K

ARM CORTEX ASSEMBLER PRIMER

KEVIN THOMAS

DIRECTOR OF TEST AUTOMATION

mytechnotalent.com

AGENDA

LINKER SCRIPT

On p50 of the STM32F401CCUx datasheet, we see the memory map. We notice that starting at 0x00000000 we have flash aliased which will be mapped to 0x08000000, flash memory during the linking process. We also see SRAM starting at 0x20000000 and we see our peripherals starting at 0x40000000. On p43 of the reference manual we see another table indicating similar info. On page 45 of the reference manual we see how the flash sectors are organized. We begin by creating our linker script, st_nucleo_f4.ld and define the MEMORY and SECTIONS based on p50 of the STM32F401CCUx datasheet. When we write Assembler instructions follow three steps.

1. assemble (creates a relocatable object file)
   arm-none-eabi-as -g main.s -o main.o
2. link (map all the 0x0000000 addresses to physical addresses)
   arm-none-eabi-ld main.o -o main.elf -T st_nucleo_f4.ld
3. flash
   openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg -c "program main.elf 0x08000000 verify reset exit"

We can dive into the object file with the below commands.

* display the contents of the section headers
  arm-none-eabi-objdump -h main.o
* display assembler contents of all sections
  arm-none-eabi-objdump -D main.o | less
* display assembler contents of executable sections
  arm-none-eabi-objdump -d main.o
* display the full contents of all sections requested
  arm-none-eabi-objdump -s main.o | less
* display information about the contents of ELF format files
  arm-none-eabi-readelf -a main.elf | less
* list symbols in file
  arm-none-eabi-nm main.elf | less

Keep in mind when you are using objdump there may be data in the binary which objdump will try to interpret as instructions so you can ignore those parts.

```
/**
 * FILE: st_nucleo_f4.ld
 *
 * DESCRIPTION:
 * This file contains the linker script
 * utilizing the STM32F401 Nucleo-64 microcontroller.
 *
 * AUTHOR: Kevin Thomas
 * CREATION DATE: September 1, 2023
 * UPDATE Date: September 1, 2023
 */

MEMORY
{
  FLASH : ORIGIN = 0x08000000, LENGTH = 512K
  SRAM  : ORIGIN = 0x20000000, LENGTH = 96K
}

SECTIONS
{
  .isr_vector :
  {
    *(.isr_vector)
  } >FLASH
  .text :
  {
    *(.text)
  } >FLASH
  .data (NOLOAD) :
  {
    . = . + 0x400;
 _estack = .;
 *(.data)
  } >SRAM
}
```

VECTOR TABLE

The ARM Cortex M4 has a vector table which the first thing loaded into memory on boot. On page 202 of the reference manual, we see the vector table. These will also get mapped during the linking process. The the very base of memory is the MSP or master stack pointer followed by the Reset_Handler function. The rest of the handlers follow the Reset_Handler followed by the IRQ or interrupt requests.

* 85 IRQ's
* 15 system exceptions
* master stack pointer
* 85+15+1 = 101 * 4 = 404 bytes or 0x194 bytes

```
/**
 * The STM32F401CCUx vector table. Note that the proper constructs
 * must be placed on this to ensure that it ends up at physical address
 * 0x00000000.
 */
.global isr_vector
.section .isr_vector, "a"
.type isr_vector, %object
isr_vector:
 .word _estack
 .word Reset_Handler
  weak NMI_Handler
  weak HardFault_Handler
  weak MemManage_Handler
  weak BusFault_Handler
  weak UsageFault_Handler
 .word 0
 .word 0
 .word 0
 .word 0
  weak SVC_Handler
  weak DebugMon_Handler
 .word 0
  weak PendSV_Handler
  weak SysTick_Handler
 .word 0
  weak EXTI16_PVD_IRQHandler              // EXTI Line 16 interrupt /PVD through EXTI line detection
interrupt
  weak TAMP_STAMP_IRQHandler              // Tamper and TimeStamp interrupts through the EXTI line
  weak EXTI22_RTC_WKUP_IRQHandler           // EXTI Line 22 interrupt /RTC Wakeup interrupt through
the EXTI line
  weak FLASH_IRQHandler                // FLASH global interrupt
  weak RCC_IRQHandler                 // RCC global interrupt
...
```

REGISTERS

The ARM Cortex M4 has 13 general-purpose registers (R0-R12), 1 stack pointer register (R13 or SP), 1 link register (R14 or LR), 1 program counter (R15 or PC) and a status register (xPSR) and 3 interrupt mask registers. There are peripheral registers as well located on page 38 of the reference manual. All registers are 32-bits wide or 4 bytes wide.

* ASSEMBLE AND LINK w/ SYMBOLS:
* 1. arm-none-eabi-as -g main.s -o main.o
* 2. arm-none-eabi-ld main.o -o main.elf -T st_nucleo_f4.ld
* 3. openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg -c "program main.elf 0x08000000 verify reset exit"
* DEBUG w/ SYMBOLS:
* 1. openocd -f board/st_nucleo_f4.cfg
* 2. arm-none-eabi-gdb main.elf
* 3. target remote :3333
* 4. monitor reset halt
* 5. b __start
* 6. c

```
r0             0x40020400       1073873920
r1             0x282            642
r2             0x35010041       889258049
r3             0x0              0
r4             0x0              0
r5             0x0              0
r6             0x0              0
r7             0x42             66
r8             0x0              0
r9             0x0              0
r10            0x0              0
r11            0x0              0
r12            0x0              0
sp             0x20000400       0x20000400
lr             0x800022d        134218285
pc             0x800043c        0x800043c <Conditional_Branching>
xPSR           0x61000000       1627389952
fpscr          0x0              0
msp            0x20000400       0x20000400
psp            0x0              0x0
primask        0x0              0
```

MOV IMMEDIATE

```
// Move an immedaite value into R0.
MOV_Immediate:
 MOV  R0, #0x42              // move 0x42 into R0
 BX  LR                      // return to caller
```

```
(gdb) disas
Dump of assembler code for function MOV_Immediate:
=> 0x080003ca <+0>:     mov.w   r0, #66 ; 0x42
   0x080003ce <+4>:     bx      lr

(gdb) s
halted: PC: 0x080003ce
(gdb) x/x $r0
0x42:   0x019d0000
```

MOVS REGISTER

```
// Move an immediate value into R0 and R1 and then move R1 into R0.
MOVS_Register:
 MOV  R0, #0x42          // move 0x42 into R0
 MOV  R1, #0x43          // move 0x43 into R1
 MOVS R0, R1             // move R1 into R0
 BX  LR                  // return to caller
```

```
(gdb) disas
Dump of assembler code for function MOVS_Register:
=> 0x080003d0 <+0>:     mov.w   r0, #66 ; 0x42
   0x080003d4 <+4>:     mov.w   r1, #67 ; 0x43
   0x080003d8 <+8>:     movs    r0, r1
   0x080003da <+10>:    bx      lr

(gdb) s
(gdb) x/x $r0
0x42:   0x019d0000

(gdb) s
(gdb) x/x $r1
0x43:   0x00019d00

(gdb) s
(gdb) x/x $r0
0x43: 0x00019d00
```

LDR IMMEDIATE

```
/// Load an address into R0, load value inside R0 into R1, set a bit and then store back the value into the address.
LDR_Immediate:
 LDR  R0, =0x40023830          // load address of RCC_AHB1ENR register (p118 RM)
 LDR  R1, [R0]                 // load value inside RCC_AHB1ENR register
 ORR  R1, #(1<<1)              // set the GPIOBEN bit
 STR  R1, [R0]                 // store value into RCC_AHB1ENR register
 BX  LR                        // return to caller
```

```
(gdb) disas
Dump of assembler code for function LDR_Immediate:
=> 0x080003dc <+0>:     ldr     r0, [pc, #136]  ; (0x8000468 <Interrupts+6>)
   0x080003de <+2>:     ldr     r1, [r0, #0]
   0x080003e0 <+4>:     orr.w   r1, r1, #2
   0x080003e4 <+8>:     str     r1, [r0, #0]
   0x080003e6 <+10>:    bx      lr

(gdb) s
(gdb) x/x $r0
0x40023830:     0x00000007

(gdb) s
(gdb) s
(gdb) x/x $r1
0x7:    0x00019d08

(gdb) s
(gdb) x/x $r0
0x40023830:     0x00000007
```

LDR OFFSET

```
// Load the SRAM base address into R0, load an offset address into R1 and left shift the offset in R1 by 1
which multiplies the value
// by 2 and then add it to the base value producing 0x20000020 to be stored back into R2. Set a 1 in the
LSB and store back the value
// into R0.
LDR_Offset:
 LDR  R0, =0x20000000              // SRAM base (p42 RM)
 LDR  R1, =0x10                    // offset address
 LDR  R2, [R0, R1, LSL #1]         // 0x20000000 + (0x10 * 2) = 0x20000020, then store what is inside
0x20000020 into R2
 ORR  R2, #(1<<0)                  // put a 1 in LSB with the contents inside R2
 STR  R2, [R0]                     // store the contents inside R2 into 0x20000000
 BX  LR                            // return to caller
```

```
(gdb) disas
Dump of assembler code for function LDR_Offset:
=> 0x080003e8 <+0>:     mov.w   r0, #536870912  ; 0x20000000
   0x080003ec <+4>:     mov.w   r1, #16
   0x080003f0 <+8>:     ldr.w   r2, [r0, r1, lsl #1]
   0x080003f4 <+12>:    orr.w   r2, r2, #1
   0x080003f8 <+16>:    str     r2, [r0, #0]
   0x080003fa <+18>:    bx      lr

(gdb) s
(gdb) x/x $r0
0x20000000:     0x35010041

(gdb) s
(gdb) s
(gdb) x/x $r2
0x35010040:     0x00000000
(gdb) x/x 0x20000020
0x20000020:     0x35010040

(gdb) s
(gdb) s
(gdb) x/x $r2
0x35010041:     0x00000000
(gdb) x/x $r0
0x20000000:     0x35010041
```

MOVW MOVT

```
/// Move half word into the MSB of R0 and then another half word into the LSB of R0 and load an address
in RAM and store that value into it.
MOVW_MOVT:
 MOVW R0, #0x5678                // move half word into MSB of R0
 MOVT R0, #0x1234                // move half word into LSB of R0
 LDR  R1, =0x20002000            // load address in RAM
 STR  R0, [R1]                   // store word into the value stored inside R1
 BX  LR                          // return to caller
```

```
(gdb) disas
Dump of assembler code for function MOVW_MOVT:
=> 0x080003fc <+0>:     movw    r0, #22136      ; 0x5678
   0x08000400 <+4>:     movt    r0, #4660       ; 0x1234
   0x08000404 <+8>:     mov.w   r1, #536879104  ; 0x20002000
   0x08000408 <+12>:    str     r0, [r1, #0]
   0x0800040a <+14>:    bx      lr

(gdb) s
(gdb) x/x $r0
0x5678: 0x00000000

(gdb) s
(gdb) x/x $r0
0x12345678:     0x00000000

(gdb) s
(gdb) s
(gdb) x/x $r1
0x20002000:     0x12345678
```

FLAGS

/// Move a negative value into R0 and R1 and add them to set the N, C and T bits in the xPSR (0b10100001000000000000000000000000).
// Load the max 32-bit signed value into R0 and load a value into R1 and add them to set the N, V and T bits inside the xPSR
// (0b10010001000000000000000000000000). Load the max 32-bit unsigned value into R0 and load a value into R1 and add them to
// set the Z, C and T bits inside the xPSR (0b01100001000000000000000000000000).
Flags:

```
MOV  R0, #-1                  // move a negative value into R0
MOV  R1, #-2                  // move a negative value into R1
ADDS R0, R1                   // N, C, T bits set in xPSR (negative, carry and thumb) (p573 ARM)
LDR  R0, =0x7FFFFFFF          // load the max 32-bit signed value into R0
LDR  R1, =0x01                // load a value into R1 that when added to R0 will overflow
ADDS R0, R1                   // N, V, T bits set in xPSR (negative, overflow and thumb)
LDR  R0, =0xFFFFFFFF          // load the max 32-bit unsigned value into R0
LDR  R1, =0x01                // load a value into R1 that when added to R0 will overflow
ADDS R0, R1                   // Z, C, T bits set in xPSR (zero, carry, thumb)
BX  LR                        // return to caller
```

```
(gdb) disas
Dump of assembler code for function Flags:
=> 0x0800040c <+0>:     mov.w   r0, #4294967295
   0x08000410 <+4>:     mvn.w   r1, #1
   0x08000414 <+8>:     adds    r0, r0, r1
   0x08000416 <+10>:    mvn.w   r0, #2147483648 ; 0x80000000
   0x0800041a <+14>:    mov.w   r1, #1
   0x0800041e <+18>:    adds    r0, r0, r1
   0x08000420 <+20>:    mov.w   r0, #4294967295
   0x08000424 <+24>:    mov.w   r1, #1
   0x08000428 <+28>:    adds    r0, r0, r1
   0x0800042a <+30>:    bx      lr

(gdb) s
(gdb) s
(gdb) s
(gdb) x/x $xPSR
0xa1000000 // 0b10100001000000000000000000000000

(gdb) s
(gdb) s
(gdb) s
(gdb) x/x $xPSR
0x91000000 // 0b10010001000000000000000000000000

(gdb) s
(gdb) s
(gdb) s
(gdb) x/x $xPSR
0x61000000 // 0b01100001000000000000000000000000
```

BIT SET CLEAR

```
// Load the GPIOB_MODER register into R0 and move the value inside the register into R1, perofrm at bit
clear and then a bit set and store the value back into the register.
Bit_Set_Clear:
 LDR R0, =0x40020400            // load address of GPIOB_MODER register (p158 RM)
 LDR R1, [R0]                   // load value inside GPIOB_MODER register
 AND R1, #~(1<<1)               // clear the MODER0 bit
 ORR R1, #(1<<1)                // set the MODER0 bit
 STR R1, [R0]                   // store value into GPIOB_MODER register
 BX  LR                         // return to caller
```

```
(gdb) disas
Dump of assembler code for function Bit_Set_Clear:
=> 0x0800042c <+0>:     ldr     r0, [pc, #112]  ; (0x80004a0 <Interrupts+60>)
   0x0800042e <+2>:     ldr     r1, [r0, #0]
   0x08000430 <+4>:     bic.w   r1, r1, #2
   0x08000434 <+8>:     orr.w   r1, r1, #2
   0x08000438 <+12>:    str     r1, [r0, #0]
   0x0800043a <+14>:    bx      lr

(gdb) s
(gdb) s
(gdb) x/x $r1
0x282:  0x6801487c

(gdb) s
(gdb) x/x $r1
0x280:  0x487c6001

(gdb) s
(gdb) x/x $r1
0x282:  0x6801487c

(gdb) s
(gdb) x/x $r0
0x40020400:     0x00000282
```

CONDITIONAL BRANCHING

```
// Load the EXTI_PR register into R0 and the contents into R1 and test if bit 13 is a 0
(0b00100000000000000000010000000000)
// then move a random mock value into R0 and compare against antother random mock velue and branch
appropriately.
Conditional_Branching:
 LDR R0, =0x40013C14                  // load the address of EXTI_PR register (p211 RM)
 LDR R1, [R0]                         // load value inside EXTI_PR register
 TST R1, #(1<<13)                     // read the PR13 bit, if 0, then BEQ
 BEQ .Conditional_TST_Finish          // branch if equal
 NOP                                  // dummy logic
.Conditional_TST_Finish:
 MOV R0, #0x40                        // random mock value
 CMP R0, #0x40                        // compare random mock value to another value
 BNE .Conditional_TST_Finish          // branch if not equal
 BLE .Conditional_CMP_Finish          // branch if less than or equal
 NOP                                 // dummy logic
.Conditional_CMP_Finish:
 BX  LR                              // return to caller
```

```
(gdb) disas
Dump of assembler code for function Conditional_Branching:
=> 0x0800043c <+0>:     ldr     r0, [pc, #100]  ; (0x80004a4 <Interrupts+64>)
   0x0800043e <+2>:     ldr     r1, [r0, #0]
   0x08000440 <+4>:     tst.w   r1, #8192       ; 0x2000
   0x08000444 <+8>:     beq.n   0x8000448 <.Conditional_TST_Finish>
   0x08000446 <+10>:    nop
End of assembler dump.

(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) disas
Dump of assembler code for function .Conditional_TST_Finish:
=> 0x08000448 <+0>:     mov.w   r0, #64 ; 0x40
   0x0800044c <+4>:     cmp     r0, #64 ; 0x40
   0x0800044e <+6>:     bne.n   0x8000448 <.Conditional_TST_Finish>
   0x08000450 <+8>:     ble.n   0x8000454 <.Conditional_CMP_Finish>
   0x08000452 <+10>:    nop

(gdb) s
(gdb) s
(gdb) s
(gdb) disas
Dump of assembler code for function .Conditional_TST_Finish:
   0x08000448 <+0>:     mov.w   r0, #64 ; 0x40
   0x0800044c <+4>:     cmp     r0, #64 ; 0x40
   0x0800044e <+6>:     bne.n   0x8000448 <.Conditional_TST_Finish>
=> 0x08000450 <+8>:     ble.n   0x8000454 <.Conditional_CMP_Finish>
   0x08000452 <+10>:    nop
```

LOOPS

```
// Move a value into R0 and we have a NOP representing any logic you would want run then subtract 1
from the counter
// and compare against 0 otherwise loop.
Loops:
 MOV  R0, #0x10                 // move a value into R0
.For_Loop:
 NOP                            // dummy logic
 SUBS R0, #0x01                 // subtract a value from R0
 CMP  R0, #0x00                 // subtract R0 from a value without updating R0
 BNE  .For_Loop                 // branch if not equal
 BX  LR                         // return to caller
```

```
(gdb) disas
Dump of assembler code for function Loops:
=> 0x08000456 <+0>:     mov.w   r0, #16

(gdb) s
(gdb) disas
Dump of assembler code for function .For_Loop:
=> 0x0800045a <+0>:     nop
   0x0800045c <+2>:     subs    r0, #1
   0x0800045e <+4>:     cmp     r0, #0
   0x08000460 <+6>:     bne.n   0x800045a <.For_Loop>
   0x08000462 <+8>:     bx      lr
(gdb) x/x $r0
0x10:   0x0800019d

(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) x/x $r0
0x10: 0x0800019d
(gdb) x/x $r0
0xf:    0x00019d0

(gdb) s
(gdb) s
(gdb) s
(gdb) s
(gdb) x/x $r0
0x10: 0x0800019d
(gdb) x/x $r0
0xe:    0x019d0800
...
```

INTERRUPTS

```
// Review EXTI15_10_IRQHandler and USART1_IRQHandler. The vector table's weak reference must be
// changed to a .word as we define the handler. Review blue button press and observe 'A' displayed
// on UART and typing a char in the UART displays the char. These actions are independent of the
// main loop and will interrupt the main loop.
Interrupts:
 BX  LR
```

```
/**
 * @brief USART1 interrupt handler.
 *
 * @details This is the interrupt handler function for USART1 interrupts. It is called when
 *     USART1 generates an interrupt request. Inside this handler, it calls the
 *     `USART1_Callback` function to handle the USART1 interrupt, and then returns to
 *     the main program execution.
 *
 * @param None
 * @retval None
 */
.section .text.USART1_IRQHandler
.weak USART1_IRQHandler
.type USART1_IRQHandler, %function
USART1_IRQHandler:
 PUSH {LR}                      // push return address onto stack
 LDR R0, =0x40011000            // load address of USART1_SR register
 LDR R1, [R0]                   // load value inside USART1_SR register
 TST R1, #(1<<5)                // read the RXNE register
 BL  USART1_Callback            // call function
 POP {LR}                       // pop return address from stack
 BX  LR                         // return from the function
```

STUXNET DEMO

Our situation is we have been contracted by a classified organization to infiltrate the Natanz Nuclear Facility. The intel we have been given is that a normal rate of delay is 64 ms for the centrifuge to spin from a classified source and that if it ever reached 8 it would spin the centrifuge out of control and destroy the facility which is our goal ;) The intel provided also explains that by entering in a 1 into the terminal will act as a kill switch in the event that an Operator obverses something wrong.

The other piece of intel is that the motor is controlled by a UNL2003 driver with a stepper motor and it is designed in a full drive sequence mode which two coils are energized at a time that means two windings of stepper motor energized together. Therefore, motor runs at full torque.

Let's assemble...

```
arm-none-eabi-as -g main.s -o main.o
arm-none-eabi-ld main.o -o main.elf -T stm32f401ccux.ld
arm-none-eabi-objcopy -O binary --strip-all main.elf main.bin
openocd -f interface/stlink-v2.cfg -f target/stm32f4x.cfg -c "program main.bin 0x08000000 verify reset exit"
```

Terminal 1:

```
openocd -f board/st_nucleo_f4.cfg
```

Terminal 2:

```
arm-none-eabi-gdb main.bin
target remote :3333
monitor reset halt
```

Let's look at first 1000 instructions.

```
(gdb) x/1000i 0x08000000
```

Arch Ref Manual page 682, we see NVIC_ISER0-NVIC_ISER15 so we see the address starting at 0xE000E100 to 0xE000E13C. So lets search our binary for each as they are 4-bytes long.

```
(gdb) find 0x08000000, 0x0A000000, 0xe000e100
Pattern not found.
```

Ok let's try the next one.

```
(gdb) find 0x08000000, 0x0A000000, 0xe000e1040x80004b8
1 pattern found.
```

Great! Let's examine what is at the address minus a few bytes.

```
(gdb) x/100i 0x8000300
```

Notice there are bytes followed by two bytes #13 and #10 this can signify a /r/n so we might have our UART message, at least one of them.

```
0x800044a:   cmp     r0, #49 ; 0x31
```

This is a 1 and we know from our intel this is the kill switch value.

```
0x80003ae: mov.w r7, #64 ; 0x40
0x80003b2:   bl      0x80002d8
0x80003b6:   cmp     r7, #64 ; 0x40
```

We know from our intel that 64ms is the value we need to change to 8ms so this could be it! The first 0x40 is the actual speed and the 2nd 0x40 is the expected value.

Fire up STM32CubeProgrammer.

0x0800044a, we see D1012831 so we need to change the last byte to 01.
0x080003ae, we see the value of 0740F04F so we need to change the 40 to 08.
0x080003b6, we see the value of D1254F40 so we need to change the 40 to 08.

BLAM!