- a07g-exploring-the-CLI
  - 0 Install Percepio
  - 1 Software Architecture
    - Overall Project Updates / Differences from A00G proposal
    - Updated Hardware Requirements Specification (HRS)
    - Updated Software Requirements Specification (SRS)
    - Block Diagram for software tasks
    - Flowcharts and State Machines
      - SAMD21 Baremetal Super Loop
  - 2 Understanding the Starter Code
    - 1: What does “InitializeSerialConsole()” do? In said function, what is “cbufRx” and “cbufTx”? What type of data structure is it?
    - 2: How are “cbufRx” and “cbufTx” initialized? Where is the library that defines them (please list the *C file they come from)
    - 3: Where are the character arrays where the RX and TX characters are being stored at the end? Please mention their name and size
    - 4: Where are the interrupts for UART character received and UART character sent defined?
    - 5: What are the callback functions that are called when
      - A character is received? (RX)
      - A character has been sent? (TX)
    - 6: Explain what is being done on each of these two callbacks and how they relate to the cbufRx and cbufTx buffers
      - usart_read_callback()
      - usart_write_callback()
    - 7: Draw a diagram that explains the program flow for UART receive – starting with the user typing a character and ending with how that characters ends up in the circular buffer “cbufRx”. Please make reference to specific functions in the starter code
    - 8: Draw a diagram that explains the program flow for the UART transmission – starting from a string added by the program to the circular buffer “cbufTx” and ending on characters being shown on the screen of a PC (On Teraterm, for example). Please make reference to specific functions in the starter code
    - 9: What is done on the function “startStasks()” in main.c? How many threads are started?
  - 3 Debug Logger Module
  - 4 Wiretap the convo

a07g-exploring-the-CLI

- Team Number: 9
- Team Name: Shorts & Sparks
- Team Members: James Steeman and Timothy Zhang
- GitHub Repository URL: https://github.com/ese5160/final-project-a07g-a14g-t09-shorts-sparks
- Description of test hardware: Macbook pro m2 (macOS) and lab desktops (windows) (development boards, sensors, actuators, laptop + OS, etc)

0 Install Percepio

1 Software Architecture

Overall Project Updates / Differences from A00G proposal

Original	Current
Foam cyllinders 3d model prototypes	Copper clad FR4 for milling PCB prototypes

Original	Current
r(z), y, theta (cyllinder) cnc	x, y, z cnc
Battery backup power in wall power outage case	Approved removal of battery as PCB milling times are faster so it is less expected and a less significant issue to lose power in the middle of a job

Updated Hardware Requirements Specification (HRS)

Req ID	Requirement	Review
HRS-01	The CNC router shall use 3 stepper motors to control X, Y, Z axis	N/A
HRS-02	The CNC router shall use 1 DC motor for spindle	N/A
HRS-03	The DC spindle shall be driven with a mosfet driver to allow for PWM speed control	N/A
HRS-04	The system shall have external non-voilatile memory (microSD) of no less than 512MB for storing G-code and current progress (in any pause scenario)	N/A
HRS-05	The CNC router shall be able to engrave FR4 copper boards	N/A
HRS-06	The CNC shall have a milling area of at least 100 by 70mm (most common FR4 board size)	N/A
HRS-07	The CNC should have a milling area of at least 150 by 100mm (some larger boards have this dimention)	N/A
HRS-08	The CNC shall have a veritcal Z axis travel of at least 30mm (length of an engraving bit)	N/A
HRS-09	The CNC shall run off a wall outlet during operation	N/A

Req ID	Requirement	Review
HRS-10	The CNC shall operate on 24V DC and draw no more than 10 A	N/A
HRS-11	The system shall use the SAMW25 as the main microcontroller and Wi-Fi communication IC	N/A
HRS-12	The stepper motors shall use encoders for closed loop feedback	N/A
HRS-13	A SAMD21 for each axis shall handle the closed loop control	N/A
HRS-14	Each axis shall have limit switches	N/A
HRS-15	Each axis shall use a leadscrew drive with linear rods for accuracy and rigidity	N/A
HRS-16	The system shall have user buttons on the PCB	N/A
HRS-17	The system shall have status indicator LEDs on the PCB	N/A
HRS-18	The system shall have a power (on/off) switch	N/A

Updated Software Requirements Specification (SRS)

Req ID	Requirement	Review
SRS-01	The system shall be able to control all 3 stepper motors in its cyllindrical coordinates	N/A
SRS-02	The MCU in the SAMW25 module shall run an RTOS	N/A
SRS-03	The user shall be able to remotely operate the machine, setting presets, start/pause etc	N/A

Req ID	Requirement	Review
SRS-04	The system shall be able to send status updates and job progress to the web portal for the user	N/A
SRS-05	The system shall be able to read and interoperate encoder values as positions	N/A
SRS-06	The system shall be able to automatically pause the job when hitting limit switch	N/A
SRS-07	The system shall be able to perform X/Y zeroing (calibration) with limit switches	N/A
SRS-08	The system shall be able to automatically perform bed level calibration using the contuctive nature of the endmill and copper FR4 board	N/A
SRS-09	The webpage portal shall handle remote uploading of jobs	N/A
SRS-10	The webpage portal shall handle GCode generation from PCB gerber files leveraging existing software (e.g. FlatCAM)	N/A
SRS-11	The webpage portal shall handle GCode to motor instruction conversion	N/A
SRS-12	An OTAFU shall be implemented	N/A
SRS-13	A CLI shall be implemented	N/A
SRS-14	The communication between MCUs shall use I2C (and some gpio)	N/A
SRS-15	The communication between MUC and SD card shall use SPI	N/A

Block Diagram for software tasks

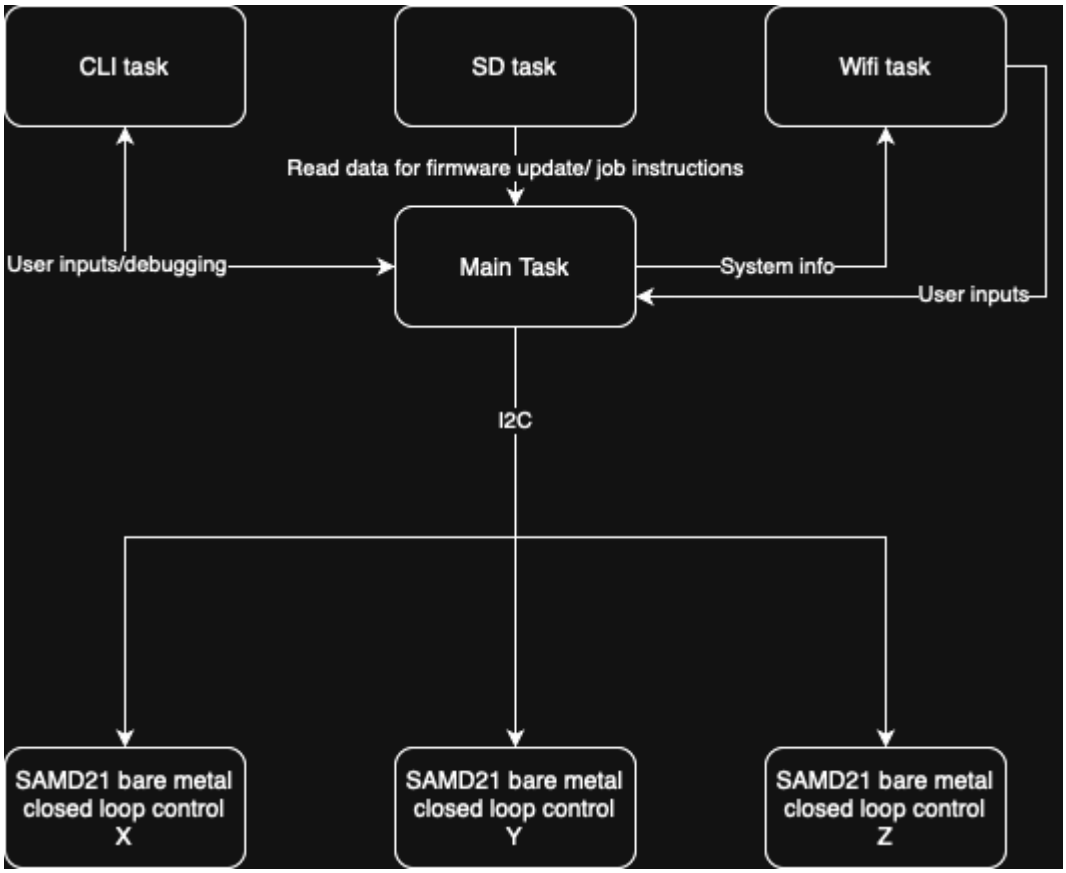
Due to the nature of this project having a lot of exceptions, the actual "tasks" handled by the main MCU is not a lot besides the wifi, cli, and SD card. The main MCU serves as a

hub for data distribution (like an orchestra conductor if you will) as it simply sends instructions to corresponding sub MCUs for closed loop feedback.

The actual sensor/actuators are handled by the SAMD21s in bare metal, and the generation of GCode/conversion to motor instructions happen in the web portal before reaching the SAMW25.

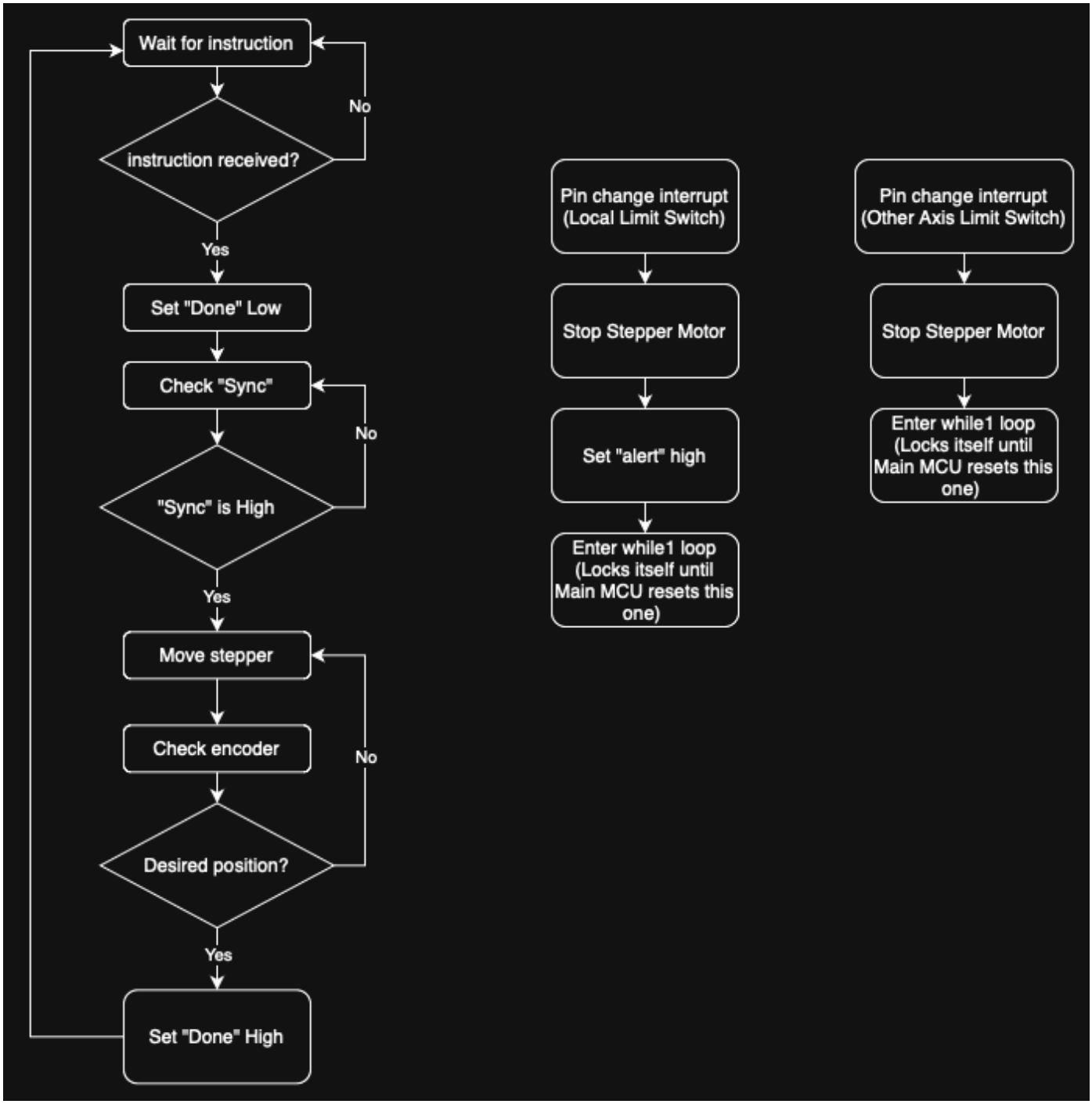
Hence, unfortunately this makes it seem like not a real "RTOS" as all of the implementation happens in a single task.

The next setions shoes more in depth the specifics of the bare metal closed loop motor control flow chart and the main task flow diagram.



Flowcharts and State Machines

SAMD21 Baremetal Super Loop



2 Understanding the Starter Code

1: What does “InitializeSerialConsole()” do? In said function, what is “cbufRx” and “cbufTx”? What type of data structure is it?

It initializes the UART circular buffers (RX and TX), configures the usart module and callback functions, set the priority for USART interrupts, and begins reading to the buffer.

```
/**
 * @brief Initializes the UART and registers callbacks.
 */
void InitializeSerialConsole(void)
{
    // Initialize circular buffers for RX and TX
   cbufRx = circular_buf_init((uint8_t *)rxCharacterBuffer, RX_BUFFER_SIZE);
   cbufTx = circular_buf_init((uint8_t *)txCharacterBuffer, TX_BUFFER_SIZE);

    // Configure USART and Callbacks
   configure_usart();
    configure_usart_callbacks();
    NVIC_SetPriority(SERCOM4_IRQn, 10);

    usart_read_buffer_job(&usart_instance, (uint8_t *)&latestRx, 1); //
Kicks off constant reading of characters

   // Add any other calls you need to do to initialize your Serial Console
}
```

cbufRx and cbufTx are circular buffer handlers: pointers to the circular buffer structure defined in circular_buffer.c

```
// The definition of our circular buffer structure is hidden from the user
struct circular_buf_t {
  uint8_t * buffer;
   size_t head;
   size_t tail;
   size_t max; //of the buffer
   bool full;
};
```

2: How are “cbufRx” and “cbufTx” initialized? Where is the library that defines them (please list the *C file they come from)

They are initialized by calling a function circular_but_init() with parameters cooresponding to rx and tx and their buffer sizes, respectively. This function is defined in circular_buffer.c. It creates an instance of the cbuf type struct using malloc to allocate sufficient memory on the heap. Then it sets the values of the elements of the struct accordingly with the parameters of the function call, and returns the newly created cbuf by reference (cbuf_handle_t is a pointer to the memory where the data is stored).

in serial console.c

```
cbufRx = circular_buf_init((uint8_t *)rxCharacterBuffer, RX_BUFFER_SIZE);
cbufTx = circular_buf_init((uint8_t *)txCharacterBuffer, TX_BUFFER_SIZE);
```

in circular_buffer.c

```
// The definition of our circular buffer structure is hidden from the user
struct circular_buf_t {
   uint8_t * buffer;
   size_t head;
   size_t tail;
   size_t max; //of the buffer
   bool full;
};
```

and

```
cbuf_handle_t circular_buf_init(uint8_t* buffer, size_t size)
{
   // assert(buffer && size);

   cbuf_handle_t cbuf = malloc(sizeof(circular_buf_t));
   //assert(cbuf);

   cbuf->buffer = buffer;
   cbuf->max = size;
   circular_buf_reset(cbuf);

   // assert(circular_buf_empty(cbuf));

   return cbuf;
}
```

3: Where are the character arrays where the RX and TX characters are being stored at the end? Please mention their name and size

Tip: Please note cBufRx and cBufTx are structures.

The characters are being stored in the data segment (global variables) in rxCharacterBuffer and txCharacterBuffer respectively. The size of each is defined by a corresponding macro: RX_BUFFER_SIZE and TX_BUFFER_SIZE. Each array is 512 characters.

in SerialConsole.c

```
char rxCharacterBuffer[RX_BUFFER_SIZE];    ///< Buffer to store received
characters
char txCharacterBuffer[TX_BUFFER_SIZE];    ///< Buffer to store characters
to be sent
```

4: Where are the interrupts for UART character received and UART character sent defined?

The usart interrupt handler is defined in _usart_interrupt_handler. The interrupt handler function checks a variety of usart status codes to handle tx and rx communication by calling the approprate callback funtions based on system status.

in usart_interrupt.c

```
/**
 * \internal
 * Handles interrupts as they occur, and it will run callback functions
 * which are registered and enabled.
 *
 * \param[in]  instance  ID of the SERCOM instance calling the interrupt
 *                       handler.
 */
void _usart_interrupt_handler(uint8_t instance)
   {
      /* omitted implementation in READEME.md for brevity */
   }
```

See line 451 of usart_interrupt.c for full function implementation.

This is set to the be the handler function for the corresponding sercom in usart.c

```
_sercom_set_handler(instance_index, _usart_interrupt_handler);
```

5: What are the callback functions that are called when

A character is received? (RX)

usart_read_callback()

A character has been sent? (TX)

usart_write_callback()

```
/*****************************************************************************
***
 * Callback Functions

*****************************************************************************
**/

/*****************************************************************************/
/**
 * @fn     void usart_read_callback(struct usart_module *const usart_module)
 * @brief  Callback called when the system finishes receives all the bytes
requested from a UART read job
           Students to fill out. Please note that the code here is dummy
code. It is only used to show you how some functions work.
 * @note

*****************************************************************************
*/
void usart_read_callback(struct usart_module *const usart_module)
{
   // ToDo: Complete this function
}

/*****************************************************************************/
/**
 * @fn         void usart_write_callback(struct usart_module *const
usart_module)
 * @brief      Callback called when the system finishes sending all the
bytes requested from a UART read job
 * @note

*****************************************************************************
*/
void usart_write_callback(struct usart_module *const usart_module)
{
   if (circular_buf_get(cbufTx, (uint8_t *)&latestTx) != -1) // Only
continue if there are more characters to send
   {
      usart_write_buffer_job(&usart_instance, (uint8_t *)&latestTx, 1);
   }
}
```

6: Explain what is being done on each of these two callbacks and how they relate to the cbufRx and cbufTx buffers

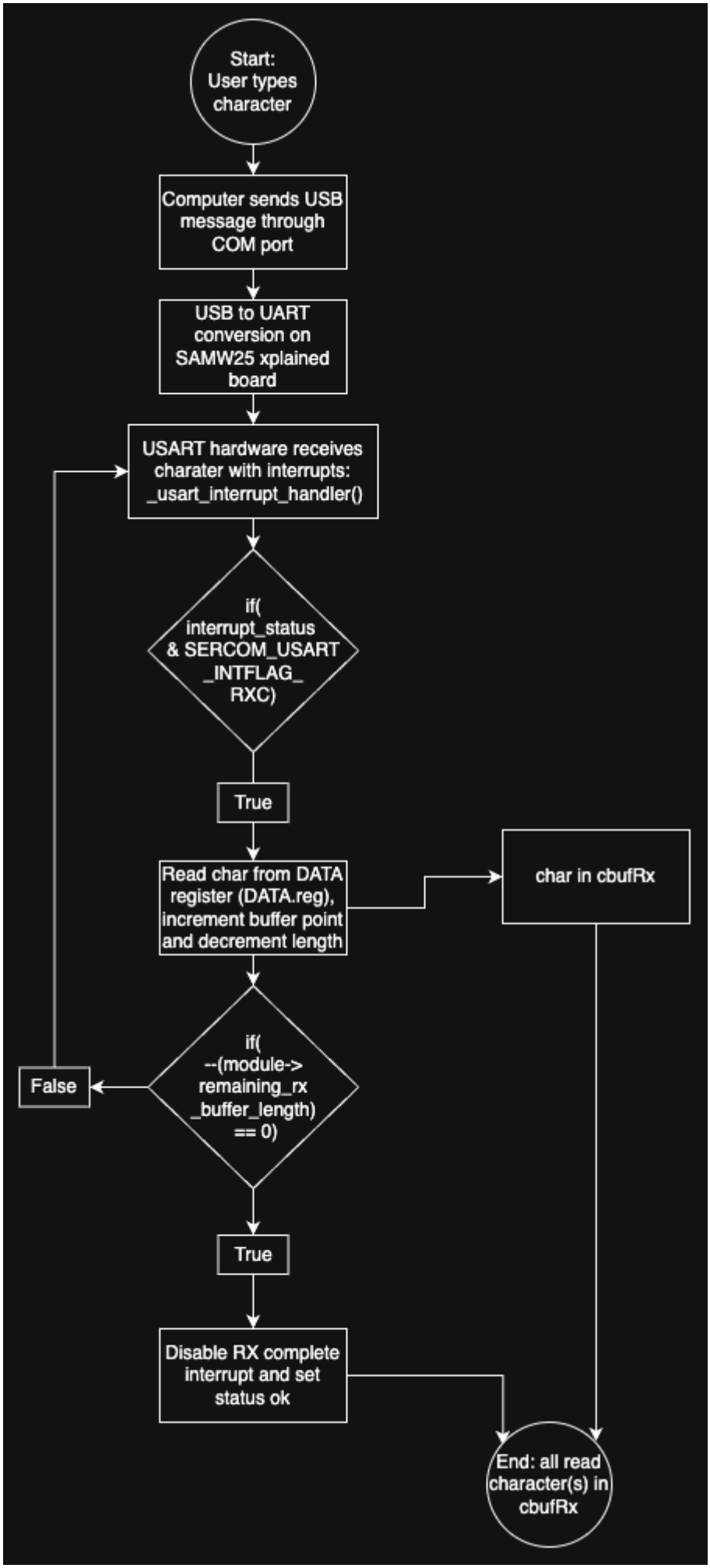
usart_read_callback()

This callback function processes data received on RX by reading from what is stored in the cbufRx buffer, until the reading pointer meets to the write/data pointer.

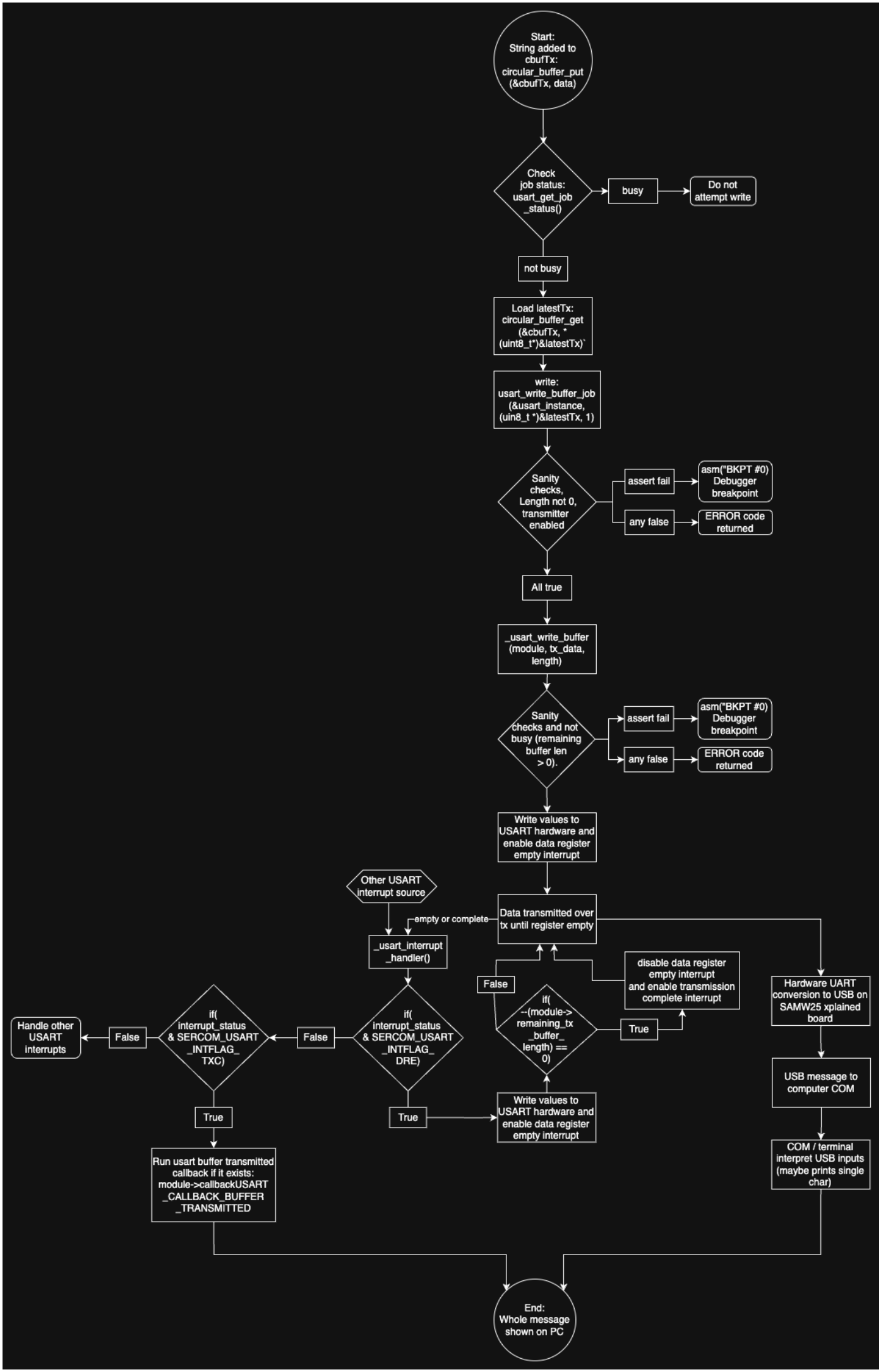
usart_write_callback()

This callback function processes data to send on TX by reading from what is stored in the cbufTx buffer. If there are more characters to send, it will continue the transmission on TX.

7: Draw a diagram that explains the program flow for UART receive – starting with the user typing a character and ending with how that characters ends up in the circular buffer “cbufRx”. Please make reference to specific functions in the starter code


Start:
User types
character
Computer sends USB
message through
COM port
USB to UART
conversion on
SAMW25 xplained
board
USART hardware receives
charater with interrupts:
_usart_interrupt_handler()
if(
interrupt_status
& SERCOM_USART
INTFLAG
RXC)
True
Read char from DATA
register (DATA.reg),
increment buffer point
and decrement length
char in cbufRx
if(
--(module->
remaining_rx
_buffer_length)
== 0)
False
True
Disable RX complete
interrupt and set
status ok
End: all read
character(s) in
cbufRx

8: Draw a diagram that explains the program flow for the UART transmission – starting from a string added by the program to the circular buffer “cbufTx” and ending on characters being shown on the screen of a PC (On Teraterm, for example). Please make reference to specific functions in the starter code


Start:
String added to cbufTx:
circular_buffer_put (&cbufTx, data)
Check job status:
usart_get_job _status()
busy
Do not attempt write
not busy
Load latestTx:
circular_buffer_get (&cbufTx, *(uint8_t*)&latestTx)`
write:
usart_write_buffer_job (&usart_instance, (uin8_t *)&latestTx, 1)
Sanity checks, Length not 0, transmitter enabled
assert fail
asm("BKPT #0) Debugger breakpoint
any false
ERROR code returned
All true
_usart_write_buffer (module, tx_data, length)
Sanity checks and not busy (remaining buffer len > 0).
assert fail
asm("BKPT #0) Debugger breakpoint
any false
ERROR code returned
Write values to USART hardware and enable data register empty interrupt
Other USART interrupt source
empty or complete
Data transmitted over tx until register empty
_usart_interrupt _handler()
if(interrupt_status & SERCOM_USART _INTFLAG_ DRE)
False
if(interrupt_status & SERCOM_USART _INTFLAG_ TXC)
False
Handle other USART interrupts
True
Run usart buffer transmitted callback if it exists: module->callbackUSART _CALLBACK_BUFFER _TRANSMITTED
True
Write values to USART hardware and enable data register empty interrupt
if(--(module-> remaining_tx _buffer_ length) == 0)
False
True
disable data register empty interrupt and enable transmission complete interrupt
Hardware UART conversion to USB on SAMW25 xplained board
USB message to computer COM
COM / terminal interpret USB inputs (maybe prints single char)
End:
Whole message shown on PC

9: What is done on the function “startStasks()” in main.c? How many threads are started?

There is a StartTasks() function in main.c that gets and prints (to serial console) the current free heap size before starting the tasks, initizlizes tasks and gets and prints coorepsponding free heap size after each task. Currently, there is only one thread started, the CLI_TASK.

```
/*****************************************************************************/
/**
 * function          StartTasks
 * @brief            Initialize application tasks in this function
 * @details
 * @param[in]        None
 * @return           None

******************************************************************************
*/
static void StartTasks(void)
{

   snprintf(bufferPrint, 64, "Heap before starting tasks: %d\r\n",
xPortGetFreeHeapSize());
   SerialConsoleWriteString(bufferPrint);

   // CODE HERE: Initialize any Tasks in your system here

   if (xTaskCreate(vCommandConsoleTask, "CLI_TASK", CLI_TASK_SIZE, NULL,
CLI_PRIORITY, &cliTaskHandle) != pdPASS)
   {
      SerialConsoleWriteString("ERR: CLI task could not be
initialized!\r\n");
   }

   snprintf(bufferPrint, 64, "Heap after starting CLI: %d\r\n",
xPortGetFreeHeapSize());
   SerialConsoleWriteString(bufferPrint);
}
```

3 Debug Logger Module

Commit your functioning Debug Logger Module to your GitHub repo, and make comments that are in Doxygen style.

LogMessage() function completed in SerialConsole.c

```
/**
 * @brief Logs a message at the specified debug level.
 */
void LogMessage(enum eDebugLogLevels level, const char *format, ...)
{
   if (level >= getLogLevel()) {
      char buffer[TX_BUFFER_SIZE];  ///< Buffer to hold the formatted log
message
      va_list arguments;            ///< Variable argument list

      // Start the argument list after format
      va_start(arguments, format);
      // Reformat the message (safely) and store it in the buffer
      vsnprintf(buffer, TX_BUFFER_SIZE, format, arguments);
      // Send formatted message in variable buffer to serial console
      SerialConsoleWriteString(buffer);
      va_end(arguments);
   }
}
```

4 Wiretap the convo

Questions

1: What nets must you attach the logic analyzer to? (Check how the firmware sets up the UART in SerialConsole.c!)

in SerialConsole.c

```
static void configure_usart(void)
{
   struct usart_config config_usart;
   usart_get_config_defaults(&config_usart);

   config_usart.baudrate = 115200;
   config_usart.mux_setting = EDBG_CDC_SERCOM_MUX_SETTING;
   config_usart.pinmux_pad0 = EDBG_CDC_SERCOM_PINMUX_PAD0;
   config_usart.pinmux_pad1 = EDBG_CDC_SERCOM_PINMUX_PAD1;
   config_usart.pinmux_pad2 = EDBG_CDC_SERCOM_PINMUX_PAD2;
   config_usart.pinmux_pad3 = EDBG_CDC_SERCOM_PINMUX_PAD3;
   while (usart_init(&usart_instance,
                  EDBG_CDC_MODULE,
                  &config_usart) != STATUS_OK)
   {
```

```
    }

    usart_enable(&usart_instance);
}
```

right click and finding the definition for any of the EDBG_CDC_Sercom ... macros leads to the following:

in samw25_xplained_pro.h

```
/** \name Embedded debugger CDC Gateway USART interface definitions
 * @{
 */
#define EDBG_CDC_MODULE SERCOM4
#define EDBG_CDC_SERCOM_MUX_SETTING USART_RX_3_TX_2_XCK_3
#define EDBG_CDC_SERCOM_PINMUX_PAD0 PINMUX_UNUSED
#define EDBG_CDC_SERCOM_PINMUX_PAD1 PINMUX_UNUSED
#define EDBG_CDC_SERCOM_PINMUX_PAD2 PINMUX_PB10D_SERCOM4_PAD2
#define EDBG_CDC_SERCOM_PINMUX_PAD3 PINMUX_PB11D_SERCOM4_PAD3
```

This shows that we are using SERCOM4, with RX on PAD 3 and pin PB11, and with TX on PAD 2 and pin PB10. Additonally, at least one ground connection must be made to the logic analyzer to share common ground between the samw system and the logic anlyzer. All channels' grounds are internally connected in the logic anlyzer so a single ground connection suffices.

Alternatively, pins could be soldered onto the 3 pin UART Debug section that has GND, TX, RX, but for soldering simplicity, I will try to use the existing header pins first.

2: Where on the circuit board can you attach / solder to?

Both PB10 and PB11 are broken out to the header block on the xplained pro dev board, so connections can be attached to these header pins.

3: What are critical settings for the logic analyzer?

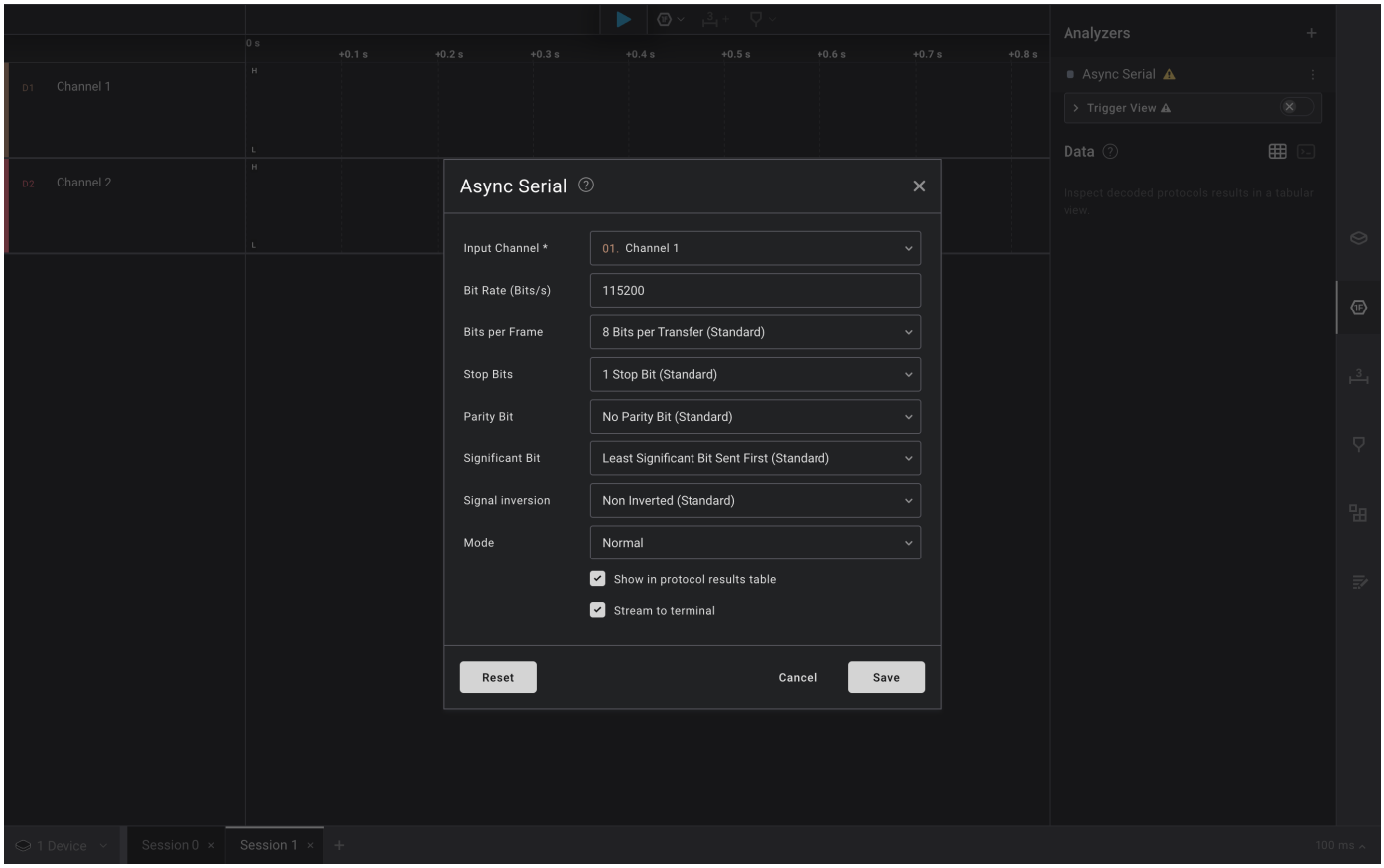
The critical setting for the logic analyzer are setting both pins active and setting them up with async serial analyzers. In the async serial analyzer, the most crictal settings are the baud/bit rate and other settings defiing the implementation of the UART protocol, including bits per frame, number of stop bit, parity bit presence or lack thereof, LSB/MSB ordering, and signal inverting.

From usart.c we can see the protocol defaults, which were set initially in the configure_usart() function, which then redefines baud rate to 115200 and sets the pad/pins to finish configuring the hardware module.

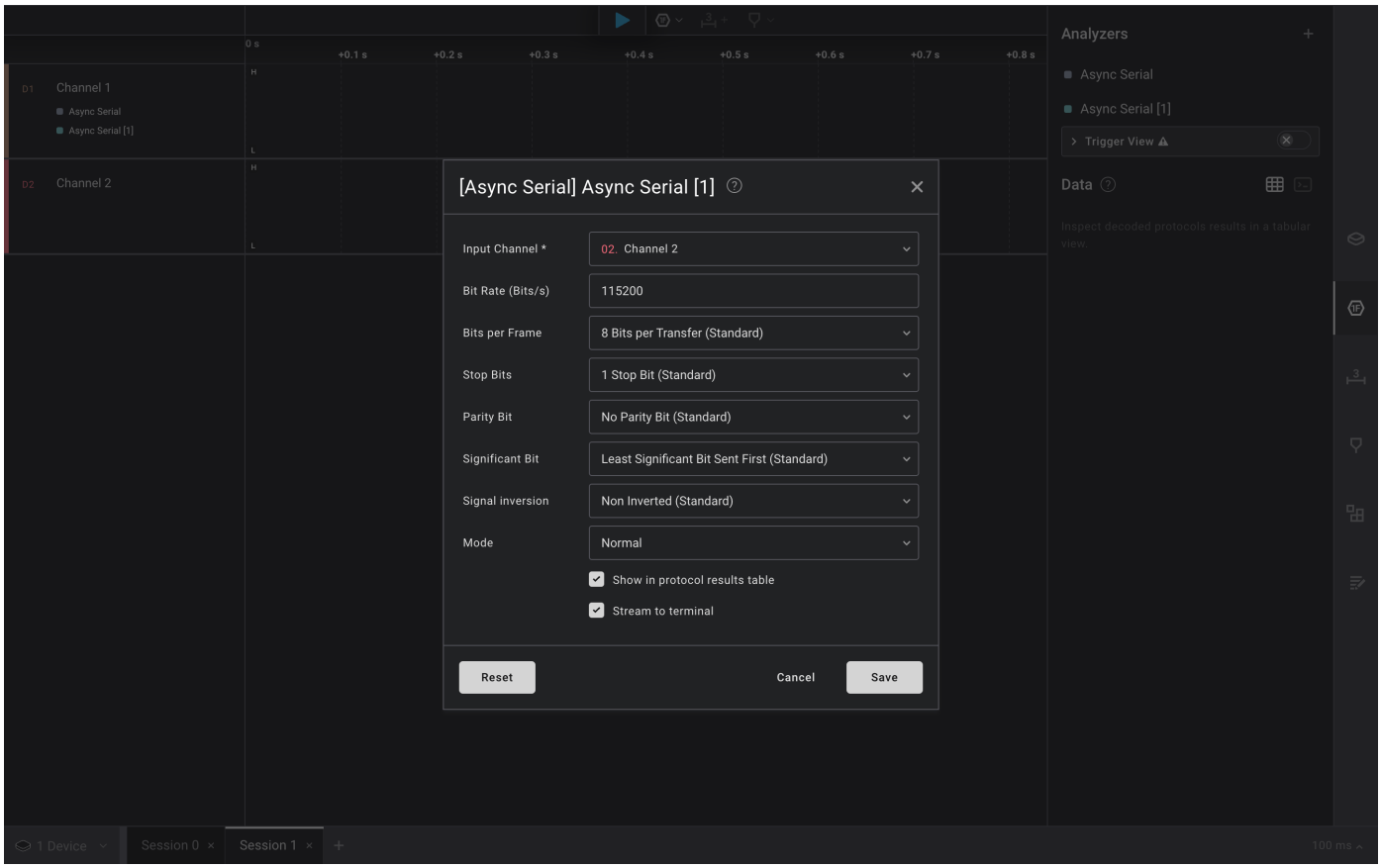
```
/**
 * \brief Initializes the device to predefined defaults
 *
 * Initialize the USART device to predefined defaults:
 * - 8-bit asynchronous USART
 * - No parity
 * - One stop bit
 * - 9600 baud
 * - Transmitter enabled
 * - Receiver enabled
 * - GCLK generator 0 as clock source
 * - Default pin configuration
 *
 * The configuration struct will be updated with the default
 * configuration.
 *
 * \param[in,out] config  Pointer to configuration struct
 */
static inline void usart_get_config_defaults(
      struct usart_config *const config) {/*omitted*/}
```

These protocol defaults are reflected in the settings chosen for the logic analyzer, shown in the screenshots below.

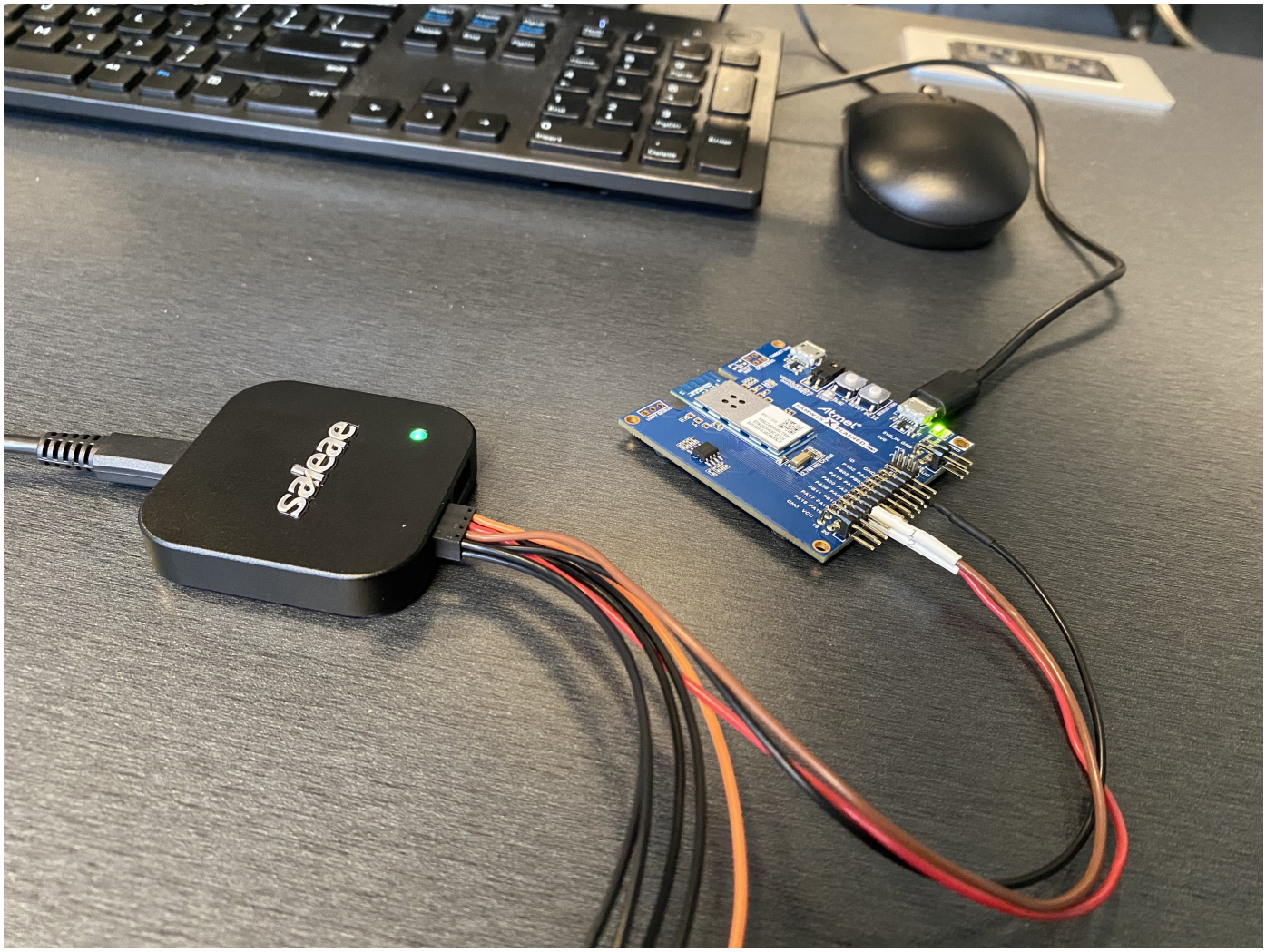
Channel 1: **RX**



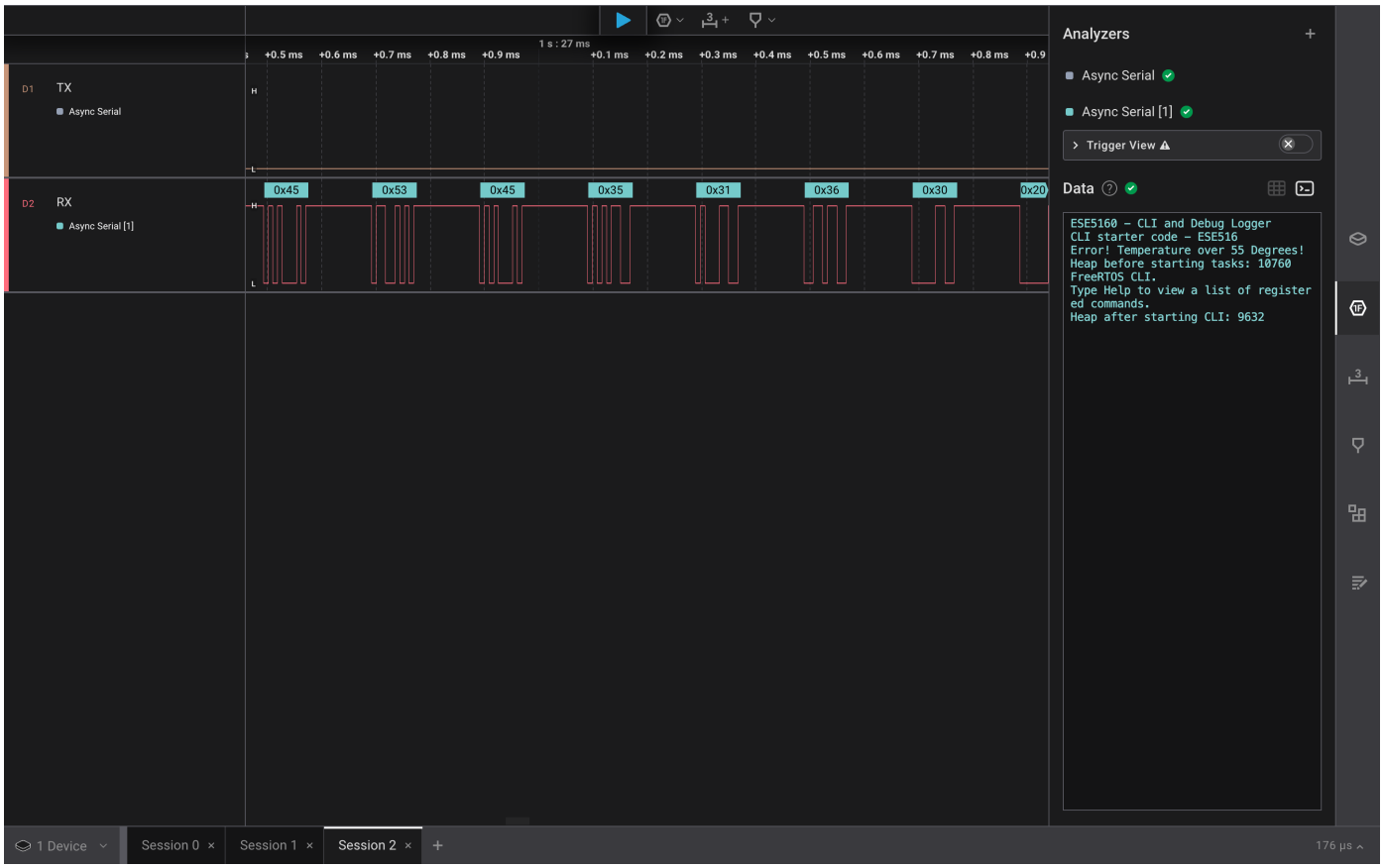
Channel 2: **TX**



Hardware Photo



Decoded Screenshot



Small Capture .sal file

SAL file for capture before entire CLI implementation (just the startup messages)

5 Complete the CLI

Commit your functioning CLI code to your GitHub repo, and make comments that are in Doxygen style.

CLI Starter Code Folder

6 Add CLI commands

Code

CLI Starter Code Folder

Video Link

Drive Video

7 Using Percepio

Removed from assignment requirements.