

# Stack Frames (11A)

---

---

Copyright (c) 2014 - 2020 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Please send corrections (or suggestions) to [youngwlim@hotmail.com](mailto:youngwlim@hotmail.com).

This document was produced by using LibreOffice.

# Based on

---

ARM System-on-Chip Architecture, 2<sup>nd</sup> ed, Steve Furber

Introduction to ARM Cortex-M Microcontrollers  
– Embedded Systems, Jonathan W. Valvano

Digital Design and Computer Architecture,  
D. M. Harris and S. L. Harris

ARM assembler in Raspberry Pi  
Roger Ferrer Ibáñez

<https://thinkingeek.com/arm-assembler-raspberry-pi/>

# Activation records (1)

## a stack frame

a frame of data that gets pushed onto the stack.

## a call stack

divided up into contiguous pieces called stack frames;  
here, a stack frame would represent a **function call** and its **argument** data.

- **return address**
- **arguments**
- **local variables.**

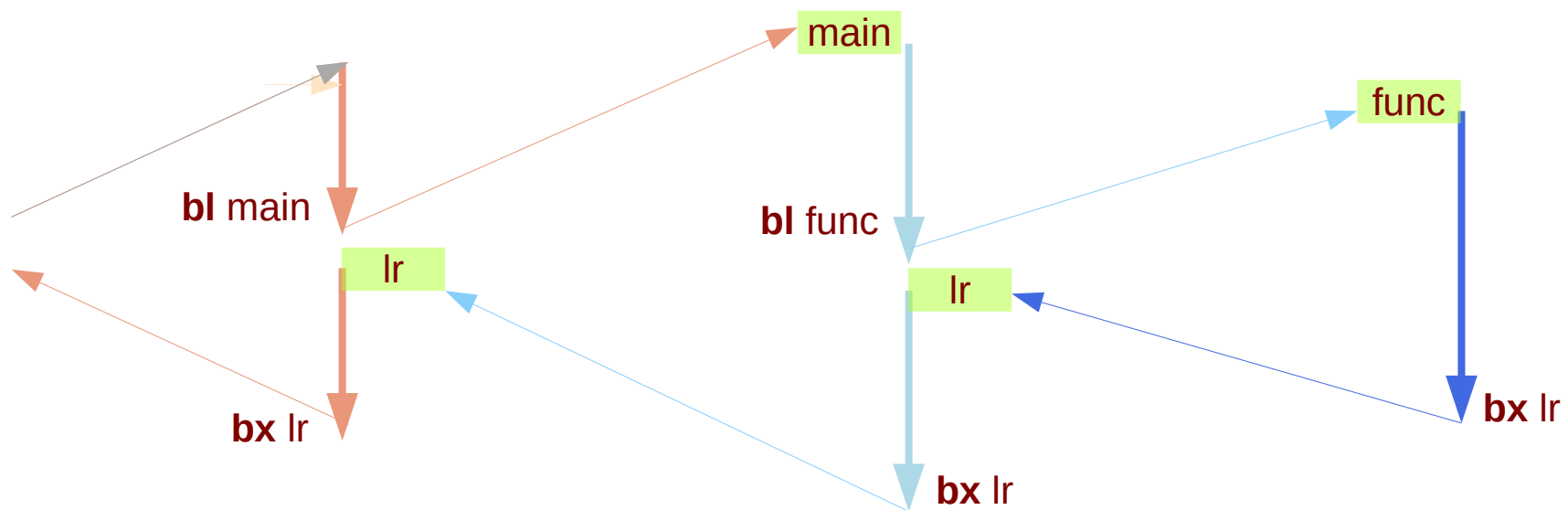
architecture-dependent.

processor knows the size of each frame  
and moves the **stack pointer** accordingly  
as **frames** are pushed and popped off the stack.

<https://stackoverflow.com/questions/10057443/explain-the-concept-of-a-stack-frame-in-a-nutshell>

# Saving and restoring of return address (1)

Nested function calls :  
**LR** must not be overwritten



<https://thinkingeek.com/2013/02/02/arm-assembler-raspberry-pi-chapter-9/>

## Activation records (2)

when your program is started,  
the **call stack** has only one frame, that of the function **main()**.  
This is called the **initial frame** or the **outermost frame**.

each time a function is called,  
a new frame is added.  
each time a function returns,  
the frame for that function invocation is eliminated.

for a recursive function,  
there can be many frames for the same function.

the frame for the currently executing function  
is called the **innermost frame**.  
the most recently created frame

[http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.neutrino.prog%2Ftopic%2Fusing\\_gdb\\_StackFrames.html](http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.neutrino.prog%2Ftopic%2Fusing_gdb_StackFrames.html)

# Activation records (3)

Inside your program,  
**stack frames** are identified by their addresses.

A stack frame consists of many bytes,  
each of which has its own address;

each kind of computer has a convention for choosing one byte  
whose address serves as the address of the frame.

Usually this address is kept in a register called the frame pointer register  
while execution is going on in that frame.

[http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.neutrino.prog%2Ftopic%2Fusing\\_gdb\\_StackFrames.html](http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.neutrino.prog%2Ftopic%2Fusing_gdb_StackFrames.html)

# Activation records (4)

GDB assigns numbers to all existing stack frames, starting with 0 for the **innermost** frame, 1 for the frame that called it, and so on upward.

These numbers don't really exist in your program; they're assigned by GDB to give you a way of designating stack frames in GDB commands.

[http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.neutrino.prog%2Ftopic%2Fusing\\_gdb\\_stackframes.html](http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.neutrino.prog%2Ftopic%2Fusing_gdb_stackframes.html)



# Activation records (5)

## **a call stack**

a stack data structure that stores information about the **active subroutines** of a computer program.

This kind of stack is also known as an **execution stack**, **program stack**, **control stack**, **run-time stack**, or **machine stack**, and is often shortened to just "the stack".

Although maintenance of the **call stack** is important for the proper functioning of most software, the details are normally hidden and automatic in high-level programming languages.

Many computer instruction sets provide special instructions for manipulating stacks.

[https://en.wikipedia.org/wiki/Call\\_stack](https://en.wikipedia.org/wiki/Call_stack)

# Activation records (6)

A call stack is used for several related purposes, but the main reason for having one is to keep track of the point to which each **active subroutine** should return control when it finishes executing.

An **active subroutine** is one that has been called, but is yet to complete execution, after which control should be handed back to the point of call.

Such **activations** of subroutines may be nested to any level (recursive as a special case), hence the stack structure.

[https://en.wikipedia.org/wiki/Call\\_stack](https://en.wikipedia.org/wiki/Call_stack)

# Activation records (7)

---

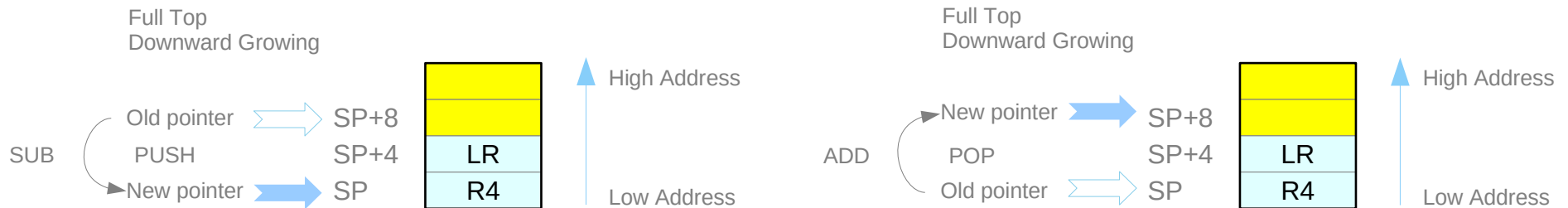
For example, if a subroutine DrawSquare calls a subroutine DrawLine from four different places, DrawLine must know where to return when its execution completes.

To accomplish this, the address following the instruction that jumps to DrawLine, the return address, is pushed onto the top of the call stack with each call.

[https://en.wikipedia.org/wiki/Call\\_stack](https://en.wikipedia.org/wiki/Call_stack)

# Activation records (1)

```
function:                                ; Keep callee-saved registers  
  
    push {r4, lr}                        ; Keep the callee saved registers  
    ...                                   ; code of the function  
    pop {r4, lr}                          ; Restore the callee saved registers  
    bx lr                                  ; Return from the function
```



<https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>

## Activation records (2)

```
function:                                ; Keep callee-saved registers */
                                        ; Keep the callee saved registers.
                                        ; We added r5 to keep the stack 8-byte aligned
push {r4, r5, fp, lr}                   ; but the important thing here is fp */
mov fp, sp                               ; fp ← sp. Keep dynamic link in fp */
...                                       ; code of the function */
mov sp, fp                               ; sp ← fp. Restore dynamic link in fp */
pop {r4, r5, fp, lr}                   ; Restore the callee saved registers.
                                        ; This will restore fp as well */
bx lr                                    ; Return from the function */
```

<https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>

# Activation records (3)

function:

```
push{r4, r5, fp, lr}
mov fp, sp
sub sp, sp, #8
...
mov sp, fp
pop {r4, r5, fp, lr}

bx lr
```

; Keep callee-saved registers

; Keep the callee saved registers.

; We added r5 to keep the stack 8-byte aligned

; but the important thing here is fp

; fp ← sp. Keep dynamic link in fp

; Enlarge the stack by 8 bytes

; code of the function

; sp ← fp. Restore dynamic link in fp

; Restore the callee saved registers.

; This will restore fp as well

; Return from the function

<https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>

# Activation records (4)

```
void sq(int *c) {  
    (*c) = (*c) * (*c);  
}
```

```
sq:  
    ldr r1, [r0]      ; r1 ← (*r0)  
    mul r1, r1, r1    ; r1 ← r1 * r1  
    str r1, [r0]      ; (*r0) ← r1  
    bx lr             ; Return from the function
```

<https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>

# Activation records (5)

```
int sq_sum5(int a, int b, int c, int d, int e) {  
    sq(&a);  
    sq(&b);  
    sq(&c);  
    sq(&d);  
    sq(&e);  
    return a + b + c + d + e;  
}
```

<https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>



# Activation records (6)

sq\_sum5:

```
push {fp, lr}      ; Keep fp and all callee-saved registers.
mov fp, sp         ; Set the dynamic link

                    ; Allocate space for 4 integers in the stack
                    ; Keep parameters in the stack

sub sp, sp, #16    ; sp ← sp - 16.
str r0, [fp, #-16] ; *(fp - 16) ← r0
str r1, [fp, #-12] ; *(fp - 12) ← r1
str r2, [fp, #-8]  ; *(fp - 8) ← r2
str r3, [fp, #-4]  ; *(fp - 4) ← r3
```

<https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>

# Activation records (7)

/\* At this point the stack looks like this

Value	Address (es)
r0	[fp, #-16], [sp]
r1	[fp, #-12], [sp, #4]
r2	[fp, #-8], [sp, #8]
r3	[fp, #-4], [sp, #12]
fp	[fp], [sp, #16]
lr	[fp, #4], [sp, #20]
e	[fp, #8], [sp, #24]

v

Higher  
addresses

\*/

<https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>

## Activation records (8)

```
sub r0, fp, #16      ; r0 ← fp - 16
bl sq                ; call sq(&a);
sub r0, fp, #12      ; r0 ← fp - 12
bl sq                ; call sq(&b);
sub r0, fp, #8       ; r0 ← fp - 8
bl sq                ; call sq(&c);
sub r0, fp, #4       ; r0 ← fp - 4
bl sq                ; call sq(&d)
add r0, fp, #8       ; r0 ← fp + 8
bl sq                ; call sq(&e)
```

<https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>

# Activation records (9)

```
ldr r0, [fp, #-16]    ; r0 ← *(fp - 16). Loads a into r0
ldr r1, [fp, #-12]    ; r1 ← *(fp - 12). Loads b into r1
add r0, r0, r1        ; r0 ← r0 + r1
ldr r1, [fp, #-8]     ; r1 ← *(fp - 8). Loads c into r1
add r0, r0, r1        ; r0 ← r0 + r1
ldr r1, [fp, #-4]     ; r1 ← *(fp - 4). Loads d into r1
add r0, r0, r1        ; r0 ← r0 + r1
ldr r1, [fp, #8]      ; r1 ← *(fp + 8). Loads e into r1
add r0, r0, r1        ; r0 ← r0 + r1

mov sp, fp           ; Undo the dynamic link
pop {fp, lr}         ; Restore fp and callee-saved registers
bx lr                ; Return from the function
```

<https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>

# Activation records (10)

```
/* squares.s */
```

```
.data
```

```
.align 4
```

```
message: .asciz "Sum of 1^2 + 2^2 + 3^2 + 4^2 + 5^2 is %d\n"
```

```
.text
```

```
sq:
```

```
<<defined above>>
```

```
sq_sum5:
```

```
<<defined above>>
```

```
.globl main
```

<https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>

# Activation records (11)

main:

```
push {r4, lr}           ; Keep callee-saved registers
```

```
; Prepare the call to sq_sum5
```

```
mov r0, #1             ; Parameter a ← 1
```

```
mov r1, #2             ; Parameter b ← 2
```

```
mov r2, #3             ; Parameter c ← 3
```

```
mov r3, #4             ; Parameter d ← 4
```

```
; Parameter e goes through the stack,
```

```
; so it requires enlarging the stack
```

```
mov r4, #5             ; r4 ← 5
```

```
sub sp, sp, #8         ; Enlarge the stack 8 bytes,
```

```
; we will use only the
```

```
; topmost 4 bytes
```

```
str r4, [sp]           ; Parameter e ← 5
```

```
bl sq_sum5             ; call sq_sum5(1, 2, 3, 4, 5)
```

```
add sp, sp, #8         ; Shrink back the stack
```

<https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>

# Activation records (12)

```
; Prepare the call to printf
mov r1, r0                ; The result of sq_sum5
ldr r0, address_of_message
bl printf                 ; Call printf

pop {r4, lr}             ; Restore callee-saved registers
bx lr
```

```
address_of_message: .word message
```

```
$ ./square
```

```
Sum of 1^2 + 2^2 + 3^2 + 4^2 + 5^2 is 55
```

<https://thinkingeek.com/2013/02/07/arm-assembler-raspberry-pi-chapter-10/>

# Callee Saved Registers

function:

```
push { r4, lr } /* Keep the callee saved registers */  
    code of the function  
pop { r4, lr } /* Restore the callee saved registers */  
bx lr  
/* Return from the function */
```



# Dynamic Link

function:

```
push { r4, r5, fp, lr }
```

```
mov fp, sp
```

code of the function

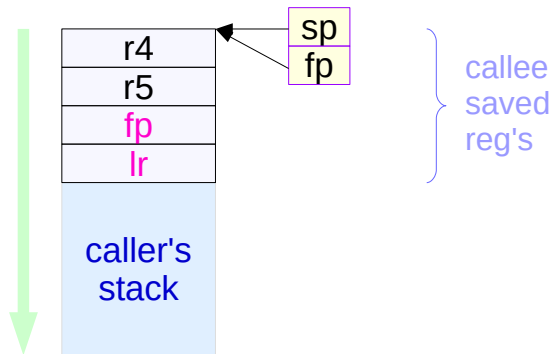
```
mov sp, fp
```

```
pop { r4, r5, fp, lr }
```

```
bx lr
```

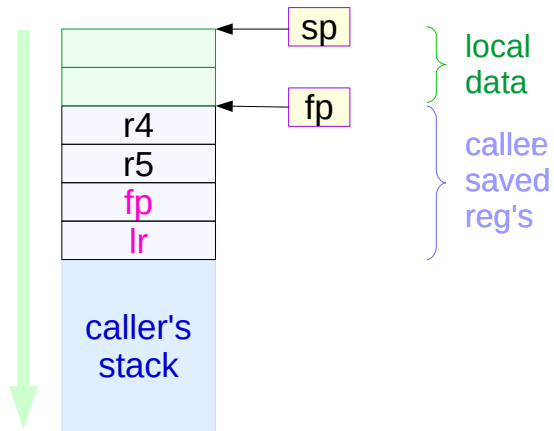
```
/* fp ← sp . Keep dynamic link in fp */
```

```
/* sp ← fp. Restore dynamic link in fp */
```

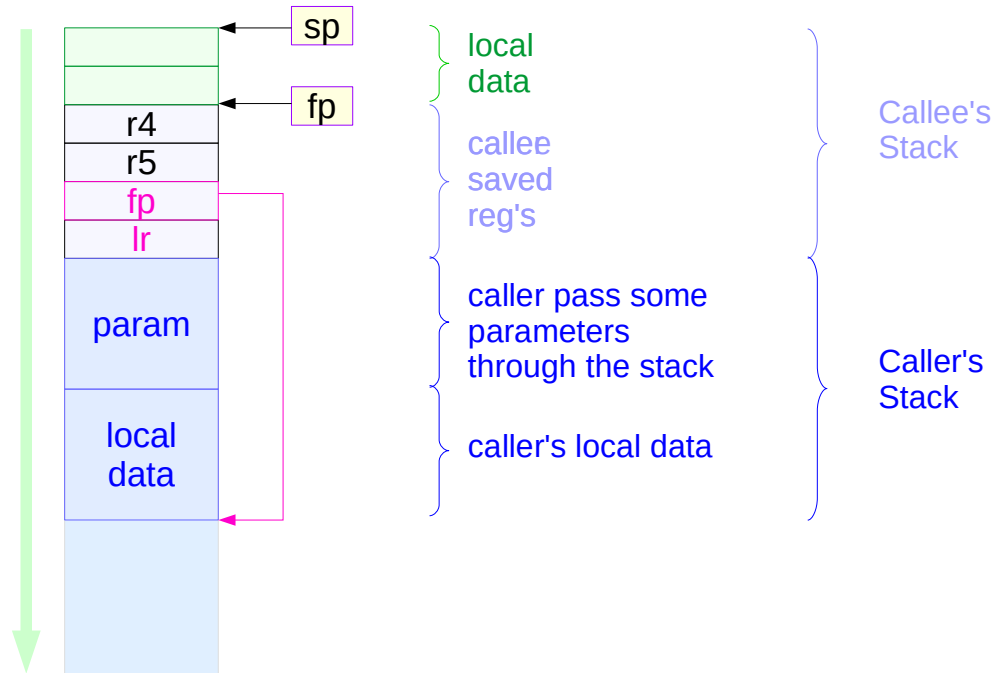


# Local Data

```
function:  
push { r4, r5, fp, lr }  
sub sp, sp, #8      /* 8 bytes local data space */  
mov fp, sp  
    code of the function  
mov sp, fp  
pop { r4, r5, fp, lr }  
bx lr
```



# Local Data and Parameters



# Local Data Generating Examples

```
void sq(int *c)
{
    (*c) = (*c) * (*c);
}
```

```
int sq_sum5(int a, int b, int c, int d, int e)
{
    sq(&a);
    sq(&b);
    sq(&c);
    sq(&d);
    sq(&e);
    return a + b + c + d + e;
}
```

```
...
sq_sum5(1, 2, 3, 4, 5);
...
```

callee  
function

- **sq** received a reference
- registers do not have an address
- allocate temporary local storage

caller  
function

# Callee Function Code (1)

```
sq_sum5:  
push { fp, lr }  
mov fp, sp  
sub sp, sp, #16
```

```
str r0, [ fp, #-16 ]    *( fp - 16 ) ← r0  
str r1, [ fp, #-12 ]    *( fp - 12 ) ← r1  
str r2, [ fp, #-8 ]     *( fp - 8 ) ← r2  
str r3, [ fp, #-4 ]     *( fp - 4 ) ← r3
```

```
mov sp, fp  
pop { fp, lr }  
bx lr
```

```
sq:  
ldr r1, [ r0 ]          r1 ← ( *r0 )  
mul r1, r1, r1          r1 ← r1 * r1  
str r1, [ r0 ]          ( *r0 ) ← r1  
bx lr
```

```
sub r0, fp, #16        r0 ← fp - 16  
bl sq                  call sq ( &a )  
sub r0, fp, #12        r0 ← fp - 12  
bl sq                  call sq ( &b )  
sub r0, fp, #8         r0 ← fp - 8  
bl sq                  call sq ( &c )  
sub r0, fp, #4         r0 ← fp - 4  
bl sq                  call sq ( &d )  
add r0, fp, #8         r0 ← fp + 8  
bl sq                  call sq ( &e )
```

```
ldr r0, [ fp, #-16 ]   r0 ← *( fp - 16 ) :a  
ldr r1, [ fp, #-12 ]   r1 ← *( fp - 12 ) :b  
add r0, r0, r1         r0 ← r0 + r1  
ldr r1, [ fp, #-8 ]    r1 ← *( fp - 8 ) :c  
add r0, r0, r1         r0 ← r0 + r1  
ldr r1, [ fp, #-4 ]    r1 ← *( fp - 4 ) :d  
add r0, r0, r1         r0 ← r0 + r1  
ldr r1, [ fp, #8 ]     r1 ← *( fp + 8 ) :e  
add r0, r0, r1         r0 ← r0 + r1
```

# Callee Function Code (2)

```
sq_sum5:  
push { fp, lr }  
mov fp, sp  
sub sp, sp, #16
```

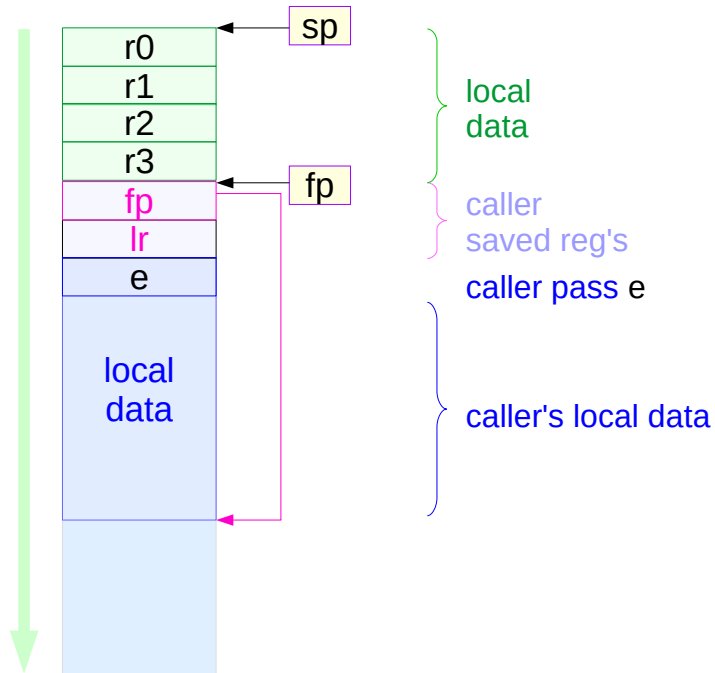
```
str r0, [ fp, #-16 ]    *( fp - 16 ) ← r0  
str r1, [ fp, #-12 ]    *( fp - 12 ) ← r1  
str r2, [ fp, #-8 ]     *( fp - 8 ) ← r2  
str r3, [ fp, #-4 ]     *( fp - 4 ) ← r3
```

```
mov sp, fp  
pop { fp, lr }  
bx lr
```

At this point the stack looks like this  
| Value | Address (es)

+-----+-----+-----+			
	r0   [ fp, #-16 ],	[sp]	
	r1   [ fp, #-12 ],	[sp, #4]	local
	r2   [ fp, #-8 ],	[sp, #8]	data
	r3   [ fp, #-4 ],	[sp, #12]	
	fp   [ fp ],	[sp, #16]	callee saved
	lr   [ fp, #4 ],	[sp, #20]	registers
	e   [ fp, #8 ],	[sp, #24]	caller pass
	v		e parameters
Higher addresses			

# Callee Function Code (3)



At this point the stack looks like this  
 | Value | Address (es)

	Value	Address (es)	
r0	[ fp, #-16 ]	[ sp ]	local data
r1	[ fp, #-12 ]	[ sp, #4 ]	
r2	[ fp, #-8 ]	[ sp, #8 ]	
r3	[ fp, #-4 ]	[ sp, #12 ]	
fp	[ fp ]	[ sp, #16 ]	callee saved registers
lr	[ fp, #4 ]	[ sp, #20 ]	caller pass e parameters
e	[ fp, #8 ]	[ sp, #24 ]	

v  
Higher addresses

# Caller Function Code

```
.data
.align 4

message:
.asciz "Sum of 1^2 + 2^2 + 3^2 + 4^2 +
5^2 is %d\n"

.text

sq: <<defined above>>
sq_sum5: <<defined above>>

.globl main
main:

push { r4, lr }

pop { r4, lr }

bx lr
```

```
mov r0, #1      a ← 1
mov r1, #2      b ← 2
mov r2, #3      c ← 3
mov r3, #4      d ← 4

mov r4, #5      r4 ← 5

sub sp, sp, #8
str r4, [sp]    e ← 5

bl sq_sum5     sq_sum5 ( 1, 2, 3, 4, 5 )

add sp, sp, #8

mov r1, r0
ldr r0, address_of_message

bl printf

address_of_message: .word message
```



# APCS Register Use Convention

---

R11	fp	Frame Pointer
R12	ip	Scratch register / specialist use by linker
R13	sp	Lower end of current stack frame
R14	lr	Link address / scratch register
R15	pc	Program counter

# LR and FP Registers

---

SP    where the stack **is**  
FP    where the stack **was**

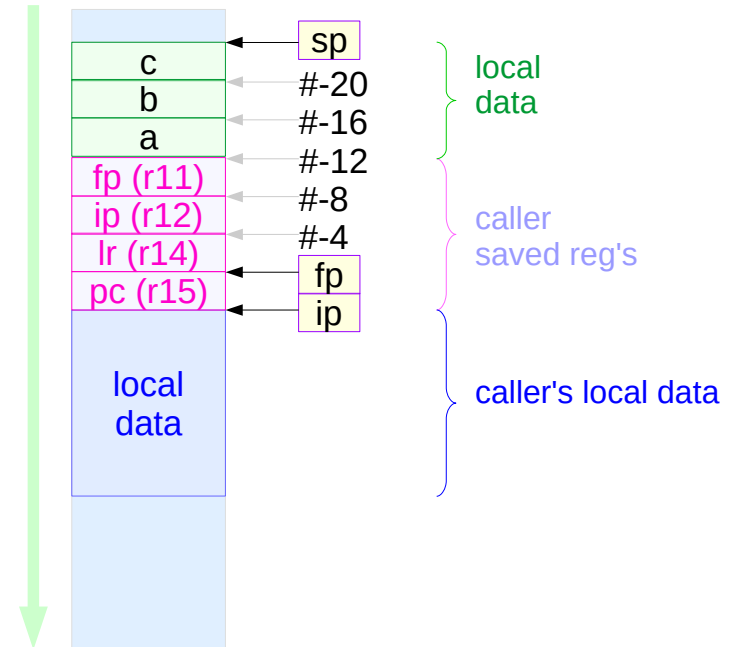
PC    where you **are**  
LR    where you **were**

<http://stackoverflow.com/questions/15752188/arm-link-register-and-frame-pointer>

# -fno-omit-frame-pointer

```
main:  
mov     ip, sp  
stmfd  sp!, { fp, ip, lr, pc }  
sub     fp, ip, #4  
sub     sp, sp, #12  
ldr     r2, [fp, #-16]  
ldr     r3, [fp, #-20]  
add     r3, r3, r2  
str     r3, [fp, #-24]  
sub     sp, fp, #12  
ldmfd  sp, {fp, sp, pc}
```

```
main()  
{  
    volatile int a, b, c;  
    c = a + b;  
}
```

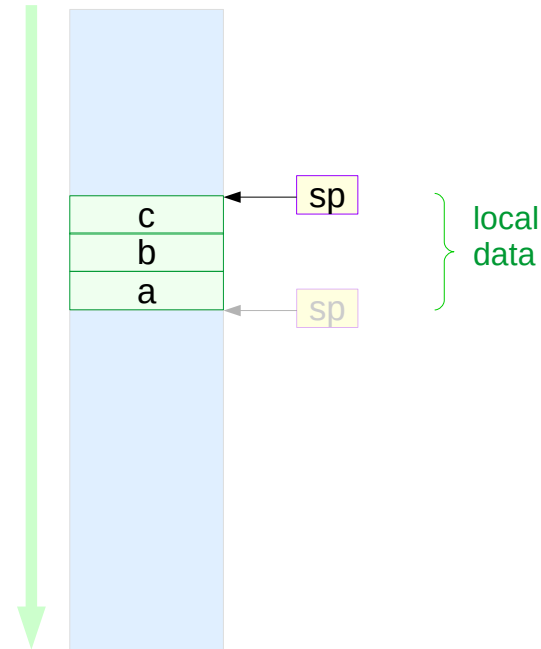


<https://community.arm.com/thread/7092>

# -fomit-frame-pointer

```
main:  
sub    sp, sp, #12  
ldr    r2, [sp, #8]  
ldr    r3, [fp, #4]  
add    r3, r3, r2  
str    r3, [sp, #0]  
sub    sp, sp, #12
```

```
main()  
{  
    volatile int a, b, c;  
    c = a + b;  
}
```



<https://community.arm.com/thread/7092>

# Local Variables

- Dynamic allocation / release allows for reuse of RAM
- Limited scope of access (making it private) provides for data protection
- Only the program that created the local variable can access it
- Since an interrupt will save registers, the code is reentrant
- Since absolute addressing is not used, the code is relocatable
  
- We can use symbolic names for the variables making it easier to understand
- The number of variables is only limited by the size of the stack
- Because it is more general, it will be easier to add additional variables
- 

Computer Organization and Design ARM Edition: The Hardware Software Interface by D. A. Patterson and J. L. Hennessy

# Global Variables

```
void MyFunction (void) {  
    static uint32_t count = 0;  
    count++;  
}
```

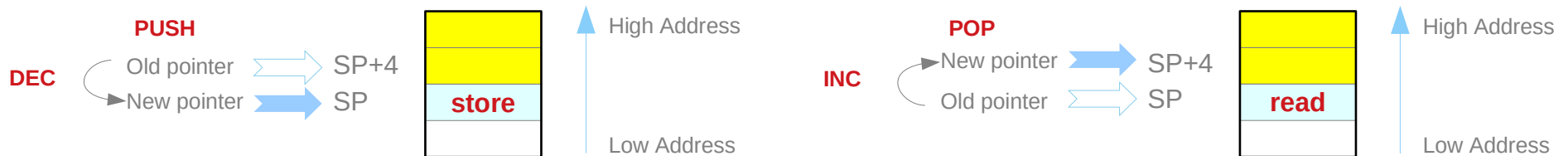
```
static int32_t myPrivateGlobalVariable; // accessible by this file only  
void static MyPrivateFunction (void) {  
}
```

```
const int16_t Slope=21;  
const uint8_t SinTable[8] = {0, 50, 98, 142, 180, 212, 236, 250};
```

# LIFO Stack

- Program segments should have an matching number of **pushes** and **pops**
- Stack accesses (push or pop) should not be performed outside the **allocation area**
- Stack reads and writes should not be performed within the **free area**
- Stack push should first **decrement** SP by 4, then **store** the data
- Stack pop should first read the data, then **increment** SP by 4

**Full Top  
Descending Stack**



Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

DEC

# LIFO Stack

```
LDR    R0, [SP, #4]    ; R0 = the next to the top bytes

SUB    R1, SP, #8     ; R1 points to the free area
STR    R0, [R1]       ; Store contents of R0 into free area (** illegal **)
LDR    R2, [R1]       ; Read contents of free area into R2 (** illegal **)

PUSH   {R0, R1}      ; Store contents of R0, R1 onto the stack
```



# Local variables on the stack

```
Func    PUSH    {R4, R5, R8, LR}    ; save registers as needed
        ; 1) allocate local variables
        ; 2) body of the function, access local variables
        ; 3) deallocate local variables
        POP     {R4, R5, R8, PC}
```

# Initializing a local array

```
void Set(void) {  
    uint32_t data[10];  
    int i;  
    for (i=0; i<10; i++) {  
        data[i] = i;  
    }  
}
```

Set	SUB	SP, SP, #40	; 1) allocate 10 words
	MOVS	R0, #0x00	; 2) i = 0
	B	test	; 2)
Loop	LSL	R1, R0, #2	; 2) 4*i
	STR	R0, [SP, R1]	; 2) access
	ADDS	R0, R0, #1	; 2) i++
Test	CMP	R0, #10	; 2)
	ADD	SP, SP, #40	; 3) deallocate
	BX	LR	

Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

# 1. Binding

---

Sum EQU 0 ; 32-bit local variable, stored on the stack

## 2. Allocation

```
MOV    R0, #0
MOV    R1, #2
PUSH   {R0, R1}           ; allocate and initialize two 32-bit variables

SUB    SP, #8             ; allocate two 32-bit variables
```

## 3. Access

```
LDR    R1, [SP, #sum]    ; R1 = sum
ADD    R1, R0            ; R1 = R0 + sum
STR    R1, [SP, #sum]    ; sum = R0 + sum
```

```
LDR    R0, [SP, #sum]    ; R0 = sum
LSR    R0, R0, #4        ;
STR    R0, [SP, #sum]    ; sum = sum / 4
```

# 4. Deallocation

---

```
ADD    SP, #4           ; deallocate sum
```

# Stack Frames

---

- Parameters
- Return address
- Saved registers
- Local variables

# Stack Frames

```
uint32_t calc(void) {  
    uint32_t    sum, n;  
    for (n=1000; n>0; n--) {  
        sum = sum + n;  
    }  
    return sum;  
}
```



# Stack Frame Implementation Example 1 (1)

```
; *** binding phase ****  
sum    EQU    0        ; 32-bit unsigned number  
n      EQU    4        ; 32-bit unsigned number  
  
; *** 1) allocation ***  
calc   PUSH   {R4, LR}  
       SUB    SP, #8    ; allocate n, sum
```

# Stack Frame Implementation Example 2 (2)

```
; *** 2) access ***
    MOV    R0, #0
    STR    R0, {R11, #sum}    ; sum = 0
    MOV    R1, #1000
    STR    R1, [R11, #n]      ; n = 1000
loop  LDR    R1, [R11, #n]      ; R1 = n
    LDR    R0, [R11, #sum]    ; R0 = sum
    ADD    R0, R1             ; R0 = sum + n
    STR    R0, [R11, #sum]    ; sum = sum + n
    LDR    R1, [R11, #n]      ; R1 = n
    SUBS   R1, #1             ; n-1
    STR    R1, [R11, #n]      ; n = n - 1
    BNE    loop

; *** 3) deallocation ***
    ADD    SP, #8             ; deallocation
    POP    {R4, R5, R11, PC} ; R0 = sum
```

Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

# Stack Frame Implementation Example 2 (1)

```
; *** binding phase ****  
sum    EQU    0        ; 32-bit unsigned number  
n      EQU    4        ; 32-bit unsigned number  
  
; *** 1) allocation ***  
calc   PUSH   {R4, R5, R11, LR}  
       SUB    SP, #8    ; allocate n, sum  
       MOV    R11, SP  ; frame pointer
```

## Stack Frame Implementation Example 2 (2)

```
; *** 2) access ***
    MOV    R0, #0
    STR    R0, {R11, #sum}    ; sum = 0
    MOV    R1, #1000
    STR    R1, [R11, #n]      ; n = 1000
loop  LDR    R1, [R11, #n]      ; R1 = n
    LDR    R0, [R11, #sum]    ; R0 = sum
    ADD    R0, R1              ; R0 = sum + n
    STR    R0, [R11, #sum]    ; sum = sum + n
    LDR    R1, [R11, #n]      ; R1 = n
    SUBS   R1, #1              ; n-1
    STR    R1, [R11, #n]      ; n = n - 1
    BNE    loop

; *** 3) deallocation ***
    ADD    SP, #8              ; deallocation
    POP    {R4, R5, R11, PC}  ; R0 = sum
```

Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

# Parameter Passing

Call by value	- safe, simple, good for small amounts of data
Call by reference	- parameter can be input or output, good for large amounts
Registers	- fast and simple
Stack	- flexible, good for large amounts of data
Global variables	- simple and poorstyle

# Call by value (1)

```
uint32_t next(uint32_t ang) {  
    ang++;  
    if (ang == 200) {  
        ang = 0;  
    }  
    return ang;  
}
```

```
void main (void) {  
    uint32_t angle = 0; // 0 to 199  
    Stepper_Init();  
    while (1) {  
        Stepper_Step();  
        angle = next(angle);  
    }  
}
```

# Call by value (2)

```
; R0 is the angle
next    ADD    R0, #1           ; add to copy
        CMP    R0, #200
        BNE   skip
        MOV   R0, #0           ; roll over
skip    BX     LR              ; 0 to 199
angle   EQU   0
main    SUB    SP, #4          ; allocate
        MOV   R0, #0
        STR   #0, [SP, #angle]
        BL   Stepper_Init
loop    BL   Stepper_Step
        LDR   R0, [SP, #angle] ; R0 = angle
        BL   next
        STR   R0, [SP, #angle] ; update
        B    loop
```

# Call by reference A (1)

```
uint32_t next(uint32_t *pt) {
    (*pt) = (*pt) + 1;
    if ((*pt) == 200) {
        (*pt) = 0;
    }
    return *pt;
}

void main (void) {
    uint32_t angle = 0; // 0 to 199
    Stepper_Init();
    while (1) {
        Stepper_Step();
        angle = next(&angle);
    }
}
```



# Call by reference A (2)

```
; R0 is points to the angle
next    LDR    R1, [R0]           ; *pt
        ADD    R1, #1           ; increment
        CMP    R1, #200
        BNE    skip
        MOV    R1, #0           ; roll over
skip    STR    R1, [R0]         ; update
        BX    LR                ; 0 to 199
angle   EQU    0
main    SUB    SP, #4           ; allocate
        MOV    R0, #0
        STR    #0, [SP, #angle]
        BL    Stepper_Init
loop    BL    Stepper_Step
        LDR    R0, [SP, #angle] ; R0 = angle
        BL    next
        STR    R0, [SP, #angle] ; update
        B     loop
```

# Call by reference B (1)

```
static int32_t Xx, Yy;           // position

void where(    int32_t *xpt,
              int32_t *ypt    ) {
    (*xpt) = Xx;                // return Xx
    (*ypt) = Yy;                // return Yy
}

void func(void) {
    int32_t myX, myY;
    where(&myX, &myY);
    // do something based on myX, myY
}
```

# Call by reference B (2)

```
Xx      SPACE      4          ; private to where
Yy      SPACE      4
where   LDR        R2, =Xx
        LDR        R2, [R2]    ; value of Xx
        STR        R2, [R0]    ; pass data
        LDR        R3, =Yy
        LDR        R3, [R3]    ; value of Yy
        STR        R3, [R1]    ; pass data
        BX        LR
myX     EQU        0          ; 32-bit
myY     EQU        4
func    PUSH       {R4, LR}
        SUB        SP, #8      ; allocate
        MOV        R0, SP      ; R0 = &myX
        ADD        R1, SP, #myY ; R1 = &myY
        BL        where
        ; do something base on myX, myY
        ADD        SP, #8      ; deallocate
        POP       {R4, PC}
```

# Parameter Passing

```
; Reg R0 = Port A,    Reg R1 = Port B
; Reg R3 = PortC,    Reg R3 = Port D
getPorts    LDR    R0, =GPIO_PORTA_DATA_R
            LDR    R0, [R0]                ; value of Port A
            LDR    R1, =GPIO_PORTB_DATA_R
            LDR    R1, [R1]                ; value of Port B
            LDR    R2, =GPIO_PORTC_DATA_R
            LDR    R2, [R2]                ; value of Port C
            LDR    R3, =GPIO_PORTD_DATA_R
            LDR    R3, [R3]                ; value of Port D
            BX     LR
            *** calling sequence ***
            BL     getPorts
; Reg R0, R1, R2, R3 have four results
```

# Parameter Passing – (1) using registers

```
; Inputs:      R0, R1
; Outputs:    R2 = R0 - R1
Sub1  SUB      R2, R0, R1
      BX      LR

      LDR      R0, =A
      LDR      R0, [R0]          ; R0 has the value of A
      LDR      R1, =B
      LDR      R1, [R1]          ; R1 has the value of B
      BL      Sub1
      LDR      R0, =C
      STR      R2, [R0]          ; C = A - B
```

# Parameter Passing – (2) using the stack

```
; Inputs:      In1 In2 on stack
; Outputs:    Out= In1 – In2 on stack
In1      EQU      8
In2      EQU      4
Out      EQU      0
Sub2     LDR      R0, [SP, #In1]
         LDR      R1, [SP, #In2]
         SUB      R2, R0, R1
         STR      R2, [SP, #Out]
         BX      LR

         LDR      R0, =A
         LDR      R0, [R0]          ; R0 has the value of A
         LDR      R1, =B
         LDR      R1, [R1]          ; R1 has the value of B
         PUSH    {R0, R1}          ; input parameters
         SUB      SP, #4            ; place for output
         BL      Sub2
         POP     {R2}              ; result
         LDR      R0, =C
         STR      R2, [R0]          ; C = A – B
         ADD     SP, #8            ; balance stack
```

Introduction to ARM Cortex-M Microcontrollers – Embedded Systems, Jonathan W. Valvano

# Parameter Passing – (3) using the stack (a)

```
; Inputs:      In1 In2 on stack
; Outputs:    Out= In1 – In2 on stack
In1      EQU      20
In2      EQU      16
Out      EQU      12
local    EQU      0
Sub3     PUSH     {R11, LR}
          SUB      SP, #4           ; allocate
          MOV      R11, SP         ; frame pointer
          LDR      R0, [R11, #In1]
          LDR      R1, [R11, #In2]
          SUB      R2, R0, R1
          STR      R2, [R11, #Out]
          ADD      SP, #4           ; deallocate
          POP      {R11, PC}
```

# Parameter Passing – (3) using the stack (b)

```
LDR    R0, =A
LDR    R0, [R0]           ; R0 has the value of A
LDR    R1, =B
LDR    R1, [R1]           ; R1 has the value of B
PUSH   {R0, R1}           ; input parameters
SUB    SP, #4             ; place for output
BL     Sub3
POP    {R2}               ; result
LDR    R0, =C
STR    R2, [R0]           ; C = A – B
ADD    SP, #4             ; deallocate stack
```



# Parameter Passing – (4) using global variables

```
; Inputs:      A, B
; Outputs:    C = A - B
Sub4  LDR      R0, =A
      LDR      R0, [R0]          ; R0 has the value of A
      LDR      R1, =B
      LDR      R1, [R1]          ; R1 has the value of B
      SUB      R2, R1, R0        ; A - B
      LDR      R0, =C
      STR      R0, [R0]
      BX      LR

      BL      Sub4
```

# Parameter Passing – (5) using memory locations

```
; Wait for Flag to become 1
Wait    LDR    R0, =Flag
Loop    LDR    R1, [R0]          ; R1 = Flag
        CMP   R1, #1
        BNE   loop             ; wait until 1
        MOV   R1, #0           ;
        STR   R1, [R0]        ; Flag = 0
        BX   LR
```

```
SysTick_Handler
        LDR   R0, =Flag        ; R0 = &Flag
        MOV   R1, #1
        STR   R1, [R0]        ; Flag = 1
        BX   LR               ; return form interrupt
```

# Compiler's local and global variable implementation (1)

Out = (99 \* In) / 100;

```
LDR    R1, [PC, #208]    ; (R1 + 1) = &In
MOVS   R2, #0x64         ; R2 = 100
LDRB   R0, [R1, #0x01]   ; R0 = In
ADD    R0, R0, R0, LSL #5 ; R0 = R0 + 32*R0 = 33 * In
ADD    R0, R0, R0, LSL #1 ; R0 = R0 + 2 *R0 = 99 * In
UDIV   R0, R0, R2         ; 99 * In / 100
STRB   R0, [R1, #0x02]   ; Out = 99 * In / 100
```

# Compiler's local and global variable implementation (1)

```
uint32_t combine (  
    uint32_t  msb,  
    uint32_t  lsb) {  
    return msb << 8 + lsb;  
}
```

```
Combine  MOV     R3, R0      ; R0 = msb  
        ADD     R3, R1, #0x08 ; lsb + 8  
        LSL     R0, R2, R3   ; msb << (8 + lsb)  
        BX     LR
```

# Compiler's local and global variable implementation (2)

```
int32_t  G;          // global
int32_t sub(int32_t *pt // R0
            int32_t index, // R1
            int32_t values) { // R2
    pt[index] -= value;
    return value;
}

Void main(void) {
    int32_t z[20]; //local
    G = 5;         // access global
    z[0] = 6;     // access local
    G = sub(z, 1, 2);
}
```

# Compiler's local and global variable implementation (3)

```
; R0 is *pt      ; R1 is index
; R2 is value
Sub      MOV      R3, R0          ; R3 is *pt
        LDR      R0, [R3, R1, LSL #2] ;
        SUBS    R0, R0, R2
        STR      R0, [R3, R1, LSL #2]
        MOV     R0, R2          ; return value
        BX     LR

Main     PUSH    {R4, LR}
        SUB     SP, SP, #0x50    ; allocate z
        MOVS   R0, #0x05
        LDR    R1, [PC, #340]    ; R1 = &G
        MOVS   R2
```



---

## References

- [1] [http://wiki.osdev.org/ARM\\_RaspberryPi\\_Tutorial\\_C](http://wiki.osdev.org/ARM_RaspberryPi_Tutorial_C)
- [2] <http://blog.bobuhiro11.net/2014/01-13-baremetal.html>
- [3] <http://www.valvers.com/open-software/raspberry-pi/>
- [4] <https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/downloads.html>