ASM+C & C+ASM
ABI std. compliancy
Code optimization

Paolo Bernardi

Cross compiler

- A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running
- For example, a compiler that runs on a Windows 10 PC but generates a code that runs on ARM SoC is a cross compiler
- Cross compiling is a typical step for embedded application written in C language
- When integrating ASM functions, a special care is needed
  - To correctly identify the functions arguments
  - To use the proper resources to return a result
- The knowledge of the ABI standard is fundamental in this context.

Branch to main.c from startup.s

- Startup.s

; Have to import the main function address

; Executes a branch to main

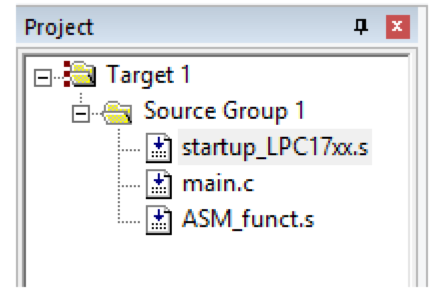
```
IMPORT  __main
LDR     R0, =__main
BX      R0
```

- Main.c

```
int main(){
    while(1) ;
}
```

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

ABI_C+ASM project (I)
(in the example folder - startup_LPC17xx.s)



- In our setup we find the following source codes:
  - startup_LPC17xx.s
  - main.c
  - ASM_funct.s
- The startup_LPC17xx.s file includes the branch to main in the reset handler

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

ABI_C+ASM project (II)
(in the example folder - main.c)

- The main.c file is a C source code
  - It invokes a function called ASM_funct with 6 parameters
  - After executing the called function, it enters in an endless loop.

```c
extern int ASM_funct(int, int, int, int, int, int);

int main(void){

  int i=0xFFFFFFFF, j=2, k=3, l=4, m=5, n=6;
  volatile int r=0;

  r = ASM_funct(i, j, k, l, m, n);

  while(1);
}
```

ABI_C+ASM project (III)
(in the example folder - ASM_funct.s)

- Inline ASM

  __ASM("SVC #0x10");

- External ASM function invoked by a C function

  r = ASM_funct(i, j, k, l, m, n);

Where parameters are stored?

How to return results?

```
                AREA asm_functions, CODE, READONLY
                EXPORT  ASM_funct
ASM_funct
                ; save current SP for a faster access
                ; to parameters in the stack
                MOV   r12, sp
                ; save volatile registers
                STMFD sp!,{r4-r8,r10-r11,lr}
                ; extract argument 4 and 5 into R4 and R5
                LDR   r4, [r12]
                LDR   r5, [r12,#4]
                ; setup a value for R0 to return
                MOV   r0, r5
                ; restore volatile registers
                LDMFD sp!,{r4-r8,r10-r11,pc}

                END
```

Parameters are in R0-R3 (a1-a4)

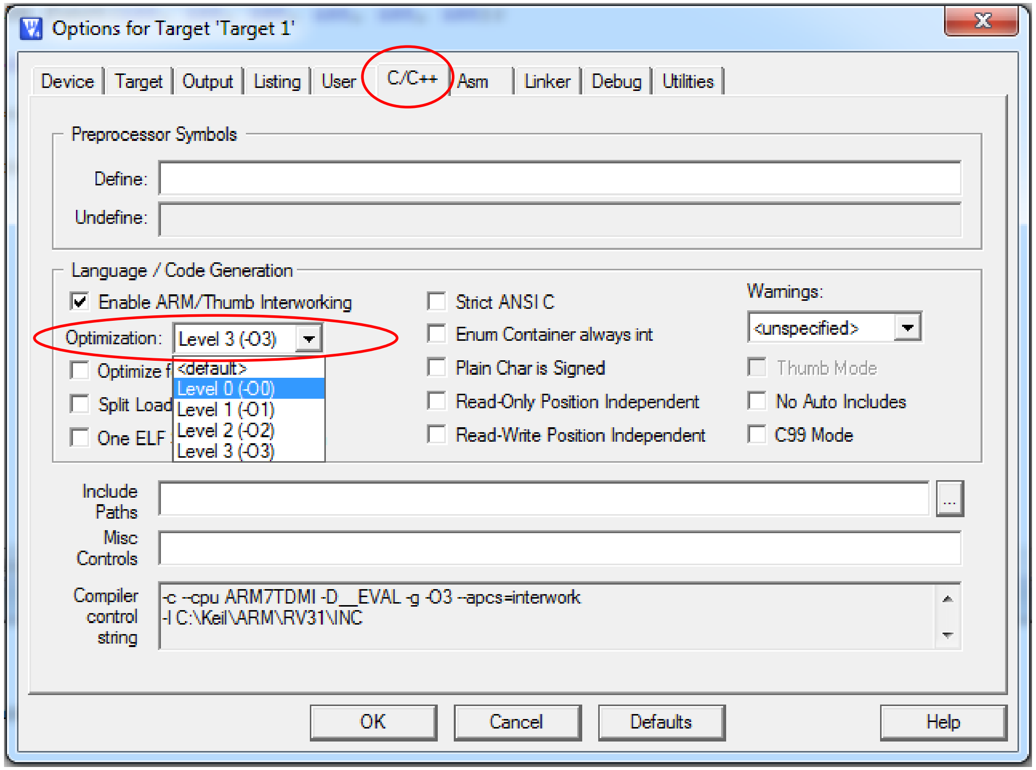
Stacked parameters

C and ASM helpful directives

- EXPORT : makes visible a function outside the file defining it
- IMPORT : makes visible a function from other files
- extern : Permits to import a variable from other file (where it is defined)

COMPILER OPTIMIZATION

- As we are now working in C language, the compiler can be asked to optimize the resulting executable file (i.e., the actual machine code).



Compiler optimization and the volatile attribute

- Higher optimization levels can reveal problems in some programs that are not apparent at lower optimization levels
- This happens when, for example, missing the volatile qualifiers
- The declaration of a variable as volatile tells the compiler that the variable can be modified at any time externally to the implementation
  - by the operating system,
  - by another thread of execution such as an interrupt routine or signal handler,
  - by hardware.

Example on volatile variables

Nonvolatile version of buffer loop

```
int buffer_full;
int read_stream(void)
{
   int count = 0;
   while (!buffer_full)
   {
      count++;
   }
   return count;
}
```

Volatile version of buffer loop

```
volatile int buffer_full;
int read_stream(void)
{
   int count = 0;
   while (!buffer_full)
   {
      count++;
   }
   return count;
}
```

Example on volatile variables

Nonvolatile version of buffer loop

```
int buffer_full;
i
{
}
```

```
read_stream PROC
   LDR      r1, |L1.28|
   MOV      r0, #0
   LDR      r1, [r1, #0]
|L1.12|
   CMP      r1, #0
   ADDEQ    r0, r0, #1
   BEQ      |L1.12|      ; infinite loop
   BX       lr
   ENDP
|L1.28|
   DCD      ||.data||
   AREA ||.data||, DATA, ALIGN=2
buffer_full
   DCD      0x00000000
```

Buffer_full

Volatile version of buffer loop

```
volatile int buffer_full;
in
{
}
```

```
read_stream PROC
   LDR      r1, |L1.28|
   MOV      r0, #0
|L1.8|
   LDR      r2, [r1, #0];  ; buffer_full
   CMP      r2, #0
   ADDEQ    r0, r0, #1
   BEQ      |L1.8|
   BX       lr
   ENDP
|L1.28|
   DCD      ||.data||
   AREA ||.data||, DATA, ALIGN=2
   buffer_full
      DCD      0x00000000
```

ABI standard for ARM

Application Binary Interface for the ARM® Architecture

The Base Standard

Document number: ARM IHI 0036B, current through ABI release 2.09

Date of Issue: 10th October 2008, reissued 30th November 2012

Abstract

This document describes the structure of the Application Binary Interface (ABI) for the ARM architecture, and links to the documents that define the base standard for the ABI for the ARM Architecture. The base standard governs inter-operation between independently generated binary files and sets standards common to ARM-based execution environments.

Keywords

ABI for the ARM architecture, ABI base standard, embedded ABI

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register [illegible]ned by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

Table 2, Core registers and AAPCS usage

Can be freely used to hold local variables

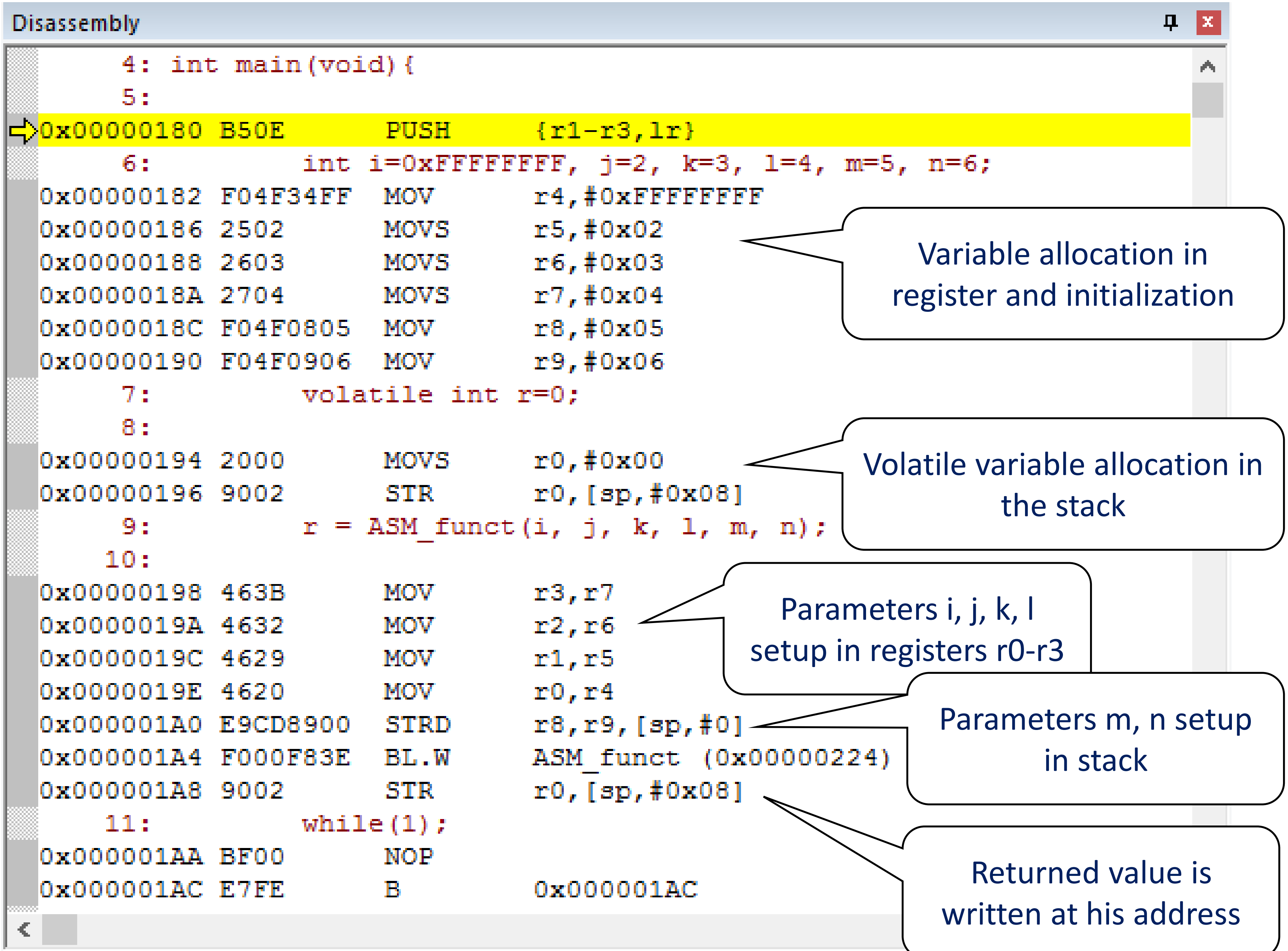
If there are more than 4 formal arguments, they have to be saved in the stack

Passing arguments

- The first four registers r0-r3 (a1-a4) are used to pass argument values into a subroutine and to return a result value in r0-r1 from a function.
  - A subroutine must preserve the contents of the registers r4-r8, r10, r11 and SP
- The base standard provides for passing arguments in core registers (r0-r3) and on the stack.
  - For subroutines that take a small number of parameters, only registers are used, greatly reducing the overhead of a call.

STACK management

- The stack implementation is *full-descending*, with the current extent of the stack held in the register SP (r13).
- The stack will, in general, have both a *base* and a *limit* though in practice an application may not be able to determine the value of either.



Disassembly
4: int main(void){
5:
0x00000180 B50E PUSH {r1-r3,lr}
6: int i=0xFFFFFFFF, j=2, k=3, l=4, m=5, n=6;
0x00000182 F04F34FF MOV r4,#0xFFFFFFFF
0x00000186 2502 MOVS r5,#0x02
0x00000188 2603 MOVS r6,#0x03
0x0000018A 2704 MOVS r7,#0x04
0x0000018C F04F0805 MOV r8,#0x05
0x00000190 F04F0906 MOV r9,#0x06
7: volatile int r=0;
8:
0x00000194 2000 MOVS r0,#0x00
0x00000196 9002 STR r0,[sp,#0x08]
9: r = ASM_funct(i, j, k, l, m, n);
10:
0x00000198 463B MOV r3,r7
0x0000019A 4632 MOV r2,r6
0x0000019C 4629 MOV r1,r5
0x0000019E 4620 MOV r0,r4
0x000001A0 E9CD8900 STRD r8,r9,[sp,#0]
0x000001A4 F000F83E BL.W ASM_funct (0x00000224)
0x000001A8 9002 STR r0,[sp,#0x08]
11: while(1);
0x000001AA BF00 NOP
0x000001AC E7FE B 0x000001AC
Variable allocation in register and initialization
Volatile variable allocation in the stack
Parameters i, j, k, l setup in registers r0-r3
Parameters m, n setup in stack
Returned value is written at his address

What happens if don’t declare r as volatile?

- A warning is signaled about missing usesfuleness of variable r
- An optimization of the machine code is implemented by the compiler.

Build Output

```
Rebuild target 'Target 1'
assembling startup_LPC17xx.s...
compiling main.c...
main.c(7): warning:  #550-D: variable "r" was set but never used
        int r;
main.c: 1 warning, 0 errors
assembling ASM_funct.s...
linking...
Program Size: Code=360 RO-data=408 RW-data=0 ZI-data=608
".\ABI_C+ASM.axf" - 0 Error(s), 1 Warning(s).
```

What happens if don’t declare r as volatile?

Disassembly

```
      4: int main(void){
      5:
0x00000180 B51C        PUSH      {r2-r4,lr}
      6:         int i=0xFFFFFFFF, j=2, k=3, l=4, m=5, n=6;
      7:         int r;
      8:
0x00000182 F04F34FF  MOV       r4,#0xFFFFFFFF
0x00000186 2502        MOVS      r5,#0x02
0x00000188 2603        MOVS      r6,#0x03
0x0000018A 2704        MOVS      r7,#0x04
0x0000018C F04F0805  MOV       r8,#0x05
0x00000190 F04F0906  MOV       r9,#0x06
      9:         r = ASM_funct(i, j, k, l, m, n);
     10:
0x00000194 463B        MOV       r3,r7
0x00000196 4632        MOV       r2,r6
0x00000198 4629        MOV       r1,r5
0x0000019A 4620        MOV       r0,r4
0x0000019C E9CD8900  STRD      r8,r9,[sp,#0]
0x000001A0 F000F83C  BL.W      ASM_funct (0x0000021C)
     11:         while(1);
0x000001A4 BF00        NOP
0x000001A6 E7FE        B         0x000001A6
```

Variable r is not allocated and the ASM function is considered as “void” function