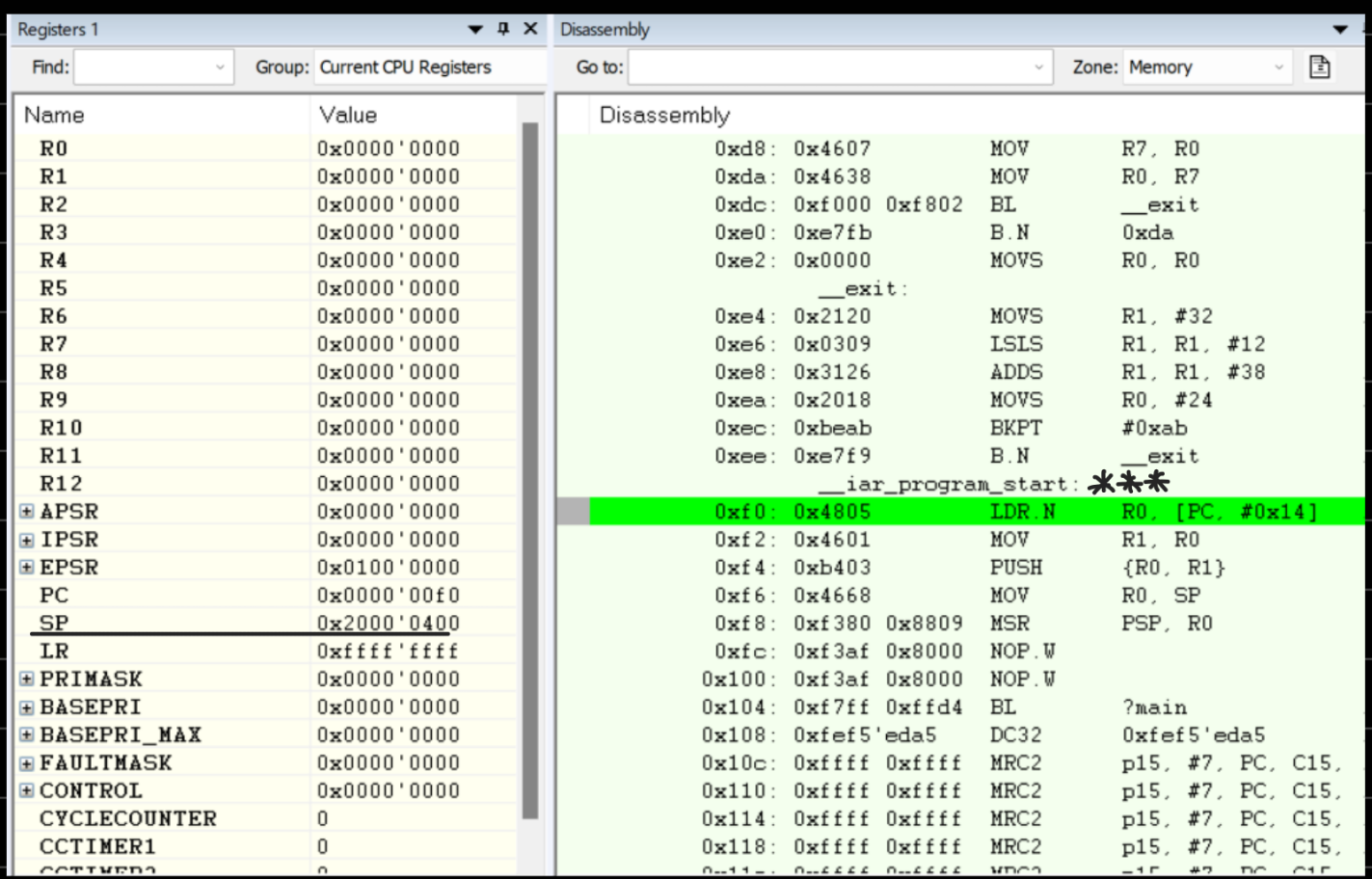
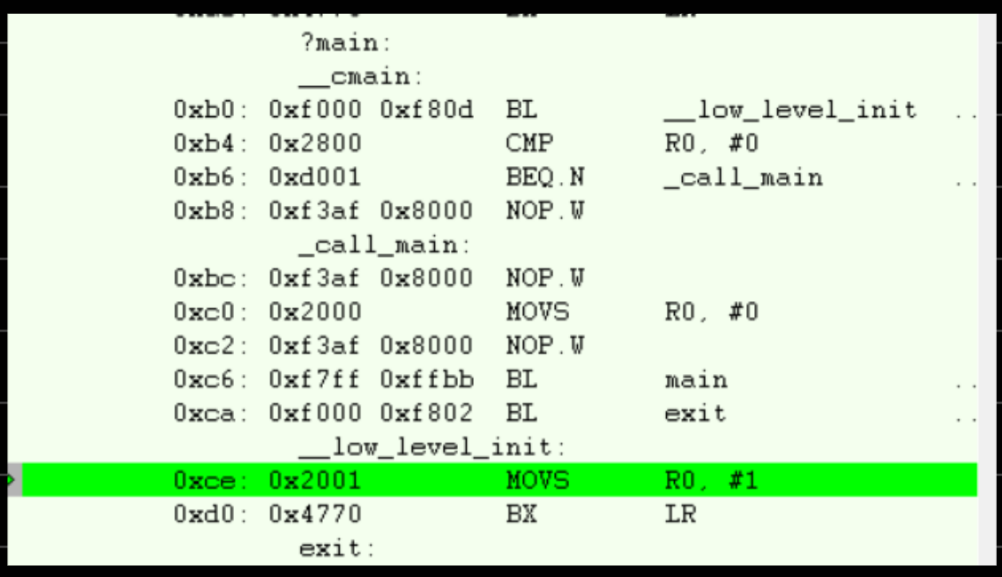
What is startup code

In IAR set project option debugger → (uncheck) run to main. When we compile our program Stellaris we find that main does not run from beging. We start from the label "_iar_program_start", with SP pointing to 0x2000 0400



The first interesting branch call is to the function with label "?main" (not the actual main func) using BL (branch with link). The next call is to the label "_low_level_init"



This is where custom hardware initialization can occur. We see R0 gets loaded with 1 and sent back to just after where the "_low_level_init" was initially called (0xb4) if R0 and 0 is equal there is a branch to "_call_main". In our case no branching occurs because R0=1

Next we generate a linker map file. In IAR project options, set linker → list → (check) generate linker map. After rebuilding our code we can see the linker map "c.map".

The "module summary" section of the linker file breaks down how the memory in the RAM is being used in our code.

```
    Module                ro code  rw data
    ------                -------  -------
command line/config:
    ----------------------------------
    Total:

D:\EEE\Embedded Systems\Tiva Launchpad\Projects\Lesson 12\Debug\Obj: [1]
    delay.o                    20
    main.o                     92
    ----------------------------------
    Total:                    112

dl7M_tln.a: [2]
    exit.o                      4
    low_level_init.o            4
    ----------------------------------
    Total:                      8

rt7M_tl.a: [3]
    XXexit.o                   12
    cexit.o                    10
    cmain.o                    30
    cstartup_M.o               28
    vector_table_M.o           64
    vectortrap_M.o              2
    ----------------------------------
    Total:                    146

    Gaps                        2
    Linker created                      1'024
----------------------------------------------
    Grand Total:              268       1'024
```

RW code uses 1024 bytes

Next the "Placement summary" section is used for showing how the MCU ROM memory is allocated for code. ".intvec" is between address 0x0 - 0x140 of ROM. ".text" indicates our code.

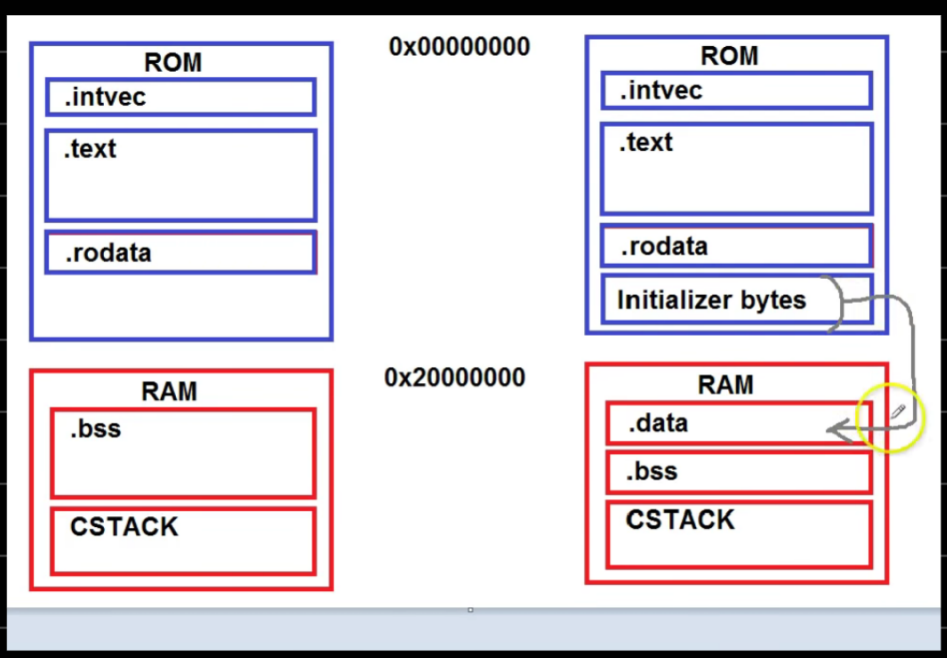
Initialization of code

The "initialyzer bytes" section (in the ROM space) in the linker shows that it is equal to the ".data" section (in the RAM space).

```
  .rodata                const          0x194     0x8  main.o [1]
  Initializer bytes      const          0x19c     0x4  <for P2-1>
  .rodata                const          0x1a0     0x0  copy_init3.o [3]
                                      - 0x1a0   0x160

"P2", part 1 of 2:                                0x4
  P2-1                              0x2000'0000   0x4  <Init block>
    .data                inited     0x2000'0000   0x4  main.o [1]
                                  - 0x2000'0004   0x4
```

What happens is the linker reorders the variables and their data in the Initializer bytes section and copies them to the data section in the ram.



The .bss section contains all uninitialized data.

The tiva cortex startup

Two important questions:

Q1. How did stack pointer get its initial value?

Q2. How did program counter start at "_iar_program_start"?

Name	Value
R0	0x0000'0000
R1	0x0000'0000
R2	0x0000'0000
R3	0x0000'0000
R4	0x0000'0000
R5	0x0000'0000
R6	0x0000'0000
R7	0x0000'0000
R8	0x0000'0000
R9	0x0000'0000
R10	0x0000'0000
R11	0x0000'0000
R12	0x0000'0000
APSR	0x0000'0000
IPSR	0x0000'0000
EPSR	0x0100'0000
PC	0x0000'0178
SP	0x2000'0408
LR	0xffff'ffff

Disassembly

```
        0x160: 0x4607          MOV       R7, R0
        0x162: 0x4638          MOV       R0, R7
        0x164: 0xf000 0xf802   BL        __exit
        0x168: 0xe7fb          B.N       0x162
        0x16a: 0x0000          MOVS      R0, R0
                 __exit:
        0x16c: 0x2120          MOVS      R1, #32
        0x16e: 0x0309          LSLS      R1, R1, #12
        0x170: 0x3126          ADDS      R1, R1, #38
        0x172: 0x2018          MOVS      R0, #24
        0x174: 0xbeab          BKPT      #0xab
        0x176: 0xe7f9          B.N       __exit
                 __iar_program_start:
        0x178: 0x4805          LDR.N     R0, [PC, #0x
        0x17a: 0x4601          MOV       R1, R0
        0x17c: 0xb403          PUSH      {R0, R1}
        0x17e: 0x4668          MOV       R0, SP
        0x180: 0xf380 0x8809   MSR       PSP, R0
        0x184: 0xf3af 0x8000   NOP.W
```

?

① Ans:- The data at address 0x0 in ARM is hardwired to 0x2000 0408 and is copied to SP on startup.

Disassembly

Go to: Zone: Memory

Disassembly

```
                __vector_table:
        0x0:  0x2000'0408      DC32      CSTACK$$Limit
        0x4:  0x0000'0179      DC32      __iar_program_start
        0x8:  0x0000'00fb      DC32      BusFault_Handler
        0xc:  0x0000'00fb      DC32      BusFault_Handler
       0x10:  0x0000'00fb      DC32      BusFault_Handler
       0x14:  0x0000'00fb      DC32      BusFault_Handler
       0x18:  0x0000'00fb      DC32      BusFault_Handler
       0x1c:  0x0000'0000      DC32      __vector_table
       0x20:  0x0000'0000      DC32      __vector_table
       0x24:  0x0000'0000      DC32      __vector_table
       0x28:  0x0000'0000      DC32      __vector_table
       0x2c:  0x0000'00fb      DC32      BusFault_Handler
       0x30:  0x0000'00fb      DC32      BusFault_Handler
       0x34:  0x0000'0000      DC32      __vector_table
       0x38:  0x0000'00fb      DC32      BusFault_Handler
       0x3c:  0x0000'00fb      DC32      BusFault_Handler
int main()
{
                main:
        0x40:  0xb53e          PUSH      {R1-R5, LR}
  Window w1 = { {p1.x, p1.y}, {234, 0xCACA} };
        0x42:  0x4816          LDR.N     R0, ??main_0
        0x44:  0xe9d0 0x2300   LDRD      R2, R3, [R0]
        0x48:  0xe9cd 0x2300   STRD      R2, R3, [SP]
  Window w1 = { {p1.x, p1.y}, {234, 0xCACA} };
```

② Ans:- The PC is also loaded with 0x00000179 which takes us to the "_iar_program_start" label. The least significant bit is zeroed in the PC due to ARM thum instructions

The vector table is also present in address 0x0. However the IAR vector table does not matchup with the one in TIVA's datasheet, thus the generic IAR vector table needs to be removed.

Initializing our own vector tables
- → contains reset address of SP
- → start address of all exception handlers
- → least significant bit of all addresses must be 1 (For thumb mode)

In week 12 (lesson 12) code we can see that there is a symbol "CStack$$Limit". This is a placeholder created for initializing the base stack address during startup of TIVA MCU.

Disassembly

Go to: CSTACK$$Limit Zone: Memory

Disassembly

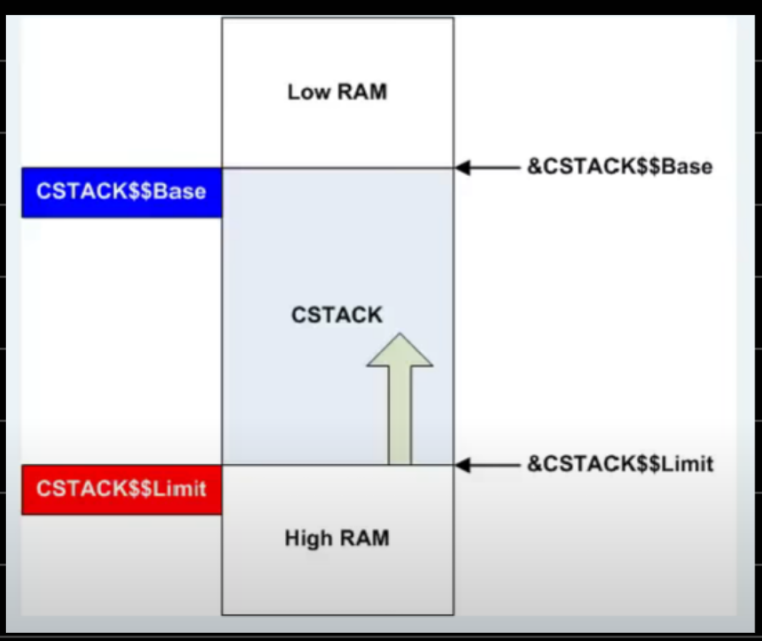
```
__vector_table:
        0x0: 0x2000'0408    DC32      CSTACK$$Limit
        0x4: 0x0000'0179    DC32      __iar_program_sta
        0x8: 0x0000'00fb    DC32      BusFault_Handler
        0xc: 0x0000'00fb    DC32      BusFault_Handler
       0x10: 0x0000'00fb    DC32      BusFault_Handler
       0x14: 0x0000'00fb    DC32      BusFault_Handler
       0x18: 0x0000'00fb    DC32      BusFault_Handler
       0x1c: 0x0000'0000    DC32      __vector_table
       0x20: 0x0000'0000    DC32      __vector_table
       0x24: 0x0000'0000    DC32      __vector_table
       0x28: 0x0000'0000    DC32      __vector_table
       0x2c: 0x0000'00fb    DC32      BusFault_Handler
       0x30: 0x0000'00fb    DC32      BusFault_Handler
       0x34: 0x0000'0000    DC32      __vector_table
       0x38: 0x0000'00fb    DC32      BusFault_Handler
       0x3c: 0x0000'00fb    DC32      BusFault_Handler
int main()
```

```
CSTACK$$Limit:
0x2000'0408: 0xffff        DC16      -1
0x2000'040a: 0xffff        DC16      -1
0x2000'040c: 0xffff        DC16      -1
0x2000'040e: 0xffff        DC16      -1
0x2000'0410: 0xffff        DC16      -1
0x2000'0412: 0xffff        DC16      -1
0x2000'0414: 0xffff        DC16      -1
0x2000'0416: 0xffff        DC16      -1
0x2000'0418: 0xffff        DC16      -1
0x2000'041a: 0xffff        DC16      -1
0x2000'041c: 0xffff        DC16      -1
0x2000'041e: 0xffff        DC16      -1
0x2000'0420: 0xffff        DC16      -1
0x2000'0422: 0xffff        DC16      -1
0x2000'0424: 0xffff        DC16      -1
0x2000'0426: 0xffff        DC16      -1
0x2000'0428: 0xffff        DC16      -1
0x2000'042a: 0xffff        DC16      -1
0x2000'042c: 0xffff        DC16      -1
0x2000'042e: 0xffff        DC16      -1
0x2000'0430: 0xffff        DC16      -1
```

We will use this symbol as well to initialize our user defined startup code.

*Read IAR C-language extensions for this

Iar linker uses the symbols "CSTACK$$Base" to indicate the starting address of a section, and the symbol "CSTACK$$Limit" to indicate the sections ending address in stack. As with programs, the "CSTACK$$Limit" is the initial stack pointer that is placed and the address grows towards low ram, where the address of "CSTACK$$Base" is then placed



When we try to initialize the stack pointer in our startup code to the address of "CSTACK$$Limit", our build fails because these section symbols are created by the linker AFTER the compilation. To resolve this use keyword "extern int" when declaring CSTACK:

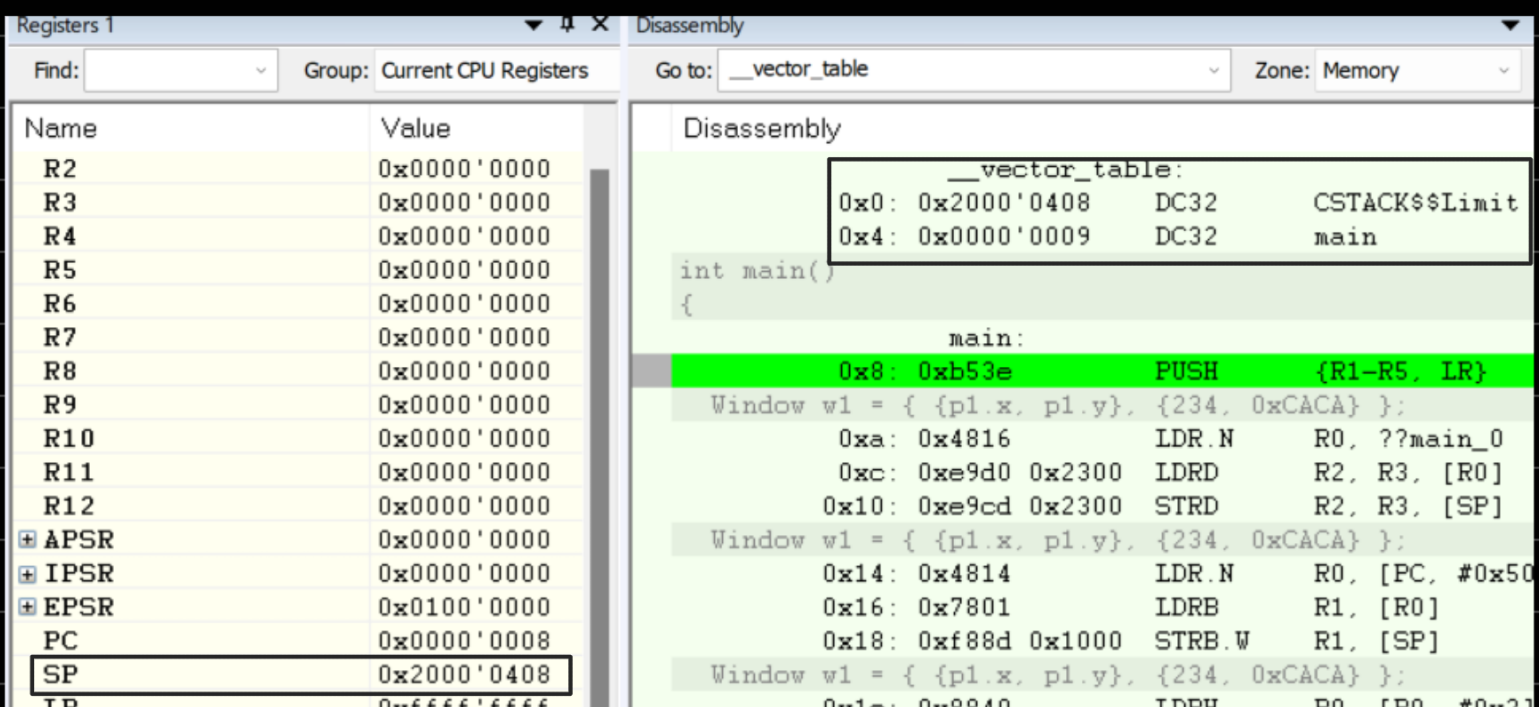
CSTACK$$Limit

```
/* Startup Code */

extern int CSTACK$$Limit;       // This declares our section symbol, but the variable isn't saved
                                // in memory due to 'extern'

int const __vector_table[] @ ".intvec" = { (int)&CSTACK$$Limit, 0x9 };     //convention for naming startup code
```

This creates symbol CSTACK and initializes pointer correctly:



* CSTACK$$Limit address can change based on stack size allocation in IAR

Next we initialize the rest of the vector table according to the vector table of the MCU datasheet.

Firstly we need to provide the reset in our vector table. The symbol used for this is "__iar_program_start" (as seen from linker script and dissassembly of lesson 12).

We can now implement our vector table using the datasheet. All prototypes and symbols for vector table entries are given/declared in the "tm4c_cmsis.h" header file. There are reserved spots in the vector table of the datasheet, to implement this we simply put a 0 in the same position of our vector table.

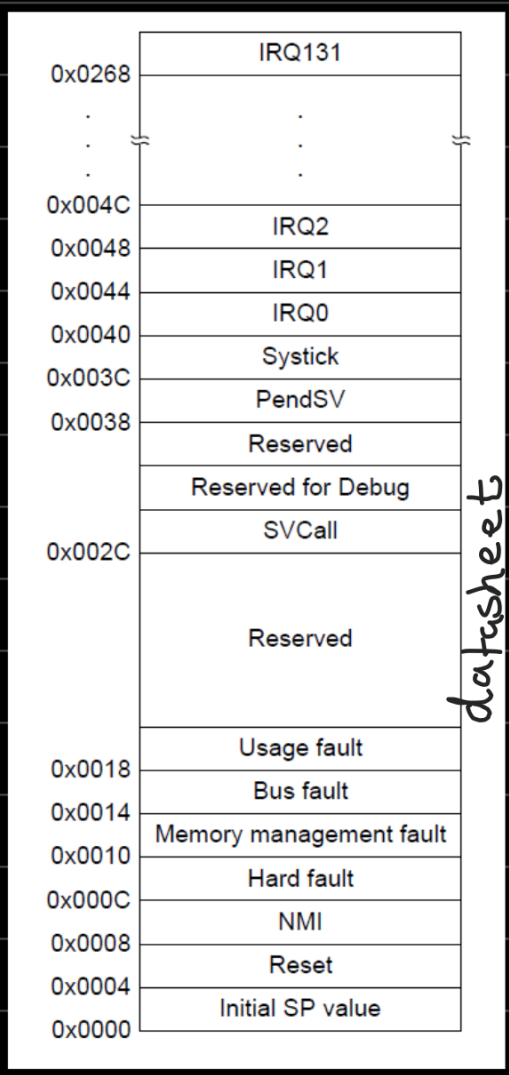
vector table prototypes in "tm4c_cmsis.h" :

```
/*
 * ================================================
 * ---------- Interrupt Handler Prototypes ---------
 * ================================================
 */
void NMI_Handler(void);
void HardFault_Handler(void);
void MemManage_Handler(void);
void BusFault_Handler(void);
void UsageFault_Handler(void);
void HardFault_Handler(void);

void SVC_Handler(void);
void DebugMon_Handler(void);
void PendSV_Handler(void);
void SysTick_Handler(void);

void GPIOPortA_IRQHandler(void);
void GPIOPortB_IRQHandler(void);
void GPIOPortC_IRQHandler(void);
void GPIOPortD_IRQHandler(void);
void GPIOPortE_IRQHandler(void);
void UART0_IRQHandler(void);
void UART1_IRQHandler(void);
void SSI0_IRQHandler(void);
void I2C0_IRQHandler(void);
void PWMFault_IRQHandler(void);
void PWMGen0_IRQHandler(void);
void PWMGen1_IRQHandler(void);
void PWMGen2_IRQHandler(void);
void QEI0_IRQHandler(void);
void ADCSeq0_IRQHandler(void);
void ADCSeq1_IRQHandler(void);
```

VS



Now to complete the vector table, we need to write the code for the other exception handlers, such as the hardfault handler, etc. We can use the function assert_failed (already defined in "Tm4c_cmsis.h" and use it in our exception handlers.

main.c | tm4c_cmsis.h | delay.c | Startup_tm4c.c *

```
/**
 * @brief  Setup the initial configuration of the microcontroller
 * @param  none
 * @return none
 *
 * Initialize the system clocking according to the user configuration.
 */
extern void SystemInit (void);

/**
 * Assertion handler
 *
 * @brief  A function to handle an assertion.
 *         This function should be defined at the application level
 *         and should never return to the caller.
 */
extern void assert_failed (char const *file, int line);
```

"assert_failed" takes 2 arguements → a pointer to a file and the line number at which the call occurred at. With this we can code the exception fault's :

```
void Hardfault_Handler()
{
  assert_failed("HARDFAULT", __LINE__);
}

void MemManage_Handler()
{
  assert_failed("MemManage", __LINE__);
}

void BusFault_Handler()
{
  assert_failed("BusFault", __LINE__);
}

void UsageFault_Handler()
{
  assert_failed("UsageFault", __LINE__);
}
```

Now to finish the final exception handlers that are not faults such as "SVCall", "PendSV", etc, we can provide them "weak aliases" that indicate that if certain handlers are not used then we can use a symbol to represent these exception handlers

```
void Unused_Handler(void)
{
  assert_failed("Unused", __LINE__);
}

// Defining the final unused exception handlers using weak aliases
#pragma weak SVC_Handler = Unused_Handler
#pragma weak DebugMon_Handler = Unused_Handler
#pragma weak PendSV_Handler = Unused_Handler
#pragma weak SysTick_Handler = Unused_Handler
```

unfortunately, our startup table won't compile yet, and we need to create a board support package file

```
/* Board Support Package */

#include "tm4c_cmsis.h"

void assert_failed (char const *file, int line)
{
  /* This code is modified by user according to project requirements */

  NVIC_SystemReset(); //System reset by TMSIS
}
```

finally, our own vector table:

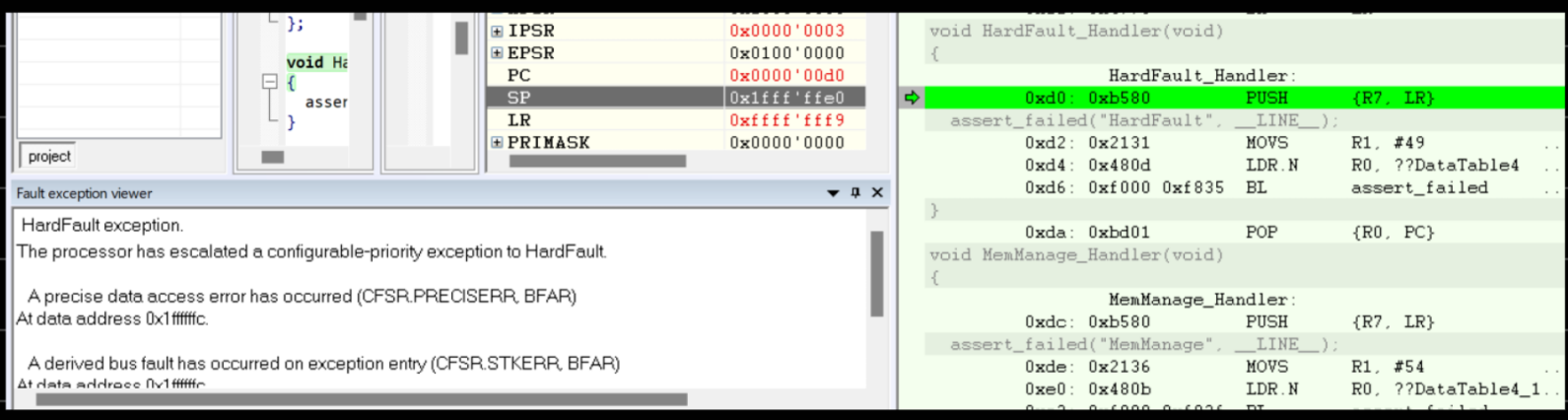
```
Disassembly
                __vector_table:
        0x0:  0x2000'0408     DC32        CSTACK$$Limit
        0x4:  0x0000'01f5     DC32        __iar_program_start
        0x8:  0x0000'0177     DC32        NMI_Handler
        0xc:  0x0000'00d1     DC32        HardFault_Handler
       0x10:  0x0000'00dd     DC32        MemManage_Handler
       0x14:  0x0000'00e9     DC32        BusFault_Handler
       0x18:  0x0000'00f5     DC32        UsageFault_Handler
       0x1c:  0x0000'0000     DC32        __vector_table
       0x20:  0x0000'0000     DC32        __vector_table
       0x24:  0x0000'0000     DC32        __vector_table
       0x28:  0x0000'0000     DC32        __vector_table
       0x2c:  0x0000'0101     DC32        Unused_Handler
       0x30:  0x0000'0101     DC32        Unused_Handler
       0x34:  0x0000'0000     DC32        __vector_table
       0x38:  0x0000'0101     DC32        Unused_Handler
       0x3c:  0x0000'0101     DC32        Unused_Handler
int main()
```

Now we can use "fault injections" to see how the exception handlers execute. First set a breakpoint at delay function in assembly and then change SP value to 0x20000000 when delay is reached.

```
R11   0x0000'0000
R12   0xabcd'007b
APSR  0x2000'0000
IPSR  0x0000'0000
EPSR  0x0100'0000
PC    0x0000'00bc
SP    0x2000'0000
LR    0x0000'0091
```

```
{
                delay:
        0xbc:  0xb401         PUSH        {R0}
        0xbe:  0xe002         B.N         ??delay_0          ...
  --iter;
                ??delay_1:
        0xc0:  0x9800         LDR         R0, [SP]
        0xc2:  0x1e40         SUBS        R0, R0, #1
```

single stepping leads to HardFault_handler as SP falls below RAM address:



However hardfault handler does not work properly since it falls beyond the stack. This freezes the MCU and hence needs to be fixed. To do this declare all exception handlers as "Stackless." Next repeat the fault injection process again and single step from delay. After the stackless declaration, the program goes to Hardfault which calls assert_failed and then "NVIC_SystemReset" which resets our MCU.