Getting started with programming

A general framework for an embedded C program:

```
// 0. Documentation Section
// main.c
// Comments here about what it does
// Authors: Daniel Valvano, Jonathan Valvano and Ramesh Yerraballi
// Date: August 30, 2020
// Comments here describe the hardware connections

// 1. Pre-processor Directives Section
// Constant declarations to access port registers using
// symbolic names instead of addresses
#define GPIO_PORTF_DATA_R (*((volatile unsigned long *)0x400253FC))

// 2. Declarations Section
// Global Variables go here

// Function Prototypes
void PortF_Init(void);

// 3. Subroutines Section
// main is mandatory for a C Program to be executable
int main(void){
  PortF_Init(); // Call initialization
// other initialization goes here
  while(1){
// main loop executed over and over
  }
}
// Function implementations go at end
void PortF_Init(void){
// implement functions as needed
}
```

this is a function to initialize Port F.

The same general program framework in Assembly:

```
; 0) Initial comments explain what it does
; main.s
; Authors: Daniel Valvano, Jonathan Valvano and Ramesh Yerraballi
; Date: August 11, 2020

; Explain the hardware connections

; 1) Constant declarations to access port registers
GPIO_PORTF_DATA_R EQU 0x400253FC

; 2) Global variables go in RAM
    AREA DATA, ALIGN=2

; 3) Stuff to make it fit together
    EXTERN TExaS_Init
;Programs and constants go in ROM
    AREA |.text|, CODE, READONLY, ALIGN=2
    THUMB
    EXPORT Start
; 4) main program
Start
    BL PortF_Init ; initialize Port F
;other initialization

loop
;the main loop executes over and over
```

*semicolon statements are comments

start label to indicate main program

function label (here program jumps to the actual initialization function

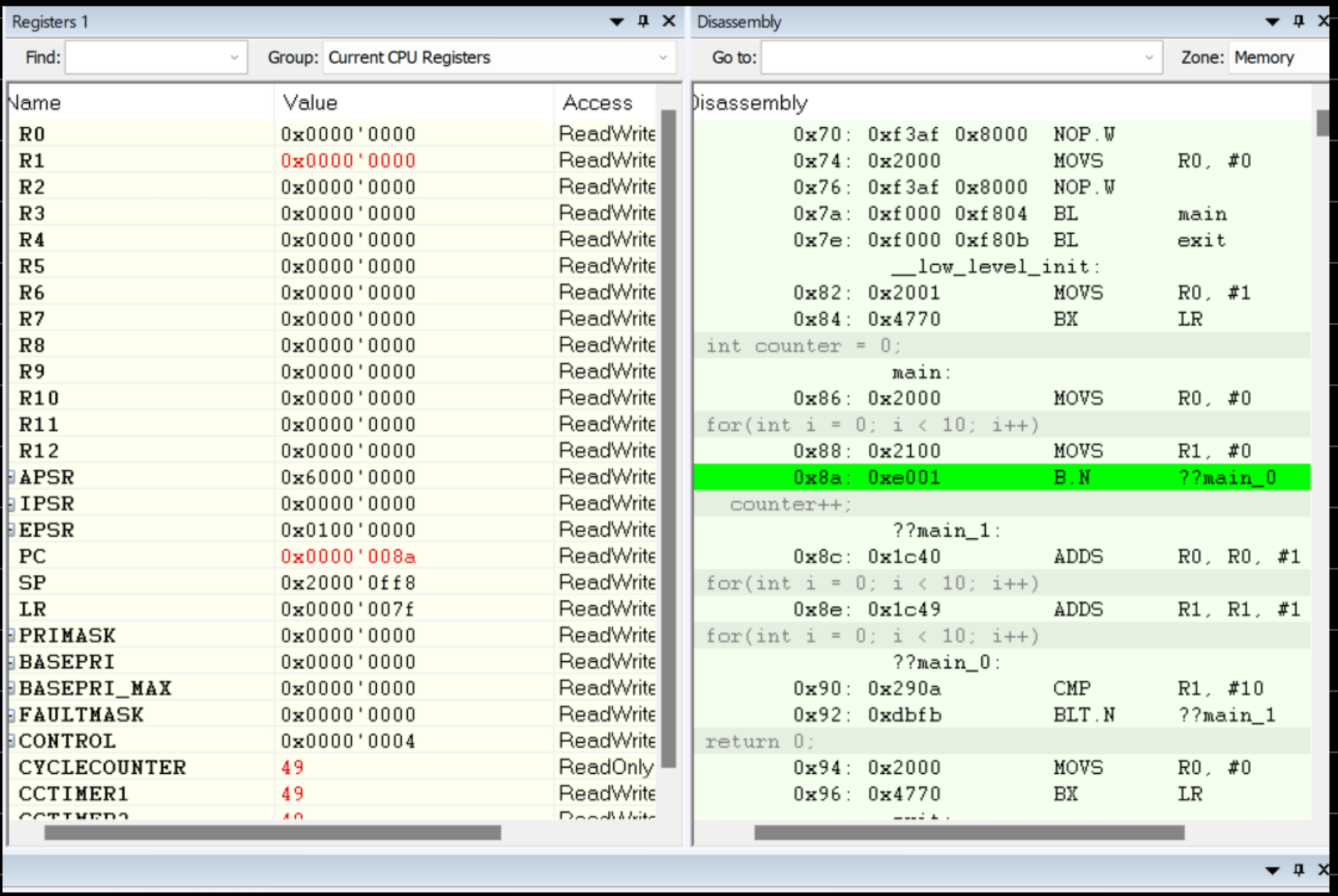
```
    B loop
; 5) function implementations go at the end

PortF_Init
; function implementation
    BX LR

; 6) stuff to make it fit together
    ALIGN
    END
```

PortF_Init → Function for initialization

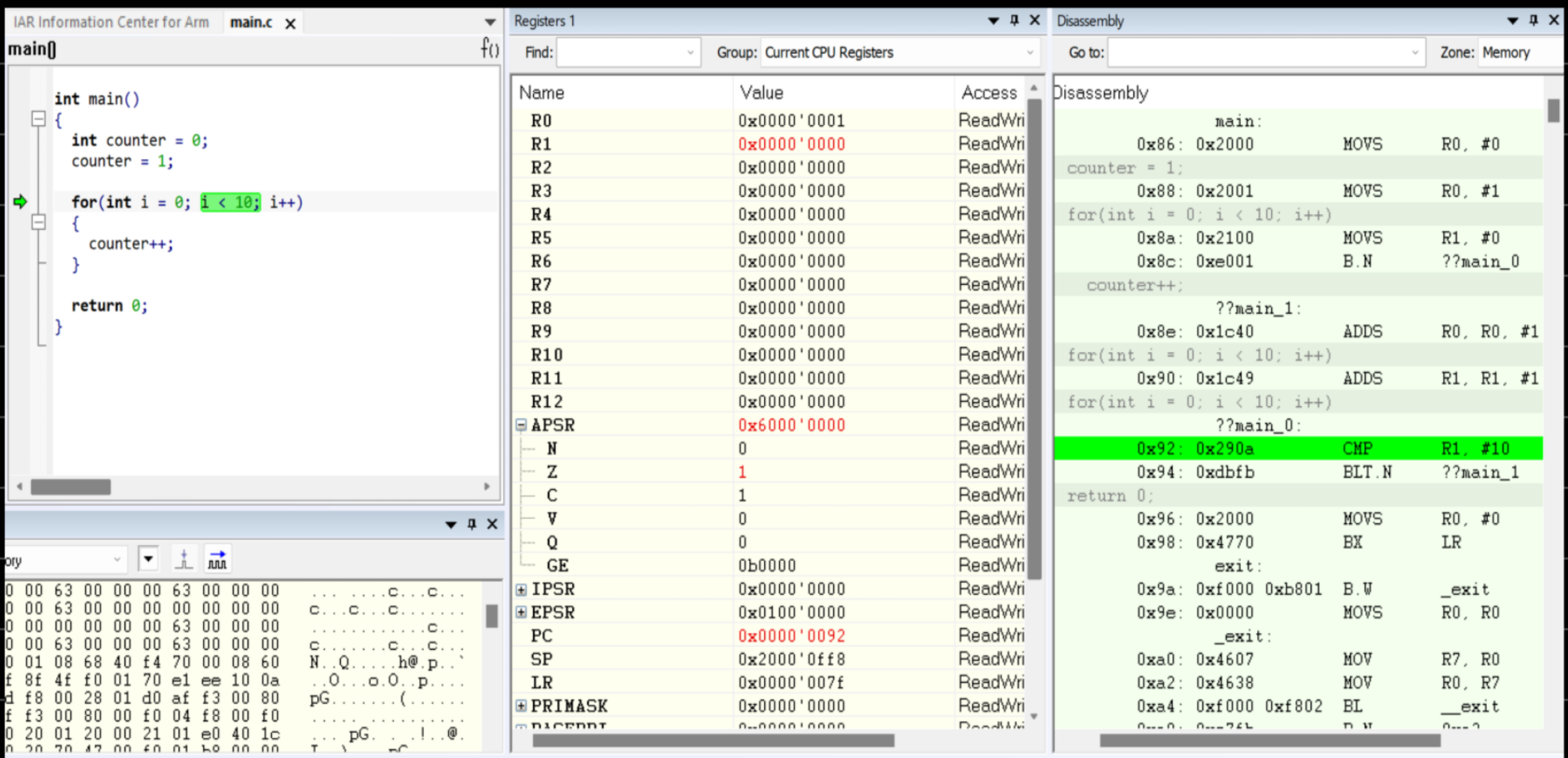
Program flow → Embedded C goes from top to bottom and increments the address. The current address of the program is saved in the PC register. Machine code before the label "main:" denote the MCU program within the ROM space.



Registers 1
Find:
Group: Current CPU Registers
Name Value Access
R0 0x0000'0000 ReadWrite
R1 0x0000'0000 ReadWrite
R2 0x0000'0000 ReadWrite
R3 0x0000'0000 ReadWrite
R4 0x0000'0000 ReadWrite
R5 0x0000'0000 ReadWrite
R6 0x0000'0000 ReadWrite
R7 0x0000'0000 ReadWrite
R8 0x0000'0000 ReadWrite
R9 0x0000'0000 ReadWrite
R10 0x0000'0000 ReadWrite
R11 0x0000'0000 ReadWrite
R12 0x0000'0000 ReadWrite
APSR 0x6000'0000 ReadWrite
IPSR 0x0000'0000 ReadWrite
EPSR 0x0100'0000 ReadWrite
PC 0x0000'008a ReadWrite
SP 0x2000'0ff8 ReadWrite
LR 0x0000'007f ReadWrite
PRIMASK 0x0000'0000 ReadWrite
BASEPRI 0x0000'0000 ReadWrite
BASEPRI_MAX 0x0000'0000 ReadWrite
FAULTMASK 0x0000'0000 ReadWrite
CONTROL 0x0000'0004 ReadWrite
CYCLECOUNTER 49 ReadOnly
CCTIMER1 49 ReadWrite
Disassembly
Go to:
Zone: Memory
Disassembly
0x70: 0xf3af 0x8000 NOP.W
0x74: 0x2000 MOVS R0, #0
0x76: 0xf3af 0x8000 NOP.W
0x7a: 0xf000 0xf804 BL main
0x7e: 0xf000 0xf80b BL exit
__low_level_init:
0x82: 0x2001 MOVS R0, #1
0x84: 0x4770 BX LR
int counter = 0;
main:
0x86: 0x2000 MOVS R0, #0
for(int i = 0; i < 10; i++)
0x88: 0x2100 MOVS R1, #0
0x8a: 0xe001 B.N ??main_0
counter++;
??main_1:
0x8c: 0x1c40 ADDS R0, R0, #1
for(int i = 0; i < 10; i++)
0x8e: 0x1c49 ADDS R1, R1, #1
for(int i = 0; i < 10; i++)
??main_0:
0x90: 0x290a CMP R1, #10
0x92: 0xdbfb BLT.N ??main_1
return 0;
0x94: 0x2000 MOVS R0, #0
0x96: 0x4770 BX LR

MCU program is at address 0x0000 008a and so is the PC register

Loops → alters the program flow and the PC register skips some of the addresses in main and goes directly to the loop (starts with comparing i in R1 register to 10). Address of PC jumps to 0x92 from 0x8a



IAR Information Center for Arm
main.c
main()
int main()
{
int counter = 0;
counter = 1;
for(int i = 0; i < 10; i++)
{
counter++;
}
return 0;
}
Registers 1
Find:
Group: Current CPU Registers
Name Value Access
R0 0x0000'0001 ReadWri
R1 0x0000'0000 ReadWri
R2 0x0000'0000 ReadWri
R3 0x0000'0000 ReadWri
R4 0x0000'0000 ReadWri
R5 0x0000'0000 ReadWri
R6 0x0000'0000 ReadWri
R7 0x0000'0000 ReadWri
R8 0x0000'0000 ReadWri
R9 0x0000'0000 ReadWri
R10 0x0000'0000 ReadWri
R11 0x0000'0000 ReadWri
R12 0x0000'0000 ReadWri
APSR 0x6000'0000 ReadWri
N 0 ReadWri
Z 1 ReadWri
C 1 ReadWri
V 0 ReadWri
Q 0 ReadWri
GE 0b0000 ReadWri
IPSR 0x0000'0000 ReadWri
EPSR 0x0100'0000 ReadWri
PC 0x0000'0092 ReadWri
SP 0x2000'0ff8 ReadWri
LR 0x0000'007f ReadWri
PRIMASK 0x0000'0000 ReadWri
Disassembly
Go to:
Zone: Memory
Disassembly
main:
0x86: 0x2000 MOVS R0, #0
counter = 1;
0x88: 0x2001 MOVS R0, #1
for(int i = 0; i < 10; i++)
0x8a: 0x2100 MOVS R1, #0
0x8c: 0xe001 B.N ??main_0
counter++;
??main_1:
0x8e: 0x1c40 ADDS R0, R0, #1
for(int i = 0; i < 10; i++)
0x90: 0x1c49 ADDS R1, R1, #1
for(int i = 0; i < 10; i++)
??main_0:
0x92: 0x290a CMP R1, #10
0x94: 0xdbfb BLT.N ??main_1
return 0;
0x96: 0x2000 MOVS R0, #0
0x98: 0x4770 BX LR
exit:
0x9a: 0xf000 0xb801 B.W _exit
0x9e: 0x0000 MOVS R0, R0
_exit:
0xa0: 0x4607 MOV R7, R0
0xa2: 0x4638 MOV R0, R7
0xa4: 0xf000 0xf802 BL __exit

After CMP the application program status register APSR N bit to 1 (not zero is true) and then resets. After that counter is incremented and PC address goes to 0x8e and then PC goes to 0x90 where i is incremented. Program flow then goes to 0x92 for CMP between i and 10 and the process repeats till i=10 and the Z and C bits become 1 which indicates iteration end.

The 0x94 address line BLT instruction decides how the branch jump (to an address) occurs, and kicks in when N=1 of APSR.

isassembly

```
        0x86: 0x2000          MOVS     R0, #0
counter = 1;
        0x88: 0x2001          MOVS     R0, #1
for(int i = 0; i < 10; i++)
        0x8a: 0x2100          MOVS     R1, #0
        0x8c: 0xe001          B.N      ??main_0
  counter++;
                ??main_1:
        0x8e: 0x1c40          ADDS     R0, R0, #1
for(int i = 0; i < 10; i++)
        0x90: 0x1c49          ADDS     R1, R1, #1
for(int i = 0; i < 10; i++)
                ??main_0:
        0x92: 0x290a          CMP      R1, #10
        0x94: 0xdbfb          BLT.N    ??main_1
return 0;
        0x96: 0x2000          MOVS     R0, #0
        0x98: 0x4770          BX       LR
                exit:
        0x9a: 0xf000 0xb801   B.W      _exit
        0x9e: 0x0000          MOVS     R0, R0
                _exit:
        0xa0: 0x4607          MOV      R7, R0
        0xa2: 0x4638          MOV      R0, R7
        0xa4: 0xf000 0xf802   BL       __exit
        0xa8: 0xe7fb          B.N      0xa2
```

To decide how the branching in a loop works, we take the instruction at jump address (0x94) instruction → 0xdbfb look up instruction encoding. In this case first bit = d and encoding T1. Next nibble denotes condition, in this case b. Last two nibbles are signed offset. In this case last two nibble = fb

= 1111 1011 = -5

add last nibble

From PC address to get next PC address to jump to (which is the beginnig of the branch).

∴ next address will be = 0x94 - 5 - 1 = 0x8E

-5-1 because cortex M cpu uses thumb instructions which are always even aligned