life.augmented

C code optimization on ARM Cortex-M architectures

10 March 2025

V2.1

Introduction

Agenda

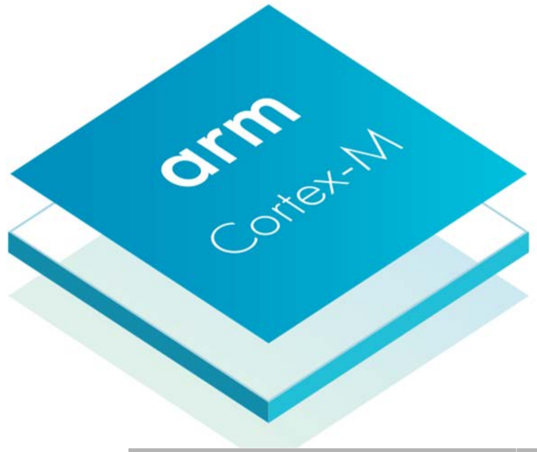
- C Language for Embedded
- ARM Cortex-M Architecture
- Cortex-M fundamentals
- Optimizing C
- EABI Introduction
- Optimizing for Cortex-M
- Knowing your Compiler
- Q&A

life.augmented

Cortex-M and EABI

life.augmented

Cortex-M Architectures



- There are several Cortex-M devices implementing different ARM architectures.

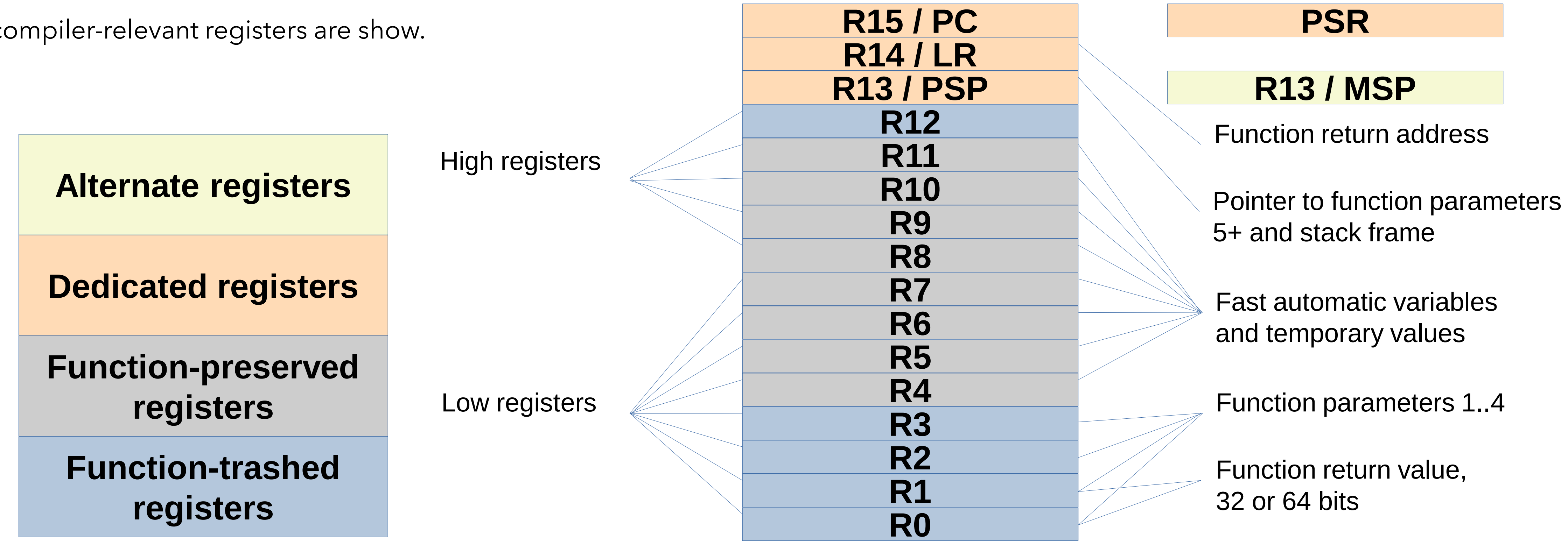
ARMv6-M	Cortex-M0 Cortex-M0+	Simplified architecture, low cost, less efficient high registers usage because simplified instruction set, no HW division instruction.
ARMv7-M	Cortex-M3	Rich instruction set, high performance devices. Efficient high registers usage.
ARMv7-MF	Cortex-M4 Cortex-M7	Introduces FPU support.
ARMv8-M-BL	Cortex-M23	Baseline ARMv8-M architecture, it is the “new” ARMv6-M. Introduces TrustZone and some other new features, shares limitations.
ARMv8-M ARMv8.1-M	Cortex-M33 Cortex-M52 Cortex-M55 Cortex-M85	Mainline ARMv8-M, introduces TrustZone and some other new advanced features.

life.augmented

ARM EABI – The integer registers file

From the programmer point of view the registers file is the most important architectural aspect, registers type and organization knowledge is important for writing good code for embedded devices.

Only compiler-relevant registers are show.

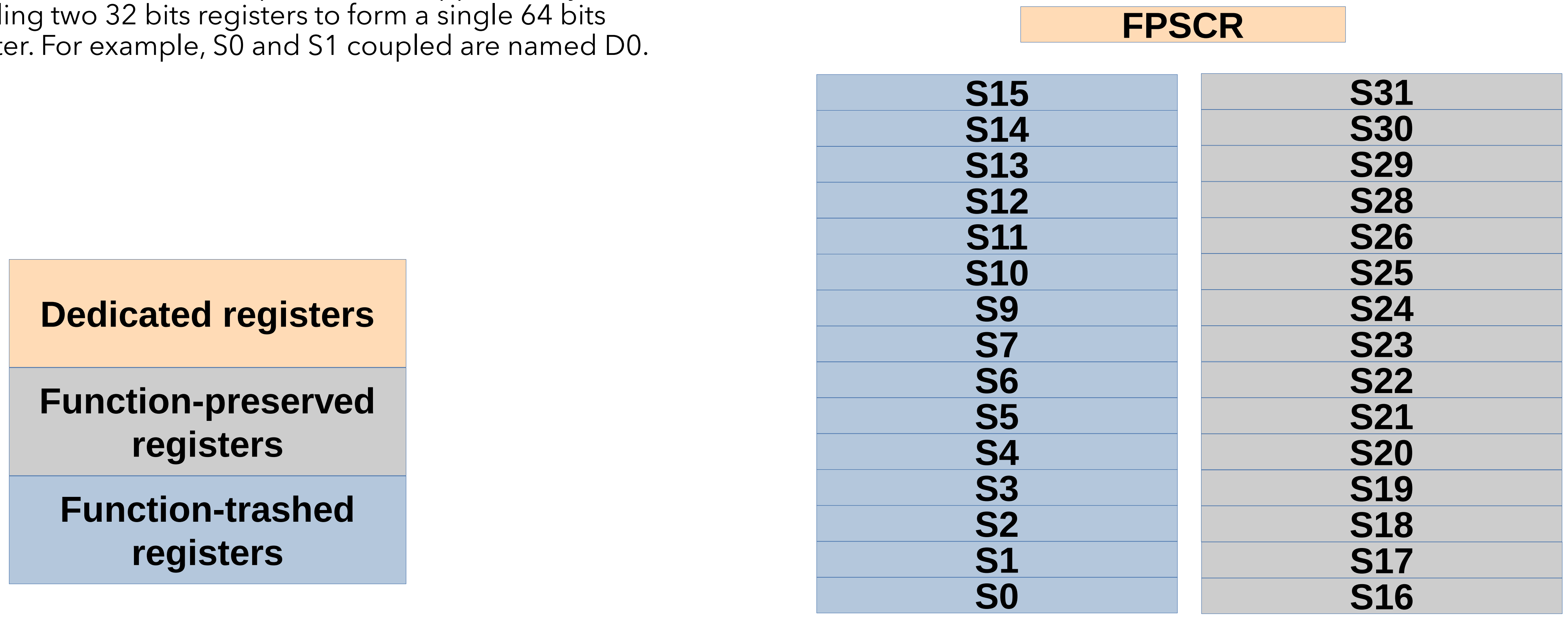


Alternate registers
Dedicated registers
Function-preserved registers
Function-trashed registers
High registers
Low registers
R15 / PC
R14 / LR
R13 / PSP
R12
R11
R10
R9
R8
R7
R6
R5
R4
R3
R2
R1
R0
PSR
R13 / MSP
Function return address
Pointer to function parameters 5+ and stack frame
Fast automatic variables and temporary values
Function parameters 1..4
Function return value, 32 or 64 bits

ARM EABI – The FP registers file

Note that FP registers are 32 bits wide and can store single-precision float numbers.

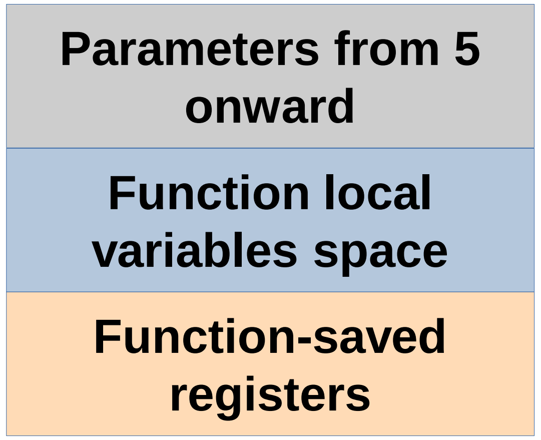
On some devices double precision is supported by coupling two 32 bits registers to form a single 64 bits register. For example, S0 and S1 coupled are named D0.

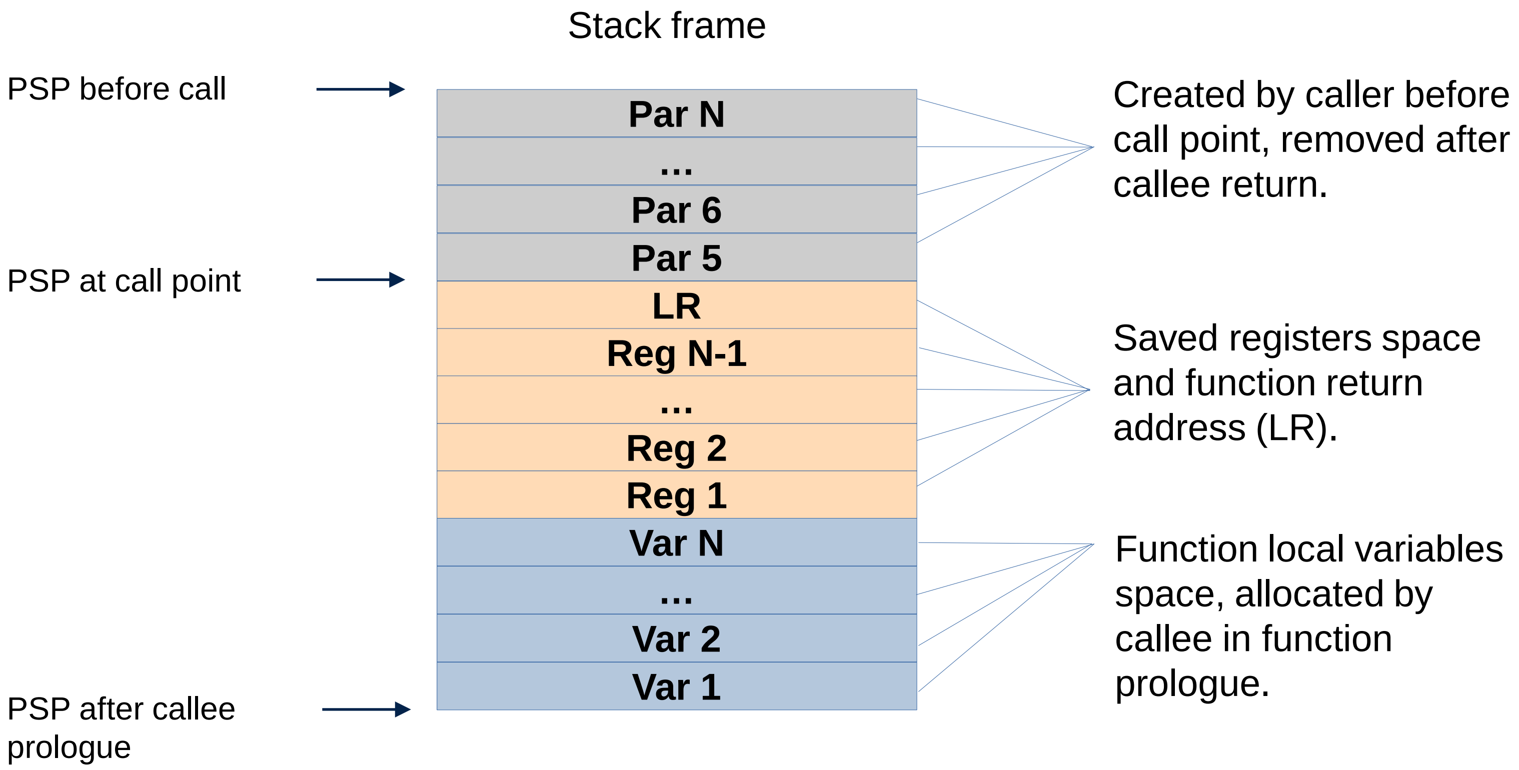


ARM EABI – The stack frame

The function stack frame is an area of stack containing extra function parameters and local variables.

Note, the stack pointer must be always aligned to 8 bytes boundaries.




Stack frame
PSP before call
PSP at call point
PSP after callee prologue
Par N
…
Par 6
Par 5
LR
Reg N-1
…
Reg 2
Reg 1
Var N
…
Var 2
Var 1
Created by caller before call point, removed after callee return.
Saved registers space and function return address (LR).
Function local variables space, allocated by callee in function prologue.

The C compiler

life.augmented

C language for embedded

- Created by Dennis Ritchie in 70s.
- Still going strong for embedded development.
  - No runtime/OS assumptions, required for bare metal programming.
  - Predictable output, 1 to 1 relation between written C code and generated ASM code.
  - Can be seen as some kind of high-level assembler.
  - Has limitations and allows for mistakes. Tools and use standards exist to mitigate the problems.
  - Excellent and stable compilers.
- Emerging alternatives.
  - Often addressing the wrong problems. Still no ideal language for embedded/system development.

life.augmented

GCC compiler

- 1st open-source compiler, almost 40 years history.
- Excellent ARM support by ARM itself.
- Yearly releases for Windows, MAC and Linux.
- Excellent generated code.
- Excellent documentation.
- World networking infrastructure largely relying on GCC.
- Support for multiple languages.
- Using version 12.2 in following examples.
- LLVM getting closer.

life.augmented

Variables access

life.augmented

Accessing static variables

Accessing variables, apparently a trivial task. This is simple example of a function increasing 3 static variables:

```c
static int a;
static int b;
static int c;

void test1(void) {

  a++;
  b++;
  c++;
}
```

Simple right?

It generates some quite complex asm code:

```
8000360 <test1>:
8000360:    b430        push    {r4, r5}
8000362:    4d07        ldr     r5, [pc, #28]   <- Taking A address
8000364:    4c07        ldr     r4, [pc, #28]   <- Taking B address
8000366:    682a        ldr     r2, [r5, #0]
8000368:    6823        ldr     r3, [r4, #0]
800036a:    4807        ldr     r0, [pc, #28]   <- Taking C address
800036c:    1c51        adds    r1, r2, #1
800036e:    1c5a        adds    r2, r3, #1
8000370:    6803        ldr     r3, [r0, #0]
8000372:    6029        str     r1, [r5, #0]
8000374:    3301        adds    r3, #1
8000376:    6022        str     r2, [r4, #0]
8000378:    6003        str     r3, [r0, #0]
800037a:    bc30        pop     {r4, r5}
800037c:    4770        bx      lr
800037e:    bf00        nop                     <- Just for alignment
8000380:    20000814    .word   0x20000814      <- A address
8000384:    20000810    .word   0x20000810      <- B address
8000388:    2000080c    .word   0x2000080c      <- C address
```

Total: 44 bytes of code, note registers stacking.

Variables grouping optimization

An effective optimization is to group static variables into a structure.

```
static struct {
  int a;
  int b;
  int c;
} group;

void test2(void) {

  group.a++;
  group.b++;
  group.c++;
}
```

It generates a much simpler asm code:

```
8000390 <test2>:
8000390:    4b05        ldr     r3, [pc, #20]   <- Address taken once
8000392:    e9d3 0100   ldrd    r0, r1, [r3]
8000396:    689a        ldr     r2, [r3, #8]
8000398:    3001        adds    r0, #1
800039a:    3101        adds    r1, #1
800039c:    3201        adds    r2, #1
800039e:    e9c3 0100   strd    r0, r1, [r3]
80003a2:    609a        str     r2, [r3, #8]
80003a4:    4770        bx      lr
80003a6:    bf00        nop                     <- Just for alignment
80003a8:    20000800    .word   0x20000800      <- Structure address
```

Total: 28 bytes of code, note less registers used, no stack used.

By grouping variables, the compiler “knows” the relative offset between variables and can take the memory address just once.

Registers pressure

life.augmented

Registers pressure

Register pressure is a concept that refers to the limited number of registers available in a processor and the allocation pressure on these registers when executing a program.

In the context of the C language, register pressure can have a significant impact on the performance of a program. C programs often use many variables, and the compiler must allocate these variables to registers or memory locations.

When there are more variables than available registers, the compiler must spill some of the variables to memory, which can result in slower program execution and increased stack usage.

Therefore, managing register pressure is an important consideration for optimizing the performance of C programs.

Factors contributing to register pressure:

- Number of function parameters.
- Number of function automatic variables.
- Calling functions within functions.
- Accessing numerous global variables (the pointer to the variable is kept within a register).
- Functions inlining.
- Called functions visibility from caller point.
- Pressure varies across the lines within a function, <u>the impacting value is the highest one</u>.

life.augmented

Registers pressure - details

Factors contributing to register pressure:

- Number of function automatic variables. Each variable takes one register (two for 64 bits variables).
- Calling functions within functions. Calling functions invalidates registers R0, R1, R2, R3, R12 making them unavailable to variables allocation.
- Accessing numerous global variables. Each global variable is accessed by pointer, the pointer takes one register, the temporary variable storage takes another register.
- Complex expressions requiring multiple intermediate results, common sub-expressions elimination.

Contributing factor	Explanation
Number of automatic variables	Increase pressure by 1 or 2 for each overlapping automatic variable at the variable use point.
Calling a function	Increase pressure by 5 at the function call point.
Accessing global variables	Increase pressure by 2 for each overlapping global variable use point (pointer to the variable and value).
Accessing a variable allocated in stack frame.	Increase pressure by 1 at the variable use point (value brought temporarily into a register).

life.augmented

Registers pressure – overlapping variables

Overlapping variables are those whose life span overlaps the life span of other variables:

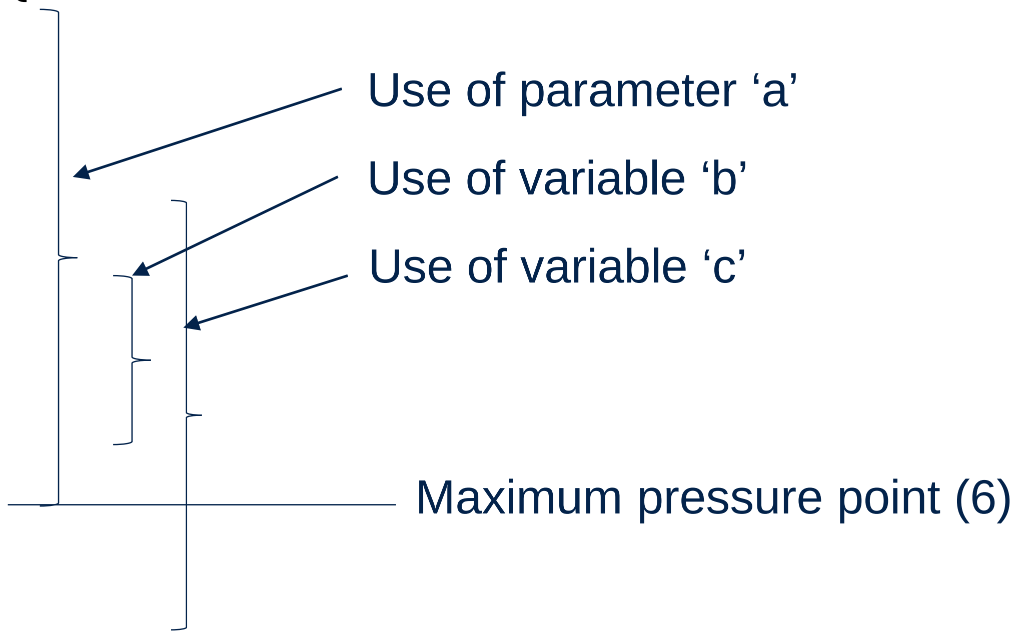
```
void func(int a) {
  int b;
  int c;


  . . .




  foo(a);

  . . .
}
```



Arranging the code to make variables and parameters have non-overlapping life spans reduces register pressure. Parameters life span always start at function begin.

Registers pressure – example

Example code:

```
static int var1;

__attribute__((noinline))
void foo1(int p1, int p2) {

  var1 -= p1 + p2;
}

int test4(int a) {
  int b;
  int c;

  var1++;
  b = var1;
  foo1(a, b);
  var1++;
  c = var1 + a;

  return c;
}
```

Resulting asm code:

```
80003f0 <test4>:
80003f0:    b538        push    {r3, r4, r5, lr}  ; Prologue, note R3 pushed for alignment
80003f2:    4c06        ldr     r4, [pc, #24]     ; Pointer to var1
80003f4:    6821        ldr     r1, [r4, #0]      ; Value of var 1
80003f6:    3101        adds    r1, #1            ; Increase value of var1
80003f8:    4605        mov     r5, r0            ; Moving ‘a’ in R5 because function call
80003fa:    6021        str     r1, [r4, #0]      ; Store new value of var1
80003fc:    f7ff fff0   bl      80003e0 <foo1>
8000400:    6823        ldr     r3, [r4, #0]      ; Taking var1 value again
8000402:    3301        adds    r3, #1            ; Increase value of var1
8000404:    6023        str     r3, [r4, #0]      ; Store new value of var1
8000406:    1958        adds    r0, r3, r5
8000408:    bd38        pop     {r3, r4, r5, pc}  ; Restoring registers and return
800040a:    bf00        nop                       <- Just for alignment
800040c:    20000800    .word   0x20000800
```

Note how because pressure the compiler is forced to stack 4 registers in the function prologue and restore them in the epilogue.

Function types

life.augmented

Functions

Functions are an essential part of the C programming language. They are used to break down a large program into smaller, more manageable pieces of code. Functions allow developers to reuse code, making it easier to maintain and update programs. They also help to improve the readability and organization of code, making it easier to understand and debug. In C, functions can be used to perform a wide range of tasks, from simple calculations to complex operations.

Functions are also the main target for code optimization often being a major bottleneck for code execution.

C functions:

- Code organization and readability.
- Avoiding redundant or duplicated code.
- Important target for code optimization.
- There are many kind of functions, choosing the right one.
- Optimizing often requires non-standard C extensions.

Global functions

This is the most common type of functions but also the one causing most problems to the compiler for code optimization.

- Calling a global function trashes r0..r3, r12 registers forcing the compiler to move data contained there to other registers or in memory locations within the function stack frame.
- External functions are "opaque", the compiler cannot tell if the function really trashes all those registers or just some of them, so it must assume "all trashed".
- External functions are also memory barriers, calling a global function makes the compiler assume that global variables potentially changed, making the compiler not trust variables cached in registers, basically forcing a re-fetch.

This is the general declaration form:

```
int fname(int par1, .., int parN, ...) {

  /* Function body.*/

  return 0;
}
```

Functions can have an unlimited number of parameters and optionally have a variable number of parameters using the "…" notation.

Functions can return a result or "`void`".

When possible, **avoid**:

- **Using more that 4 parameters.**
- **Using variable number of parameters.**
- **Returning types larger than 32 bits.**

A good idea is to use Link Time Optimization that makes the compiler aware of function at whole application level

life.augmented

Local functions

Local functions are visible to the compiler from places where functions are called offering some hints for optimizations.

- Calling a local function trashes r0..r3, r12 registers forcing the compiler to move data there contained to other registers or in memory locations within the function stack frame.
- Local functions are “transparent”, the compiler can tell if the function really trashes all those registers or just some of them, so it can optimize registers usage.
- Local functions, being transparent, tell the compiler if/which static variables are modified by the function. This allows the compiler to not re-fetch unmodified variables.
- Local functions are eligible for inlining.
- Avoid naming conflicts.

This is the general declaration form:

```
static int fname(int par1, .., int parN, ...) {

  /* Function body.*/

  return 0;
}
```

Functionally local functions are equivalent to global functions except their visibility is limited to the local module. When possible, functions should be declared as static.

Inline functions

Inline functions code is “dissolved” into the caller function code.

- There is no registers trashing because the function is not actually called but incorporated in the caller.
- Less stack usage because no function is called. This saves stack RAM, especially important when an RTOS is used.
- Being the code “dissolved” in the caller allows the compiler to optimize caller and callee together improving overall registers pressure.
- Inline functions are often placed in header files as replacement for preprocessor macros, replacing macros with inline functions is recommended because stronger checking and no risk of side effects.

This is the general declaration form:

```
static int fname(int par1, .., int parN, ...) {

  /* Function body.*/

  return 0;
}
```

Note that the “`inline`” keyword does not enforce inlining, it only gives the compiler a hint that inlining is desirable. Avoid:

- Making large inline functions, this can make code size explode.
- Avoid inline functions in .c files, make those plan “” functions and let the compiler if/when inline those. Inline functions are better declared in .h files.

Forced inline functions

Forced inline functions are not a standard C feature, it is done using compiler-specific extension.

Luckly CMSIS offers an abstraction that allows to use this feature in a portable way.

- Forced inline functions are guaranteed to be inlined
- Same advantages/disadvantages as normal inline functions.
- Being a CMSIS abstraction this is a Cortex-M specific feature.
- GCC can do this on any architecture by using the function attribute:

```
__attribute__((always_inline))
```

This is the general declaration form:

```
#include "cmsis.h"

__STATIC_FORCEINLINE int fname(int par1, .., int
parN) {

  /* Function body.*/

  return 0;
}
```

Alternatively:

```
__attribute__((always_inline))
int fname(int par1, .., int parN) {

  /* Function body.*/

  return 0;
}
```

Cold and hot functions

Cold and hot functions give the compiler a hint about functions that should be compiled with specific options.

Cold functions are those rarely executed, the compiler optimizes cold functions for size rather than for speed.

Hot functions are those that affect system performance significatively, the compiler optimizes hot functions for speed rather than for size.

- GCC can do this on any architecture by using the function attributes:

  ```
  __attribute__((cold))
  __attribute__((hot))
  ```

- Cold and hot functions are GCC-specific extensions, other compilers may or may not have this capability.

Cold functions:

```
__attribute__((cold))
int fname(int par1, .., int parN, ...) {

  /* Function body.*/

  return 0;
}
```

Hot functions:

```
__attribute__((hot))
int fname(int par1, .., int parN, ...) {

  /* Function body.*/

  return 0;
}
```

Pure functions

Calls to functions that have no observable effects on the state of the program other than to return a value may lend themselves to optimizations such as common subexpression elimination. Declaring such functions with the pure attribute allows the compiler to avoid emitting some calls in repeated invocations of the function with the same argument values.

The pure attribute prohibits a function from modifying the state of the program that is observable by means other than inspecting the function's return value.

However, functions declared with the pure attribute can safely read any nonvolatile objects and modify the value of objects in a way that does not affect their return value or the observable state of the program.

This is the general declaration form:

```
__attribute__((pure))
int fname(int par1, .., int parN) {

  /* Function body.*/

  return 0;
}
```

Pure functions:

- Can improve code size/speed.
- Pure functions are GCC-specific extensions, other compilers may or may not have this capability.

Const functions

Calls to functions whose return value is not affected by changes to the observable state of the program and that have no observable effects on such state other than to return a value may lend themselves to optimizations such as common subexpression elimination. Declaring such functions with the const attribute allows the compiler to avoid emitting some calls in repeated invocations of the function with the same argument values.

The const attribute prohibits a function from reading objects that affect its return value between successive invocations. However, functions declared with the attribute can safely read objects that do not change their return value, such as non-volatile constants.

The const attribute imposes greater restrictions on a function's definition than the similar pure attribute

This is the general declaration form:

```
__attribute__((const))
int fname(int par1, .., int parN) {

  /* Function body.*/

  return 0;
}
```

Const functions:

- Can improve code size/speed.
- Const functions are GCC-specific extensions, other compilers may or may not have this capability.

Other GCC attributes

Other GCC function attributes affecting optimization, see the GCC manual for details:

Attribute	Explanation
aligned	Makes a function start address aligned to a specified power of 2 number. This can lead to performance improvements in system with flash-prefetch features or cache memories.
flatten	Makes a function “flat”, all functions called from within the function are inlined if possible.
leaf	Gives compiler a hint that a function is not escaping the execution flow using function pointers.
noinline	Prevents a function to be inlined.
noreturn	Used for function that do not return to the caller function. This can improve generated code.

life.augmented

Function-related tricks

life.augmented

Head calls #1

A specific optimization can be performed when a function calls another function that takes some of its parameters as parameters. See this example:

```
void foo2(int p1, int p2, int p3) {

  var1 -= p1 + p2 + p3;
}

int test5(int a, int b) {

  foo2(18, a, b);

  var1++;

  return var1;
}
```

This results in the following code:

```
8000430 <test5>:
8000430:   b508       push    {r3, lr}
8000432:   4603       mov     r3, r0          <- reordering
8000434:   460a       mov     r2, r1          <- reordering
8000436:   2012       movs    r0, #18
8000438:   4619       mov     r1, r3          <- reordering
800043a:   f7ff ffe9  bl      8000410 <foo2>
800043e:   4b02       ldr     r3, [pc, #8]
8000440:   6818       ldr     r0, [r3, #0]
8000442:   3001       adds    r0, #1
8000444:   6018       str     r0, [r3, #0]
8000446:   bd08       pop     {r3, pc}
8000448:   20000800   .word   0x20000800
```

The order of parameters forces the compiler to reorder registers before calling foo2().

Head calls #2

Simple change, just keep parameters order and position the same as in caller function:

```
void foo2(int p1, int p2, int p3) {

  var1 -= p1 + p2 + p3;
}

int test5(int a, int b) {

  foo2(a, b, 18);

  var1++;

  return var1;
}
```

This results in the following code:

```
8000450 <test6>:
8000450:    b508        push    {r3, lr}
8000452:    2212        movs    r2, #18
8000454:    f7ff ffdc   bl      8000410 <foo2>
8000458:    4b02        ldr     r3, [pc, #8]
800045a:    6818        ldr     r0, [r3, #0]
800045c:    3001        adds    r0, #1
800045e:    6018        str     r0, [r3, #0]
8000460:    bd08        pop     {r3, pc}
8000462:    bf00        nop                     <- Just for alignment
8000464:    20000800    .word   0x20000800
```

Simple change and much better resulting code.

Tail calls #1

An important optimization is when a function calls another function as very last thing in this body. Example:

```
int foo3(int p1, int p2) {

  return p1 + p2;
}

int test7(int a, int b) {
  int sum;

  sum = foo3(a, b);

  var1++;

  return sum;
}
```

This results in the following code:

```
8000480 <test7>:
8000480:    b508        push    {r3, lr}        <- stacking LR
8000482:    f7ff fff5   bl      8000470 <foo3>
8000486:    4a02        ldr     r2, [pc, #8]
8000488:    6813        ldr     r3, [r2, #0]
800048a:    3301        adds    r3, #1
800048c:    6013        str     r3, [r2, #0]
800048e:    bd08        pop     {r3, pc}        <- unstacking LR
8000490:    20000800    .word   0x20000800
```

Apparently, nothing wrong but there are non-obvious problems in this.

Tail calls #2

The increase of var1 could be moved before the function call making the call of foo3() the last statement in the function body:

```
int foo3(int p1, int p2) {

  return p1 + p2;
}

int test8(int a, int b) {

  var1++;

  return foo3(a, b);
}
```

This results in the following code:

```
80004a0 <test8>:
80004a0:    4a02        ldr     r2, [pc, #8]
80004a2:    6813        ldr     r3, [r2, #0]
80004a4:    3301        adds    r3, #1
80004a6:    6013        str     r3, [r2, #0]
80004a8:    f7ff bfe2   b.w     8000470 <foo3> <- Note here, it's a branch not a call
80004ac:    20000800    .word   0x20000800
```

Now the caller function does not need to stack/unstack LR anymore because foo3() does not return in test8() but directly in test8() caller.

In addition, there is less stack usage because test8() and foo3() stack frames overlaps instead of nesting, saving stack is especially important when using and RTOS.

Function crossing #1

One onerous operation is to bring variables across a function call. Example:

```
void foo4(int p1, int p2) {

  var1 += p1 + p2;
}

void test9(int a, int b) {

  var1 += a;

  foo4(a, b);

  var1 += b;
}
```

This results in the following code:

```
80004d0 <test9>:
80004d0:    b538        push    {r3, r4, r5, lr}
80004d2:    4c05        ldr     r4, [pc, #20]    <- Address of “var1” in R4
80004d4:    6823        ldr     r3, [r4, #0]
80004d6:    4403        add     r3, r0
80004d8:    460d        mov     r5, r1           <- Variable “b” moved in R5
80004da:    6023        str     r3, [r4, #0]
80004dc:    f7ff fff0   bl      80004c0 <foo4>
80004e0:    6823        ldr     r3, [r4, #0]
80004e2:    442b        add     r3, r5
80004e4:    6023        str     r3, [r4, #0]
80004e6:    bd38        pop     {r3, r4, r5, pc}
80004e8:    20000800    .word   0x20000800
```

Note that there is a lot of stacking/unstacking done to bring R4, R5, LR values across the function call.

Function crossing #2

Now we make foo5() return “p2”.

```
int foo5(int p1, int p2) {

  var1 += p1 + p2;

  return p2;
}

void test10(int a, int b) {

  var1 += a;

  b = foo5(a, b);

  var1 += b;
}
```

Now “b” does not need to be brought **around** foo5(), it is bridged **within** foo5().

This results in the following code:

```
8000510 <test10>:
8000510:    b510        push    {r4, lr}        <- Less stacking
8000512:    4c05        ldr     r4, [pc, #20]
8000514:    6823        ldr     r3, [r4, #0]
8000516:    4403        add     r3, r0
8000518:    6023        str     r3, [r4, #0]
800051a:    f7ff ffe9   bl      80004f0 <foo5>
800051e:    6823        ldr     r3, [r4, #0]
8000520:    4403        add     r3, r0
8000522:    6023        str     r3, [r4, #0]
8000524:    bd10        pop     {r4, pc}        <- Less unstacking
8000526:    bf00        nop                     <- Just for alignment
8000528:    20000800    .word   0x20000800
```

Code is a bit smaller, there are less stacked registers making it faster and resulting in less stack space used.

Branches optimization

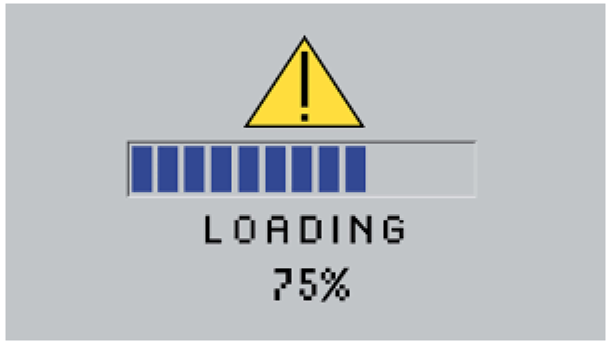
life.augmented

Optimizing branches

Most microcontrollers include prefetch mechanism on flash controllers. Because of this jumping out an instruction sequence can have a cost of several cycles, flash is slow, you can have many wait states when the prefetch queue is empty.

The problem is that compilers are mostly unable to decide if to jump when a condition is true or false.

- Flash memories are “slow” compared to CPUs this is why several wait states are to be expected.
- Flash-stored programs take an execution impact when jumping out of an instruction flow.
- Sometime compiler cannot guess the best way to arrange code, manual optimization is required.



Likely and Unlikely conditions #1

GCC offers support for giving hints to the compiler about probability of a condition to be true or false.

For simplicity we define a couple macros like this:

```
#define likely(x) __builtin_expect(!!(x), 1)
#define unlikely(x) __builtin_expect(!!(x), 0)
```

The parameter “x” of the macro is a Boolean expression, the macros just return the expression result but also give the compiler a hint about if the condition is likely or unlikely to be true.

Here an example taken from ChibiOS:

```
static inline
ch_priority_queue_t *ch_insert(ch_priority_queue_t *pqp,
                               ch_priority_queue_t *p) {

  /* Scanning priority queue, the list is assumed to be
     mostly non-empty.*/
  do {
    pqp = pqp->next;
  } while (unlikely(pqp->prio >= p->prio));

  /* Insertion on prev.*/
  p->next       = pqp;
  p->prev       = pqp->prev;
  p->prev->next = p;
  pqp->prev     = p;

  return p;
}
```

life.augmented

Likely and Unlikely conditions #2

Some examples:

```
void test11(void) {

  if (var1 > 0) {
    var1++;
    foo6();
  }
}

void test12(void) {

  if (likely(var1 > 0)) {
    var1++;
    foo6();
  }
}

void test13(void) {

  if (unlikely(var1 > 0)) {
    var1++;
    foo6();
  }
}
```

```
8000540 <test11>:
8000540:     4a05         ldr          r2, [pc, #20]
8000542:     b508         push         {r3, lr}
8000544:     6813         ldr          r3, [r2, #0]
8000546:     2b00         cmp          r3, #0
8000548:     dc00         bgt.n        800054c <test11+0xc> <- note
800054a:     bd08         pop          {r3, pc}
800054c:     3301         adds         r3, #1
800054e:     6013         str          r3, [r2, #0]
8000550:     f7ff ffee    bl           8000530 <foo6>
8000554:     bd08         pop          {r3, pc}

8000560 <test12>:
8000562:     b508         push         {r3, lr}
8000564:     6813         ldr          r3, [r2, #0]
8000566:     2b00         cmp          r3, #0
8000568:     dd03         ble.n        8000572 <test12+0x12> <- note
800056a:     3301         adds         r3, #1
800056c:     6013         str          r3, [r2, #0]
800056e:     f7ff ffdf    bl           8000530 <foo6>
8000572:     bd08         pop          {r3, pc}

8000580 <test13>:
8000580:     4a05         ldr          r2, [pc, #20]
8000582:     b508         push         {r3, lr}
8000584:     6813         ldr          r3, [r2, #0]
8000586:     2b00         cmp          r3, #0
8000588:     dc00         bgt.n        800058c <test13+0xc> <- note
800058a:     bd08         pop          {r3, pc}
800058c:     3301         adds         r3, #1
800058e:     6013         str          r3, [r2, #0]
8000590:     f7ff ffce    bl           8000530 <foo6>
8000594:     bd08         pop          {r3, pc}
```

life.augmented

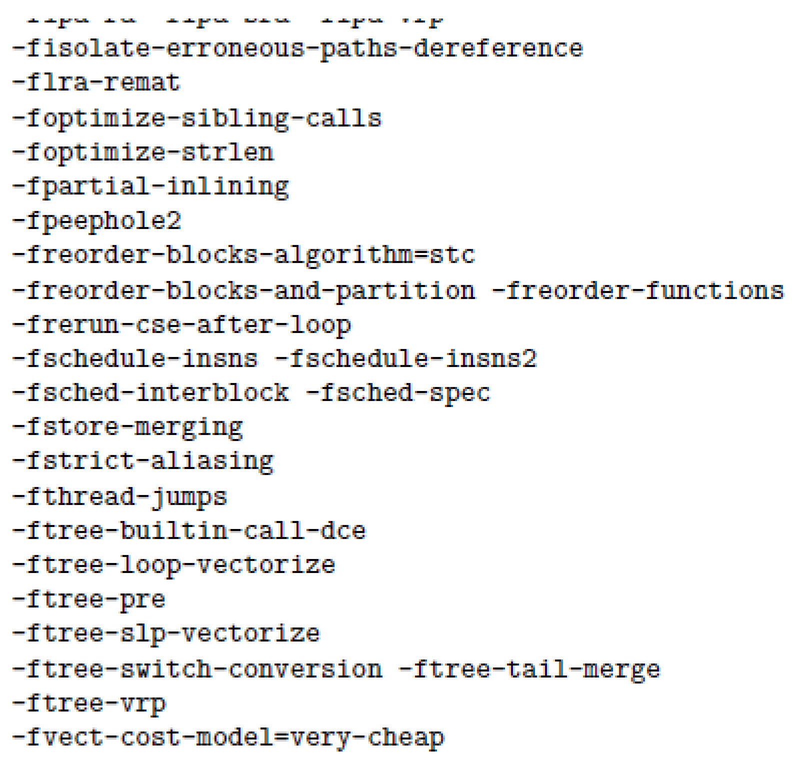
Project settings optimizations

life.augmented

Importance of setting up a project correctly

Compilers have a lot of options that can be used to optimize the final code, lets see some options available on GCC (and CLANG):

- Global optimization level: -O0, -Og, -Os, -O1, -O2, -O3 (also –Oz for CLANG).
- Functions alignment: -falign-functions=32
  - Important for MCUs with caches or flash prefetch buffers.
- Correct libraries selection: --specs=nano.specs
  - Make sure to use compact libraries when more types are available.



```
-fisolate-erroneous-paths-dereference
-flra-remat
-foptimize-sibling-calls
-foptimize-strlen
-fpartial-inlining
-fpeephole2
-freorder-blocks-algorithm=stc
-freorder-blocks-and-partition -freorder-functions
-frerun-cse-after-loop
-fschedule-insns -fschedule-insns2
-fsched-interblock -fsched-spec
-fstore-merging
-fstrict-aliasing
-fthread-jumps
-ftree-builtin-call-dce
-ftree-loop-vectorize
-ftree-pre
-ftree-slp-vectorize
-ftree-switch-conversion -ftree-tail-merge
-ftree-vrp
-fvect-cost-model=very-cheap
```

life.augmented

Use of memories

Modern MCUs can have multiple RAM memory types and banks, for example Caches, DTCM, TCM, etc. Also often multiple banks are available, on STM32 those are named SRAM1, SRAM2 etc.

What is the correct way to use those? We need to design a correct memory layout for our application, this is very important for our code performance.

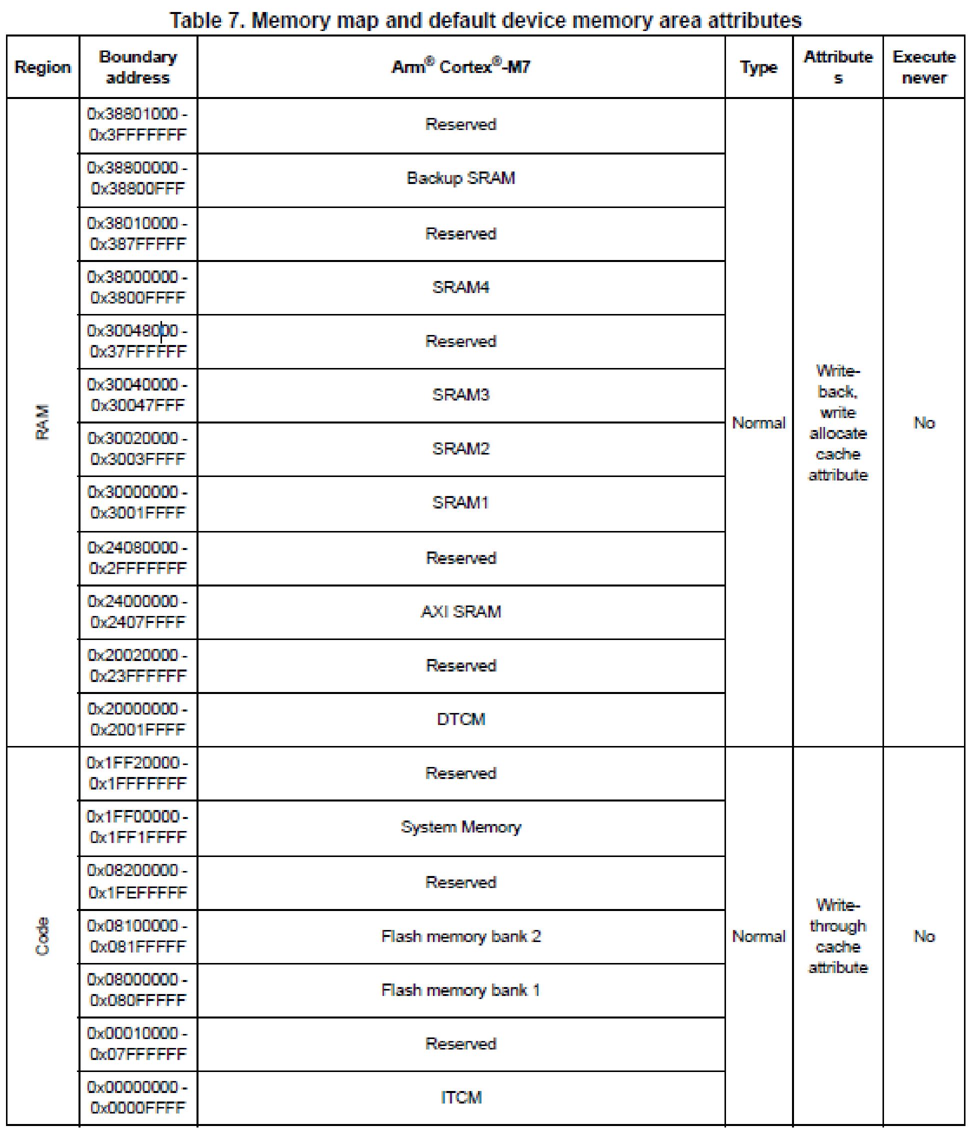


Table 7. Memory map and default device memory area attributes

Region	Boundary address	Arm® Cortex®-M7	Type	Attributes	Execute never
RAM	0x38801000 - 0x3FFFFFFF	Reserved	Normal	Write-back, write allocate cache attribute	No
	0x38800000 - 0x38800FFF	Backup SRAM			
	0x38010000 - 0x387FFFFF	Reserved			
	0x38000000 - 0x3800FFFF	SRAM4			
	0x30048000 - 0x37FFFFFF	Reserved			
	0x30040000 - 0x30047FFF	SRAM3			
	0x30020000 - 0x3003FFFF	SRAM2			
	0x30000000 - 0x3001FFFF	SRAM1			
	0x24080000 - 0x2FFFFFFF	Reserved			
	0x24000000 - 0x2407FFFF	AXI SRAM			
	0x20020000 - 0x23FFFFFF	Reserved			
	0x20000000 - 0x2001FFFF	DTCM			
Code	0x1FF20000 - 0x1FFFFFFF	Reserved	Normal	Write-through cache attribute	No
	0x1FF00000 - 0x1FF1FFFF	System Memory			
	0x08200000 - 0x1FEFFFFF	Reserved			
	0x08100000 - 0x081FFFFF	Flash memory bank 2			
	0x08000000 - 0x080FFFFF	Flash memory bank 1			
	0x00010000 - 0x07FFFFFF	Reserved			
	0x00000000 - 0x0000FFFF	ITCM			

life.augmented

Use of DTCM

DTCM is a data SRAM very close to a CPU core, it can have advantages and limitations, for example:

Pros:

- Very close to the core, DTCMs have very often zero wait states access, fastest data memory in the system,

Cons:

- Limited in size, DTCMs are usually very small 8..64kB.
- Accessibility limitations, ITCMs can be accessed only by the core owning it. Access from other cores or other bus masters (DMAs) is very slow or not possible at all, carefully read your device documentation.

Right use for a DTCM:

- MSP area (Main Stack in Cortex-M), this will ensure lowest latency and jitter for IRQ servicing.
- PSP area (Process Stack in Cortex.M), this will make sure that C automatic variable are accessed efficiently.
- Task Stacks for your RTOS, this will make sure that C automatic variables private to tasks are accessed efficiently.
- Very critical data structures, for example, if you have a small set of variables accessed by your critical control loop then DTCM is the right place.

Wrong use for a DTCM:

- DMA-accessible buffers. On Cortex-M external access to DTCM incurs in a high number of extra cycles and could be not possible at all depending on your device implementation.
- Non-critical data, DTCMs are small, use the resource wisely.
- Code, use ITCM instead.

life.augmented

Use of ITCM

ITCM is a code SRAM very close to a CPU core, it can have advantages and limitations, for example:

Pros:

- Very close to the core, ITCMs have very often zero wait states access, fastest code memory in the system,

Cons:

- Limited in size, ITCMs are usually very small 8..64kB.
- Accessibility limitations, DTCMs can be accessed only by the core owning it. Access from other cores or other bus masters (DMAs) is very slow or not possible at all, carefully read your device documentation.

Right use for an ITCM:

- Cortex-M vectors table, putting the vector table in ITCM makes sure that IRQs are serviced with lowest latency and jitter.
- Critical ISRs code.
- Critical tasks code (if using an RTOS).
- Critical functions.
- In general, critical control loops.

Wrong use for an ITCM:

- Code shared with other cores.

Use of System SRAMs

System SRAMs are the largest SRAM resources in your MCUs:

Pros:

- Good performance but not as good as TCMs (if TCMs are present).
- Equidistant from all bus masters (cores, DMAs, complex peripherals).
- Often segmented in multiple banks (separate bus slaves) allowing concurrent access by multiple bus masters.

Cons:

- Not as good as DTCM for data and ITCM for code.

Right use for System SRAMs:

- Basically, anything is not fitting in TCMs should be placed in System SRAMs.

life.augmented

Use of multiple System SRAMs and Caches

Why there are multiple SRAM blocks in my MCU?

Multiple SRAMs are often placed sequentially so you can access the whole SRAM as a single contiguous SRAM area but is this the right thing to do? Most likely not, if you use other bust masters such as DMAs, Ethernet controllers, etc. then you want to optimize buffers allocation in memory master, avoid access conflicts.

Few recommendations:

- Place your DMA buffers in their own SRAM bank, make the bank non-cacheable if your MCU has also a data cache.
  - This makes sure that DMAs and CPU do not conflict very often.
  - Simplifies cache management.
- Make your Ethernet descriptors and buffers in their own SRAM bank and make the bank non-cacheable.
  - Ethernet can be very a traffic-intensive bus master, you don’t want it to conflict with your CPU accesses.
- Data structures and buffers accessed by multiple masters and CPUs are best placed in a non-cacheable SRAM, the cache coherency is not handled on Cortex-M and SW cache management overhead is heavier than just making shared data non-cacheable. Use the Cortex-M MPU to declare non-cacheable regions.t
- When designing your application memory usage make sure to have fully understood the device bus architecture, some devices are really tricky, see the STM32H7xx for example. Devices with multiple cores are even more complex.

life.augmented

Final suggestions

life.augmented

Few random suggestions

We need to make sure that we are trying to optimize the right thing and verify that we are doing it right, few suggestions.

- Identify critical code and focus your optimization efforts there.
- Measure your code often:
  - Code size.
  - Execution speed.
  - Memory usage.
- Make sure to check for speed for improvements under ideal measurement conditions:
  - Measure under the final compiler optimization options.
  - Run benchmarks on a zero-wait-states memory or you can experience measurement artifacts.
- Local optimizations are good, but a good SW architecture is even more important, design for efficiency, use the right data structures. There are early tradeoffs to be done.
- Allocate critical code and data on fast memories like DTCMs/ITCMs.
  - Understand your MCU architecture, not all memories are equal.
  - There are configurations affecting performance.

life.augmented

Topics not yet covered

There are other optimization-related topics that should be covered:

- Choosing the right C types for your variables.
- Assigning data structures to correct memories on NUMA devices.
- Improving your ISRs.
- Use OOP, language does not matter.
- Pass pointers or references to structures as parameters, not individual variables.
- Enforce alignment of functions and loops.
- Importance of instrumenting code for easy and repeatable benchmarking.

life.augmented

Thank you

life.augmented