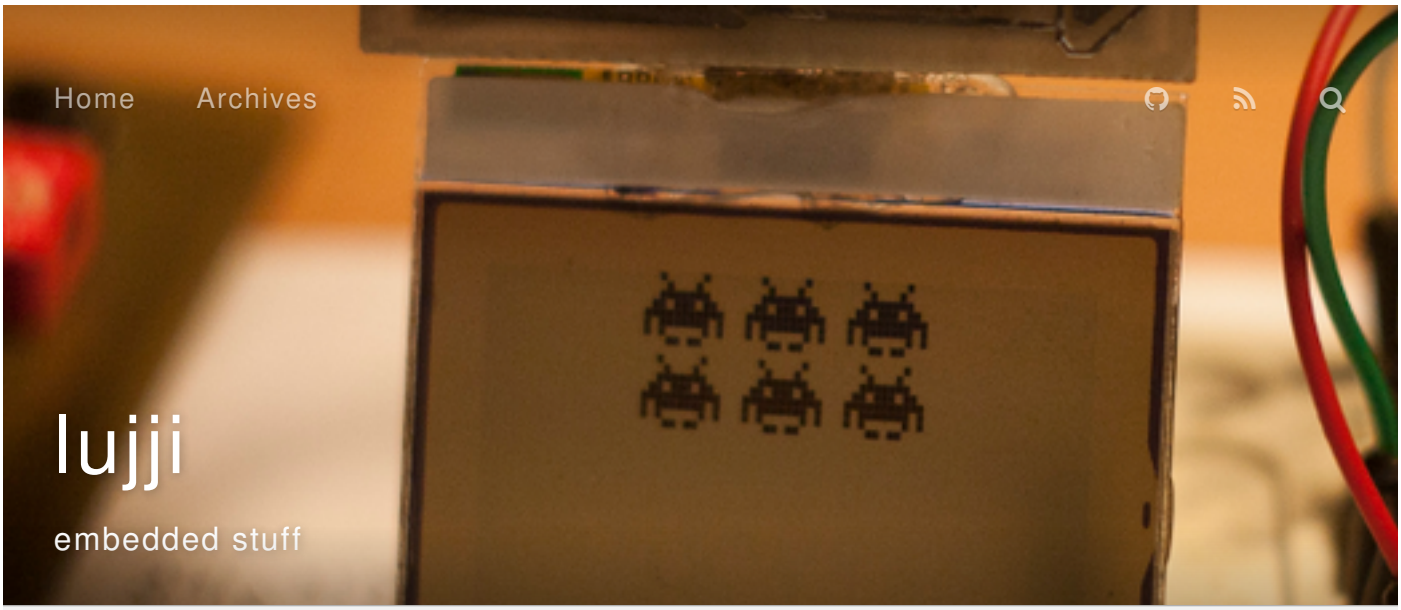


2017-08-01

Mixing C and assembly on STM8

This guide discusses how we should (and should not) speed up our code with inline assembly and explains how to write separate assembly routines that can be used within C.

Inline assembly and optimizations

Let's take a simple code snippet for toggling an IO pin:

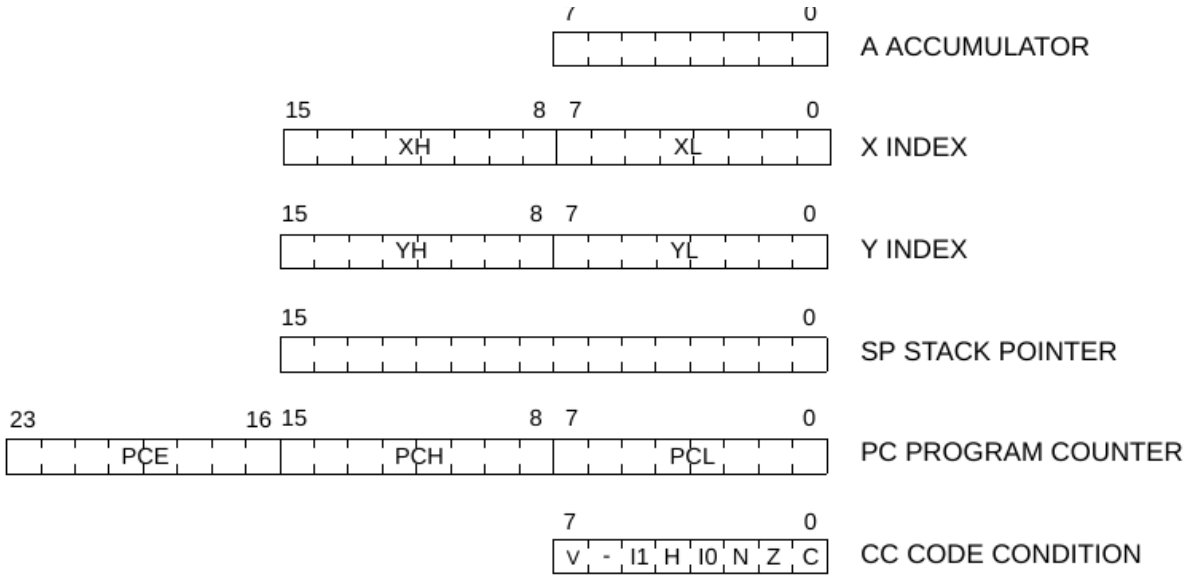
```
1  PD_ODR ^= (1 << PIN4);
```

Now let's look at the assembly instructions generated by SDCC:

```
1  ld  a, 0x500f
2  xor a, #0x10
3  ldw x, #0x500f
4  ld  (x), a
```

That's 4 instructions just to toggle a pin, I'm pretty sure we can do better than that.

First, let's familiarize ourselves with CPU registers: we have an 8-bit accumulator register A and two 16-bit registers X and Y. The stack pointer is 16-bit wide and the program counter has 24 bits, but we're only using the lower 16 bits on processors with <64k of flash.



CPU registers

You can find all instructions and other CPU-related stuff is in the programming manual. STM8 has 3 dedicated instructions that take only one cycle to execute: Bit Set (BSET), Bit Reset (BRES) and Bit Complement (BCPL). The last instruction is used to flip a single bit leaving other bits unchanged. We can use these instructions to control individual IO pins as fast as possible:

```
#define PIND4_SET()     __asm__("bset 0x500f, #4")
#define PIND4_RESET()   __asm__("bres 0x500f, #4")
#define PIND4_TOGGLE()  __asm__("bcpl 0x500f, #4")
```

Another usage is clearing pending interrupt flags:

```
void tim4_isr() __interrupt(TIM4_ISR) {
    /* ... */
    __asm__("bres 0x5344, #0"); // TIM4_SR &= ~(1 << TIM4_SR_UIF)
}
```

To be honest I'm not a big fan of inline assembly - it makes code less readable and harder to maintain. In fact, these optimizations should have been made by the compiler in the first place. SDCC has a rule based pattern matching optimizer, which can be extended with our custom rules. We can use the following pattern that matches the example above:

```
// reg ^= (1 << 4) -> bcpl reg, #4
replace restart {
    ld a, %1
    xor a, #0x10
    ldw %2, #%1
```

```
    ld (%2), a
} by {
    bcpl %1, #4
} if notUsed('a')
```

Save this rule under 'extra.def' and compile with `--peep-file extra.def` option. Since I didn't find any better solution, I wrote a script that generates patterns for every single bit shift. You can find the rule as well as the python script on github.

Accessing C symbols from assembly

SDCC generates symbol names for C variables with an underscore - knowing that makes it possible to access these variables from assembly. Let's write a small function that increments a 16-bit variable `val`:

```
volatile uint16_t val = 0;

void inc_val() {
    __asm
        ldw x, _val
        incw x
        ldw _val, x
    __endasm;
}
```

There's a slight issue with this function, though: we're modifying a commonly used register X, which means that if some value was loaded before calling the function, it will be lost. The compiler does not know about this - it just places assembly instructions where we told it to. The proper way is to save the contents of the registers before altering them and restore them afterwards.

That being said, in our case saving registers is not really necessary. There are two calling conventions for assembly functions: caller saves and callee saves. The first one means that functions are allowed to modify registers as they please and function caller is responsible for saving and restoring context. The second one means that any register modified by the function must be restored by the function itself when the it returns.

According to the documentation, SDCC uses caller saves convention by default, which means that we can implement our functions without saving the context. But I would still prefer doing it the 'right way', since this would allow inlining the function without any consequences:

```
inline void inc_val() {
    __asm
```

```
        pushw x
        ldw x, _val
        incw x
        ldw _val, x
        popw x
    __endasm;
}
```

Separate assembly functions

OK, but what if we wanted to build our own function with ~~blackjack~~ parameters and return value? Well, for the return value SDCC seems to follow this convention: accumulator is used for storing 8-bit return value, index register X for 16-bit values, and both X and Y are used if we need to return a 32-bit value. Things are a bit more complicated with function parameters, so it's better to explain this with an example. Let's implement a fast `memcpy` that would copy up to 255 bytes. First we declare a prototype with external linkage:

```
extern void fast_memcpy(uint8_t *dest, uint8_t *src, uint8_t len);
```

Next we create a file called util.s where we implement this function in assembly:

```
.module UTIL
.globl _fast_memcpy
.area CODE
_fast_memcpy:
    ldw x, (0x03, sp) ; dest
    ldw y, (0x05, sp) ; src
loop0$:
    tnz (0x07, sp)    ; if (len == 0)
    jreq loop0_end$
    ld a, (y)         ; loop body
    ld (x), a
    incw x
    incw y
    dec (0x07, sp)    ; len--
    jra loop0$
loop0_end$:
    ret
```

All right, let's figure out what's going on here. First of all we have `.globl` - that means we make a symbol accessible from the outside world, and `.area` - code section. Now for the function itself -

the first instruction is `ldw x, (0x03, sp)`. Here's how you read it: we get a value from the stack located at [SP + 3]. This value is then treated as a memory address, and the processor loads the value from that address into register X. Just like with pointers in C you can think of `ldw (x), y` as `*((uint16_t *) &x) = y`.

But what's the deal with those values - 0x03 and 0x05? When we call a function, we (unsurprisingly) issue a `call` instruction. The programming manual describes what the instruction does: it saves the high and low bytes of Program Counter (PC) register on the stack and loads PC with the destination address of the function being called. At the end of our function we issure a `ret` instruction which restores PC. Stack pointer decreases when you push something on the stack, so if we offset it by 1, we get the address of the last byte that was pushed on the stack (which is PCH), if we offset it by 2 we get PCL and if we offset it by 3 - bingo! We get the first argument that was passed to the function. Since the first two arguments are pointers, each of them will occupy 2 bytes on the stack. So the offset for the second argument would be 0x03 + 2 = 0x05.

The rest of the code is pretty much self-explanatory: we jump to `loop_end` if the third argument (len) is 0, otherwise we continue with the main loop which copies source byte into destination address, increments the pointers and decrements len. The last thing is to assemble our source:

```
1  sdasstm8 -lo util.s
```

Options `-lo` tell the assembler to create list and object files respectively. That's it, now we can link `util.rel` with our program and call the assembly subroutine directly from C code.

As always, code is on github.

#assembly #sdcc #stm8 Comments Share

NEWER POSTS

Serial bootloader for STM8

OLDER POSTS

Executing code from RAM on STM8

Alan • 4 years ago

Hi Lujji! Very nice post! Thanks for sharing it! BR, Alan

1 • Reply • Share ›

tovis • 3 years ago

Well done! Big help for start. I'm still struggling on UART1, setup is OK, isr (minimal :) also working. Now I'm trying to connect using circular buffers, long time ago I have use that for MCS51 (not SDCC). But it was written in pure assembler, where I can put buffers on boundaries, which is quite effective when you calculate modulo for indexes. Have you seen some trick for absolute allocation?
I even try "__code __at (0x9ff0) char Id[5] = ”SDCC”;" aimed from SDCC manual, but it gives me "syntax error: token -> '__at'. Using .area and .org from assembler is hopeless :(

• Reply • Share ›

lujji Mod ➚ tovis • 3 years ago • edited

I don't think that "code" attribute is implemented for STM8 port. You can try `__at(0x9ff0) char id[5]` if you want to tell the compiler where to place the variable in program memory. But since you're talking about buffers, I assume that you'll be accessing them from RAM, not flash.

• Reply • Share ›

tovis ➚ lujji • 3 years ago • edited

How you know that, the "code" attribute does not implemented? "__at" without CODE or DATA is work, even for SRAM/DATA area - at least I do not have syntax errors. I can not see these values "DATA" in map file, all about 26 bytes have attributes (REL,CON) - what means for me that absolute data was not placed at marked places :(
By the way, after coding simple circular buffer and see what compiler done (in asm file), I have decided rewrite hole thing in assembler. I love stm8 addressing modes, a specially ([longptr.w],X) "Long Pointer Indirect Long Indexed addressing mode" is amazing for me! Using asm files created by compiler, I have see that int return values are awaited in X register (I have not tried but pointers too), and type

article reminds me of my past. Now I'm trying to learn stm8 and effective programming of them in C language simultaneously. I hope Philipp Klaus Krause and Ben Shi (official STM8 SDCC maintainers) will read your articles and wil make sdcc more optimized in the very core.

Frankly speaking, I was thinking than sdcc is as bad as people say about it. Until one day I've tried one of the best, very expensive for commercial use "professional" IAR STM8 compiler and found out it's not that good either: With the best optimization possible

return a ^= (1 << 4);

was compiled into

0080D6 01 RRWA X, A
0080D7 A810 XOR A, #0x10
0080D9 02 RLWA X, A
0080DA 81 RET

So that's why I've never coded in C for small systems before. And that's why I love your blogs cause bare metal coding allows us at least not to waste resources to crappy libraries and wrappers.

Reply • Share ›

lujji Mod ➚ Анна Макарова • 3 years ago

I don't think sdcc is bad. In fact, I think it's great since it's the only compiler that allows me to work with stm8 under linux.

Reply • Share ›

ARCHIVES

RECENT POSTS

Serial bootloader for STM8
Mixing C and assembly on STM8
Executing code from RAM on STM8
Bare metal programming: STM8 (Part 2)
Bare metal programming: STM8

lujji at protonmail com