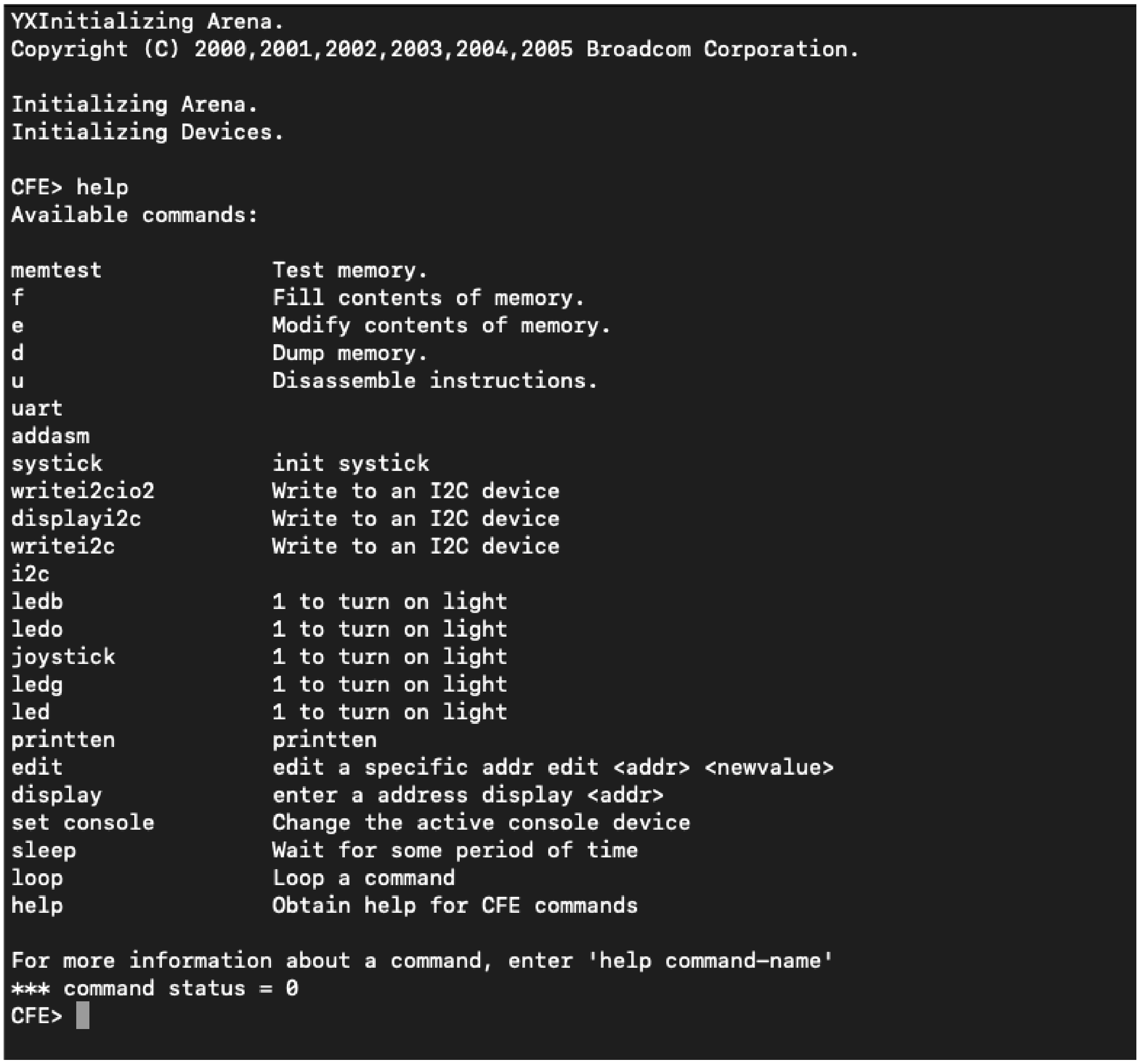
CFE: Common Firmware Environment on STM32L476G Board

What is CFE?

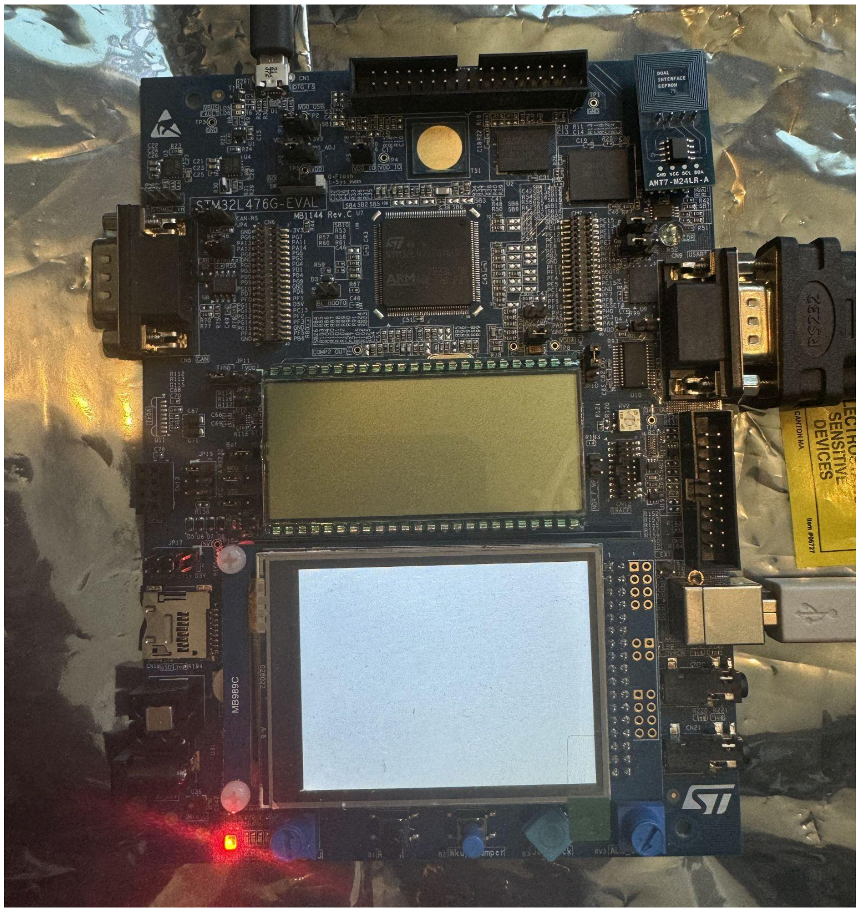
CFE (Common Firmware Environment) is an open-source, command-driven interface UI. Its primary responsibility is to initialize the CPU, caches, memory controllers, LEDs, and other peripherals on the STM32 board. For more information, visit the Wikipedia page or the source code on GitHub.



```
YXInitializing Arena.
Copyright (C) 2000,2001,2002,2003,2004,2005 Broadcom Corporation.

Initializing Arena.
Initializing Devices.

CFE> help
Available commands:

memtest             Test memory.
f                   Fill contents of memory.
e                   Modify contents of memory.
d                   Dump memory.
u                   Disassemble instructions.
uart
addasm
systick             init systick
writei2cio2         Write to an I2C device
displayi2c          Write to an I2C device
writei2c            Write to an I2C device
i2c
ledb                1 to turn on light
ledo                1 to turn on light
joystick            1 to turn on light
ledg                1 to turn on light
led                 1 to turn on light
printten            printten
edit                edit a specific addr edit <addr> <newvalue>
display             enter a address display <addr>
set console         Change the active console device
sleep               Wait for some period of time
loop                Loop a command
help                Obtain help for CFE commands

For more information about a command, enter 'help command-name'
*** command status = 0
CFE> 
```

Hardware Setup

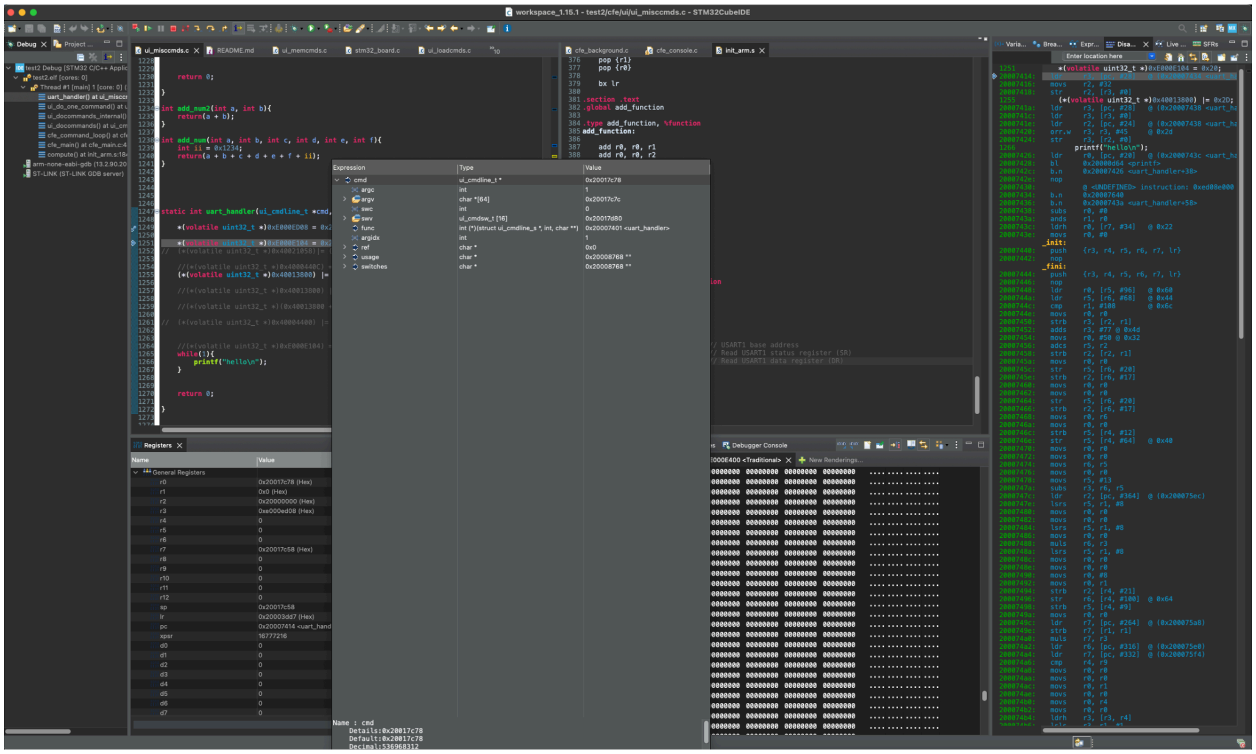
The board used for software development is the STM32L476G-EVAL evaluation board. It features a Cortex®-M4 CPU and various peripherals.



Communication with the device is done via a USB Serial connection (black cable), while the white USB cable is used for debugging and powering the device, enabling communication with the host to download and debug code. More details about the board can be found here.

Software Setup

The software used is STM32CubeIDE, an advanced C/C++ development platform with peripheral configuration, code generation, compilation, and debugging features such as disassembly, memory access/modification, and address access/modification. Below is an example of single-stepping through code with the disassembly on the right and registers at the bottom to observe changes in real time. Learn more about STM32CubeIDE here.



workspace_1.15.1 - test2/cfe/ui/ui_misccmds.c - STM32CubeIDE

```c
    return 0;

}

int add_num2(int a, int b){
    return(a + b);
}

int add_num(int a, int b, int c, int d, int e, int f){
    int ii = 0x1234;
    return(a + b + c + d + e + f + ii);
}




static int uart_handler(ui_cmdline_t *cmd,

    *(volatile uint32_t *)0xE000ED08 = 0x2

    *(volatile uint32_t *)0xE000E104 = 0x2
//  (*(volatile uint32_t *)0x40021058)|= (

    //(*(volatile uint32_t *)0x4000440C) =
    (*(volatile uint32_t *)0x40013800)  |=

    //(*(volatile uint32_t *)0x40013800) |

    //(*(volatile uint32_t *)(0x40013800 +

//  (*(volatile uint32_t *)0x40004400)  |=


    //(*(volatile uint32_t *)0xE000E104) =
    while(1){
        printf("hello\n");
    }


    return 0;

}
```

```
    pop {r1}
    pop {r0}

    bx lr

.section .text
.global add_function

.type add_function, %function
add_function:

    add r0, r0, r1
    add r0, r0, r2
```

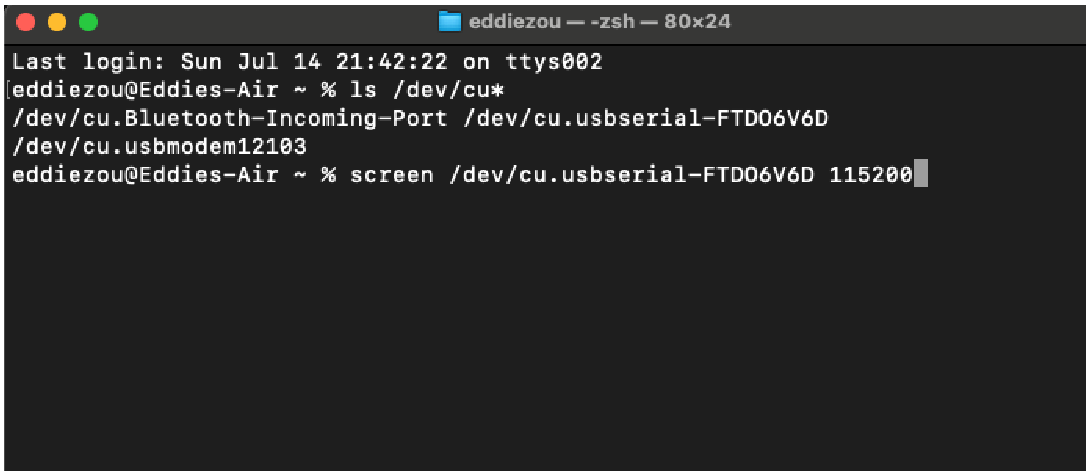
Expression	Type	Value
cmd	ui_cmdline_t *	0x20017c78
argc	int	1
argv	char *[64]	0x20017c7c
swc	int	0
swv	ui_cmdsw_t [16]	0x20017d80
func	int (*)(struct ui_cmdline_s *, int, char **)	0x20007401 <uart_handler>
argidx	int	1
ref	char *	0x0
usage	char *	0x20008768 ""
switches	char *	0x20008768 ""

```
Name : cmd
    Details:0x20017c78
    Default:0x20017c78
    Decimal:536968312
```

Name	Value
General Registers	
r0	0x20017c78 (Hex)
r1	0x0 (Hex)
r2	0x20000000 (Hex)
r3	0xe000ed08 (Hex)
r4	0
r5	0
r6	0
r7	0x20017c58 (Hex)
r8	0
r9	0
r10	0
r11	0
r12	0
sp	0x20017c58
lr	0x20003dd7 (Hex)
pc	0x20007414 <uart_hand
xpsr	16777216
d0	0
d1	0
d2	0
d3	0
d4	0
d5	0
d6	0
d7	0

```
1251            *(volatile uint32_t *)0xE000E104 = 0x20;
20007414:   ldr     r3, [pc, #28]   @ (0x20007434 <uart_ha
20007416:   movs    r2, #32
20007418:   str     r2, [r3, #0]
1255            (*(volatile uint32_t *)0x40013800) |= 0x2D;
2000741a:   ldr     r3, [pc, #28]   @ (0x20007438 <uart_ha
2000741c:   ldr     r3, [r3, #0]
2000741e:   ldr     r2, [pc, #24]   @ (0x20007438 <uart_ha
20007420:   orr.w   r3, r3, #45     @ 0x2d
20007424:   str     r3, [r2, #0]
1266                printf("hello\n");
20007426:   ldr     r0, [pc, #20]   @ (0x2000743c <uart_ha
20007428:   bl      0x20000d64 <printf>
2000742c:   b.n     0x20007426 <uart_handler+38>
2000742e:   nop
20007430:           @ <UNDEFINED> instruction: 0xed08e000
20007434:   b.n     0x20007640
20007436:   b.n     0x2000743a <uart_handler+58>
20007438:   subs    r0, #0
2000743a:   ands    r1, r0
2000743c:   ldrh    r0, [r7, #34]   @ 0x22
2000743e:   movs    r0, #0
          _init:
20007440:   push    {r3, r4, r5, r6, r7, lr}
20007442:   nop
          _fini:
20007444:   push    {r3, r4, r5, r6, r7, lr}
20007446:   nop
20007448:   ldr     r0, [r5, #96]   @ 0x60
2000744a:   ldr     r5, [r6, #68]   @ 0x44
2000744c:   cmp     r1, #108        @ 0x6c
2000744e:   movs    r0, r0
20007450:   strb    r3, [r2, r1]
20007452:   adds    r3, #77 @ 0x4d
20007454:   movs    r0, #50 @ 0x32
20007456:   adcs    r5, r2
20007458:   strb    r2, [r2, r1]
2000745a:   movs    r0, r0
2000745c:   str     r5, [r6, #20]
2000745e:   strb    r2, [r6, #17]
20007460:   movs    r0, r0
20007462:   movs    r0, r0
20007464:   str     r5, [r6, #20]
20007466:   strb    r2, [r6, #17]
20007468:   movs    r0, r6
2000746a:   movs    r0, r0
2000746c:   strb    r5, [r4, #12]
2000746e:   str     r5, [r4, #64]   @ 0x40
20007470:   movs    r0, r0
20007472:   movs    r0, r0
20007474:   movs    r6, r5
20007476:   movs    r0, r0
20007478:   movs    r5, #13
2000747a:   subs    r3, r6, r5
2000747c:   ldr     r2, [pc, #364]  @ (0x200075ec)
2000747e:   lsrs    r5, r1, #8
20007480:   movs    r0, r0
20007482:   movs    r0, r0
20007484:   lsrs    r5, r1, #8
20007486:   movs    r0, r0
20007488:   muls    r6, r3
2000748a:   lsrs    r5, r1, #8
2000748c:   movs    r0, r0
2000748e:   movs    r0, r0
20007490:   movs    r0, #8
20007492:   movs    r0, r1
20007494:   strb    r2, [r4, #21]
20007496:   str     r6, [r4, #100]  @ 0x64
20007498:   strb    r5, [r4, #9]
2000749a:   movs    r0, r0
2000749c:   ldr     r7, [pc, #264]  @ (0x200075a8)
2000749e:   strb    r7, [r1, r1]
200074a0:   muls    r7, r3
200074a2:   ldr     r6, [pc, #316]  @ (0x200075e0)
200074a4:   ldr     r7, [pc, #332]  @ (0x200075f4)
200074a6:   cmp     r4, r9
200074a8:   movs    r0, r0
200074aa:   movs    r0, r0
200074ac:   movs    r0, r1
200074ae:   movs    r0, r0
200074b0:   movs    r0, r4
200074b2:   movs    r0, r0
200074b4:   ldrh    r3, [r3, r4]
```

For the serial monitor, Mac's built-in screen function is utilized.

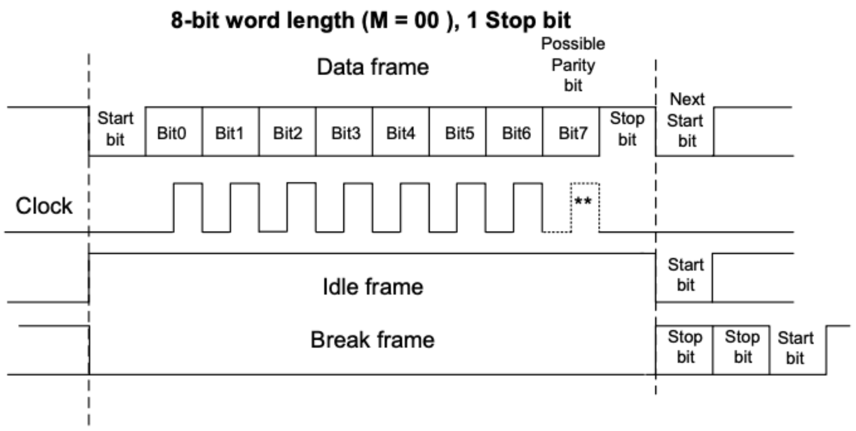


eddiezou — -zsh — 80×24

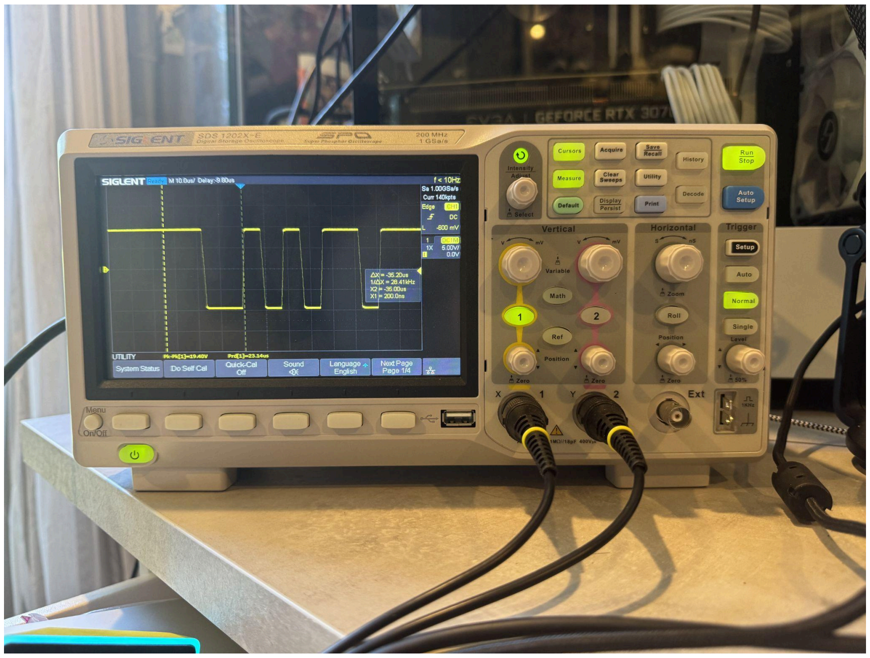
```
Last login: Sun Jul 14 21:42:22 on ttys002
eddiezou@Eddies-Air ~ % ls /dev/cu*
/dev/cu.Bluetooth-Incoming-Port /dev/cu.usbserial-FTDO6V6D
/dev/cu.usbmodem12103
eddiezou@Eddies-Air ~ % screen /dev/cu.usbserial-FTDO6V6D 115200
```

UART

The UART setup involves several configurations, starting with enabling the CLOCK and PWR, followed by writing to the address in GPIOB. The bit length is configured by writing to GPIO MODER. Below is an observation of the character 'd' being read by the oscilloscope. The pins RXD and TXD are monitored, with the initial start bit followed by bits 0-7 (0010 0110). After reversing, this gives us 0x64, corresponding to 'd', the character input into the CFE.



Transmission and reception processes were verified by sending known characters and observing the corresponding signal patterns on the oscilloscope, confirming the correct implementation of UART configuration.



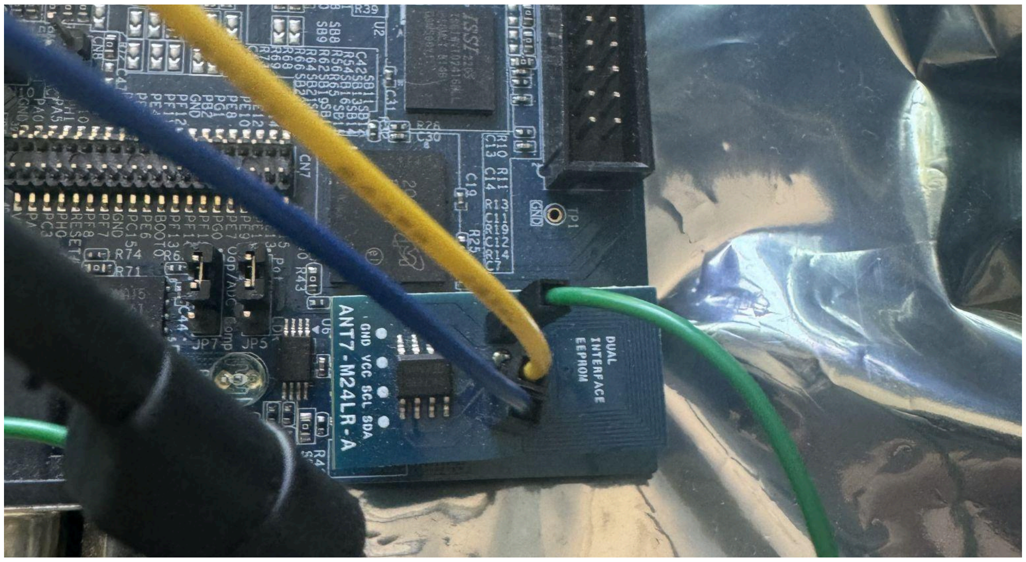
I2C

To enable other peripherals, the I2C peripheral on the STM32 microcontroller needs to be configured. The initialization code starts by enabling the clocks for the System Configuration Controller, Power Interface, and GPIOG. GPIO pins are then configured for I2C communication by setting their speed, output type, pull-up/pull-down resistors, and alternate functions. The I2C peripheral is disabled to configure its timing, address, acknowledgment, general call, and no-stretch modes before re-enabling it. For data transmission, the control and address registers are set up, data is placed in the TX register, and it is sent. For data reception, the STOP condition and busy flag are awaited, data is read from the RX register, and printed to the console. This demonstrates the steps to initialize, transmit, and receive data using the I2C peripheral.

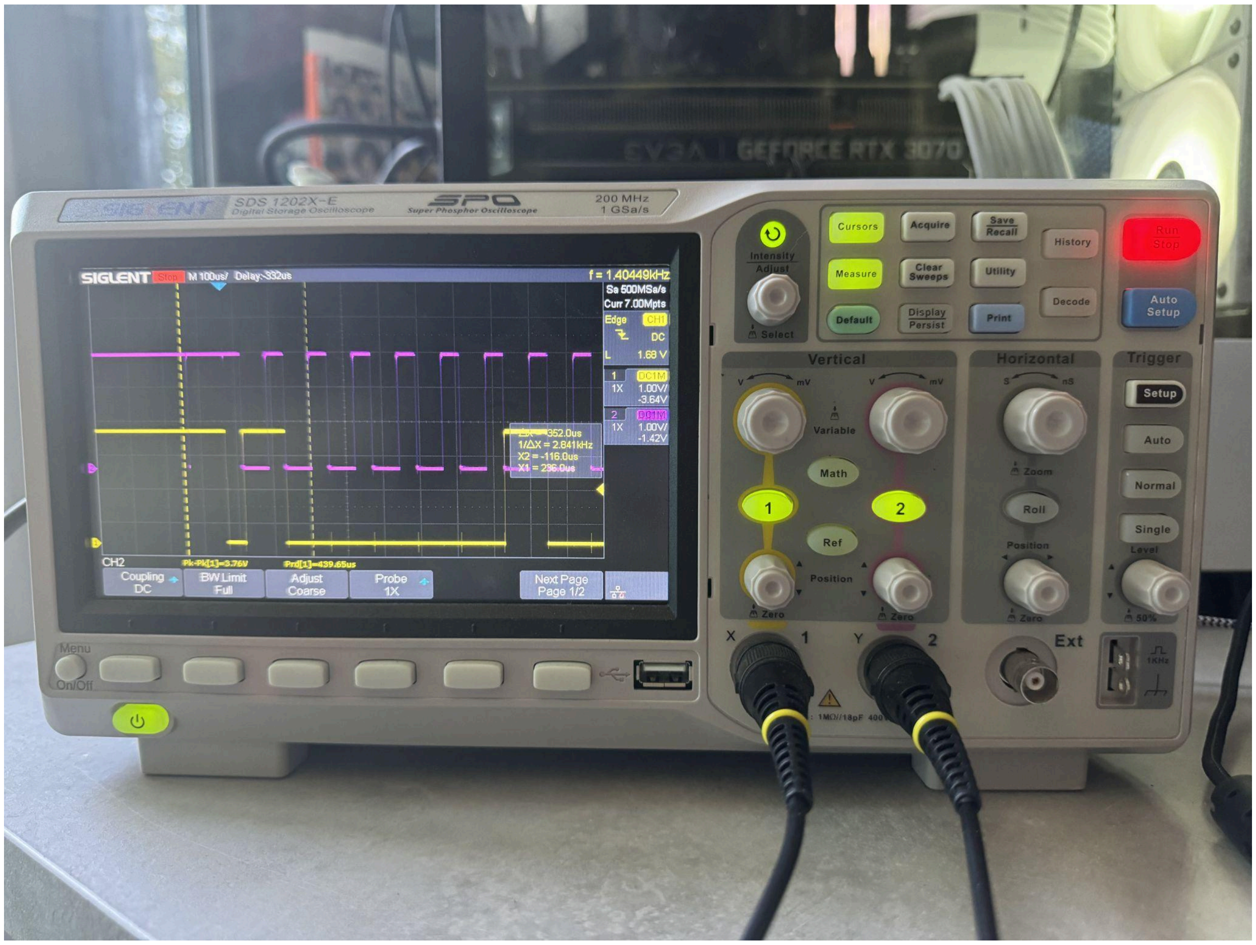
Using sample code and single-stepping through disassembly, we identified and corrected missing address writes. Below is an example of examining registers, where r3 at 0x44 is written with r2.

```
 636        hi2c->ErrorCode = HAL_I2C_ERROR_NONE;
0800538a:   ldr     r3, [r7, #4]
0800538c:   movs    r2, #0
0800538e:   str     r2, [r3, #68]   @ 0x44
```

Transmitting and receiving are tested by executing the function displayi2c which initiates I2C and displays an address depending on the user's argument. For the input we are using displayi2c 0x02 which outputs Received data: 0x00000003.



The oscilloscope connected to SCL and SDA monitors the I2C. The purple trace represents the clock, while the yellow trace represents the data being read.



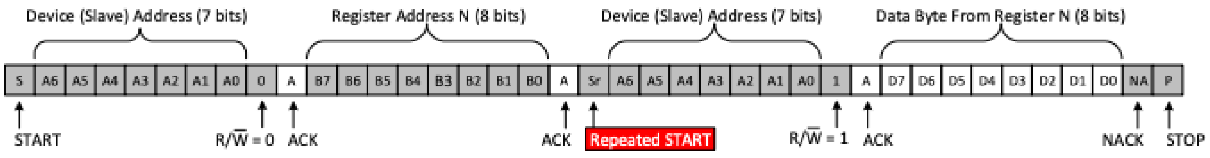

Device (Slave) Address (7 bits)
Register Address N (8 bits)
Device (Slave) Address (7 bits)
Data Byte From Register N (8 bits)
S A6 A5 A4 A3 A2 A1 A0 0 A B7 B6 B5 B4 B3 B2 B1 B0 A Sr A6 A5 A4 A3 A2 A1 A0 1 A D7 D6 D5 D4 D3 D2 D1 D0 NA P
START
R/W̄ = 0
ACK
ACK
Repeated START
R/W̄ = 1
ACK
NACK
STOP

Figure 9. Example I²C Read from Slave Device's Register

The data sequence includes the initial start bit, slave address, register address, device slave address, and the input data 00000011 (0x3). For more information, refer to this document from Texas Instruments and the STMPE811 datasheet. From our testing we can determine that it accurately displays and reads from the I2C.

Reset_Handler and Vector Table Setup
The interrupt vector table defines handlers for reset, SysTick, and USART1 interrupts. In the Reset Handler, GPIO and UART clocks are configured, and GPIO pins are set to alternate functions for UART. The UART clock is enabled, and UART configuration, including control and baud rate settings, is established.

To enable the SysTick Handler, reference the Cortex M4 datasheet for the required register changes. Initialization involves configuring the SysTick Control and Status Register to enable the timer and interrupts, setting the reload value to 1,000,000 for timing intervals, and clearing the current value register to start counting from zero.

The SysTick Handler saves the current processor state, calls the function mytest_1 for custom processing, restores the processor state, and then returns from the interrupt. Testing mytest prints a message to ensure correct register saving and continuation after the interrupt.

To enable UART interrupts, write to the NVIC interrupt set-enable register. The function enables UART by setting appropriate bits in the UART control register. An infinite loop continuously prints "hello" to the console, demonstrating that the UART handler is operational and capable of transmitting data. After receiving the interrupt, the code branches to mytest again to ensure proper UART handler functionality.

```
.section .text
.global USART1_IRQHandler
.type USART1_IRQHandler, %function
USART1_IRQHandler:

      push {r0-r12, lr}


      bl mytest_1

    //ldr r0, =0x40013800      // USART1 base address
    //ldr r1, [r0, #0x1C]      // Read USART1 status register (SR)
    //ldr r2, [r0, #0x24]      // Read USART1 data register (DR)


      pop {r0-r12, lr}

    bx lr
```