Lecture 6: ARM Cortex-M4 Software Development

Seyed-Hosein Attarzadeh-Niaki

Some slides due to ARM

Review

- ARM Cortex-M4 ISA
  - ARM, Thumb, and Thumb 2 instructions
  - Load and store instructions
  - Stack operations
  - Arithmetic and logic instructions
  - Branch and call instructions
  - Other instructions
- ARM assembly language syntax
- Examples

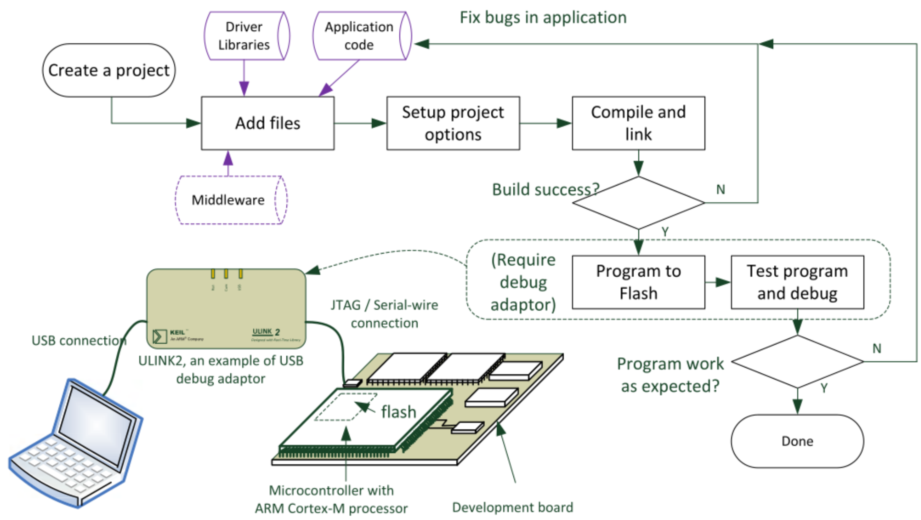
Outline

- Software development flow
- C For Embedded Systems
  - Data types
  - Bitwise operations
- Application Binary Interface (ABI)
  - Memory requirements
  - Calling procedures

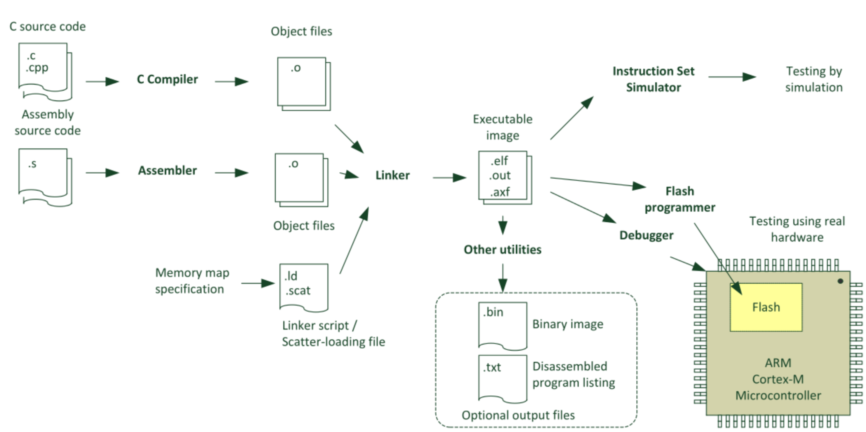
Requirements for Microcontroller Software Development

- Development suites
  - e.g., Keil (MDK-ARM), Coocox, GCC (+Eclipse), Arduino, etc.
- Development boards
  - e.g., STM Nucleo64
- Debug adaptor
  - To download the code to the board and debug
- Software device driver (C code and header files)
  - Definitions of peripheral registers
  - Access functions for configuring and accessing the peripherals
- Examples
- Documentation and resources
- Other equipment
  - Peripherals (LCD, …), Test (Logic analyzer, …), etc.

Simplified Software Development Flow

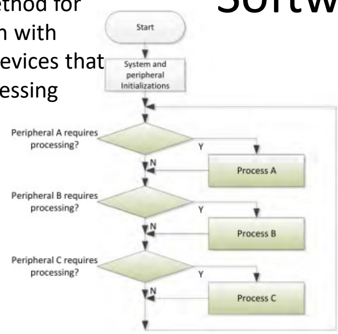


Common Software Compilation Flow

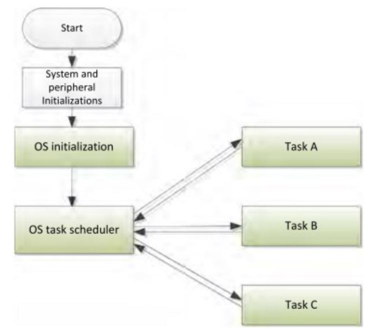


Software Flow

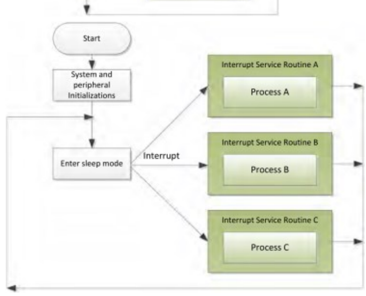
Polling method for application with multiple devices that need processing



Using an RTOS to handle multiple tasks

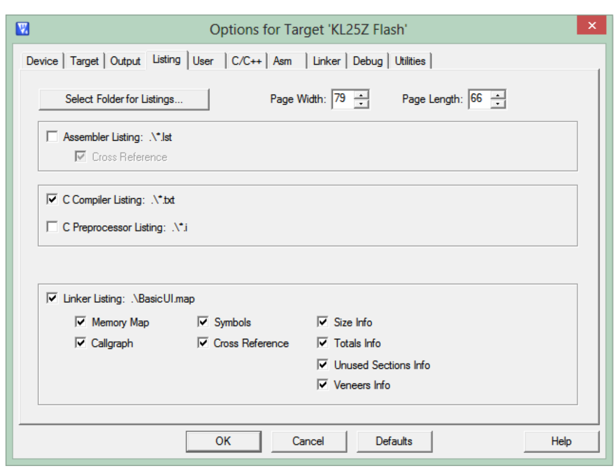


Simple interrupt-driven application



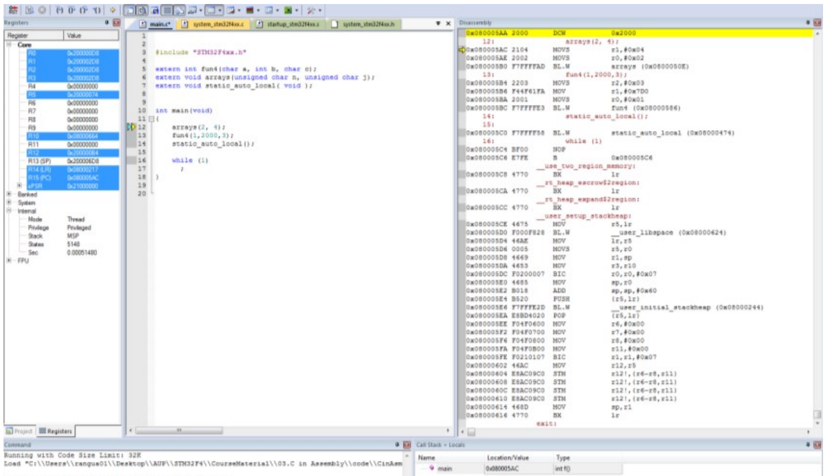
Examining Assembly Code Before Debugger

- Compiler can generate assembly code listing for reference
- Select in project options



Examining Disassembled Program in Debugger

View->Disassembly Window



C FOR EMBEDDED SYSTEMS

Why Is Data Type Important?

Performance

- Using 64-bit data type for saving 32-bit variable will cause
  - Waste RAM resource
  - Twice RAM access time
  - Additional arithmetic instructions

Overflow

- High level language programs do not provide indications when overflow occurs and the program just fails silently.
- If you use a short int to hold the number of seconds of a day, the second count will overflow from 32,767 to -32,768.

Coercion

- The compiler will convert the data types on incompatible assignments. These implicit data type is called **coercion**.
- The compiler may or may not give you warning when coercion occurs
- If the variable is signed and the data sized is increased, the new bits are filled with the sign bit (most significant bit) of the original value.
- When you assign a larger data type to a smaller data type variable, the higher order bits will be truncated.

C Data Types in ARM

- Standard C data types might differ in size (and range) in different architectures

Data type	Size	Range Min	Range Max
char	1 byte	-128	127
unsigned char	1 byte	0	255
short int	2 bytes	-32,768	32,767
unsigned short int	2 bytes	0	65,535
int	4 bytes	-2,147,483,648	2,147,483,647
unsigned int	4 bytes	0	4,294,967,295
long	4 bytes	-2,147,483,648	2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295	
long long	8 bytes	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long long	8 bytes	0	18,446,744,073,709,551,615

ISO C99 integer data types and their ranges

Data type	Size	Range Min	Range Max
int8_t	1 byte	-128	127
uint8_t	1 byte	0 to	255
int16_t	2 bytes	-32,768	32,767
uint16_t	2 bytes	0	65,535
int32_t	4 bytes	-2,147,483,648	2,147,483,647
uint32_t	4 bytes	0	4,294,967,295
int64_t	8 bytes	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
uint64_t	8 bytes	0	18,446,744,073,709,551,615

Size and Range of Cortex-M Data Types

C and C99 (stdint.h) Data Type	Number of Bits	Range (Signed)	Range (Unsigned)
char, int8_t, uint8_t	8	−128 to 127	0 to 255
short int16_t, uint16_t	16	−32768 to 32767	0 to 65535
int, int32_t, uint32_t	32	−2147483648 to 2147483647	0 to 4294967295
Long	32	−2147483648 to 2147483647	0 to 4294967295
long long, int64_t, uint64_t	64	−(2^63) to (2^63 - 1)	0 to (2^64 - 1)
Float	32	$-3.4028234 \times 10^{38}$ to 3.4028234×10^{38}	
Double	64	$-1.7976931348623157 \times 10^{308}$ to $1.7976931348623157 \times 10^{308}$	
long double	64	$-1.7976931348623157 \times 10^{308}$ to $1.7976931348623157 \times 10^{308}$	
Pointers	32	0x0 to 0xFFFFFFFF	
Enum	8 / 16/ 32	Smallest possible data type, except when overridden by compiler option	
bool (C++ only), _Bool (C only)	8	True or false	
wchar_t	16	0 to 65535	

Data Conversion Functions in C

- `stdlib.h` header file has some useful functions to convert integer to string or string to integer.

Function Name	Description
`int atoi(char *str)`	Converts the string str to integer
`long atol(char *str)`	Converts the string str to long
`void itoa(int n, char *str)`	Converts the integer n to characters in string str
`void ltoa(int n, char *str)`	Converts the long n to characters in string str
`float atof(char *str)`	Converts the characters from string str to float

Bit Operations in C

- Set and Clear
  - Anything ORed with a 1 results in a 1; anything ORed with a 0 results in no change.
  - Anything ANDed with a 1 results in no change; anything ANDed with a 0 results in a zero.
  - Anything EX-ORed with a 1 results in the complement; anything EX-ORed with a 0 results in no change.
- Test
  - When it is necessary to test a given bit to see if it is high or low, the unused bits are masked and then the remaining data is tested.
  - Example:
    `if (var1 & 0x20)`

A	B	AND (A & B)	OR (A | B)	EX-OR (A^B)	Invert ~B
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	1
1	1	1	1	0	0

Example: Write a C program to monitor bit 5 of var1. If it is HIGH, change value of var2 to 0x55; otherwise, change value of var2 to 0xAA.

```
while(1)
{
    /* check bit 5 (6th bit) of var1 */
    if (var1 & 0x20)
        /* this statement is executed if bit 5 is a 1 */
        var2 = 0x55;
    else
        /* this statement is executed if bit 5 is a 0 */
        var2 = 0xAA;
}
```


Using shift operator to generate mask

- To generate a mask with bit n set to 1, use the expression: `1 << n`
- If more bits are to be set in the mask, they can be “or” together. To generate a mask with bit n and bit m set to 1, use the expression: `(1 << n) | (1 << m)`
- `register |= (1 << 6) | (1 << 1);`

Operation	Symbol	Format of Shift Operation
Shift Right	>>	data >> number of bit-positions to be shifted right
Shift Left	<<	data << number of bit-positions to be shifted left

Setting the value in a multi-bit field

Example: set bits 28 to 30 to '101'

```
register |= 1 << 30;
register &= ~(1 << 29);
register |= 1 << 28;
```

Example: clear bits 28 to 30 and then set bits 28 & 30

```
register &= ~(7 << 28);
register |= 5 << 28;
```

```
register = register & ~(7 << 28)
                      | (5 << 28);
```


APPLICATION BINARY INTERFACE (ABI)

Application Binary Interface

- ABI defines rules which allow separately developed functions to work together

ARM Architecture Procedure Call Standard (AAPCS)

- Which registers must be saved and restored
- How to call procedures
- How to return from procedures

C Library ABI (CLIBABI)

- C Library functions

Run-Time ABI (RTABI)

- Run-time helper functions: 32/32 integer division, memory copying, floating-point operations, data type conversions, etc.

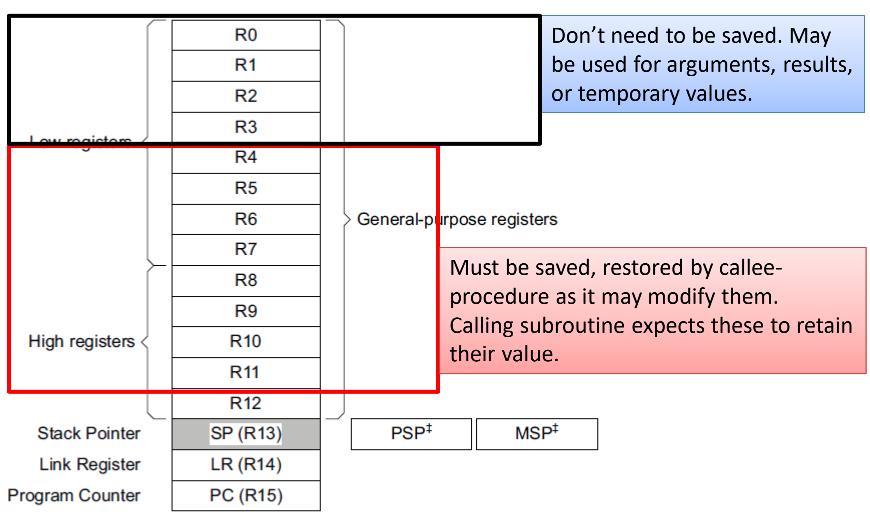
A Word on Code Optimizations

- Compiler and rest of tool-chain try to optimize code
  - Simplifying operations
  - Removing “dead” code
  - Using registers
- These optimizations often get in way of understanding what the code does
  - Fundamental trade-off: Fast or comprehensible code?
  - Compiler optimization levels: Level 0 to Level 3
- Some codes use “**`volatile`**” data type modifier to reduce compiler optimizations and improve readability

AAPCS Register Use Conventions

- Make it easier to create modular, isolated and integrated code
- For cortex-M4, there are two types of registers.
  - "Caller saved registers": registers that are not expected to be preserved upon returning from a called subroutine
    - R0-R3, R12, LR, PSR
  - "Callee-saved registers": Preserved ("variable") registers are expected to have their original values upon returning from a called subroutine
    - R4-R11

AAPCS Core Register Use



What Memory Does a Program Need?

```
int a, b;
const char c=123;
int d=31;
void main(void) {
   int e;
   char f[32];
   e = d + 7;
   a = e + 29999;
   strcpy(f,"Hello!");
}
```

- Five possible types
  - Code
  - Read-only static data
  - Writable static data
    - Initialized
    - Zero-initialized
    - Uninitialized
  - Heap
  - Stack
- What goes where?
  - Code is obvious
  - And the others?

What Memory Does a Program Need?

```
int a, b;
const char c=123;
int d=31;
void main(void) {
   int e;
   char f[32];
   e = d + 7;
   a = e + 29999;
   strcpy(f,"Hello!");
}
```

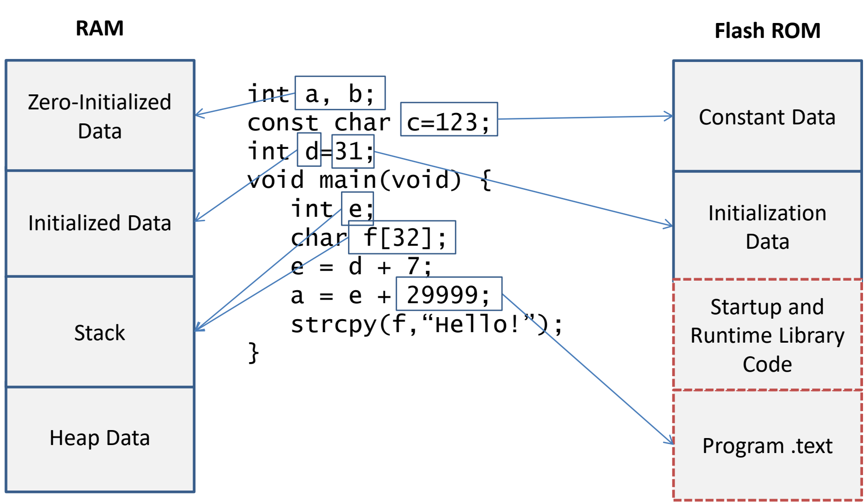
- Can the information change?
  - No? Put it in read-only, nonvolatile memory
    - Instructions
    - Constant strings
    - Constant operands
    - Initialization values
  - Yes? Put it in read/write memory
    - Variables
    - Intermediate computations
    - Return address
    - Other housekeeping data

What Memory Does a Program Need?

```
int a, b;
const char c=123;
int d=31;
void main(void) {
   int e;
   char f[32];
   e = d + 7;
   a = e + 29999;
   strcpy(f,“Hello!”);
}
```

- How long does the data need to exist? Reuse memory if possible.
  - Statically allocated
    - Exists from program start to end
    - Each variable has its own fixed location
    - Space is not reused
  - Automatically allocated
    - Exists from function start to end
    - Space can be reused
  - Dynamically allocated
    - Exists from explicit allocation to explicit deallocation
    - Space can be reused

Program Memory Use



Activation Record

- Activation records are located on the stack
  - Calling a function creates an activation record
  - Returning from a function deletes the activation record
- Automatic variables and housekeeping information are stored in a function's activation record

Lower address		(Free stack space)	
	Activation record for current function	Local storage	<- Stack ptr
		Return address	
		Arguments	
	Activation record for caller function	Local storage	
		Return address	
		Arguments	
	Activation record for caller's caller function	Local storage	
		Return address	
		Arguments	
Higher address	Activation record for caller's caller's caller function	Local storage	
		Return address	
		Arguments	

- Not all fields (LS, RA, Arg) may be present for each activation record

Type and Class Qualifiers

Const
- Never written by program, can be put in ROM to save RAM

Volatile
- Can be changed outside of normal program flow: ISR, hardware register
- Compiler must be careful with optimizations

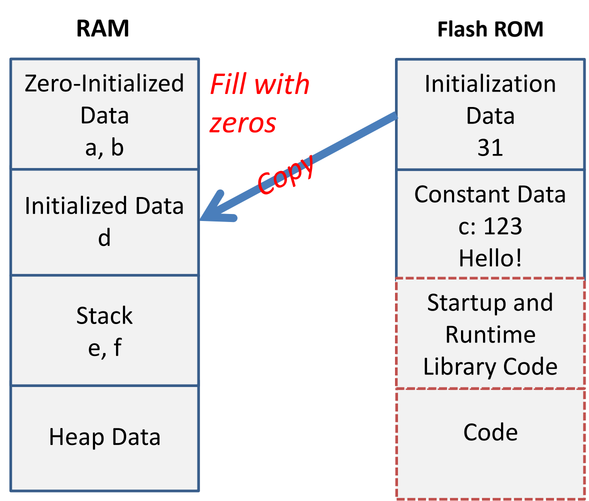
Static
- Declared within function, retains value between function invocations
- Scope is limited to function

Linker Map File

- Contains extensive information on functions and variables
  - Value, type, size, object
- Cross references between sections
- Memory map of image
- Sizes of image components
- Summary of memory requirements

C Run-Time Start-Up Module

- After reset, MCU must…
- Initialize hardware
  - Peripherals, etc.
  - Set up stack pointer
- Initialize C or C++ run-time environment
  - Set up heap memory
  - Initialize variables



Function Prolog and Epilog

- A function's P&E are responsible for creating and destroying its *activation record*
- Remember AAPCS
  – Caller saved registers r0-r3, r12 are not expected to be preserved upon returning from a called subroutine, can be overwritten
  – Callee saved ("variable") registers r4-r11 must have their original values upon returning from a called subroutine
  – Prolog must save preserved registers on stack
  – Epilog must restore preserved registers from stack
- Prolog also may
  – Handle function arguments
  – Allocate temporary storage space on stack (subtract from SP)
- Epilog
  – May deallocate stack space (add to SP)
  – Returns control to calling function

Return Address

- Return address stored in LR by bl, blx instructions
- Consider case where a() calls b() which calls c()
  – On entry to b(), LR holds return address in a()
  – When b() calls c(), LR will be overwritten with return address in b()
  – After c() returns, b() will have lost its return address
- Does this function call a subroutine?
  – Yes: must save and restore LR on stack just like other preserved registers, but LR value is popped into PC rather than LR
  – No: don't need to save or restore LR, as it will not be modified

Function Prolog and Epilog

```
int fun4(char a, int b, char c) {
  volatile int x[8];
  return a+b+c;
}
```

- Save r4 (preserved register) and link register (return address)
- Allocate 32 (0x20) bytes on stack for array x by subtracting from SP
- Compute return value, placing in return register r0
- Deallocate 32 bytes from stack
- Pop r4 (preserved register) and PC (return address)

```
 fun4 PROC
;;;102  int fun4(char a, int b, char c) {
;;;103    volatile int x[8];
00010a  b510  PUSH {r4,lr}
00010c  b088  SUB  sp,sp,#0x20
...
;;;106        return a+b+c;
00011c  1858  ADDS r0,r3,r1
00011e  1880  ADDS r0,r0,r2
;;;107    }
000120  b008  ADD  sp,sp,#0x20
000122  bd10  POP  {r4,pc}
              ENDP
```


Activation Record Creation by Prolog

Smaller address

space for x[0]	Array x	<- 3. SP after sub sp,sp,#0x20
space for x[1]		
space for x[2]		
space for x[3]		
space for x[4]		
space for x[5]		
space for x[6]		
space for x[7]		
lr	Return address	<- 2. SP after push {r4,lr}
r4	Preserved register	
	Caller's stack frame	<- 1. SP on entry to function, before push {r4,lr}

Larger address

Activation Record Destruction by Epilog

Smaller address ↑	space for x[0]	Array x	<- 1. SP before add sp,sp,#0x20
	space for x[1]		
	space for x[2]		
	space for x[3]		
	space for x[4]		
	space for x[5]		
	space for x[6]		
	space for x[7]		
	lr	Return address	<- 2. SP after add sp,sp,#20
	r4	Preserved register	
Larger address		Caller’s stack frame	<- 1. SP after pop {r4,pc}

Function Arguments and Return Values

- First, pass the arguments
  - How to pass them?
    - Much faster to use registers than stack
    - But quantity of registers is limited
  - Basic rules
    - Process arguments in order they appear in source code
    - Round size up to be a multiple of 4 bytes
    - Copy arguments into core registers (r0-r3), aligning doubles to even registers
    - Copy remaining arguments onto stack, aligning doubles to even addresses
    - Specific rules in AAPCS
- Second, call the function
  - Usually as subroutine with branch link (bl) or branch link and exchange instruction (blx)
  - Exceptions in AAPCS

Return Values

- Callee passes Return Value in register(s) or stack
- Registers
- Stack
  - Caller function allocates space for return value, then passes pointer to space as an argument to callee
  - Callee stores result at location indicated by pointer

Return value size	Registers used for passing	
	Fundamental Data Type	Composite Data Type
1-4 bytes	r0	r0
8 bytes	r0-r1	stack
16 bytes	r0-r3	stack
Indeterminate size	n/a	stack

Call Example

```
int fun2(int arg2_1, int arg2_2) {
  int i;
  arg2_2 += fun3(arg2_1, 4, 5, 6);
  …
}
```

- Argument 4 into r3
- Argument 3 into r2
- Argument 2 into r1
- Argument 0 into r0
- Call fun3 with BL instruction
- Result was returned in r0, so add to r4 (arg2_2 += result)

```
fun2 PROC
;;;85     int fun2(int arg2_1,
int arg2_2) {
...
0000e0  2306  MOVS  r3,#6
0000e2  2205  MOVS  r2,#5
0000e4  2104  MOVS  r1,#4
0000e6  4630  MOV   r0,r6

0000e8  f7fffffe   BL fun3

0000ec  1904  ADDS  r4,r0,r4
```

Call and Return Example

```
int fun3(int arg3_1, int arg3_2,
  int arg3_3, int arg3_4) {
  return   arg3_1*arg3_2*
           arg3_3*arg3_4;
}
```

```
 fun3 PROC
;;;81     int fun3(int arg3_1,
int arg3_2, int arg3_3, int
arg3_4) {
```

- Save r4 and Link Register on stack

```
0000ba  b510 PUSH {r4,lr}
```

- r0 = arg3_1*arg3_2
- r0 *= arg3_3
- r0 *= arg3_4

```
0000c0  4348 MULS r0,r1,r0
0000c2  4350 MULS r0,r2,r0
0000c4  4358 MULS r0,r3,r0
```

- Restore r4 and return from subroutine

```
0000c6  bd10 POP  {r4,pc}
```

- Return value is in r0