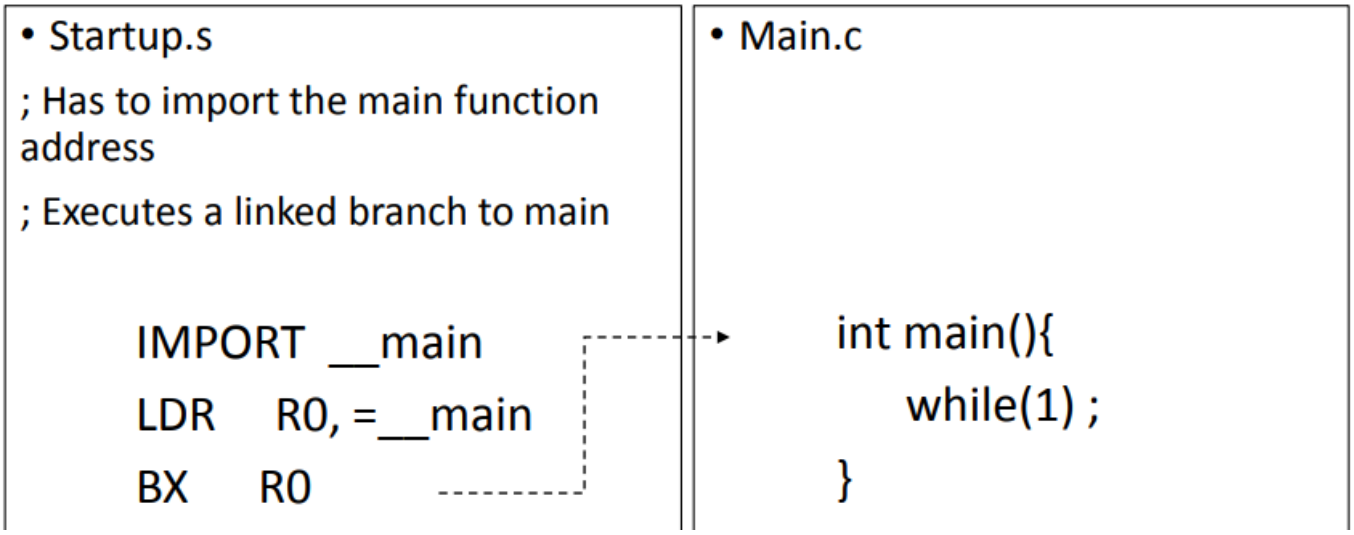
Cross compiler

E' un compilatore che riesce a creare un codice eseguibile per una piattaforma oltre a quella dove il compilatore sta girando.
Ad esempio su **Windows 10** riesco a creare un eseguibile per **Mac OS X**.
La *cross-compilazione* è tipica per applicazioni **embedded** scritte in **C**.

Da ASM a C



L'etichetta *main* del programma **c** viene importata nell'ASM.
La direttiva **IMPORT** rende visibile una funzione presente in altri file.

Da C ad ASM

```
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);
}
```

Grazie alla keyword **extern**, importiamo la funzione ASM presente in un altro file.
La variable **volatile** fa sì che la variabile R non venga *ottimizzata troppo* dal compilatore.

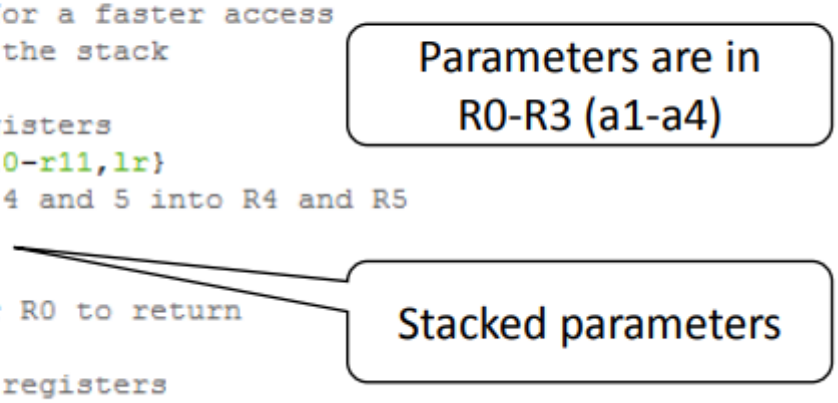
Inline ASM

E' possibile eseguire ASM direttamente *inline*.
Ad esempio **__ASM("SVC #0x10")**

External ASM

```
AREA asm_functions, CODE, READONLY
EXPORT 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}
```



La direttiva **EXPORT** fa sì che la funzione sia visibile anche ad altri file.
R12 nell'ABI per le subroutine viene usato per facilitare l'accesso allo Stack Pointer.

Attributo volatile

Alti livelli di ottimizzazione del compilatore possono generare problemi.
Per far sì che alcune variabili non *vengano* ottimizzate troppo, si utilizza l'attributo **volatile**.

Table 12. C code for nonvolatile and volatile buffer loops

Nonvolatile version of buffer loop	Volatile version of buffer loop

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

Table 13. Disassembly for nonvolatile and volatile buffer loop

Nonvolatile version of buffer loop	Volatile version of buffer loop

`read_stream PROC`
`  LDR  r1, |L1.28|` indirizzo di buffer_full
`  MOV  r0, #0`
`  LDR  r1, [r1, #0]` valore di buffer_full
`|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`
	`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`

Nella prima versione, cicla finchè **buffer_full** è diverso da 0. Tuttavia il compilatore *detecta* il ciclo infinito e quindi dentro il ciclo, per ottimizzare, non esegue la lettura di **buffer_full** ogni volta.

Nella seconda versione invece, visto che **buffer_full** è **volatile**, viene letta ad ogni iterazione del ciclo.

In questo modo, tramite interrupt esterni, è possibile modificare il valore di **buffer_full** ed eventualmente uscire dal ciclo