

STM32 Programmer for the Multi-cycle OTTER

Use this guide to set up and use the STM32 programmer.

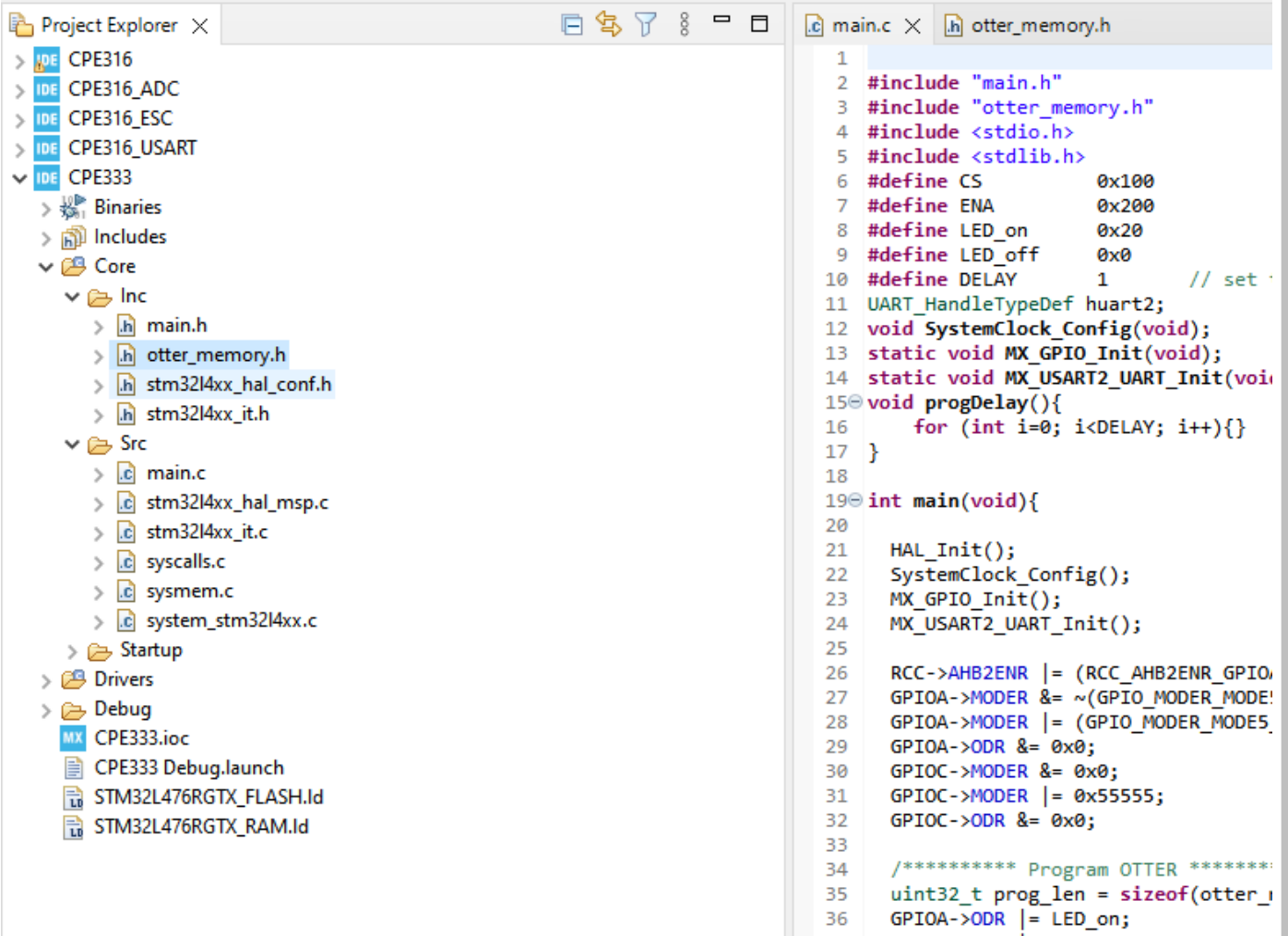
Source files found here:

https://github.com/geoneill12/OTTER_Multicycle_STM32_Programmer

Download and install the STM32CubeIDE from the vendor's website:

<https://www.st.com/en/development-tools/stm32cubeide.html>

The board used for this project is the NUCLEO-L476RG board.



Create a new STM32 project. Create a C header file under the “Inc” folder. Call this file “otter_memory.h”.

In the “main.c” file, delete everything, and replace it with the contents of the “STM32programmer.c” file from github.

Use the following pinout chart to connect the Nucleoboard to the BASYS3 board using jumper wires. All connections, including Ground connections, must be wired exactly as below.

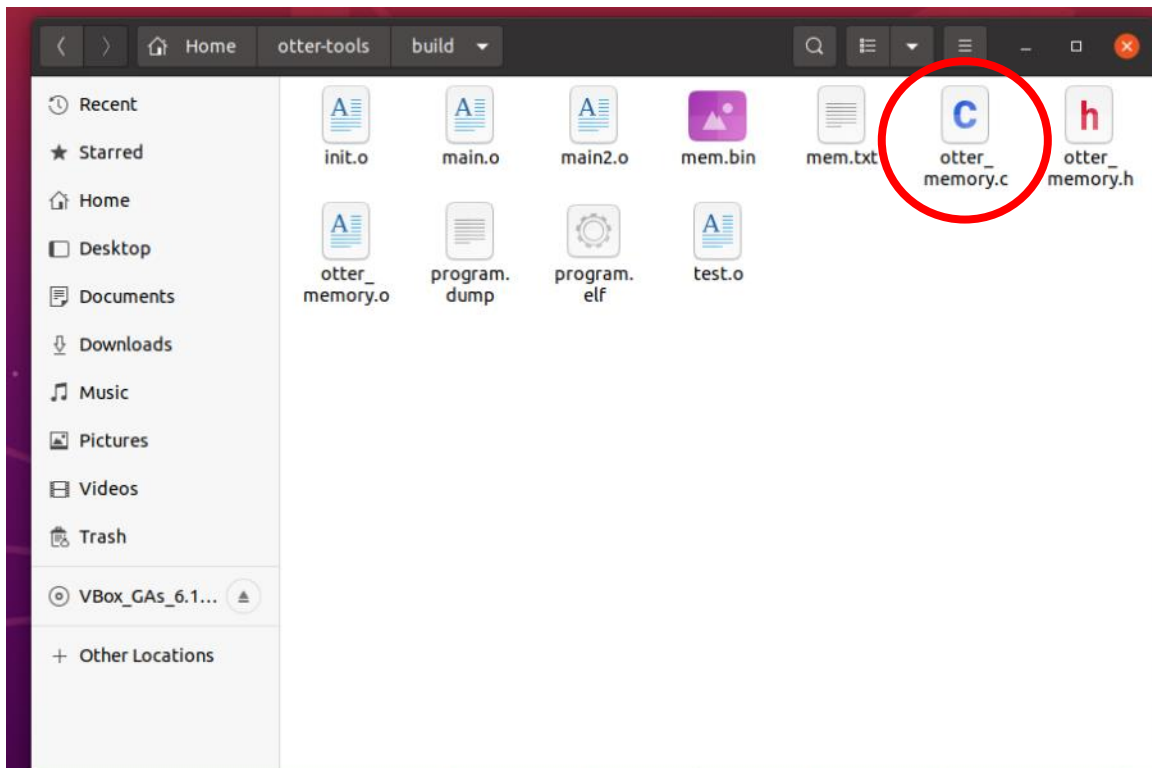
Nucleoboard		BASYS3
Pin #	Signal Name	Pin #
PC0	Data Bit 0	JB.1
PC1	Data Bit 1	JB.2
PC2	Data Bit 2	JB.3
PC3	Data Bit 3	JB.4
PC4	Data Bit 4	JB.7
PC5	Data Bit 5	JB.8
PC6	Data Bit 6	JB.9
PC7	Data Bit 7	JB.10
PC8	CS	JC.1
PC9	ENA	JC.2
GND	Ground	JB.11
GND	Ground	JC.11

Use the “OTTER_STM32_Programmer_BBD.pdf” file to modify the Multi-cycle OTTER. Sample code for both the Wrapper and the OTTERMCU are provided to show how to integrate the STM32_programmer module into the OTTER. The STM32_programmer module can be thought of as a wrapper for the memory module. In normal mode, it just passed signals through to the memory as though it weren’t there. In programmer mode, it takes control of all signals going to memory.

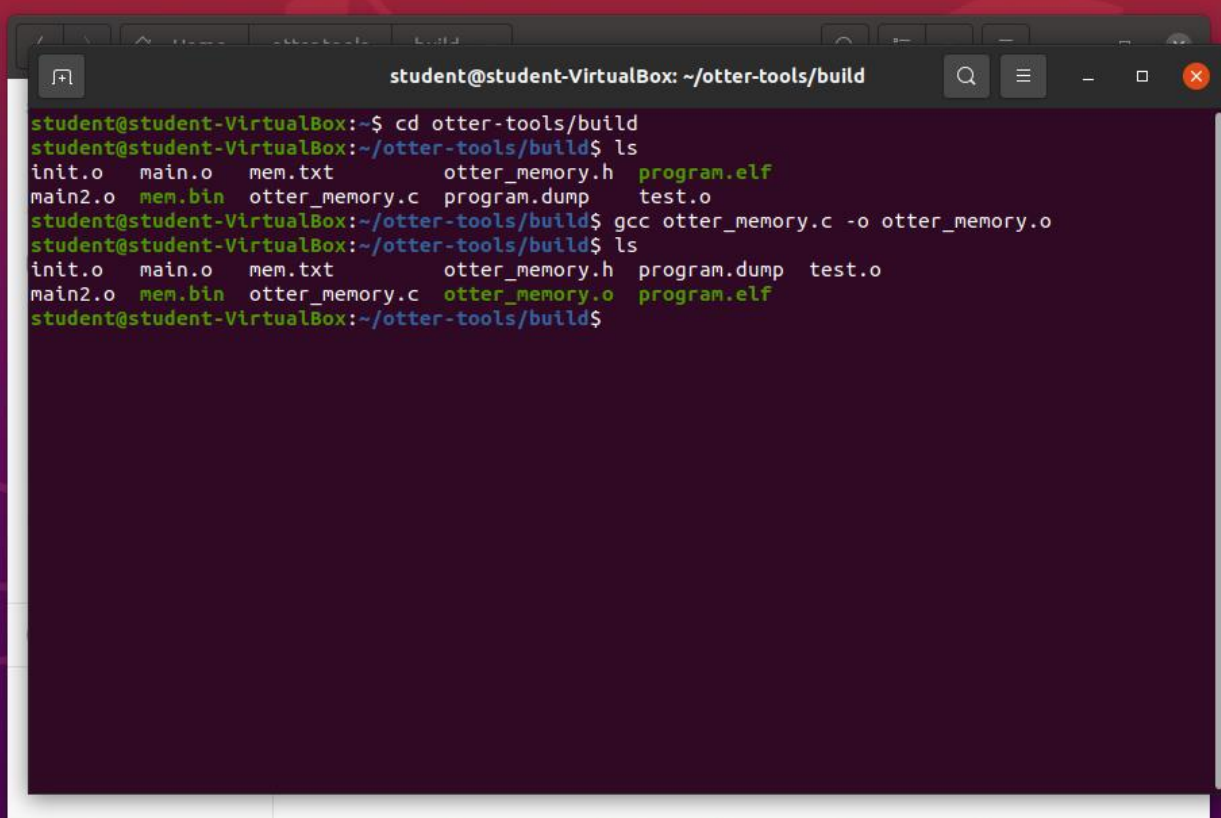
Refer to the “Basys3_constraints.xdc” file to see how to connect the pmod ports to the OTTER. Pulldown resistors must be utilized on all ports that act as inputs.

In the “otter_memory.mem” file in Vivado, fill each line with “00000013” (assembly code for “nop” instructions). Create several thousand lines of this (file provided on github). This step may not be necessary, however the programmer has not been tested without doing this.

Synthesize the design and load the new bitstream onto the BASYS3.



Start up the OTTER developer virtual machine. Copy the “otter_memory.c” file from github into the “otter-tools/build” directory.

A terminal window titled "student@student-VirtualBox: ~/otter-tools/build" is shown. The terminal output is as follows:

```
student@student-VirtualBox:~$ cd otter-tools/build
student@student-VirtualBox:~/otter-tools/build$ ls
init.o  main.o  mem.txt  otter_memory.h  program.elf
main2.o mem.bin otter_memory.c  program.dump  test.o
student@student-VirtualBox:~/otter-tools/build$ gcc otter_memory.c -o otter_memory.o
student@student-VirtualBox:~/otter-tools/build$ ls
init.o  main.o  mem.txt  otter_memory.h  program.dump  test.o
main2.o mem.bin otter_memory.c  otter_memory.o  program.elf
student@student-VirtualBox:~/otter-tools/build$
```

Open a terminal window and navigate to the “otter-tools/build” directory. Use gcc to compile the “otter_memory.c” program. Give it the name “otter_memory.o” (not necessary, but recommended to keep things organized).

Everything is now set up and ready to begin programming the OTTER!

```
student@student-VirtualBox: ~/otter-tools/src

1 #####
2 #
3 #     OTTER Tests
4 #     (37 instructions)
5 #     CalPoly 233 - Callenes
6 #
7 #####
8
9 .equ test, 0x0044
10 .equ SWITCHES, 0x11000000
11 .equ VGA_READ, 0x11040000
12 .equ LEDS, 0x11080000
13 .equ SSEG, 0x110C0000
14 .equ VGA_ADDR, 0x11100000
15 .equ VGA_COLOR, 0x11140000
16
17
18 .text
19
20 .global main
21 .type main, @function
22 main:
23     li x16, SWITCHES
24     li x11, SSEG
25     li x12, LEDS
26 loop:
27     li x14, -1; call post_test          #Test  Number of Tests
28     jal x31, add_test; call post_test   #0      17
29     jal x31, sub_test; call post_test   #1      16
30     jal x31, and_test; call post_test   #2       7
31     jal x31, or_test; call post_test    #3       7
32     jal x31, xor_test; call post_test    #4       7
33     jal x31, sll_test; call post_test    #5      19
34     jal x31, srl_test; call post_test    #6      17
35     jal x31, sra_test; call post_test    #7      20
36     jal x31, slt_test; call post_test    #8      16
37     jal x31, sltu_test; call post_test   #9      16
38     jal x31, addi_test; call post_test   #A      16
39     jal x31, andi_test; call post_test   #B       5
40     jal x31, ori_test; call post_test    #C       5
```

DEMO

Edit your code.


```
student@student-VirtualBox: ~/otter-tools
student@student-VirtualBox:~$ cd otter-tools/build
student@student-VirtualBox:~/otter-tools/build$ ls
init.o  main.o  mem.txt  otter_memory.h  program.elf
main2.o  mem.bin  otter_memory.c  program.dump  test.o
student@student-VirtualBox:~/otter-tools/build$ gcc otter_memory.c -o otter_memory.o
student@student-VirtualBox:~/otter-tools/build$ ls
init.o  main.o  mem.txt  otter_memory.h  program.dump  test.o
main2.o  mem.bin  otter_memory.c  otter_memory.o  program.elf
student@student-VirtualBox:~/otter-tools/build$ cd ..
student@student-VirtualBox:~/otter-tools$ ls
build  link.ld  Makefile  readme.md  src
student@student-VirtualBox:~/otter-tools$ cd src
student@student-VirtualBox:~/otter-tools/src$ ls
init.s  main.s
student@student-VirtualBox:~/otter-tools/src$ vim main.s
student@student-VirtualBox:~/otter-tools/src$ ls
init.s  main.s
student@student-VirtualBox:~/otter-tools/src$ vim main.s
student@student-VirtualBox:~/otter-tools/src$ cd ..
student@student-VirtualBox:~/otter-tools$ make
mkdir -p build
riscv64-unknown-elf-gcc -c -o build/main.o src/main.s -O0 -march=rv32i -mabi=ilp32
riscv64-unknown-elf-gcc -o build/program.elf build/init.o build/main.o -T link.ld -mno-relax -nostdlib -nostartfiles -mcmodel=medany -O0 -march=rv32i -mabi=ilp32
/lib/gcc/riscv64-unknown-elf/8.2.0/../../../../riscv64-unknown-elf/bin/ld: warning: section '.data' type changed to PROGBITS
riscv64-unknown-elf-objcopy -O binary --only-section=.data* --only-section=.text* build/program.elf build/mem.bin
hexdump -v -e "%08x\n" build/mem.bin > build/mem.txt
riscv64-unknown-elf-objdump -S -s build/program.elf > build/program.dump
student@student-VirtualBox:~/otter-tools$
```

DEMO

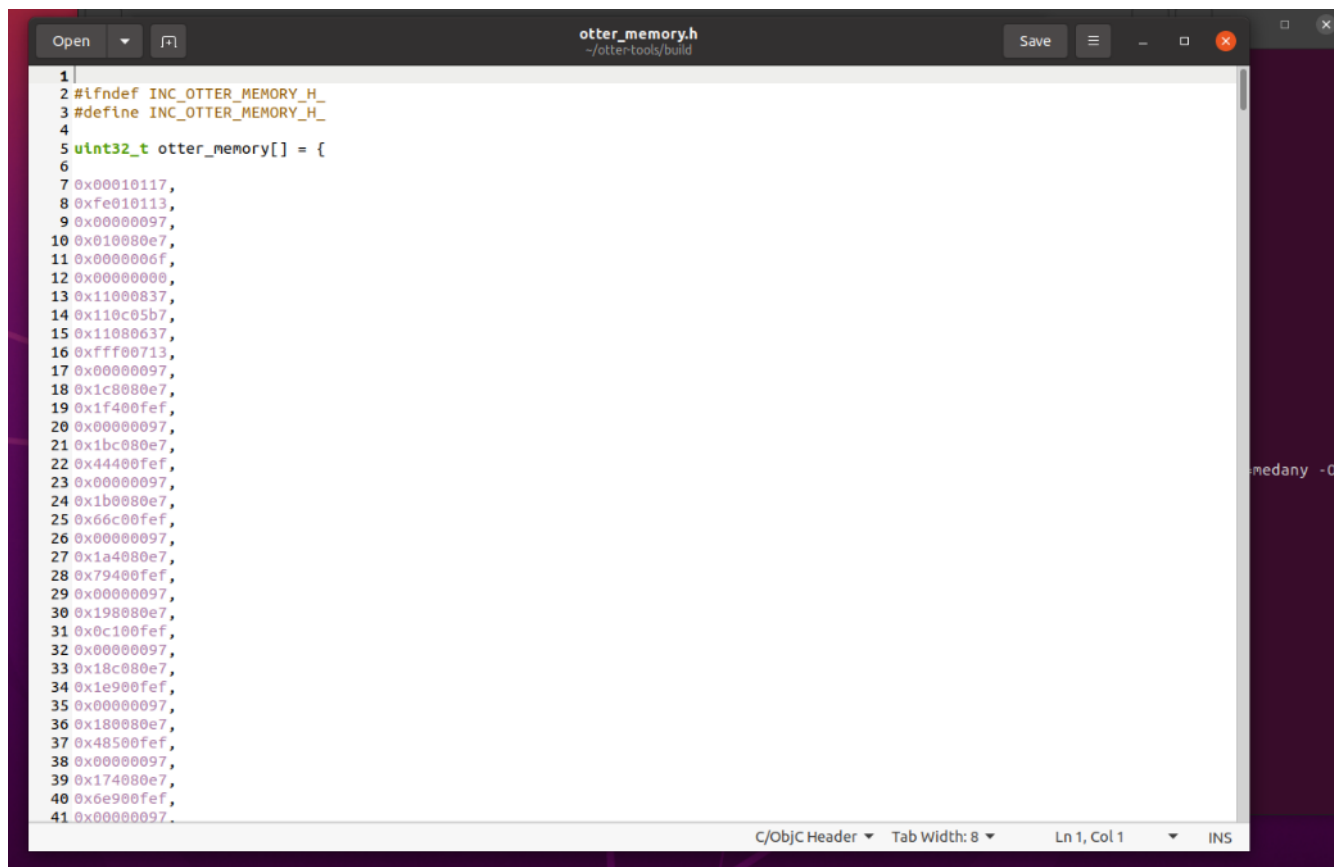
Run “make” to assemble your code.

```
student@student-VirtualBox: ~/otter-tools/build
student@student-VirtualBox:~/otter-tools/build$ ls
init.o  main.o  mem.txt  otter_memory.h  program.elf
main2.o mem.bin otter_memory.c  program.dump  test.o
student@student-VirtualBox:~/otter-tools/build$ gcc otter_memory.c -o otter_memory.o
student@student-VirtualBox:~/otter-tools/build$ ls
init.o  main.o  mem.txt  otter_memory.h  program.dump  test.o
main2.o mem.bin otter_memory.c  otter_memory.o  program.elf
student@student-VirtualBox:~/otter-tools/build$ cd ..
student@student-VirtualBox:~/otter-tools$ ls
build  link.ld  Makefile  readme.md  src
student@student-VirtualBox:~/otter-tools$ cd src
student@student-VirtualBox:~/otter-tools/src$ ls
init.s  main.s
student@student-VirtualBox:~/otter-tools/src$ vim main.s
student@student-VirtualBox:~/otter-tools/src$ cd ..
student@student-VirtualBox:~/otter-tools$ make
mkdir -p build
riscv64-unknown-elf-gcc -c -o build/main.o src/main.s -O0 -march=rv32l -mabi=ilp32
riscv64-unknown-elf-gcc -o build/program.elf build/init.o build/main.o -T link.ld -mno-relax -nostdlib -nostartfiles -mcmodel=medany -O0 -march=rv32l -mabi=ilp32
/llb/gcc/riscv64-unknown-elf/8.2.0/../../../../riscv64-unknown-elf/bin/ld: warning: section '.data' type changed to PROGBITS
riscv64-unknown-elf-objcopy -O binary --only-section=.data* --only-section=.text* build/program.elf build/mem.bin
hexdump -v -e "%08x\n" build/mem.bin > build/mem.txt
riscv64-unknown-elf-objdump -S -s build/program.elf > build/program.dump
student@student-VirtualBox:~/otter-tools$ cd build
student@student-VirtualBox:~/otter-tools/build$ ./otter_memory.o
student@student-VirtualBox:~/otter-tools/build$ ls
init.o  main2.o  main.o  mem.bin  mem.txt  otter_memory.c  otter_memory.h  otter_memory.o  program.dump  program.elf  test.o
student@student-VirtualBox:~/otter-tools/build$
```

DEMO

Navigate to the “build” directory and execute the “otter_memory.o” file.

This action will create a new “otter_memory.h” header file, with the contents of the “mem.txt” file placed into an array. Each line of the “mem.txt” file becomes an element of the array. If an “otter_memory.h” file was already present, it will delete it first.

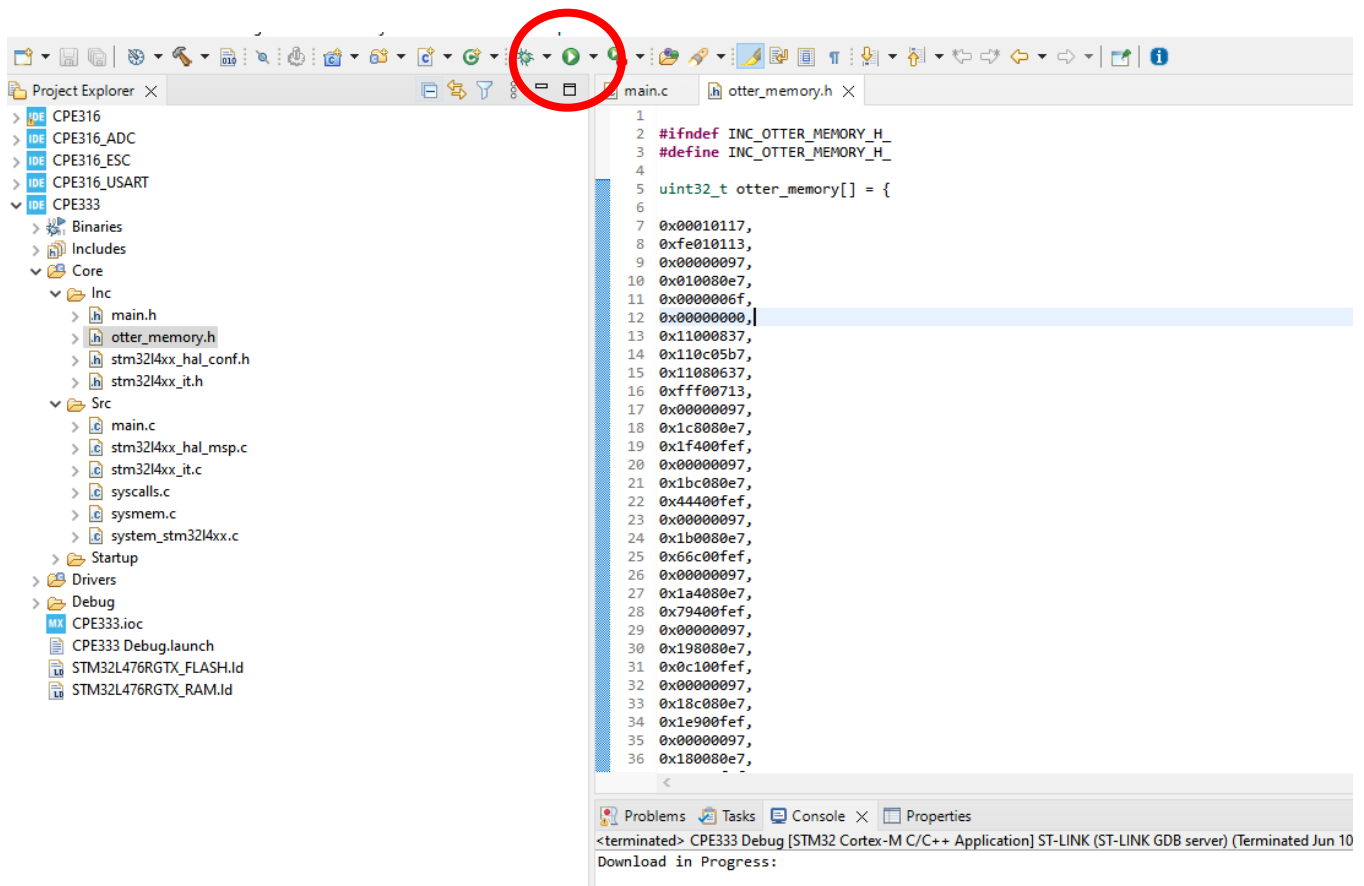


```
1
2 #ifndef INC_OTTER_MEMORY_H_
3 #define INC_OTTER_MEMORY_H_
4
5 uint32_t otter_memory[] = {
6
7 0x00010117,
8 0xfe010113,
9 0x00000097,
10 0x010000e7,
11 0x0000006f,
12 0x00000000,
13 0x11000837,
14 0x110c05b7,
15 0x11080637,
16 0xffff00713,
17 0x00000097,
18 0x1c0000e7,
19 0x1f400fef,
20 0x00000097,
21 0x1bc000e7,
22 0x44400fef,
23 0x00000097,
24 0x1b0000e7,
25 0x66c00fef,
26 0x00000097,
27 0x1a4000e7,
28 0x79400fef,
29 0x00000097,
30 0x190000e7,
31 0x0c100fef,
32 0x00000097,
33 0x18c000e7,
34 0x1e900fef,
35 0x00000097,
36 0x180000e7,
37 0x40500fef,
38 0x00000097,
39 0x174000e7,
40 0x6e900fef,
41 0x00000097.
```

DEMO

Open the “otter_memory.h” file in a text editor.

Select and copy the contents of the file using “CTRL+A” and “CTRL+C”.



DEMO

In the STM32CubeIDE, delete all the contents of the “otter_memory.h” file (“CTRL+A” and “delete”), then paste in the contents of the “otter_memory.h” file from the virtual machine (“CTRL+V”). Save the header file.

Press the green play button to run the project and upload it to the Nucleoboard. Once uploaded to the Nucleoboard, it will immediately program the OTTER. Pressing the black reset button on the Nucleoboard will cause it to program the OTTER again.

```

main.c x otter_memory.h
1 |
2 #include "main.h"
3 #include "otter_memory.h"
4 #include <stdio.h>
5 #include <stdlib.h>
6 #define CS      0x100
7 #define ENA     0x200
8 #define LED_on  0x20
9 #define LED_off 0x0
10 #define DELAY   1 // set to 1 for 50MHz OTTER; 7 gives ~1ms delay
11 UART_HandleTypeDef huart2;
12 void SystemClock_Config(void);
13 static void MX_GPIO_Init(void);
14 static void MX_USART2_UART_Init(void);
15 void progDelay(){
16     for (int i=0; i<DELAY; i++){
17 }

```

Notes

The code on github is designed to run the Nucleoboard at 80MHz, and the OTTER at 50MHz. If the OTTER's clock rate is reduced below 50MHz, then the programmer on the Nucleoboard must be slowed down as well for the OTTER to "see" the changing signals. To do this, increase the "DELAY" value (circled above). A value of 1 works for a 50MHz OTTER. Ratio this value up in proportion to however much the OTTER clock rate is slowed down (ex: set DELAY to 10 for an OTTER clock rate of 5MHz). This may require some trial and error.

How the STM32_programmer.sv module works.

The programming module is an FSM with 6 states:

State 0: Programmer passes signals through to memory. If it sees the CS line go high, it goes to state 1. Else, it stays in State 0.

State 1: Programmer holds all the memRead/memWrite lines at 0. It sets the size and sign signals to 0 for loading data into memory 1 byte at a time. It passes data from the Nucleoboard directly to the “din2” port of memory (preceded by 24 zeros). It keeps track of the address using a register. This register starts at 0, and increments by 1 each time a write event occurs. The programmer remains in State 1 until ENA goes high, at which point it goes to State 2.

State 2: Programmer writes the data to memory. It stays in State 2 until ENA goes low, at which point it goes to State 3.

State 3: Programmer increments the address register by 1. It then jumps unconditionally to State 4.

State 4: Programmer waits to see if more data is coming. If CS stays high and ENA stays low, the programmer stays in State 4. If ENA goes high again, more data is coming, so the programmer jumps back to State 2 to repeat the write process. This continues until CS goes low while in State 4, which only happens when there is no more data. When this happens, the programmer jumps to State 5.

State 5: The programmer resets the internal address register to 0, so that the OTTER can be programmed again. A reset signal is also sent to the Program Counter, Control Unit FSM, and Control Status Registers to restart to OTTER. The programmer then jumps unconditionally back to State 0. At the start of the next clock cycle, the OTTER re-starts at instruction 0.

How the STM32_programmer.c program works.

The C file that runs on the Nucleoboard has 3 main steps:

1. Set CS line high to tell the OTTER it's about to receive program data.
2. Enters a programming loop that repeats for each array element:
 - a) Send lowest 8 bits to OTTER.
 - b) Set ENA high, then low.
 - c) Shift right 8 bits, send lowest 8 bits to OTTER.
 - d) Set ENA high, then low.
 - e) Shift right 8 bits, send lowest 8 bits to OTTER.
 - f) Set ENA high, then low.
 - g) Shift right 8 bits, send lowest 8 bits to OTTER.
 - h) Set ENA high, then low.
 - i) Go back to start of the loop. Repeat this process for as many array elements (instructions) that are in the "otter_memory.h" file.
3. Reset CS line to low to tell the OTTER that no more data is coming.

NOTE: A delay is placed in between each event to allow the OTTER enough time to "see" each signal change.

How the otter_memory.c program works.

The C file that creates the “otter_memory.h” header file first deletes the “otter_memory.h” file (if already present), creates a new file, and then reads the contents of the “mem.txt” file that is automatically generated using the “make” command. It reads this file line by line and places each instruction into an array with a “0x” prefix.

Video Demo:

<https://www.youtube.com/watch?v=vsKfYsN15Cc>

The above demo shows the OTTER being programmed with the standard “test-all” program. It shows the complete process, from editing the code in the virtual machine, all the way to building the project and loading it onto the Nucleoboard. The video also shows that the OTTER can be programmed again using the reset button.

Note: In this demo, the OTTER is slightly modified so that when SWITCH15 is set high, the OTTER runs at 500Hz, and when it is set low, it runs at 50MHz. This is to allow the OTTER to be programmed quickly at 50MHz, while also giving us the option to slow down the OTTER to see the test-all program running at a slower clock rate.

Summary:

This programmer achieves extremely fast data transfer rates due to have 8 parallel data lines. Using an oscilloscope, the programmer was timed programming a 50MHz OTTER at a maximum speed of 1.65 MB/s (bytes!) with the delay function commented out of the “main.c” file. With the delay function left in, and given a “DELAY” value of 1, a 50MHz OTTER can be programmed at 573 KB/s.

Happy programming!