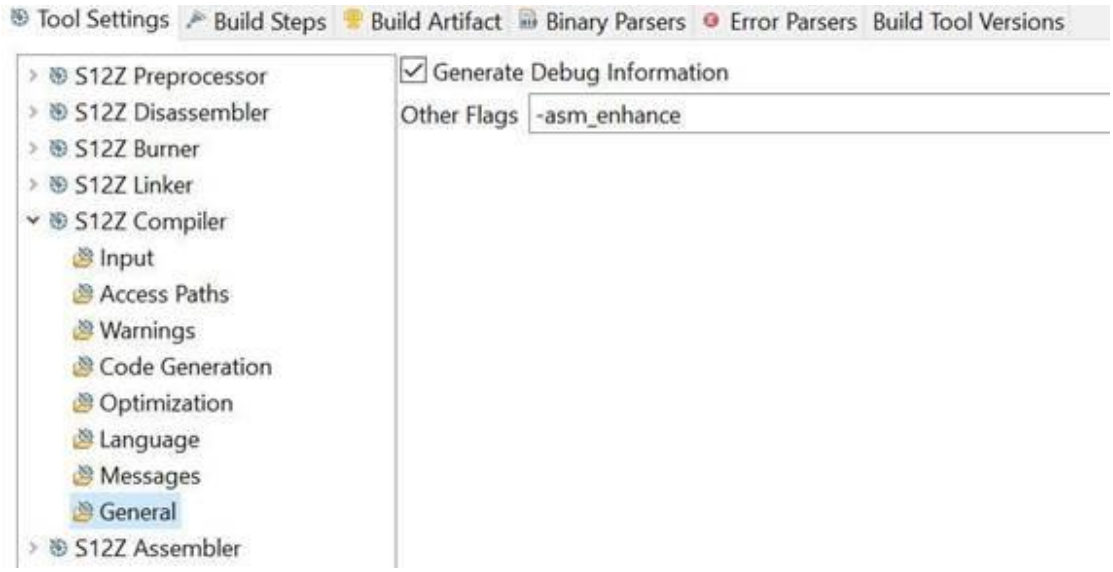
Command line option "-asm_enhance" Usage (CW for S12Z)

By Jennie Zhang

Recently we helped a customer resolve a question related with "asm_enhance"command line option usage. It's strange there is no documentation of this knowledge in help manual. After many discussions with development team, it gets clear to me. So I wrote an article on this topic for sharing.

" -asm_enhance" option is added in CodeWarrior10/CodeWarrior11 "compiler setting", "General", "Other Flags":



"-asm_enhance" command line option can be used for two distinct aspects:

- ***Enabling inline assembly data optimizations***
- ***Enabling stack effect computation through inline assembly code.***

These two aspects were all gathered under the same option because they both are referring to enhanced inline assembly – i.e. inline assembly code that gets handled almost as C code).

1. *Enabling inline assembly data optimizations*

The first aspect is related to optimizing the memory accesses to global variables inside inline assembly code, based on the selected memory model.
For example:

```
int global_var;
static void iasm_data_addr_opt() __attribute__((noinline)) {
      __asm {
            inc.w global_var
      }
}
```

In this situation, without the “-asm_enhance” option, the access to global_var will be handled as being 24-bit address regardless the selected memory model, resulting in wasting few bytes in encoding addresses that could fit in less memory.

9DFA001100 INC.W 4352

If the “-asm_enhance” option is used, the access to global_var is adjusted to the used memory model – for instance if small memory model is used, the following encoding is generated (which is shorter with 1 byte)

9D1100 INC.W 4352

When this option was added, we had some issues related to startup code (see starts12z.c) because of the combination of following facts:

- startup data is a table generated by the linker and is always placed on 24-bit addresses
- the startup code has parts written in inline assembly which are using the startup data global variable directly
- when the “-asm_enhance” option is used, the compiler will adjust the inline assembly accesses to the global variable startup data to the used memory model
- this results in wrong code (this could not even link), since the actual global variable has 24-bit address (by its placement in the linker)

In order to overcome this situation, we decided to define the startup data as “const” data if and only if the “-asm_enhance” option is used. When defining data as const, the compiler knows this will be placed on 24-bit addresses (i.e. .rodata), and will not optimize them for memory models.

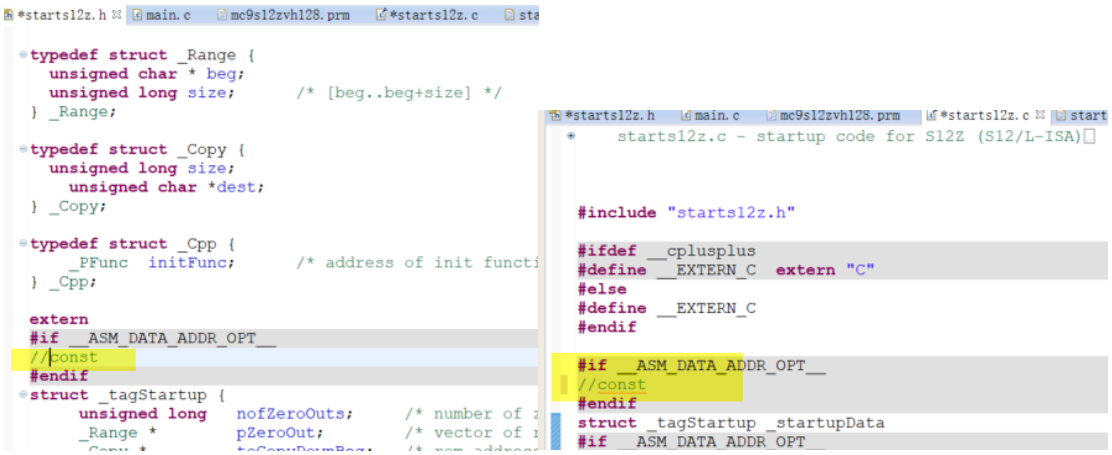
So, we decided whenever the “-asm_enhance” option is enabled, the compiler will implicitly define the “__ASM_DATA_ADDR_OPT__” macro, and we can conditionally define code using this macro.

This is the case for the startup data definition (defining it as const, if and only if the option is used).

We can test it with attached project.

This demo has “-asm_enhance” option added to compiler command line. So in starts12z.c, __ASM_DATA_ADDR_OPT__ is enabled and _startupData is defined as const.

To verify it, if you comment “const” in starts12z.c and starts12z.h



Build project again. You will see error. This verifies __ASM_DATA_ADDR_OPT__ is enabled by option “-asm_enhance”.

the const in for the _startupData is added when using “-asm_enhance” just to make sure the inline assembly code accessing the _startupData will not get optimized (with shorter addresses) – when the compiler sees that the accessed data is in ROM, the addressing mode optimization does not occur anymore. (if no “const”, the compiler will try to shorten the addresses in the code (see the code from below), but the address of _startupData in ROM will always be 24-bit and will not fit in the instruction)

```
static void DoZeroOut(void) {
      __asm {
            LD D6, _startupData.nofZeroOuts
            BEQ end                          /* nothing to do */
            LD X, _startupData.pZeroOut
      zeroOutLoop:
            LD Y,  (0,X)                     /* X points to the first range */
            LD D7, (3,X)                     /* D7 holds size */
      doZeroOut:
            CLR.b (Y+)
            DBNE D7, doZeroOut
            LEA X, (7,X)
            DBNE D6, zeroOutLoop
      end:
      }
}

static void DoCopyDown(void) {
      __asm {
            LD Y, _startupData.toCopyDownBeg
            BEQ end                          /* the pointer is NULL */
      nextItemLoop:
            LD D6, (Y+)                  /* load the size */
            BEQ end
            LD X, (Y+)                   /* load the destination */
      copyLoop:
            MOV.b (Y+), (X+)             /* copy the data */
            DBNE D6, copyLoop
            BRA nextItemLoop
      end:
      }
}
```

Related to the coloring of code in IDE, this is done by IDE, based on the defines and environment variables the IDE is aware of. It shows “#ifdef __ASM_DATA_ADDR_OPT” still gray, but since __ASM_DATA_ADDR_OPT__ is implicitly defined by the compiler when using a

certain command line option, the IDE does not have any information on this macro being defined, hence no appropriate context coloring.

2. *Enabling stack effect computation through inline assembly code*

The second aspect is related to considering the stack effect of inline assembly code.
As mentioned before, these two aspects don't have much in common, except they are referring to enhanced inline assembly code (since most of the time the inline assembly code is treated as black box).

This aspect is visible in the example below:

```
static void iasm_stack_effect() __attribute__((noinline)) {
	int local_var = global_var + 3;
	__asm {
		psh d6
	}
	global_var = local_var;
	__asm {
		pul d6
	}
}
```

In this case, if the option is not used, the compiler will not know that the stack is modified in inline assembly code, and will generate code for accessing "local_var" (which lives on stack) as if the stack does not change along the function code:
global_var = local_var;
1D601100 MOV.W (0,S),4352

If the option is enabled, the compiler will know that "psh d6" will increase the stack pointer and will adjust the access to "local_var" using the stack effect of the "psh" instruction (which basically moves the SP 4-byte further than the place the local variable was defined).
global_var = local_var;
1D641100 MOV.W (4,S),4352