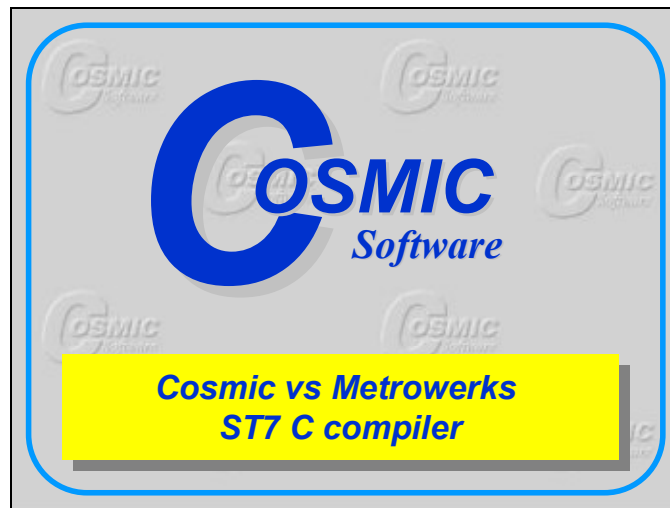


COSMIC SOFTWARE

Application Note N °65 – Cosmic vs MW ST7 compiler

Understanding the differences between Cosmic and Metrowerks ST7 C compilers



This technical article explains the main differences between Cosmic and Metrowerks ST7 C Compiler and is particularly intended for people porting an application from MW to Cosmic, as well as for people benchmarking the two compilers. This article is based on the latest compilers available as of August 2004.

Memory Models

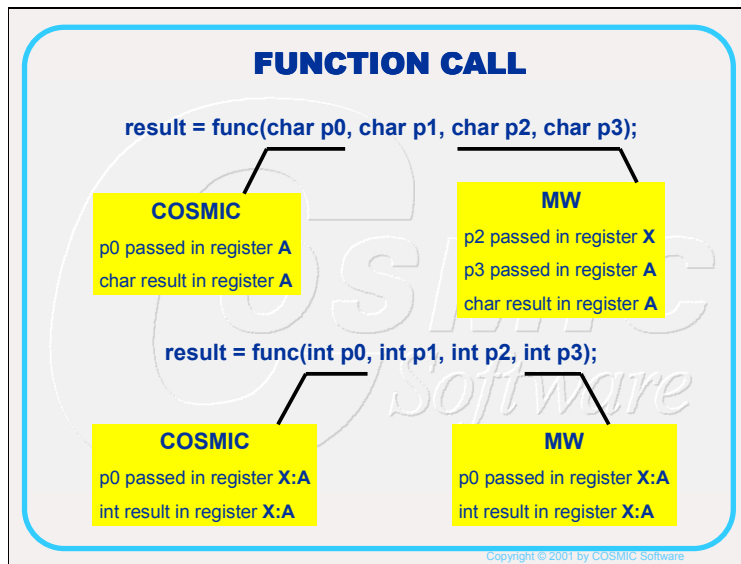
MEMORY MODELS			
MW			COSMIC
No STACK model			STACK models:
			Locals Globals
			SHORT (+mods) X S
			LARGE (+mods) X L
MEMORY models:			MEMORY models:
	Locals	Globals	
SMALL	S	S	Locals Globals
LARGE	S	L	COMPACT (+modc) S S
LARGE Extended	L	L	SHORT (+modm) S S
			SMALL (+modms) S L
			MEDIUM (+modmm) L S
			LARGE (+modml) L L

Copyright © 2001 by COSMIC Software

The main difference between the two compilers is in how memory models are managed (for an explanation of what a memory model is and how it affects your application see AN 64). Note first that MW does not offer “stack” memory models, that is, it is not possible to write reentrant or recursive applications with the MW compiler. Coming to “standard” (for the ST7) memory models, **it is important to note that the same name does not mean the same thing for the two compilers**: from the table above you can see that the SMALL model for MW is comparable to the COMPACT model for Cosmic (because they both put Locals and Globals in the short addressing range + 8 bit pointers), and that the LARGE model for Cosmic is comparable with the LARGE EXTENDED from MW.

The first implication of this is that if you make a quick benchmark by just selecting the memory model with the same name for the two compilers, MW will **look** consistently better, but you are comparing apples with oranges: for a fair comparison, you should choose memory models as explained above, or, even better, choose for each compiler the smaller model that will allow you to compile, **link** and **run** your application correctly (more on benchmarking in the last page of this document). On the same line, note that if you just take the default memory model for each compiler, the comparison won’t be any useful, as MW defaults to the smaller model, whereas Cosmic default to the more general (which is also the biggest). If you are porting an application, you should at first try to use the “comparable” memory model.

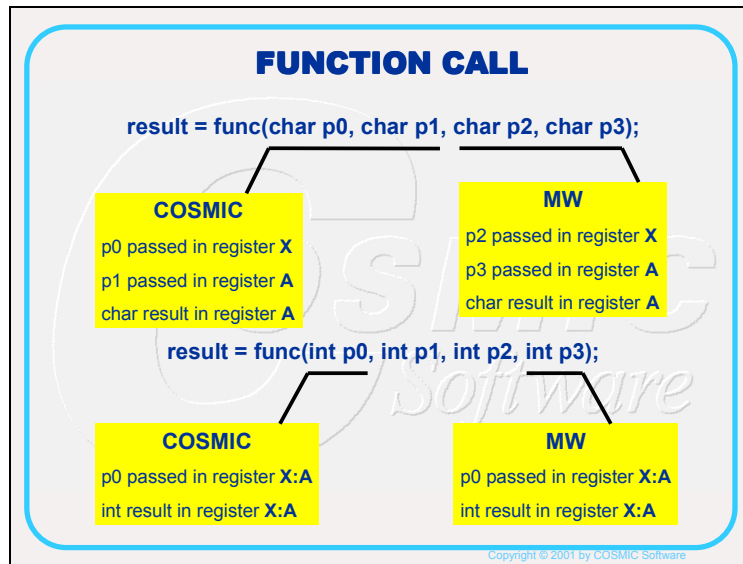
Function call and parameter passing (v4.4h and lower)



This information only apply for Cosmic compiler v4.4h and lower: from v4.5a onward please see next page.

The Cosmic and Metrowerks ST7 C compilers use different conventions for parameter passing: this is usually transparent to the user, but if you call assembler routines in your application (which, by the way, is a porting nightmare from many points of view), you will need to modify them to keep this into account. Understanding how parameters are passed can also be useful when fine tuning the application, as you might want to arrange your parameters in such a way to use registers as much as possible: for example, using a long as the first parameter with Cosmic will mean no parameters are in registers (because long is 4 bytes -> too big for registers), whereas doing the same with the last parameter for MW will result in the same problem. Bottom line is: if you care to optimize the order of your parameters, the optimal order is reversed for Cosmic and MW.

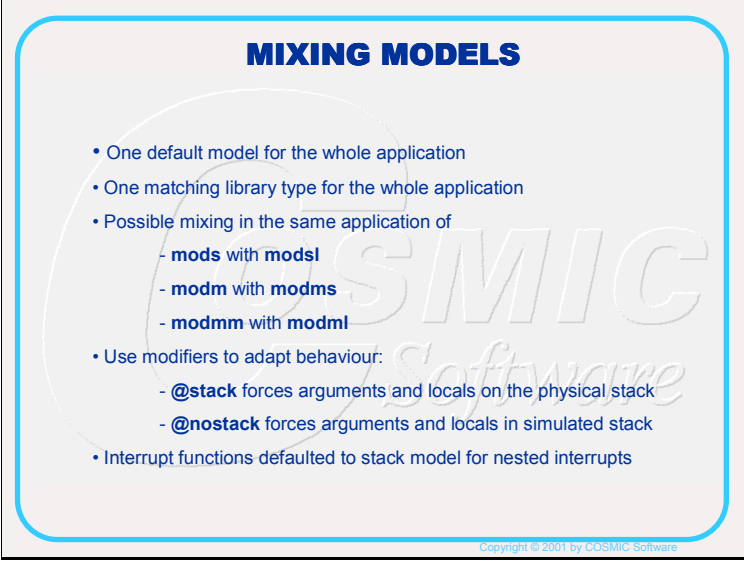
Function call and parameter passing (v4.5a and higher)



This information only apply for Cosmic compiler v4.5a and higher: for v4.4h and earlier please see the previous page.

The Cosmic and Metrowerks ST7 C compilers use different conventions for parameter passing: this is usually transparent to the user, but if you call assembler routines in your application (which, by the way, is a porting nightmare from many points of view), you will need to modify them to keep this into account. Understanding how parameters are passed can also be useful when fine tuning the application, as you might want to arrange your parameters in such a way to use registers as much as possible: for example, using a long as the first parameter with Cosmic will mean no parameters are in registers (because long is 4 bytes -> too big for registers), whereas doing the same with the last parameter for MW will result in the same problem. Bottom line is: if you care to optimize the order of your parameters, the optimal order is reversed for Cosmic and MW.

Mixing Memory Models



MIXING MODELS

- One default model for the whole application
- One matching library type for the whole application
- Possible mixing in the same application of
 - **mods** with **modsl**
 - **modm** with **modms**
 - **modmm** with **modml**
- Use modifiers to adapt behaviour:
 - **@stack** forces arguments and locals on the physical stack
 - **@nostack** forces arguments and locals in simulated stack
- Interrupt functions defaulted to stack model for nested interrupts

Copyright © 2001 by COSMIC Software

The Cosmic compiler allows for some mixing of different memory models on the same application, as explained in the slide above. This can be useful, for example, to decide which variables go to page zero and which go to the long range on a file by file basis, without having to touch the code to add `@tiny` and `@near` modifiers.

COSMIC SOFTWARE

Application Note N °65 – Cosmic vs MW ST7 compiler

Code allocation

CODE ALLOCATION

MW	COSMIC
<pre>#pragma CODE_SEG MY_ROM void func1(void) { ... } #pragma CODE_SEG DEFAULT void func2(void) { ... }</pre>	<pre>#pragma section (MY_ROM) void func1(void) { ... } #pragma section () void func2(void) { ... }</pre>

Copyright © 2001 by COSMIC Software

If your application uses named code segments (other than the default segment), the syntax to specify the segment name is slightly different for the two compilers, as shown above.

Constant allocation

CONSTANT ALLOCATION

MW	COSMIC
<pre>#pragma CONST_SEG MY_CONST const char table[] = { ... };</pre>	<pre>#pragma section const {MY_CONST} const char table[] = { ... }; .MY_CONST</pre>
	<pre>#pragma section const {} const int tsize = 10; .const</pre>
<p><i>+nocst forces constants to .text</i></p>	

Copyright © 2001 by COSMIC Software

On the same line of what happens for code, if your application uses named segments for constants (other than the default const segment), the syntax to specify the segment name is slightly different for the two compilers, as shown above. Note that the Cosmic compiler includes an option to store constants together with the code rather than in a separate segment.

DATA allocation

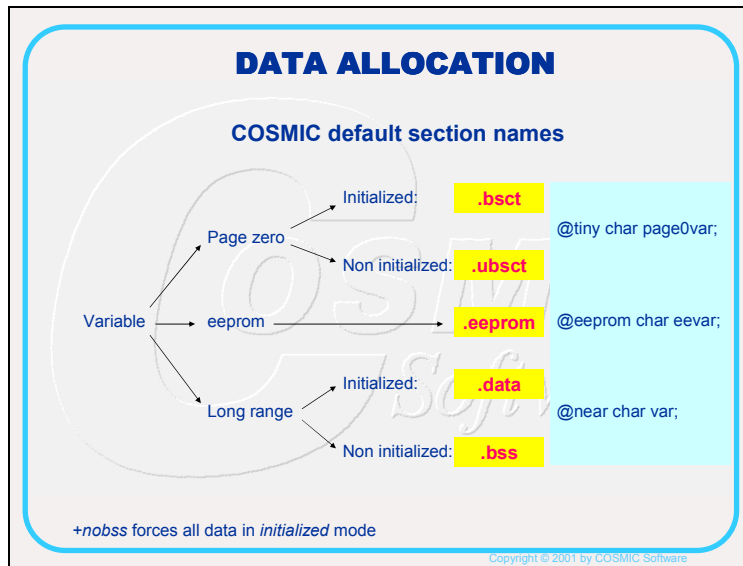
MW	COSMIC
Variables in zero page:	Variables in zero page:
<code>#pragma DATA_SEG SHORT MY_RAM0</code>	<code>#pragma section @tiny {MY_RAM0}</code>
<code>char page0var;</code>	<code>char page0var = 1; .MY_RAM0</code>
	<code>#pragma section @tiny [MY_BSS0]</code>
	<code>char page0uvar; .MY_BSS0</code>
Other variables:	Other variables:
<code>#pragma DATA_SEG MY_RAM</code>	<code>#pragma section @near {MY_RAM}</code>
<code>char var;</code>	<code>char var = 1; .MY_RAM</code>
	<code>#pragma section @near [MY_BSS]</code>
	<code>char uvar; .MY_BSS</code>

Copyright © 2001 by COSMIC Software

As with code and constants, declaring DATA segments requires a different syntax for the two compilers.

Note that, with the MW compiler, the only way to assign a variable to an addressing space (for example, short/long), is to use the syntax above whereas, with the Cosmic compiler, you can either use the syntax above (if you are porting and you want to minimize your work), or, more simply, you can use the @tiny (short) and @near (long) modifiers in the variable declaration (see next page for an example).

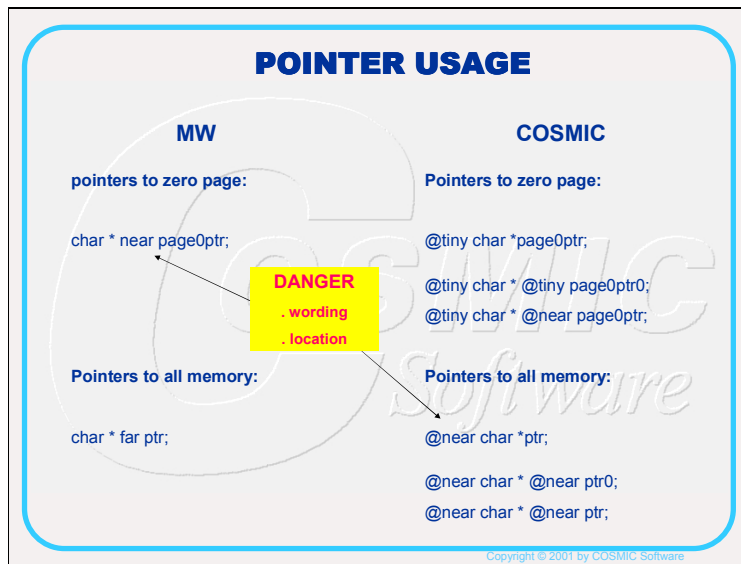
DATA allocation



This slide shows how the Cosmic compiler allocates data; using the proper modifier (@xxx, right column), will force the data in the desired addressing range (more specifically, in the default segment for that addressing range), without using any pragma.

Note that the Cosmic compiler (and libraries) fully support EEPROM variables, whereas EEPROM must be managed manually with the MW compiler.

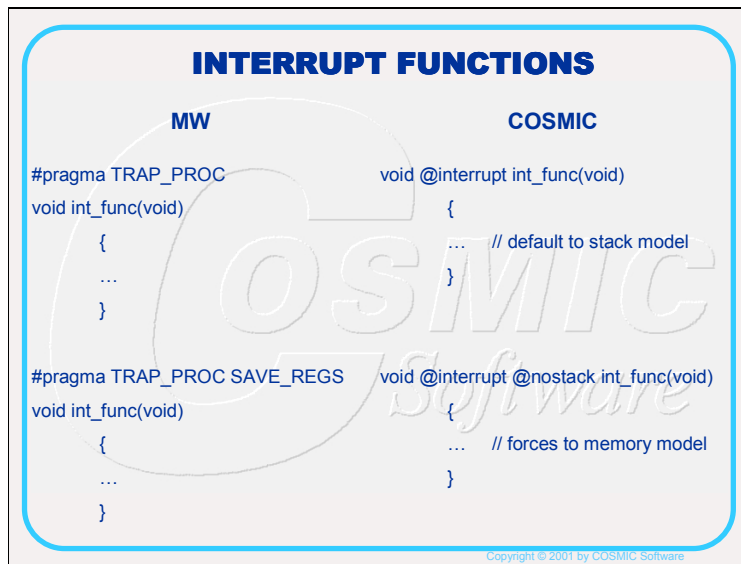
Pointer Usage



Qualified pointers (that is, pointers that do not default to the standard of the memory model being used) are tricky because they are managed in a completely different way in the two compilers. You first need to consider that the MW compiler uses the keywords “near” and “far” for the short (8 bit) and long (16 bit) addressing range respectively, whereas the Cosmic compiler uses the keywords @tiny and @near for the same. As you see, the keyword “near” means a different thing for the two compilers.

Besides this, the Cosmic compiler allows to specify the storage class for both the pointed object (before the *) and the pointer itself (after the *), whereas the MW compiler only allows to specify the pointed object (the storage class for the pointer itself can be specified via pragmas). Note that the position after the * symbol denoted the storage class for the pointed object in MW and the storage class for the pointer itself in Cosmic!

Interrupt functions



Interrupt routines are managed via pragmas by the MW compiler (with the option to save or not the compiler working registers), whereas the Cosmic compiler simply requires the @interrupt qualifier to identify an interrupt handler. The Cosmic compiler generates reentrant code by default in the interrupt function, so that managing nested interrupts is not a problem. The MW compiler requires additional code to be able to manage nested interrupts.

Interrupt functions

INTERRUPT FUNCTIONS

COSMIC compiler internal variables

- **c_x** (2 bytes) used for extending **X** register to 16 bits
- **c_y** (2 bytes) used for extending **Y** register to 16 bits
- **c_lreg** (4 bytes) used for long and float operations

*Automatically and selectively saved with the **Y** register by interrupt functions*

If there is a function call inside the interrupt routine:

- **Y, c_x, c_y** saved even if not explicitly used, unless if **@nosvf** specified
- **c_lreg** *NOT* saved if not explicitly used, unless if **@svlreg** specified

Copyright © 2001 by COSMIC Software

This slide provides more details about how interrupts are managed by the Cosmic compiler : in addition to the hardware registers already saved by the hardware, the compiler knows which registers are used in the interrupt routine and automatically save only those that need to be, thus generating small code with low interrupt latency. The only exception to this rule, is when there are function calls in the interrupt handler: in this case the compiler cannot know which registers are used or not, so it saves all the commonly used ones (**Y, c_x** and **c_y**) but does not save the ones used for float and long operations (**c_lreg**, 4 bytes), as these operations are best avoided in an interrupt routine. This default behaviour can be modified as explained in the yellow box.

Including Assembler

ASSEMBLER INCLUSION

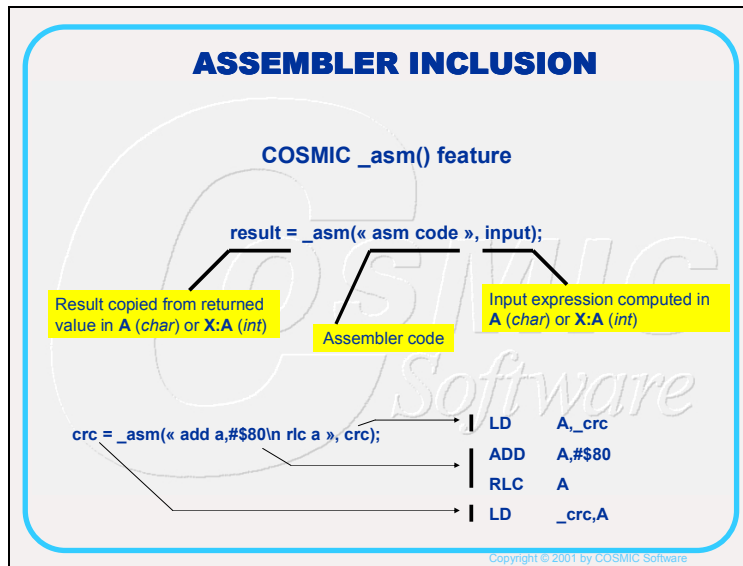
MW	COSMIC
<pre>asm LD A,X;</pre>	<pre>#asm</pre>
<pre>asm {</pre>	<pre>LD A,X</pre>
<pre>LD A,X</pre>	<pre>LD Y,A</pre>
<pre>LD Y,A</pre>	<pre>#endasm</pre>
<pre>}</pre>	<pre>result = _asm(« asm code », input);</pre>

<ul style="list-style-type: none">• Direct access to any C objects	<ul style="list-style-type: none">• Possible access to global C objects• NO ACCESS to local C objects• Use <code>_asm()</code> feature for C objects interface
--	--

Copyright © 2001 by COSMIC Software

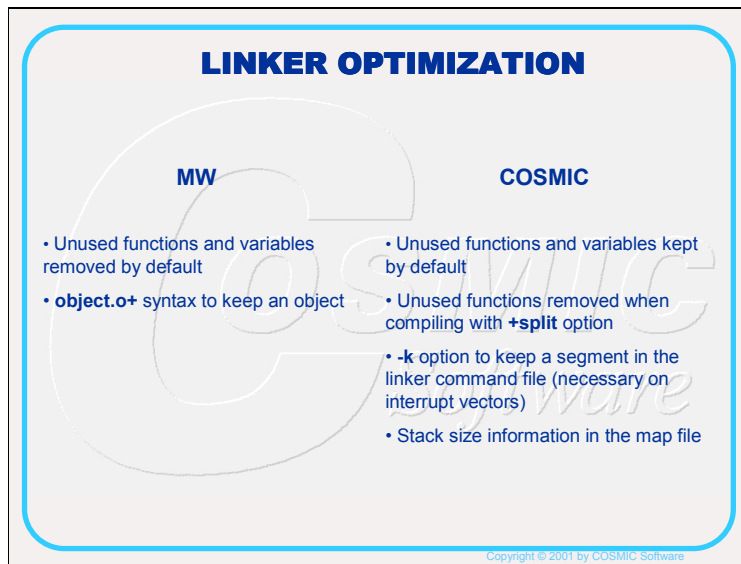
Assembler inclusion is managed in a fairly similar manner by the two compilers for what concern « simple assembler ». However, if the assembler gets more complex and starts interacting with C, more differences arise (in addition to the differences in the parameters passing convention already explained); for example, the MW compiler allows to access any object, while the Cosmic compiler allows only access to global objects, because local objects can be in the stack (stack models) thus making it impossible to access them directly. The next slide shows how to interact with local objects.

Including Assembler



The `_asm` function allows to access local objects. The compiler will generate the necessary code to access the parameters and to store the result in the specified variable.

Linker Optimizations



The MW and Cosmic linker are different in what the MW removes by default unused objects, whereas the Cosmic keeps them unless the +split option is used. This difference can play in subtle ways when doing quick benchmarks; if, for example, you decide to leave out the vector table “because it’s just to get an idea”, the MW linker will also remove all the interrupt functions (because they appear as unused), whereas the Cosmic linker will keep them, thus showing a false difference in code size.

Peripherals Access

PERIPHERALS ACCESS

MW	COSMIC
<ul style="list-style-type: none">• object file to be linked• file.o+ syntax to keep all definitions• no support of timer registers	<ul style="list-style-type: none">• header file to be included• #include <io72254.h>• no object to be linked• C syntax to simply load a register: TASR;• direct support of timer registers: TAOCR1 += 1000;

Copyright © 2001 by COSMIC Software

When it comes to hardware access, the Cosmic compiler comes with a complete set of predefined include files where all hardware registers for the most common derivatives are defined. Using these registers is as easy as assigning a value to them, and the compiler manages correctly the multi-byte registers that must be accessed in a well defined order. On the same idea of no-need to use assembler, registers that must be accessed but whose value is not significant, can be accessed with the syntax shown above.

With the MW compiler, SFR definitions must be linked in as an objet file. Using the registers in C is the same as with Cosmic, but timer registers require to use assembler as the compiler does not access them in the right order.

Benchmarking guidelines

This page provides a simple set of guidelines to follow when you are benchmarking Cosmic and MW compilers for ST7. Our experience shows that it is very easy to be misled when doing benchmarks, so here are the main points to check:

- A “good” benchmark should not only compile, but also link and run with both toolchains before you can compare the numbers: this might seem obvious, but as benchmarks are often done during an evaluation phase, sometimes there is no time to complete them properly (especially when porting from another micro). Here are a few examples of what might occur with uncomplete benchmarks:
 - o A compiler might do a great job of producing tight code by putting everything in page zero, only to find out that the linker will actually report that page zero is full and the application cannot link
 - o The linker might report everything is fine, except it might have discarded all the interrupt handlers if the interrupt table was not defined correctly: in this case the application will not run.
- Make sure you compare Memory Models that are comparable: the biggest incidence on the code size for a given application is given by where the variables are stored (short/long/stack) and this is decided mainly by the memory model. As explained earlier in this presentation, memory models with the same name are NOT comparable between Cosmic and MW. Also consider the pointer size (8 or 16 bits) and make sure the model selected use the same size for both compilers; check each compiler manual for more information on this.
- Check the main compilation options and make sure they are set consistently for both compilers; for example, some versions of the Cosmic compiler have the “factorization option” turned off by default (to facilitate debugging), whereas the same option for MW is ON by default.
- Compare the whole application performance. Comparing function by function can be useful to spot where a particular compiler is strong or weak, but in this case you have to take special care at the “side effects”:

COSMIC SOFTWARE

Application Note N °65 – Cosmic vs MW ST7 compiler

- a string manipulation can give a great benchmark if the string is in page0, but in this case you must keep into account that some other function (possibly in another module, or even in the startup file) must have copied it from ROM to RAM, and this other function takes place and time to execute, place and time that will likely not show in your bench
- A compiler might decide to inline a function in some cases: the overall application will be smaller and faster, but the module where the application is inlined will look bigger (but still faster) compared to a call to the same function in a library file
- Individual functions can be very small: a 20% difference in code size is almost meaningless if the code is 10 and 12 bytes, whereas it is much more representative if the code is 10k and 12k.
- Know what you want to compare and how to achieve it. Benchmarks usually take into account code size and execution speed; although code size is usually the most important parameter for an ST7 application, some optimizations that allow a small gain on the size against a big loss in execution speed might not be enabled by default in one or both compilers
- And, to finish, a rule of thumb: it is not scientific and there might be exceptions, but it's worth to keep in mind that:
 - If you find a difference of >10% between the two compilers (code size), there is likely some different setup between the two: check the points above.
 - If you find a difference of >20% there is almost definitely something wrong (different options, benchmark too small, too selective or biased); please contact support.