Optimizing C For Microcontrollers

Khem Raj, Comcast

Embedded Linux Conference & IOT summit - Portland OR
Agenda

- Introduction
- Knowing the Tools
- Data Types and sizes
- Variable and Function Types
- Loops
- Low Level Assembly
- RAM optimizations
- Summary

Meanwhile you are welcome to suggest more use-cases & solutions!
Knowing Tools

- Toolchains
  - many vendors e.g. GNU GCC, IAR system, ARM, ...
  - Each compiler has its own characteristics
    - Read through what compilers have to offer.
Knowing Tools - Compiler Switches

- **Code performance**
  - -O2/-O3, -Ofast

- **Code Size**
  - -Os

- **Debuggable code**
  - Og

- **Zephyr Codesize**
  - hello_world

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Code(bytes)</th>
<th>Data</th>
<th>BSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Os</td>
<td>6094</td>
<td>200</td>
<td>3648</td>
</tr>
<tr>
<td>O1</td>
<td>6568</td>
<td>200</td>
<td>3648</td>
</tr>
<tr>
<td>O2</td>
<td>6672</td>
<td>200</td>
<td>3648</td>
</tr>
<tr>
<td>O3/Ofast</td>
<td>7068</td>
<td>200</td>
<td>3648</td>
</tr>
<tr>
<td>Og</td>
<td>6748</td>
<td>200</td>
<td>3648</td>
</tr>
</tbody>
</table>
OUTPUT_FORMAT("elf32-littlearm", "elf32-bigarm", "elf32-littlearm")

MEMORY
{
    FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 512*1K
    SRAM (wx) : ORIGIN = 0x20000000, LENGTH = 96 * 1K
...
}

SECTIONS
{
    .text :
    {
        ./crt0.o(.text*)
        *(.text*)
        *(.strings)
        ....
        *(.init)
        *(.fini)
        _etext = . ;

    . = ALIGN(4);
    } > FLASH
    .data : AT (ADDR (.text) + SIZEOF (.text))
    {
        . = ALIGN(4);
        __data = . ;
        *(.data)
        *(.data*)
        *(.rodata)
        *(.rodata*)
        __edata = . ;
    } > RAM
    . = ALIGN(4);
    .bss SIZEOF(.data) + ADDR(.data) :
    {
        _bss_start = . ;
        *(.bss)
        *(COMMON)
        _end = . ;
    } > RAM
    __data_load_start = LOADADDR(.data);
    __data_load_end = __data_load_start + SIZEOF(.data);
Linker Map

-Wl,-Map=zephyr.map

Archive member included to satisfy reference by file (symbol)

drivers/built-in.o (--whole-archive)
...
kernel/lib.a(device.o) drivers/built-in.o (device_get_binding)
kernel/lib.a(errno.o) lib/built-in.o (_get_errno)
...

Allocating common symbols

Common symbol       size              file

x                       0x4               src/built-in.o
_handling_timeout       0x4               kernel/lib.a(sys_clock.o)
...

Discarded input sections

.text     0xffffffffffffffff00000000 0x0 isr_tables.o
.data     0xffffffffffffffff00000000 0x0 isr_tables.o
.bss      0xffffffffffffffff00000000 0x0 isr_tables.o

Memory Configuration

<table>
<thead>
<tr>
<th>Name</th>
<th>Origin</th>
<th>Length</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLASH</td>
<td>0xffffffff00000000</td>
<td>0x0000000000040000</td>
<td>xR</td>
</tr>
<tr>
<td>SRAM</td>
<td>0xffffffff00000000</td>
<td>0x0000000000001000</td>
<td>xW</td>
</tr>
<tr>
<td></td>
<td>0xffffffff00000000</td>
<td>0x0000000000000000</td>
<td>0xffffff00000000</td>
</tr>
</tbody>
</table>

Linker script and memory map

LOAD isr_tables.o
START GROUP
LOAD src/built-in.o
LOAD libzephyr.a
LOAD kernel/lib.a
LOAD ./arch/arm/core/offsets/offsets.o
END GROUP
LOAD
/opt/zephyr-sdk/sysroots/armv5-zephyr-eabi/usr/lib/arm-zephyr-eabi/6.2.0
/armv7-m/libgcc.a

0x0000000000000000 _image_rom_start = 0x0

text     0xffffffffffffffff00000000 0x131a
0xffffffffffffffff00000000 = 0x0
...
Binutils Tools

- **Objdump**
  - Disassemble object files

  ```
  objdump -dS zephyr.elf
  ```

- **Size**
  - Dump size information of ELF file

  ```
  size zephyr.elf
  text  data  bss  dec  hex  filename
  6118  200  3664  9982  26fe  zephyr.elf
  ```

- **elfutils**

  ```
  readelf -e zephyr.elf
  ELF Header:
  Magic:    7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           ARM
  ...
  Program Headers:
  Type Offset VirtAddr PhysAddr FileSiz MemSiz Flg Align
  LOAD  0x0000b4 0x00000000 0x00000000 0x0183c 0x0183c  RWE 0x4
  LOAD  0x0018f0 0x02000000 0x000183c 0x00074 0x00074  RW  0x4
  LOAD  0x001968 0x00000000 0x02000078 0x00000 0x00e50  RW  0x8
  LOAD  0x001964 0x00000000 0x02010000 0x00000 0x00000  RW  0x4
  ...
  Section to Segment mapping:
  Segment Sections...
  00  text devconfig rodata
  01  datas initlevel
  02  bss noinit
  03
Variables

- **Size**
  - Use local variables representable in processor WORD
    - Smaller locals can result in increased code size
  - Using variables > Processor WORD
    - Extra Load/Store, might not increase code size but degrade performance

- **Globals**
  - Compilers have to reload globals across function calls
  - Global pointers, also means the data they point to is reloaded
  - Use locals to avoid heavy accesses to globals if required
Data Types

Int and short int

```c
int foo(int x, int y)
{
    return x + y;
}
```

```assembly
foo:
    add    r0, r0, r1
    bx     lr
```

```c
short foo(short x, short y)
{
    return x + y;
}
```

```assembly
foo:
    add    r0, r0, r1
    sxth   r0, r0
    bx     lr
```
Slow and fast integers

- c99 allows “fast” and “least” integer type
  - Let compiler decide on the size
    - Fixed width X - uintX_t
    - Minimum width X - uint_leastX_t
      - Compact size
    - Fastest width X - uint_fastX_t
      - Faster execution

- Check your compilers for C standard support (needs c99)
- If using RTOS check if they provide libc
  - E.g. with Zephyr it was trying to use stdint.h from bundled libc
Portable Datatypes

- Use `uint{8,16,32,64}_t`
  - Defined in `inttypes.h`
- Avoid effects of changing size of `int` type across different processors
- Portable code
- Ensure compiler supports C99
‘const’ qualifier for variables and function parameters

- Const qualifier provides important hint to compiler
  - Data is not modified (Read-only)
- Conveys more information to reader about function from its prototype
- Compiler would be able to issue diagnostics if subsequent change to function modifies data
- Hint could enable compiler to optimize code of calling function
- Use const variable can better debugging
  - Can be held in RAM so watch out if you have smaller RAM
  - If stored in ROM, accessed using indexed addressing which is slower than immediate addressing
`const` qualifier

```c
const uint8_t a = 3;
const uint8_t b = 4;

uint8_t foo(uint8_t i)
{
    i *= a + b;
    return i;
}
```

```assembly
foo:
    ldr r3, .L3
    ldr r2, .L3+4
    ldrb r3, [r3] @ zero_extendqisi2
    ldrb r2, [r2] @ zero_extendqisi2
    add r3, r3, r2
    muls r0, r3, r0
    uxtb r0, r0
    bx lr
.L4:
    .align 2
.L3:
    .word .LANCHOR0
    .word .LANCHOR1
    .size foo, -.foo
```
Const volatile variables

- Can we have a const volatile variable?
- Yes
- Can you think of an example?
- Hardware status Registers
Global variables

extern int x;
extern void bar();

int foo(int y) {
    x++;
    if (y)
        x *= 2;
    else
        x *= 3;
    bar();
    return x;
}

foo:
    push {r4, lr}  @
    ldr  r4, .L9 @ tmp116,
    ldr  r3, [r4]  @ x, x
    adds r2, r3, #1  @ _4, x,
    lsls r3, r2, #1  @ tmp129, _4,
    cbz  r0, .L6 @ y,
    .L8:
        str  r3, [r4]  @ tmp124, x
        b1   bar  @
        ldr  r0, [r4]  @, x
        pop  {r4, pc}  @
    .L6:
        add  r3, r3, r2  @ tmp124, _4
        b    .L8  @
    .L10:
        .align 2
    .L9:
        .word  x
        .size foo, .-foo
Global Vs Local

Global

```c
int x;
void main(void)
{
    x = 0xDE;
    printk("X = %d\n", x);
}
```

local

```c
void main(void)
{
    int x;
    x = 0xDE;
    printk("X = %d\n", x);
}
```
Static Variable/Functions

- **Static Variables**
  - Persists state across functions in same compilation unit
  - Limit the visibility to compilation unit
  - Spatial locality during link time
    - Can use common base for pointer accesses

- **Static Functions**
  - Only called by functions in same compilation unit
  - Location is known during compilation (shorted jump sequence)
  - Inlining optimizations
  - Debugging
Volatile variable

● A value can change outside the program
  ○ Via ISR
  ○ Memory mapped peripherals

● Compiler does not optimize volatile variables
  ○ Some compilers offer non standard extensions
Array subscript Vs Pointer Access

**Subscript**

```c
int a[5] = {1, 11, 111, 1111, 11111};

int foo(void)
{
    int i;
    int res = 0;
    for (i = 0; i < 5; i++)
        res += a[i];
    return res;
}
```

**Pointer to array**

```c
int a[5] = {1, 11, 111, 1111, 11111};

int foo(void)
{
    int *p;
    int i;
    int res = 0;
    for (p = a, i = 0; i < 5; i++, p++)
        res += *p;
    return res;
}
```
Loops (Increment Vs Decrement)

```c
void main(void)
{
    int x = 0;
    do {
        printf("X = %d\n", x);
        x++;
    } while (x < 100);
}
```

```c
main:
push {r3, r4, r5, lr} @
movs r4, #0 @ x,
ldr r5, .L5 @ tmp112,
.L3:
mov r1, r4 @, x
mov r0, r5 @, tmp112
adds r4, r4, #1 @ x, x,
b1 printk @
cmp r4, #100 @ x,
   bne .L3 @,
pop {r3, r4, r5, pc} @
.L6:
   .align 2
.L5:
   .word .LC0
   .size main, .-main
```

```c
void main(void)
{
    int x = 100;
    do {
        printf("X = %d\n", x);
        x--;
    } while (x);
}
```

```c
main:
push {r3, r4, r5, lr} @
movs r4, #100 @ x,
ldr r5, .L5 @ tmp112,
.L3:
mov r1, r4 @, x
mov r0, r5 @, tmp112
b1 printk @
subs r4, r4, #1 @ x, x,
bne .L3 @,
pop {r3, r4, r5, pc} @
.L6:
   .align 2
.L5:
   .word .LC0
   .size main, .-main
```
Loops ( post Vs Pre Decrement )

```c
unsigned int x = 10;
do {
    if (--x) {
        printk("X = %d\n", x);
    } else {
        printk("X = %d\n", x);
        x = 10;
    }
} while (1);
```

```c
unsigned int x = 9;
do {
    if (x--) {
        printk("X = %d\n", x);
    } else {
        printk("X = %d\n", x);
        x = 9;
    }
} while (1);
```

```
main:
    push {r3, r4, r5, lr}
    movs r4, #10
    ldr r5, .L6

.L3:
    subs r4, r4, #1
    mov r1, r4
    mov r0, r5
    beq .L4
    bl printk
    b .L3

.L4:
    bl printk
    movs r4, #10
    b .L3
```

```
main:
push {r3, r4, r5, lr}
movs r4, #9
ldr r5, .L7

.L3:
    cbz r4, .L4

.L6:
    subs r4, r4, #1
    mov r1, r4
    mov r0, r5
    bl printk
    b .L3

.L4:
    mov r1, #-1
    mov r0, r5
    bl printk
    movs r4, #9
    b .L6
```
Function Parameters

- Read the ABI carefully
  - Depends on processor architectures
- ARM EABI makes 4 registers available for parameter passing
  - If params needs more than 4 registers, stack is used
  - Consider alignments while deciding on function argument sequence
Order of Function Parameters

R0 - R3 registers for parameter passing on ARM

```c
void __attribute__((noinline))
foo(int a, long long b, int c)
{
  ...
}
```

```c
main:
    push {r0, r1, r2, lr}
    ldr  r3, .L6
    ldr  r1, .L6+4
    ldr  r3, [r3]
    str  r3, [sp]
    ldr  r3, .L6+8
    ldr  r0, [r1]
    ldrd r2, [r3]
    bl   foo
```

```c
void __attribute__((noinline))
foo(int a, int c, long long b)
{
  ...
}
```

```c
main:
    ldr  r3, .L6
    ldr  r1, .L6+4
    ldr  r0, .L6+8
    ldrd r2, [r3]
    ldr  r1, [r1]
    ldr  r0, [r0]
    b    foo
```
Inline Assembly

- Used for inserting processor specific instructions into C code
- Built-in intrinsics
  - Fixed point support
  - Special instructions
- GCC Inline Assembly
- Other compilers documents their syntaxes too
- WARNING !!
  - C standards do not specify asm semantics
  - One of major source of incompatibility between compilers
Optimizing for DRAM

- Use smaller data-type
  - short integers instead of ints
- Re-organize data structure elements to eliminate padding
  - Use packed structs
- Life of local variables
  - alloca() does not release memory until return
- Use Merge constants compiler optimizations
- Check stack and heap usage
  - Adjust limits if unused
Help the compiler out!

- Compiler does not have magic crystal ball
  - It has to make worst case assumptions
    - Pointer aliasing
    - Global data across functions is not immutable
- Do-while is better than for loop when loop is always executed
  - Loop termination test can be optimized out
  - H/W Branch Predictor helps but still we can avoid lousy programming
- Use compiler provided annotations
  - Function attributes
  - Variable attributes
  - Pragmas
- Use available compiler intrinsic functions
Optimizing your code

- Stay away from “Debug mode” and “Release Mode” optimization settings
  - Same optimization level for development and deployment
- Find details about system e.g. architecture, buses, memory, flash
- Profile code before optimizing anything
  - I have a hunch !!
    - You are wrong
  - Find “Top 10” functions to optimize
- Defer to Tools as much as possible
  - Don’t fight the tools, help them
    - Often there is reason for their behaviour
- Avoid assembly