Assembly and C

R. Ferrero

Politecnico di Torino

Dipartimento di Automatica e Informatica (DAUIN)

Torino - Italy

Ways to mix assembly and C

- calling a C function from assembly file
- calling an assembly subroutine from C file
- inline assembly: assembly code in a C file

C function call from assembly

- visibility of the C function
  - how does the assembler know the name of the function?
- exchanging data
  - how are parameters passed to the C function?
  - where is the result stored?

Visibility of the C function

- Two directives notify the assembler of the name of a symbol defined in another file:

  `IMPORT` *`symbolName`* `{[options]}`

  `EXTERN` *`symbolName`* `{[options]}`

- `IMPORT` always imports the symbol.
  - The linker generates an error if the symbol is not defined elsewhere.
- `EXTERN` imports the symbol only if it is used.
  - The linker generates an error if the symbol is used in the assembly file but not defined elsewhere.

Options of IMPORT and EXTERN

- `WEAK`: prevents the linker
  - generating an error if the symbol is not defined
  - searching libraries that are not already included
- `DATA` | `CODE`: treats the symbol either as data or code when source is assembled and linked
- `SIZE =` *value*: specifies the size. If missing:
  - for `PROC` symbols: size of the code until `ENDP`
  - for other symbols: size of instruction or data on the same source line of the symbol
  - if there is no instruction or data: size is zero.

Example: startup code

```
Reset_Handler   PROC
    EXPORT  Reset_Handler [WEAK]
    IMPORT  SystemInit
    IMPORT  __main
    LDR     R0, =SystemInit
    BLX     R0
    LDR     R0, =__main
    BX      R0
    ENDP
```

__main Vs main

- `main()` is the user-defined function.
- Embedded applications need an initialization sequence before `main()` function starts. This is called the startup code or boot code.
- The ARM C library contains pre-compiled and pre-assembled code sections for startup.
- The linker includes the necessary code from the C library to create a custom startup code.
- `__main` is the entry point to the startup code in the ARM C library.

Data from Assembly to C function

- Data exchange is regulated by ARM Architecture Procedure Call Standard (AAPCS).
- The first 4 parameters are passed in `r0-r3`.
- Further parameters are passed in the stack.
  - After returning, they must be removed from stack
- C function returns the value in `r0-r3`
  - 32-bit sized type -> `r0`
  - 64-bit sized type -> `r0-r1`
  - 128-bit sized type -> `r0-r3`

Example: square root

- Square root is not available in the Thumb-2 instruction set.
- Workaround: in the assembly code, a C function that computes the square root is called.
- The C function receives one parameter and returns the integer square root.

C code

```
#include <math.h>
int intSquareRoot(int intNumber) {
    double realNumber;
    int result;
    realNumber = sqrt(intNumber);
    result = floor(realNumber+0.5);
    return result;
}
```

Assembly code

```
EXTERN intSquareRoot
MOV r0, #26 ; first parameter
BL intSquareRoot
```

- At the end, `r0` = 5.
- Note that other registers may have been changed, e.g., `r1`, `r2`, and `r3`. According to AAPCS, they are scratch registers.

Exercise

The 4th degree polynomial equation

$$ax^4 + bx^3 + cx^2 + dx^1 + e = 0$$

has 4 solutions:

$$x_{1,2} = -\frac{b}{4a} - Q \pm \frac{1}{2}\sqrt{-4Q^2 - 2p + \frac{S}{Q}}$$

$$x_{3,4} = -\frac{b}{4a} + Q \pm \frac{1}{2}\sqrt{-4Q^2 - 2p + \frac{S}{Q}}$$

Exercise

$$p = \frac{8ac - 3b^2}{8a^2}$$

$$S = \frac{8a^2d - 4abc + b^3}{8a^3}$$

$$Q = \frac{1}{2}\sqrt{-\frac{2}{3}p + \frac{1}{3a}\left(\Delta_0 + \frac{q}{\Delta_0}\right)}$$

$$\Delta_0 = \sqrt[3]{\frac{s + \sqrt{s^2 - 4q^3}}{2}}$$

$$q = 12ae - 3bd + c^2$$

$$s = 27ad^2 - 72ace + 27b^2e - 9bcd + 2c^3$$

Exercise

- Let `int solution1_grade4(int a, int b, int c, int d, int e)` be a function (in a C file) that computes the first solution of a quartic equation.
- Write the assembly code to obtain a solution of the following equation:

$x^4 - 10x^3 + 35x^2 - 50x + 32 = 0$

https://polito.padlet.org/rf/Carm

Assembly subroutine call from C

- visibility of the assembly subroutine
  - how does the linker know the name of the subroutine?
- exchanging data
  - how are parameters received by the assembly subroutine?
  - where is the result stored?

Visibility of the ASM subroutine

- In the C file:
  - there is the prototype of the subroutine
  - the prototype begins with `extern`
- In the assembly file:
  - the subroutine is implemented
  - the symbol is exported with one of the two equivalent directives:

```
EXPORT symbolName {[option]}
GLOBAL symbolName {[option]}
```

Options of EXPORT and GLOBAL

- `WEAK`: the symbol is not exported if another file exports the same symbol name.
- `DATA` | `CODE`: treats the symbol either as data or code when source is assembled and linked
- `SIZE =` *value*: specifies the size. If missing:
  - for `PROC` symbols: size of the code until `ENDP`
  - for other symbols: size of instruction or data on the same source line of the symbol
  - if there is no instruction or data: size is zero.

Data from C to Assembly routine

- Data exchange is regulated by ARM Architecture Procedure Call Standard (AAPCS).
- The first 4 parameters are received in `r0-r3`.
- Further parameters are received in the stack.
  - They must not be removed from the stack.
- The assembly subroutine returns the value in `r0-r3` (e.g., a word is passed in `r0`)
- The assembly subroutine must preserve the contents of registers `r4-r8`, `r10`, `r11`, `SP`.

Example: string concatenation

- Write a program that copies the first characters of two strings into a third string.
- The three strings are defined in the C file.
- The copying routine is written in assembly
  - it copies one byte at a time
  - controls are added for robustness, e.g., the destination string is full, there are no more characters to copy in the source strings.

Parameters and return value

- The copying subroutine receives in input:
  - pointer to string1
  - number of characters to copy from string1
  - pointer to string2
  - number of characters to copy from string2
  - pointer to string3
  - maximum length of string3
- The copying subroutine returns the number of characters copied.

Example: C code

```
#define MAX_LENGTH 20
extern int concatenateString(const char *,
     int, const char *, int, char *, int);
int main(void){
  const char *string1 = "problem solving";
  const char *string2 = "grammar book";
  char string3[MAX_LENGTH];
  int len1 = 3, len2 = 4, len3;
  len3 = concatenateString(string1, len1,
     string2, len2, string3, MAX_LENGTH);
  while(1);
}
```

Example: assembly code (I)

```
concatenateString PROC
  EXPORT concatenateString
  MOV r12, sp
  ; save volatile registers
  STMFD sp!, {r4-r8,r10-r11,lr}
  ;extract argument 4 and 5 from stack
  LDR   r4, [r12]
  LDR   r5, [r12, #4]
  SUB r5, r5, #1     ; last character must be
                     ; the zero terminator
  MOV r6, #0    ; num bytes copied to string3
```

Example: assembly code (II)

```
string1copy
  LDRB r7, [r0], #1 ;load byte from string1
  CMP r7, #0  ;check for zero terminator
  BEQ string1End
  STRB r7, [r4], #1 ;store byte in string3
  ADD r6, r6, #1
  CMP r6, r5   ;is string 3 full?
  BEQ string2End
  CMP r6, r1   ;other bytes to copy?
  BLO string1copy
string1End     MOV r8, #0;
```

Example: assembly code (III)

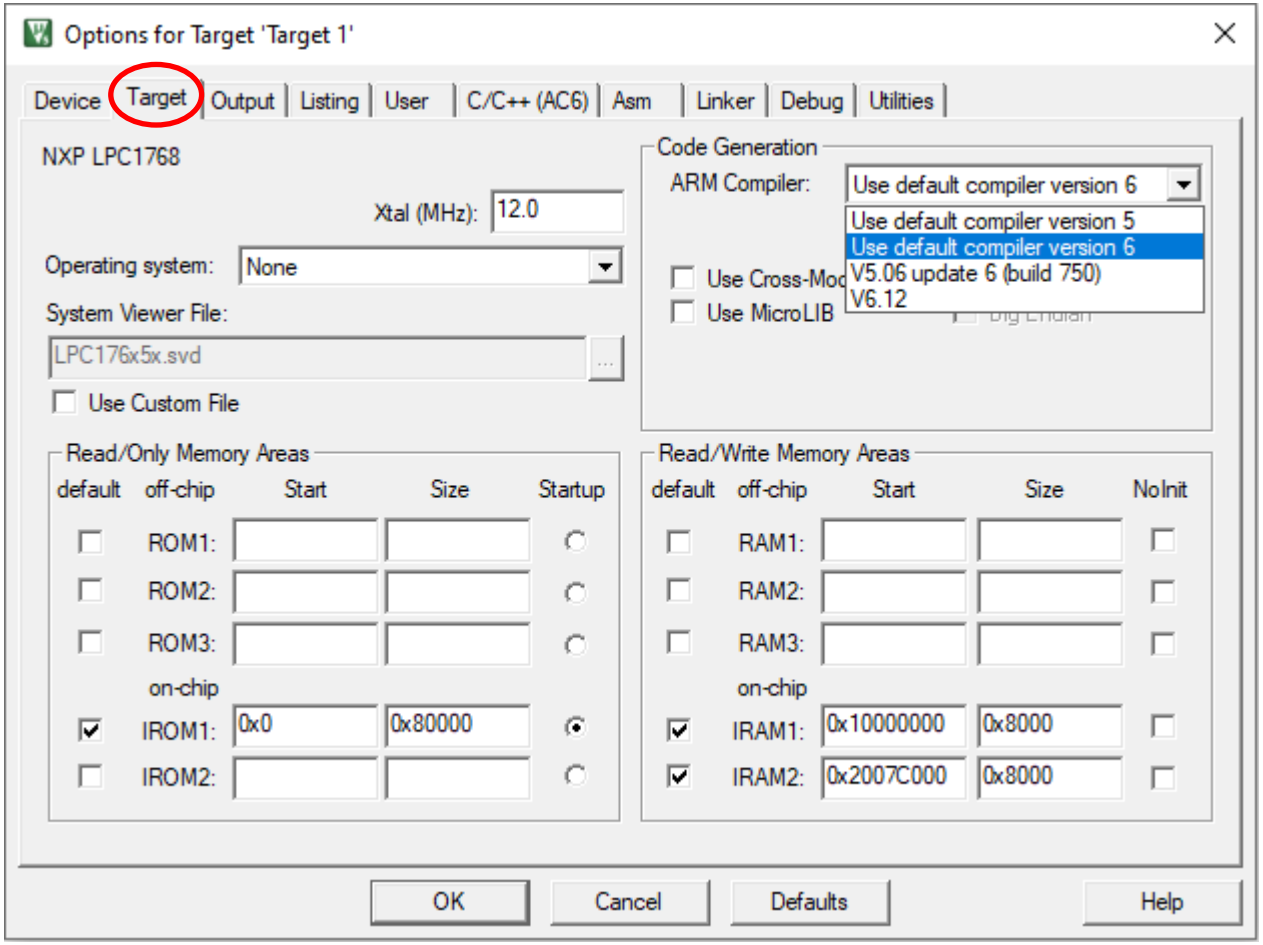
```
string2copy
  LDRB r7, [r2], #1 ;load byte from string2
  CMP r7, #0  ;check for zero terminator
  BEQ string2End
  STRB r7, [r4], #1 ;store byte in string3
  ADD r6, r6, #1
  ADD r8, r8, #1
  CMP r6, r5   ;is string 3 full?
  BEQ string2End
  CMP r8, r3   ;other bytes to copy?
  BLO string2copy
```

Example: assembly code (IV)

```
string2End
  MOV r7, #0   ;insert the zero terminator
  STRB r7, [r4], #1 ;store byte in string3
  MOV r0, r6   ;set the return value
  ;restore volatile registers
  LDMFD sp!,{r4-r8,r10-r11,pc}
  ENDP
```

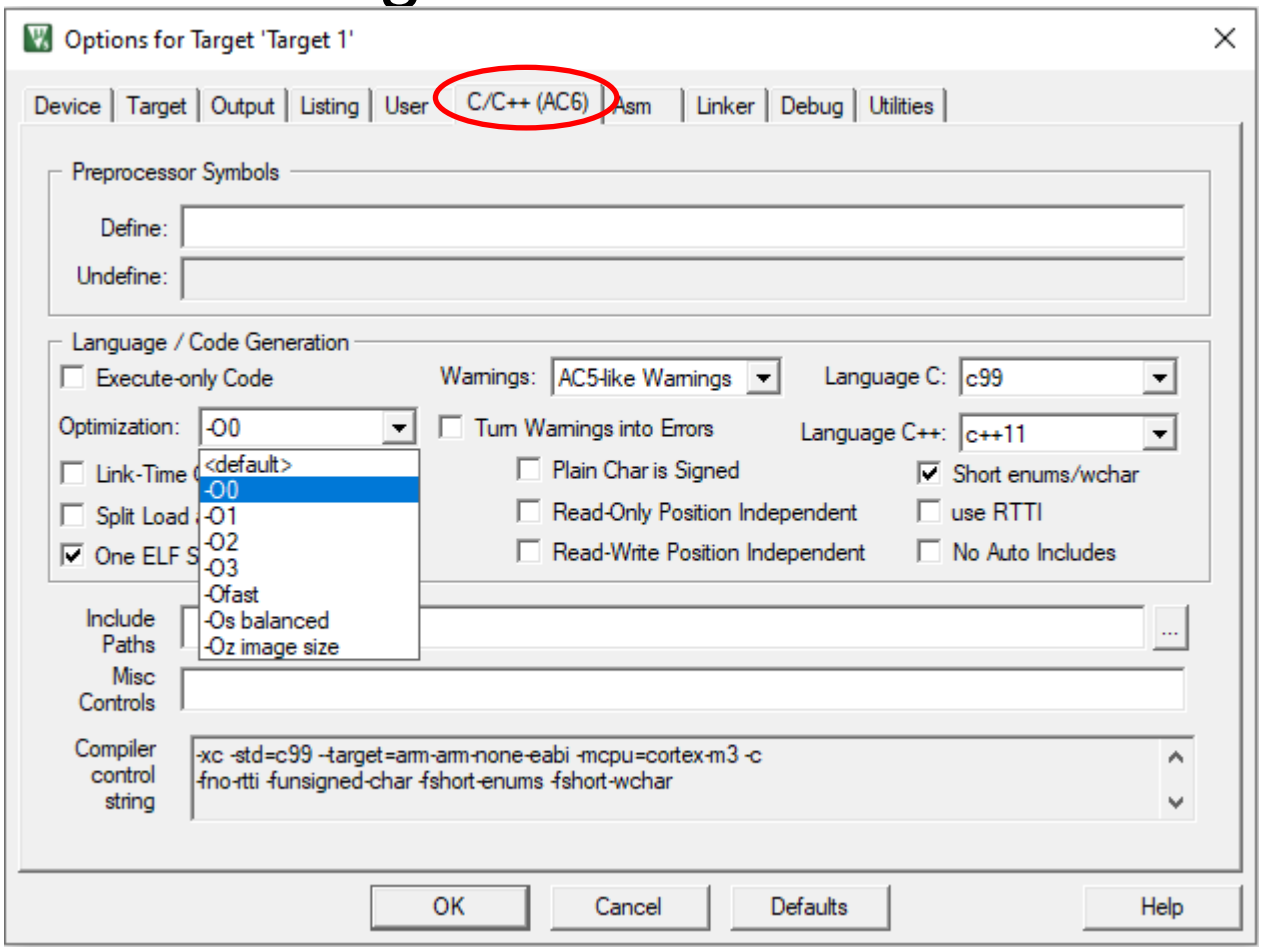
Compiler version

- In the next slides it is assumed that the last version of the compiler (version 6) is used



Compiler optimization

- The compiler can be asked to optimize the machine code generated from the C code.



Compiler optimization levels

- level 0: minimum optimization
  - good for debugging: the structure of generated code directly corresponds to the source code.
- level 1: restricted optimization
  - the generated code can be significantly smaller than level 0: this simplifies analysis of the code.
- level 2: high optimization
  - the compiler automatically inlines functions.
- level 3: maximum optimization
  - loop unrolling, more aggressive inlining.
- Rebuild (not Build) is needed to change level.

Disassembled code with level 0

```
0x00000200 2000      MOVS     r0,#0x00
0x00000202 900D      STR      r0,[sp,#0x34]
     8:   const char *string1 = "problem solving";
0x00000204 F24020A0  MOVW     r0,#0x2A0
0x00000208 F2C00000  MOVT     r0,#0x00
0x0000020C 900C      STR      r0,[sp,#0x30]
     9:   const char *string2 = "grammar book";
    10:   char string3[MAX_LENGTH];
0x0000020E F24020B0  MOVW     r0,#0x2B0
0x00000212 F2C00000  MOVT     r0,#0x00
0x00000216 900B      STR      r0,[sp,#0x2C]
0x00000218 2003      MOVS     r0,#0x03
    11:   int len1 = 3, len2 = 4;
    12:         int len3;
    13:
0x0000021A 9005      STR      r0,[sp,#0x14]
0x0000021C 2004      MOVS     r0,#0x04
0x0000021E 9004      STR      r0,[sp,#0x10]
    14:         len3 = concatenateString(string1, len1, string2, len2, string3, MAX_LENGTH);
    15:
0x00000220 980C      LDR      r0,[sp,#0x30]
0x00000222 9905      LDR      r1,[sp,#0x14]
0x00000224 9A0B      LDR      r2,[sp,#0x2C]
0x00000226 9B04      LDR      r3,[sp,#0x10]
0x00000228 46EC      MOV      r12,sp
0x0000022A F04F0E14  MOV      lr,#0x14
0x0000022E F8CCE004  STR      lr,[r12,#0x04]
0x00000232 F10D0E18  ADD      lr,SP,#0x18
0x00000236 F8CCE000  STR      lr,[r12,#0x00]
0x0000023A F000F803  BL.W     concatenateString (0x00000244)
```

- values and pointers are pushed in the stack
- parameters are load to `r0`-`r3` or pushed into the stack when call the function

Disassembled code with level 1

```
0x00000300 B088       SUB      sp,sp,#0x20
0x00000302 2014       MOVS     r0,#0x14
0x00000304 A903       ADD      r1,sp,#0x0C
    14:          len3 = concatenateString(string1, len1, string2, len2, string3, MAX_LENGTH);
    15:
0x00000306 E9CD1000   STRD     r1,r0,[sp,#0]
0x0000030A A004       ADR      r0,{pc}+0x14  ; @0x0000031C
0x0000030C A207       ADR      r2,{pc}+0x20  ; @0x0000032C
0x0000030E 2103       MOVS     r1,#0x03
0x00000310 2304       MOVS     r3,#0x04
0x00000312 F000F813   BL.W     concatenateString (0x0000033C)
0x00000316 BF00       NOP
    16:    while(1);
0x00000318 E7FE       B        0x00000318
```

- code is shorter: values and pointers are loaded in the proper registers for the subroutine call.
- a breakpoint is not hit if the corresponding instruction is removed with the optimization.

Disassembled code with level 2

```
0x000001FC B088       SUB      sp,sp,#0x20
0x000001FE 2014       MOVS     r0,#0x14
0x00000200 A903       ADD      r1,sp,#0x0C
    14:         len3 = concatenateString(string1, len1, string2, len2, string3, MAX_LENGTH);
    15:
0x00000202 A209       ADR      r2,{pc}+0x28  ; @0x00000228
0x00000204 2304       MOVS     r3,#0x04
0x00000206 E9CD1000   STRD     r1,r0,[sp,#0]
0x0000020A A003       ADR      r0,{pc}+0x10  ; @0x00000218
0x0000020C 2103       MOVS     r1,#0x03
0x0000020E F000F813   BL.W     concatenateString (0x00000238)
0x00000212 BF00       NOP
    16:   while(1);
0x00000214 E7FE       B        0x00000214
```

- differences are not significant: only the order of instructions changes

Return value is missing!

- At level 0, the value returned by the subroutine `concatenateString` is saved in the stack: e.g. `STR r0, [sp, #0x14]`
- At level ≥ 1, such instruction is missing.
- The returning value should be saved in `len3`, but the variable is never used later.
- Therefore, the compiler does not save the value and consider the subroutine as void.

Obtaining the return value

- There are two ways for forcing the acquisition of the return value:
  - add some instructions that use `len3`
  - declare `len3` as `volatile`.
- The keyword `volatile` may appear before or after the data type in the variable definition
  - `volatile int len3;`
  - `int volatile len3;`

Use of the variable: an example

```
int main(void){
  const char *string1 = "problem solving";
  const char *string2 = "grammar book";
  char string3[MAX_LENGTH];
  int len1 = 3, len2 = 4, len3;
  len3 = concatenateString(string1, len1,
string2, len2, string3, MAX_LENGTH);
  for (; len3 > 0; len3 --)
     string3[len3 - 1] += 'A' - 'a';
  while(1);
}
```

Disassembled code with level 0

```
    26:         len3 = concatenateString(string1, len1, string2, len2, string3, MAX_LENGTH);
    27:
0x00000220 980C      LDR       r0,[sp,#0x30]
0x00000222 9905      LDR       r1,[sp,#0x14]
0x00000224 9A0B      LDR       r2,[sp,#0x2C]
0x00000226 9B04      LDR       r3,[sp,#0x10]
0x00000228 46EC      MOV       r12,sp
0x0000022A F04F0E14  MOV       lr,#0x14
0x0000022E F8CCE004  STR       lr,[r12,#0x04]
0x00000232 F10D0E18  ADD       lr,SP,#0x18
0x00000236 F8CCE000  STR       lr,[r12,#0x00]
0x0000023A F000F815  BL.W      concatenateString (0x00000268)
0x0000023E 9003      STR       r0,[sp,#0x0C]
    28:         for (; len3 > 0; len3 --)
0x00000240 E7FF      B         0x00000242
0x00000242 9803      LDR       r0,[sp,#0x0C]
0x00000244 2801      CMP       r0,#0x01
0x00000246 DB0D      BLT       0x00000264
0x00000248 E7FF      B         0x0000024A
    29:             string3[len3 - 1] += 'A' - 'a';
    30:
    31:
0x0000024A 9803      LDR       r0,[sp,#0x0C]
0x0000024C A906      ADD       r1,sp,#0x18
0x0000024E 4408      ADD       r0,r0,r1
0x00000250 F8101C01  LDRB      r1,[r0,#-0x01]
0x00000254 3920      SUBS      r1,r1,#0x20
0x00000256 F8001C01  STRB      r1,[r0,#-0x01]
0x0000025A E7FF      B         0x0000025C
0x0000025C 9803      LDR       r0,[sp,#0x0C]
0x0000025E 3801      SUBS      r0,r0,#0x01
0x00000260 9003      STR       r0,[sp,#0x0C]
0x00000262 E7EE      B         0x00000242
```

`r0` contains the return value. It is saved in the stack to be used later.

Disassembled code with level 1

```
0x00000300 B088       SUB      sp,sp,#0x20
0x00000302 2014       MOVS     r0,#0x14
0x00000304 AC03       ADD      r4,sp,#0x0C
    26:         len3 = concatenateString(string1, len1, string2, len2, string3, MAX_LENGTH);
    27:
0x00000306 E9CD4000   STRD     r4,r0,[sp,#0]
0x0000030A A008       ADR      r0,{pc}+0x24  ; @0x0000032C
0x0000030C A20B       ADR      r2,{pc}+0x30  ; @0x0000033C
0x0000030E 2103       MOVS     r1,#0x03
0x00000310 2304       MOVS     r3,#0x04
0x00000312 F000F81B   BL.W     concatenateString (0x0000034C)
    28:         for (; len3 > 0; len3 --)
0x00000316 2801       CMP      r0,#0x01
0x00000318 DB06       BLT      0x00000328
    29:                 string3[len3 - 1] += 'A' - 'a';
    30:
    31:
0x0000031A 1E61       SUBS     r1,r4,#1
0x0000031C 5C0A       LDRB     r2,[r1,r0]
0x0000031E 3A20       SUBS     r2,r2,#0x20
0x00000320 540A       STRB     r2,[r1,r0]
0x00000322 3801       SUBS     r0,r0,#0x01
0x00000324 2800       CMP      r0,#0x00
0x00000326 DCF9       BGT      0x0000031C
    32:   while(1);
0x00000328 E7FE       B        0x00000328
```

optimization: `r0` is directly used, without push into and pop from the stack.

Use of volatile

- A volatile variable may change at any time, without any action from the code where the variable is currently used.
- Value may change due to:
  - peripheral registers
  - interrupt service routine
  - another task in a multi-threaded application.

Use of volatile: an example

```
int main(void){
  const char *string1 = "problem solving";
  const char *string2 = "grammar book";
  char string3[MAX_LENGTH];
  int len1 = 3, len2 = 4;
  volatile int len3;
  len3 = concatenateString(string1, len1,
string2, len2, string3, MAX_LENGTH);
  while(1);
}
```

Disassembled code with level 0

```
    26:         len3 = concatenateString(string1, len1, string2, len2, string3, MAX_LENGTH);
    27:
    28:
    29:
    30:
0x00000220 980C      LDR      r0,[sp,#0x30]
0x00000222 9905      LDR      r1,[sp,#0x14]
0x00000224 9A0B      LDR      r2,[sp,#0x2C]
0x00000226 9B04      LDR      r3,[sp,#0x10]
0x00000228 46EC      MOV      r12,sp
0x0000022A F04F0E14  MOV      lr,#0x14
0x0000022E F8CCE004  STR      lr,[r12,#0x04]
0x00000232 F10D0E18  ADD      lr,SP,#0x18
0x00000236 F8CCE000  STR      lr,[r12,#0x00]
0x0000023A F000F803  BL.W     concatenateString (0x00000244)
0x0000023E 9003      STR      r0,[sp,#0x0C]
    31:   while(1);
0x00000240 E7FF      B        0x00000242
0x00000242 E7FE      B        0x00000242
```

`r0` is saved in the stack.

Disassembled code with level 1

```
    26:         len3 = concatenateString(string1, len1, string2, len2, string3, MAX_LENGTH);
    27:
    28:
    29:
    30:
0x00000324 980C      LDR      r0,[sp,#0x30]
0x00000326 9905      LDR      r1,[sp,#0x14]
0x00000328 9A0B      LDR      r2,[sp,#0x2C]
0x0000032A 9B04      LDR      r3,[sp,#0x10]
0x0000032C 46EC      MOV      r12,sp
0x0000032E F04F0E14  MOV      lr,#0x14
0x00000332 F8CCE004  STR      lr,[r12,#0x04]
0x00000336 F10D0E18  ADD      lr,SP,#0x18
0x0000033A F8CCE000  STR      lr,[r12,#0x00]
0x0000033E F000F803  BL.W     concatenateString (0x00000348)
0x00000342 9003      STR      r0,[sp,#0x0C]
    31:   while(1);
0x00000344 E7FF      B        0x00000346
0x00000346 E7FE      B        0x00000346
```

`r0` is saved in the stack.

C file with Assembly code inside

- With inline assembly, ARM instructions can be inserted into a C function.
- Goal: operations which are not available in C can be accomplished in assembly.
- Note: the syntax of inline assembly depends on the compiler version. Only version 6 is shown in next slides.

Inline assembly

```
__asm [volatile] ("instructionList"
    : outputOperands
    [: inputOperands]);
```

- `volatile` prevents the compiler discarding statements when optimizing the code
- `instructionList`: assembler instructions separated with "\n\t"
  - the newline breaks the line
  - the tab character moves to the instruction field

Output operands

`: [asmName] "constraint" (Cname)`

- `[asmName]`: symbolic name in instruction list
  - this field is optional; if missing, the C variable is referred with its position within the operands list
- `constraint`: updating and storing info
  - `=` : the current value of the variable is overwritten
  - `+` : the value is read and then modified
  - `r` : the operand can be saved in a general register
  - `m` : the operand can be saved in memory
- `(Cname)`: C variable holding the output

Input operands

`: [asmName] constraint (Cname)`

- `[asmName]`: symbolic name in instruction list
  - this field is optional; if missing, the C variable is referred with its position within the operands list
- `constraint`: storing info
  - `r` : the operand can be saved in a general register
  - `m` : the operand can be saved in memory
- `(Cname)`: C variable or expression passed to the instruction list

Example with named operands

```
int inlineAssembly(int value)
{
    int var1, var2, res;
    __asm__ volatile(
      "AND %[asmVar2], %[asmValue], #0x000000FF\n\t"
      "LSR %[asmVar1], %[asmValue], #24\n\t"
      "ADD %[asmRes], %[asmVar1], %[asmVar2]"
    : [asmVar1] "+r" (var1),
      [asmVar2] "+r" (var2),
      [asmRes] "=r" (res)
    : [asmValue] "r" (value)
    );
    return res;
}
```

Example with positional operands

```
int inlineAssembly(int value)
{
    int var1, var2, res;
    __asm__ volatile(
      "AND %1, %3, #0x000000FF\n\t"
      "LSR %0, %3, #24\n\t"
      "ADD %2, %0, %1"
    : "=r" (var1),
      "=r" (var2),
      "=r" (res)
    : "r" (value)
    );
    return res;
}
```

Example with same in/out operand

- In C, usually the opposite of a number is obtained by multiplying the number to -1

```
int var = 10;
var = var * (-1);
```

- The multiplication can be avoided by using the `MVN` instruction, and then by adding 1:

```
__asm__ ("MVN %[myVar], %[myVar]\n\t"
     "ADD %[myVar], %[myVar], 1"
     : [myVar] "+r" (var));
```