(SPISettings(8000000, MSBFIRST, SPI_MODE0));
  mySPI.transfer(i++);
  mySPI.endTransaction();
}
```

You can use your oscilloscope to see the data traces

Say you only want to transmit SPI data, you can leave out the `pinPeripheral(12, PIO_SERCOM)` line, and you can keep using #12 however you like

We can also try SERCOM2:

<table>
<thead>
<tr>
<th>Pin Arduino 'Pin'</th>
<th>SERCOM</th>
<th>SERCOM alt</th>
</tr>
</thead>
<tbody>
<tr>
<td>PA14D2</td>
<td>SERCOM2.2</td>
<td>SERCOM4.2</td>
</tr>
<tr>
<td>PA09D3</td>
<td>SERCOM0.1</td>
<td>SERCOM2.1</td>
</tr>
<tr>
<td>PA08D4</td>
<td>SERCOM0.0</td>
<td>SERCOM2.0</td>
</tr>
<tr>
<td>PA15D5</td>
<td>SERCOM2.3</td>
<td>SERCOM4.3</td>
</tr>
</tbody>
</table>

Let's put SCK on D5 (SERCOM2.3), MISO on D3 (SERCOM2.1) and MOSI on D4 (SERCOM2.0)

```cpp
#include <SPI.h>
#include "wiring_private.h" // pinPeripheral() function

SPIClass mySPI (&sercom2, 3, 5, 4, SPI_PAD_0_SCK_3, SERCOM_RX_PAD_1);

void setup() {
  Serial.begin(115200);
```
// do this first, for Reasons
mySPI.begin();

// Assign pins 3, 4, 5 to SERCOM & SERCOM_ALT
pinPeripheral(3, PIO_SERCOM_ALT);
pinPeripheral(4, PIO_SERCOM_ALT);
pinPeripheral(5, PIO_SERCOM);
}

uint8_t i=0;
void loop() {
  Serial.println(i);
  mySPI.beginTransaction(SPISettings(8000000, MSBFIRST, SPI_MODE0));
  mySPI.transfer(i++);
  mySPI.endTransaction();
}

Note that since pin 3 and 4 use a SERCOM alt mux, we have to pass in PIO_SERCOM_ALT rather than PIO_SERCOM when we call pinPeripheral()

---

Creating a new Serial

Compared to SPI, UART is even simpler - it's only 2 pins, RX & TX. UARTs are a real pain to emulate or bitbang due to their very precise timing and their asynchronous RX lines are very difficult to do if you don't have hardware interrupt pins available.

For that reason, being able to create new Serial's is awesome

How Serial is Created Now

Luckily, Atmel & Arduino did a really great job with structuring SERCOMs so you can create and assign new ones. You can find the actual code that is used to create the "Serial" SERCOM in variants.cpp:

```cpp
Uart Serial1( &sercom0, PIN_SERIAL1_RX, PIN_SERIAL1_TX, PAD_SERIAL1_RX, PAD_SERIAL1_TX );
Uart Serial( &sercom5, PIN_SERIAL_RX, PIN_SERIAL_TX, PAD_SERIAL_RX, PAD_SERIAL_TX );
void SERCOM0_Handler()
{
  Serial1.IrqHandler();
}

void SERCOM5_Handler()
{
  Serial.IrqHandler();
}
```

If you have a Feather, you won't see the SERCOM5_Handler or Serial(&sercom5...) lines
The first hardware Serial object (on D0 and D1) is created with:

```c
Uart Serial1( &sercom0, PIN_SERIAL1_RX, PIN_SERIAL1_TX, PAD_SERIAL1_RX, PAD_SERIAL1_TX );
```

The actual definitions of those macros is available in `variants.h`:

```c
// Serial1
#define PIN_SERIAL1_RX       (0ul)
#define PIN_SERIAL1_TX       (1ul)
#define PAD_SERIAL1_TX       (UART_TX_PAD_2)
#define PAD_SERIAL1_RX       (SERCOM_RX_PAD_3)
```

You can see here that Serial1 is of type Uart and if you expand the macros:

```c
Uart Serial1( &sercom0, 0, 1, PAD_SERIAL1_RX, PAD_SERIAL1_TX ) ;
```

We can match this up with the pins we know are used for Serial1:

<table>
<thead>
<tr>
<th>Pin</th>
<th>SERCOM0.3</th>
<th>SERCOM2.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>PA11D0</td>
<td>SERCOM0.3</td>
<td>SERCOM2.3</td>
</tr>
<tr>
<td>PA10D1</td>
<td>SERCOM0.2</td>
<td>SERCOM2.2</td>
</tr>
</tbody>
</table>

You can see that the first argument (&sercom0) does in fact match our observation that Serial1 is on SERCOM0. The second and third arguments match with the RX & TX pins.

The last two args are what actually define the pin muxing. We can see these definitions in `SERCOM.h`:

```c
typedef enum
{
    SERCOM_RX_PAD_0 = 0,
    SERCOM_RX_PAD_1,
    SERCOM_RX_PAD_2,
    SERCOM_RX_PAD_3
} SercomRXPad;

typedef enum
{
    UART_TX_PAD_0 = 0x0ul, // Only for UART
    UART_TX_PAD_2 = 0x1ul, // Only for UART
    UART_TX_RTS_CTS_PAD_0 = 0x2ul, // Only for UART with TX on PAD0, RTS on PAD2 and CTS on PAD3
} SercomUartTXPad;
```

The two together let us define the muxing. For example we can define which pin is for RX:

- SERCOM_RX_PAD_0 means RX on SERCOMn.0
- SERCOM_RX_PAD_1 means RX on SERCOMn.1
SERCOM_RX_PAD_2 means RX on SERCOMn.2
SERCOM_RX_PAD_3 means RX on SERCOMn.3

and then for TX:

- UART_TX_PAD_0 means TX on SERCOMn.0
- UART_TX_PAD_2 means TX on SERCOMn.2

As you can tell, you can't quite have every possibility, for example you cannot have TX on pad 1 or pad 3. But you do have a lot of options!

**SAMD51 Serial SERCOM Pads**

The SAMD51 is a little more restrictive, you can only have TX on PAD 0 - PAD 2 is no longer permitted as it was on the SAMD21

![SERCOM Pin Muxing Diagram]

**OK so let's make a new Serial SERCOM already**

I understand, you want to make Serials! Let's try creating a new SERCOM and testing it out

Let's make a Serial device on SERCOM #1, let's look at what pin muxing options we've got:
Pin Arduino 'Pin' SERCOM SERCOM alt

<table>
<thead>
<tr>
<th>Pin</th>
<th>SERCOM 1.2</th>
<th>SERCOM 1.0</th>
<th>SERCOM 3.2</th>
<th>SERCOM 3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PA18D10</td>
<td>SERCOM1.2</td>
<td>SERCOM1.0</td>
<td>SERCOM3.2</td>
<td></td>
</tr>
<tr>
<td>PA16D11</td>
<td>SERCOM3.2</td>
<td></td>
<td></td>
<td>SERCOM3.0</td>
</tr>
<tr>
<td>PA19D12</td>
<td>SERCOM1.3</td>
<td>SERCOM1.1</td>
<td></td>
<td>SERCOM3.3</td>
</tr>
<tr>
<td>PA17D13</td>
<td>SERCOM1.1</td>
<td>SERCOM1.0</td>
<td>SERCOM3.1</td>
<td></td>
</tr>
</tbody>
</table>

We can put TX on D11 (SERCOM1.0) or D10 (SERCOM1.2), RX can be on any pin.

How about we have D10 be TX and D11 be RX?

That would mean something like:

```cpp
Uart Serial2 (&sercom1, 11, 10, SERCOM_RX_PAD_0, UART_TX_PAD_2);
```

Let's test it out!

```cpp
void SERCOM1_Handler()
{
  Serial2.IrqHandler();
}

void setup()
{
  Serial.begin(115200);
  Serial2.begin(115200);
}

uint8_t i=0;
void loop()
{
  Serial.println(i);
  Serial2.write(i++);
  delay(10);
}
```

aaaand...nothing! Why? Good question! Check out those pin definitions in the variants.cpp file:

```cpp
const PinDescription g_APinDescription[] =
...
{ PORTA, 18, PIO_TIMER, (PIN_ATTR_DIGITAL|PIN_ATTR_PWM|PIN_ATTR_TIMER),
  No_ADC_Channel, PWM3_CH0, TC3_CH0, EXTERNAL_INT_2 }, // TC3/WO[0]
{ PORTA, 16, PIO_TIMER, (PIN_ATTR_DIGITAL|PIN_ATTR_PWM|PIN_ATTR_TIMER),
  No_ADC_Channel, PWM2_CH0, TCC2_CH0, EXTERNAL_INT_0 }, // TCC2/WO[0]
```

The first argument is the port (e.g. PORTA or PORTB), the second is the port's pin number, e.g. PORTA, 18 -> PA18. The third argument 'sets' the type of pinmux we're going to use.

In this case, these are set to be used as PIO_TIMER!
You can do that by calling pinPeripheral(pinnumber, function) which is an internal function but hey we’re being hardk0re

```cpp
#include <Arduino.h> // required before wiring_private.h
#include "wiring_private.h" // pinPeripheral() function

Uart Serial2 (&sercom1, 11, 10, SERCOM_RX_PAD_0, UART_TX_PAD_2);
void SERCOM1_Handler()
{
    Serial2.IrqHandler();
}

void setup()
{
    Serial.begin(115200);

    Serial2.begin(115200);

    // Assign pins 10 & 11 SERCOM functionality
    pinPeripheral(10, PIO_SERCOM);
    pinPeripheral(11, PIO_SERCOM);
}

uint8_t i=0;
void loop()
{
    Serial.print(i);
    Serial2.write(i++);
    if (Serial2.available()) {
        Serial.print(" -> 0x"); Serial.print(Serial2.read(), HEX);
    }
    Serial.println();
    delay(10);
}
```

You can use your oscilloscope to see the data traces

Say you only want to transmit Serial data, you can leave out the pinPeripheral(10, PIO_SERCOM) line, and you can keep using #10 however you like

We can also try SERCOM2:

```
Pin  Arduino 'Pin'  SERCOM   SERCOM alt
----------------------------------------
PA14D2     SERCOM2.2  SERCOM4.2
PA09D3     SERCOM0.1  SERCOM2.1
PA08D4     SERCOM0.0  SERCOM2.0
PA15D5     SERCOM2.3  SERCOM4.3
```

Lets put TX on D4 (SERCOM2.0) and RX on D3 (SERCOM2.1)

```cpp
#include <Arduino.h> // required before wiring_private.h
#include "wiring_private.h" // pinPeripheral() function
```
Uart Serial2 (&sercom2, 3, 4, SERCOM_RX_PAD_1, UART_TX_PAD_0);
void SERCOM2_Handler()
{
    Serial2.IrqHandler();
}

void setup()
{
    Serial.begin(115200);
    Serial2.begin(115200);

    // Assign pins 3 & 4 SERCOM functionality
    pinPeripheral(3, PIO_SERCOM_ALT);
    pinPeripheral(4, PIO_SERCOM_ALT);
}

uint8_t i=0;
void loop()
{
    Serial.print(i);
    Serial2.write(i++);
    if (Serial2.available())
    {
        Serial.print(" -&gt; 0x"); Serial.print(Serial2.read(), HEX);
    }
    Serial.println();
    delay(10);
}

Note that since pin 3 and 4 use a SERCOM alt mux, we have to pass in
PIO_SERCOM_ALT rather than PIO_SERCOM when we call pinPeripheral()

**SAMD51 Interrupts**

The SAMD51 has 4 total interrupt handlers, you'll need to define all of them. Here's an
example for the SAMD51

```c
#define PIN_SERIAL2_RX 30  // PA05
#define PAD_SERIAL2_RX (SERCOM_RX_PAD_1)
#define PIN_SERIAL2_TX 18  // PA04
#define PAD_SERIAL2_TX (UART_TX_PAD_0)

Uart Serial2( &sercom0, PIN_SERIAL2_RX, PIN_SERIAL2_TX, PAD_SERIAL2_RX,
PAD_SERIAL2_TX );

Stream *SERIALOUT = &Serial2;

void SERCOM0_0_Handler()
{
    Serial2.IrqHandler();
}
void SERCOM0_1_Handler()
{
    Serial2.IrqHandler();
}
void SERCOM0_2_Handler()
{
    Serial2.IrqHandler();
}
void SERCOM0_3_Handler()
{  ```
Creating a new Wire

Compared to SPI, I2C (a.k.a Two-Wire or sometimes just referred to it's Arduino name, Wire) is even simpler - it's only 2 pins, SCL and SDA. I2C ports are a huge pain to emulate or bitbang due to the somewhat convoluted interface with multiple exceptions (repeated start! clock stretching!) and the bidirectional SDA line.

For that reason, being able to create new Wire's is awesome

How Wire is Created Now

Luckily, Atmel & Arduino did a really great job with structuring SERCOMs so you can create and assign new ones. You can find the actual code that is used to create the "Wire" SERCOM in `Wire.cpp`

```cpp
TwoWire Wire(&PERIPH_WIRE, PIN_WIRE_SDA, PIN_WIRE_SCL);
void WIRE_IT_HANDLER(void) {
  Wire.onService();
}
```

The macros are see in `variants.h`

```cpp
#define PIN_WIRE_SDA         (20u)
#define PIN_WIRE_SCL         (21u)
#define PERIPH_WIRE          sercom3
#define WIRE_IT_HANDLER      SERCOM3_Handler
```

You can see here that Wire is of type TwoWire and if you expand the macros:

```cpp
TwoWire Wire(&sercom3, 20, 21);
```

We can match this up with the pins we know are used for I2C:

```plaintext
PA22D20 / SDA   SERCOM3.0   SERCOM5.0
PA23D21 / SCL   SERCOM3.1   SERCOM5.1
```

You can see that the first argument (&sercom3) does in fact match our observation that I2C is on SERCOM3. The second and third arguments match with the SDA & SCL pins.
Unlike Serial or SPI, there are no passed in arguments for the MUXing. I2C, rather, has 'fixed' pads:

Pad 0 will always be SDA and pad 1 will always be SCL

Both the SAMD21 and SAMD51 have the same restrictions for SDA/SCL pads

OK so let's make a new I2C SERCOM already

Now we're ready, let's try creating a new SERCOM and testing it out

Let's make a Wire device on SERCOM 1

Since we have to use pads 0 & 1, let's check the mux table:

<table>
<thead>
<tr>
<th>Pin Arduino 'Pin'</th>
<th>SERCOM</th>
<th>SERCOM alt</th>
</tr>
</thead>
<tbody>
<tr>
<td>PA18D10</td>
<td>SERCOM1.2</td>
<td>SERCOM3.2</td>
</tr>
<tr>
<td>PA16D11</td>
<td>SERCOM1.0</td>
<td>SERCOM3.0</td>
</tr>
<tr>
<td>PA19D12</td>
<td>SERCOM1.3</td>
<td>SERCOM3.3</td>
</tr>
<tr>
<td>PA17D13</td>
<td>SERCOM1.1</td>
<td>SERCOM3.1</td>
</tr>
</tbody>
</table>

SDA will be on SERCOM1.0 D11 and SCL will be on SERCOM1.1 D13

The definition for the new Wire will look like this:

```c
TwoWire Wire(&sercom1, 11, 13);
```

Let's test it out!

```c
#include <Wire.h>
TwoWire myWire(&sercom1, 11, 13);
void setup() {
    Serial.begin(115200);
    myWire.begin();
```
uint8_t i=0;
void loop() {
  Serial.println(i);
  myWire.beginTransmission(0x55); // start transmission to device
  myWire.write(i); // send a byte
  myWire.endTransmission(); // end transmission, actually sending
}

OK ... that wont work. Why? Good question! Check out those pin definitions in the variants.cpp file:

```
const PinDescription g_APinDescription[] =
  {.PORTA, 16, PIO_TIMER, (PIN_ATTR_DIGITAL|PIN_ATTR_PWM|PIN_ATTR_TIMER),
    No_ADC_Channel, PWM2_CH0, TCC2_CH0, EXTERNAL_INT_0 }, // TCC2/WO[0]
  {.PORTA, 17, PIO_PWM, (PIN_ATTR_DIGITAL|PIN_ATTR_PWM), No_ADC_Channel, PWM2_CH1,
    NOT_ON_TIMER, EXTERNAL_INT_1 }, // TCC2/WO[1]
```

The first argument is the port (e.g. PORTA or PORTB), the second is the port's pin number, e.g. PORTA, 16 -> PA16. The third argument 'sets' the type of pinmux we're going to use.

In this case, these are set to be used as PIO_TIMER & PIO_PWM

We have to tell the Arduino core to change the MUX type before we reassign the SERCOM to this pin

You can do that by calling pinPeripheral(pinnumber, function) which is an internal function but hey we're being hardk0re

Note that I've also put in code to control an MCP4725 by hand, since its a very simple I2C device (write 2 bytes to 0x40 to set an analog voltage out)

```
#include <Wire.h>
#include "wiring_private.h" // pinPeripheral() function

TwoWire myWire(&sercom1, 11, 13);
#define MCP4725_CMD_WRITEDAC (0x40)
#define MCP4725_ADDR (0x62)

void setup() {
  Serial.begin(115200);
  myWire.begin();

  // Assign pins 13 & 11 to SERCOM functionality
  pinPeripheral(11, PIO_SERCOM);
  pinPeripheral(13, PIO_SERCOM);
}

uint8_t i=0;
void loop() {
```
```c
Serial.println(i);
myWire.beginTransmission(MCP4725_ADDR); // start transmission to device
myWire.write(MCP4725_CMD_WRITEDAC);
myWire.write(i++);
myWire.write((uint8_t)0x0);  // bottom four bits are 0x0
myWire.endTransmission(); // end transmission, actually sending
```

2-10K pullup resistors are required on SDA and SCL, both go to 3.3V! You can use your oscilloscope to see the data traces

We can also try SERCOM2:

<table>
<thead>
<tr>
<th>Pin Arduino 'Pin'</th>
<th>SERCOM</th>
<th>SERCOM alt</th>
</tr>
</thead>
<tbody>
<tr>
<td>PA14D2</td>
<td>SERCOM2.2</td>
<td>SERCOM4.2</td>
</tr>
<tr>
<td>PA09D3</td>
<td>SERCOM0.1</td>
<td>SERCOM2.1</td>
</tr>
<tr>
<td>PA08D4</td>
<td>SERCOM0.0</td>
<td>SERCOM2.0</td>
</tr>
<tr>
<td>PA15D5</td>
<td>SERCOM2.3</td>
<td>SERCOM4.3</td>
</tr>
</tbody>
</table>

We will have to have SDA on D4 (SERCOM2.0), SCL on D3 (SERCOM2.1)

```c
#include "Wire.h"
#include "wiring_private.h" // pinPeripheral() function

TwoWire myWire(&sercom2, 4, 3);

#define MCP4725_CMD_WRITEDAC (0x40)
#define MCP4725_ADDR (0x62)

void setup() {
  Serial.begin(115200);
  myWire.begin();

  // Assign pins 4 & 3 to SERCOM functionality
  pinPeripheral(4, PIO_SERCOM_ALT);
  pinPeripheral(3, PIO_SERCOM_ALT);
}

uint8_t i=0;
void loop() {
  Serial.println(i);
  myWire.beginTransmission(MCP4725_ADDR); // start transmission to device
  myWire.write(MCP4725_CMD_WRITEDAC);
  myWire.write(i++);
  myWire.write((uint8_t)0x0);  // bottom four bits are 0x0
  myWire.endTransmission(); // end transmission, actually sending
}
```

Note that since pin 3 and 4 use a SERCOM alt mux, we have to pass in PIO_SERCOM_ALT rather than PIO_SERCOM when we call pinPeripheral()