

**Writing Optimal C Code for ST7 with Cosmic C  
Compiler**



This technical article provides the basics for writing optimal ST7 applications by taking a real-life example and showing how it can be improved modifying both the code and the compiler options.

## The ST7 and Compiler registers

The slide is titled "ST7 Registers" in a yellow box at the top. It is divided into two columns: "Compiler" (light blue background) and "Processor" (light orange background). The "Compiler" column lists three registers: c\_x (2 bytes), c\_y (2 bytes), and c\_ireg (4 bytes). The "Processor" column lists six registers: A (accumulator 8 bits), X (index 8 bits), Y (index 8 bits with code prefix), S (stack pointer 8 bits in page 1), CC (condition code 8 bits), and PC (program counter 16 bits). The slide also features the COSMIC Software logo in the bottom left, the text "ST7 Code Optimization" in the bottom center, and a small number "2" in the bottom right corner.

Compiler		Processor	
(ram in short range)		A	accumulator 8 bits
c_x	2 bytes	X	index 8 bits
c_y	2 bytes	Y	index 8 bits (+code prefix)
c_ireg	4 bytes	S	stack pointer 8 bits in page 1
		CC	condition code 8 bits
		PC	program counter 16 bits

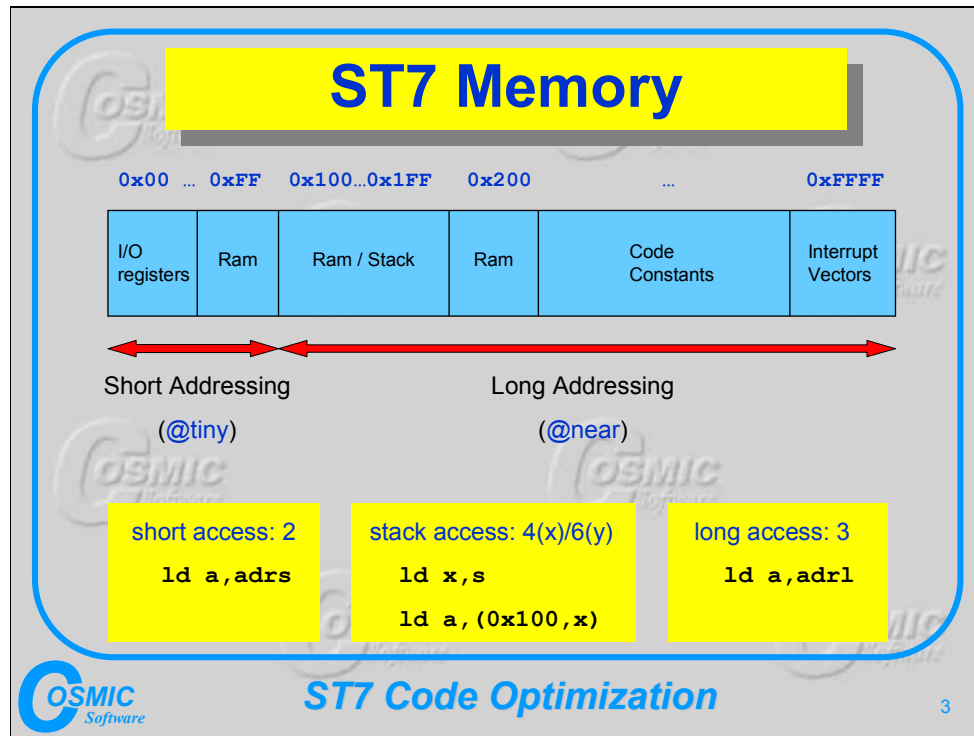
The ST7 has a set of 8 bit registers. The main registers, from the compiler point of view, are shown in the slide. Note that using the Y register requires 1 byte more than using the X register to do the same thing (because of the code prefix necessary to use Y), therefore the compiler will always try to use X when possible.

As the available hardware registers are not enough for anything but the simplest tasks, the compiler reserves 4 to 8 bytes of RAM memory for its own use and gives them conventional names as shown in the slide:

- c\_x and c\_y are two 16 bits general purpose registers that are always defined
- c\_ireg is a 4 bytes register that is used when long or float variables are used in the application

Note that the use of c\_x, c\_y and c\_ireg is transparent to the users, but you can find these symbols if you look at listing files, therefore you need to know what they are if you want to understand optimizations.

## The ST7 Memory Map



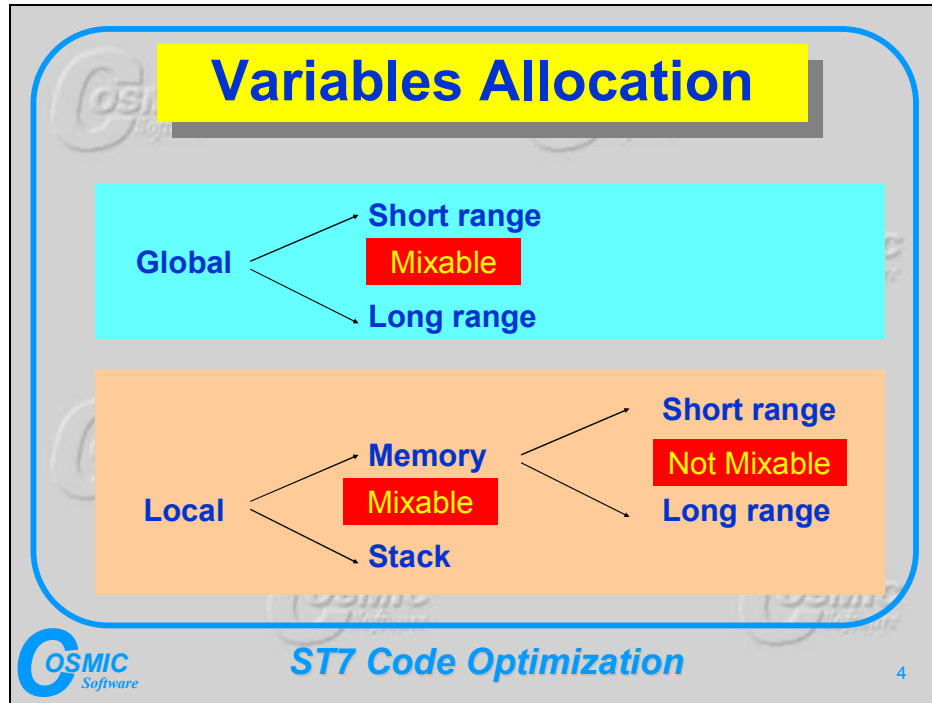
The ST7 memory map is organized as shown. The first page of 256 bytes (called Page 0, from 0x00 to 0xFF) can be accessed with the short addressing mode, which only requires two bytes, while the rest of the memory can be accessed in the long addressing mode (3 bytes) or stack addressing mode (4 bytes using X, 6 bytes using Y).

This shows clearly that it is more efficient, both in terms of code size and execution speed, to put as many variables as possible in the zero page, either using a memory model which puts variables there by default (more on this later), or using the @tiny modifier to force variables in the zero page regardless of the memory model.

Example:

```
@tiny char aa; // variable aa will be in page zero regardless of the memory model used
```

## Variables allocation



This slides gives an overview of how the compiler can allocate variables. Global variables can be either in short range or long range, including mixing the two types in the same application. Local variables and function parameters can be either in the stack (for reentrant functions) or in memory, and it is possible to mix, in, the same application, functions that are reentrant with functions that are not, but, for non-reentrant functions, local variables need to be all in the same addressing range. Of course the compiler takes care of this automatically.

## What are Memory Models?

Category	Model	Compiler Flag
Physical Stack	Stack long	+modsl
	Stack short	+mods
Static Memory	Memory large	+modml
	Memory medium	+modmm
	Memory small	+modms
	Memory short	+modm
	Memory compact	+modc

**Memory Models**

- **Physical Stack**
  - ◆ Stack long +modsl
  - ◆ Stack short +mods
- **Static Memory**
  - ◆ Memory large +modml
  - ◆ Memory medium +modmm
  - ◆ Memory small +modms
  - ◆ Memory short +modm
  - ◆ Memory compact +modc

**ST7 Code Optimization**

5

The first thing to do in order to produce efficient code is to choose a memory model that fits the application size and the processor resources. The memory model tells the compiler some default information about where to store variables and how to access them. This default behavior is applied to all variables for which the storage class is not specified by the user.

Example:

```
char aa;           // short or long addressing mode depending on the memory model
@tiny char aa;    // page zero regardless of the memory model
```

The first two memory models are called “stack” models because the local variables are put on the stack, thus making functions reentrant by default. These models are not the most efficient and should be used only when reentrancy is required or when performance is not an issue.

The last 5 memory models are called “static” model because the local variables are stored at fixed memory addresses, thus making functions that are not reentrant by default (calling the function while it is already being executed would overwrite the parameters of the first call).

Static models usually produce tight code, especially the models down in the list. In the second part of this documents we will see, taking a real-life example, how to choose the best memory model for a real application.

## CXST7 Memory Models

<i>Model</i>	<i>Stack</i>	<i>Globals</i>	<i>Pointers</i>
♦ modsl	phys	long	16 bits
♦ mods	phys	short	16 bits
♦ modml	long	long	16 bits
♦ modmm	long	short	16 bits
♦ modms	short	long	16 bits
♦ modm	short	short	16 bits
♦ modc	short	short	8 bits

This slide shows the details of the different memory models available with CXST7. As you see, the memory model has an influence on 3 parameters:

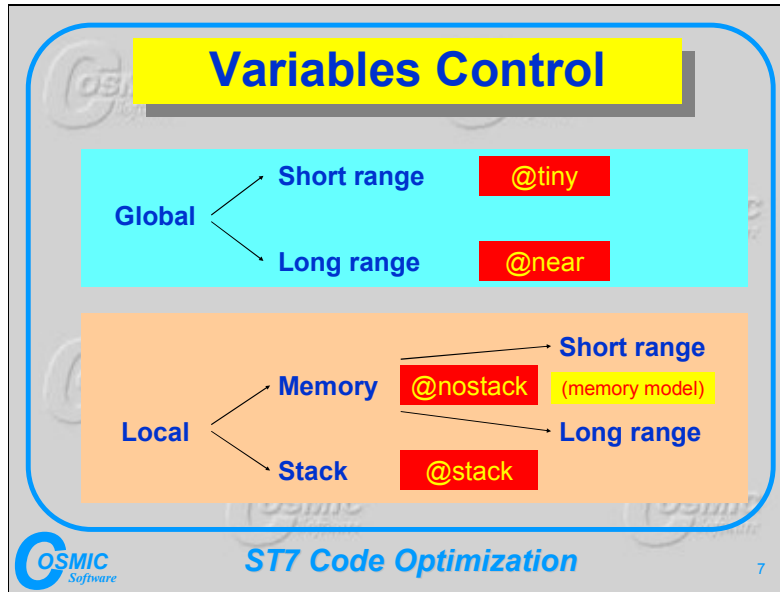
- Where and how the stack is managed
- Where global variables are stored by default
- How long the default pointer is.

In the stack models (first two lines), the actual stack (hardware) is used for local variables, and the compiler generates push and pop instructions. Global variables can be in short or long range and the pointers are 16 bits wide (that is, a pointer can point the whole 64k addressing range of the ST7).

The static models offer all combinations of simulated stack (that is, local variables at fixed addresses, either long or short) and global variables positioning (long or short).

The last model (compact) is suitable for devices that only have page zero (ST7 LITE), therefore everything need to be in there. As a consequence, the default pointer is only 8 bits as this is enough to cover the whole data space. In this case, pointers to the code space (constants, for example pointers to fixed strings) need to be declared with the @short modifier in order to make them 16 bits.

## Variables Control



This slide shows how to override the memory model settings and place variables manually. For global variables it is very simple: just use `@tiny` or `@near` in the variable declaration, and the variable will be forced respectively in the short or long addressing range.

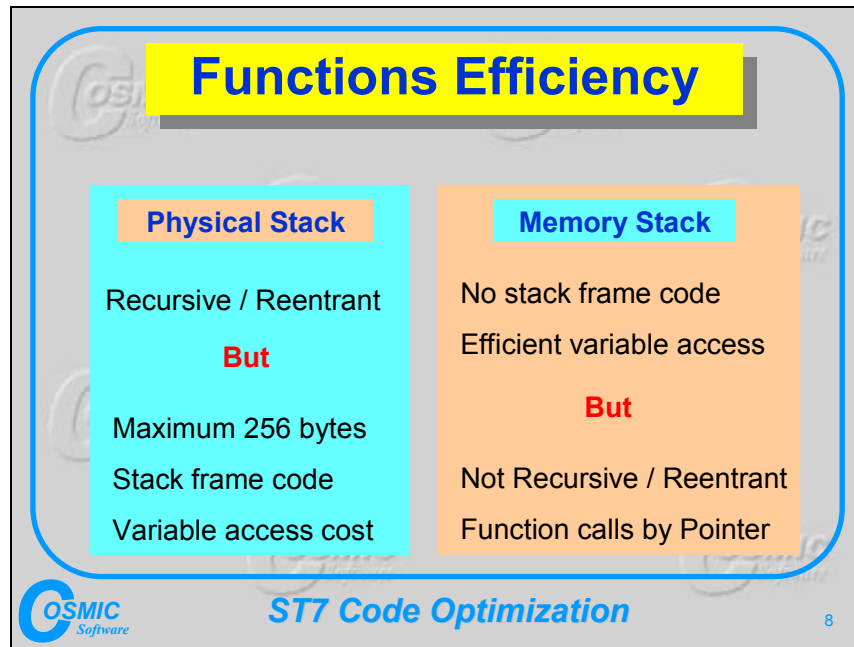
Local variables cannot be controlled one by one, but rather function by function. Declaring a function with the `@stack` attribute

```
@stack my_func(car a, char b)
{
    char c;
    // function code
}
```

will force the function to be reentrant regardless of the memory model, and therefore the local variables and parameters will be in the hardware stack.

Conversely, declaring a function with the `@nostack` attribute will force the function to use the simulated stack regardless of the memory model, and therefore the local variables and parameters will be at fixed memory addresses (short or long range, depending on the memory model).

## Functions Efficiency



This slide summarizes the advantages of using the hardware stack versus the simulated stack: the main advantage of the hardware stack is to make functions reentrant, but everything is bigger and slower. Conversely, the memory stack allows efficient code but functions are neither recursive (which is usually not too bad in an embedded application) nor reentrant (which means that you cannot call the same functions in an interrupt handler and in the main flow).



## Variables Efficiency

Short range	Long range
Shorter code / time	Limited by device
Allow direct pointers	<b>But</b>
Allow bit instructions	No direct pointers
<b>But</b>	No bit instructions
Maximum 128 bytes	

COSMIC Software ST7 Code Optimization 9

This slide summarizes the advantages of using short range variables versus long range ones: as you see, the only advantage of long range variables is that there is more space to store them (only limited by the RAM available on the device). Short range variables are more efficient from all point of view → one of the tricks to produce good code is to fill up the short range section as much as possible and/or use it for the variables that are accessed more often.

## BIT Instructions

**Bit Instructions**

```
if (flags & 0x08)
```

**Short range: 3**

```
brclr _flags,3,L123
```

**Long range: 7**

```
ld  a,_flags
bcp a,#8
jreq L123
```

**COSMIC Software** **ST7 Code Optimization** 10

Here is an example of how a variable in the short range can sometimes be dramatically more efficient than the same in long range.

If you want to test a single bit in the variable with the instruction

```
if (flags & 0x08) { }
```

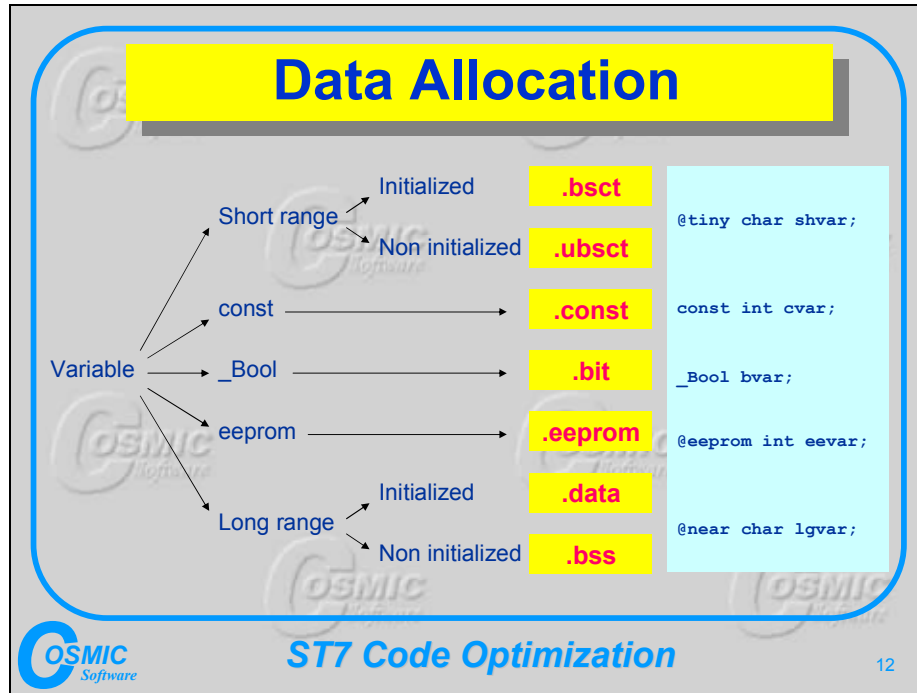
the compiler will generate a single assembler instruction (3 bytes) in the first case, while it needs 3 instructions (7 bytes) in the second case.

## Pointer Efficiency

The slide is titled "Pointer Efficiency" in a yellow box. Below the title, the C code `i = *p;` is shown in a yellow box. Two columns of assembly code are presented in light blue boxes. The left column, labeled "Short range: 3", shows two instructions: `ld a,[_p.w]` and `ld _i,a`. The right column, labeled "Long range: 13", shows six instructions: `ld a,_p`, `ld c_x,a`, `ld a,_p+1`, `ld c_x+1,a`, `ld a,[c_x.w]`, and `ld _i,a`. The slide also features the COSMIC Software logo in the bottom left, the text "ST7 Code Optimization" in the bottom center, and the number "11" in the bottom right.

Here is another example of how a variable in the short range can be dramatically more efficient than the same in long range, this time related to pointers: if you want to access a memory location via a 16 bits pointer, it requires only 2 assembler instructions (3 bytes) if the pointer is in the short addressing range, whereas, if the pointer is in the long addressing range, you first need to copy it in the short range (into the compiler working register `c_x`) for a total of 6 assembler instructions (13 bytes)

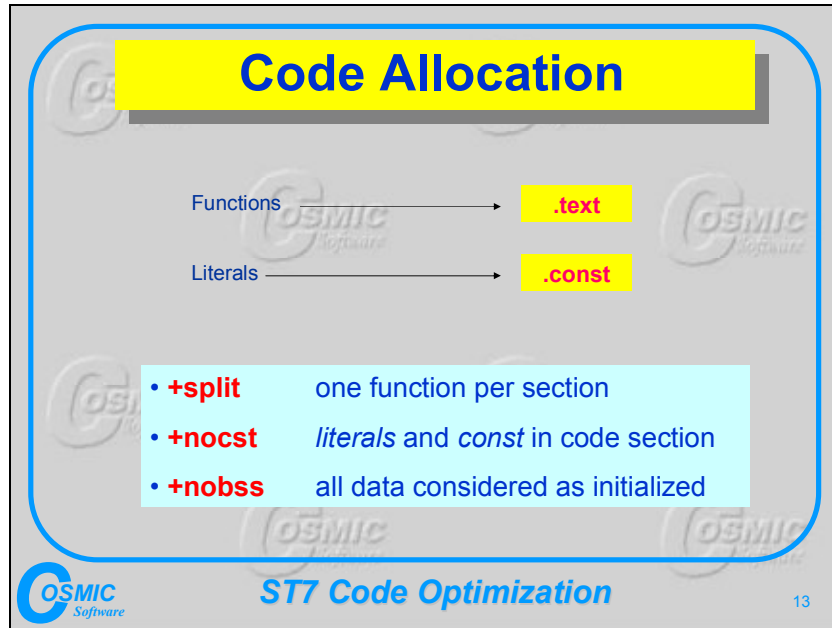
## Data allocation



This slide reminds how the compiler allocate different types of data into different sections. This is important to know because all sections used in the application must be properly declared in the linker command file.

Note that, when using initialized data, you need to use a specific startup file for the initialization to actually take place before starting to execute the user code; please refer to the CXST7 user manual for more details.

## Code Allocation



Code allocation is very simple : all code goes by default into the .text section and all literals (constants) go into the .const section.

The +split option tells the compiler to generate the code for each function into a separate section in such a way that the linker can eliminate uncalled functions: however this requires marking an entry point in the linker command file, see the CXST7 user manual for more details.

## Model Selection

**Model Selection**

- 1) Build application with **+modsl**  
largest model, no constraints
- 2) Check data / stack size in map file  
segments **.data** and **.bss** and stack usage
- 3) Rebuild with the closest model from the figures
- 4) Enhance by moving objects inside/outside short range

COSMIC Software ST7 Code Optimization 14

In this second part of this application note, we will show one possible way to optimize an existing application. Before going into the details of the steps outlined above, note that this is only one possible way; we have chosen this one because it is very general and should work with any kind and dimension of applications, but users can choose other ways (which usually means playing around with the same ideas, but in a different order).

So, supposing you have an application that already compiles, but that you don't know much else about it, here is how you can optimize it:

- First, compile it with the biggest and more general memory model available. The goal of this step is simply to build the whole application in such a way to have access to the map information generated by the compiler and linker
- Check the data / stack size in the map file
- Choose the smallest model that fits
- Refine by manually moving objects in and out of the short range

The following slides show these steps applied on a real-world application that we have helped one Customer to optimize.


## Real-world example : memory model selection

### Model Selection

**+modsl**

```
-----  
Segments  
-----  
  
start 0000e800 end 0000ef1b length 1819 segment .text  
start 0000e000 end 0000e0e1 length 225 segment .const  
start 00000080 end 00000080 length 0 segment .bsct  
start 00000080 end 00000084 length 4 segment iram  
start 00000100 end 00000103 length 3 segment .data, initialized  
start 0000ef23 end 0000ef26 length 3 segment .data, from  
start 00000084 end 00000084 length 0 segment .share  
start 00000000 end 00001ba1 length 7073 segment .debug  
start 00000103 end 00000151 length 78 segment .bss  
start 0000ffe0 end 00010000 length 32 segment .const  
start 0000ef1b end 0000ef23 length 8 segment .init
```

**4 bytes in short range****81 bytes in long range**

**ST7 Code Optimization**15

Our Customer code, when compiled with the Stack Large model (+modsl) gave the results shown above: code was 1819 bytes, 4 bytes in the short range (the c\_x and c\_y compiler registers) and 81 bytes in the long range (all the application variables). The stack usage was 77 bytes (next slide)

## Real-world example : memory model selection

# Model Selection


**+modsl**

```
-----  
Stack usage  
-----  
  
codem.o:  
_Initialization      5  (2)  
_main                 > 7  (2)  
  
ErrorP.o:  
_GetLastError        2  (2)  
_SetLastError        2  (2)  
  
Interrupt.o:  
_InterruptEnable    > 2  (2)  
_InterruptInit      2  (2)  
  
Tools.o:  
_tsInterpolate      8  (5)  
  
i2c.o:  
_I2CInit            3  (3)  
_I2CRead            6  (4)  
_I2CRW             11 (5)  
_I2CWrite           7  (5)  
  
Stack size: 77
```


4 bytes in short range

81 bytes in long range

77 bytes on the stack



try +modms



ST7 Code Optimization

16

This data suggest that the Memory Compact and Memory Short models will not fit (as both would try to put Stack and Globals, that is  $77+81=158$  bytes into the  $128-4=124$  available bytes of the short addressing range). Let's then try the Memory Small model (+modms) that will put the stack in page 0 and the globals in the long range.



## Real-world example : memory model selection

**Model Selection**

**Linker Error:**  
function \_I2CRW is reentrant

↓

in header file I2C.H:  
extern @stack BYTE I2CRW(I2CDEVADDR ucDevAddr,

in source file I2C.C:  
@stack BYTE I2CRW(I2CDEVADDR dev\_addr,

**OSMIC Software** **ST7 Code Optimization** 17

By trying to compile the same application with the Memory Small model, we get

Linker error: function \_I2CRW is reentrant

which means that this function is called both in an interrupt handler and in the main flow and should therefore be reentrant in order to work properly (when we compiled with the Stack Long model we did not have this error because, in this model, functions are reentrant by default).

Here we need to modify the code (function and prototype) in order to force this function to be reentrant using the @stack modifier as shown above.

## Real-world example : memory model selection

### Model Selection


**+modms**

-----  
Segments  
-----

```
start 0000e800 end 0000ed70 length 1392 segment .text
start 0000e000 end 0000e0e1 length 225 segment .const
start 00000080 end 00000080 length 0 segment .bsct
start 00000080 end 00000084 length 4 segment .iram
start 00000100 end 00000103 length 3 segment .data, initialized
start 0000ed78 end 0000ed7b length 3 segment .data, from
start 00000084 end 000000b9 length 53 segment .share
start 00000000 end 00001c61 length 7265 segment .debug
start 00000103 end 00000151 length 78 segment .bss
start 0000ffe0 end 00010000 length 32 segment .const
start 0000ed70 end 0000ed78 length 8 segment .init
```

**57 bytes in short range**

**81 bytes in long range**



**COSMIC**  
Software

## ST7 Code Optimization

18

Once we have modified the `_I2CRW` function to be reentrant, the application builds fine in the Memory Small model, and we can compare the results in the map file with what we got for the Stack Long Memory model.

The code is already much smaller (1392 vs 1819, that is 23% smaller) and we see that we still have free space in the zero page: this page currently holds the short range (57 bytes) and the stack (33 bytes, next slide), therefore there still  $128 - (33 + 57) = 38$  bytes available.

## Real-world example : memory model selection

### Model Selection


**+modms**

```
-----  
Shared areas  
-----  
  
ErrorP.o:  
_SetLastError      on stack (2)  
  
Interrupt.o:  
_E10IntHandler    on stack (11)  
_E11IntHandler    on stack (5)  
  
Tasks.o:  
_Task1            00000084-00000085 (2,0)  
_Task2            00000084-00000087 (4,0)  
_Task3            00000084-00000084 (1,0)  
  
Tools.o:  
_tsInterpolate    00000086-00000088 (0,3)  
  
i2c.o:  
_I2CInit          00000084-00000084 (1,0)  
_I2CRead          on stack (4)  
_I2CRW           on stack (5)  
_I2CWrite         00000087-0000008a (1,3)  
_I2CWriteWait    00000084-00000086 (0,3)  
  
Stack size: 33
```

57 bytes in short range  
**81 bytes in long range**  
33 bytes on the stack

↓

force some data in long range (@near) and  
try **+modm**

ST7 Code Optimization19

38 bytes available in page zero are far from enough for the 81 bytes currently in the long range, but here we can take a look at the code and see how the data is structured : if we're lucky there will be a few "big" data structures that can be moved in the long range by hand (with the @near modifier) and we can rebuild with the Memory Short (+modm) model that will put everything else into page zero.

## Real-world example : memory model selection

### Model Selection

```
-----  
Modules  
-----  
...  
ErrorP.o:  
start 0000eb6a end 0000eb72 length      8 section .text  
start 00000102 end 00000103 length      1 section .data  
start 0000111e end 0000125a length    316 section .debug  
  
Tasks.o:  
start 0000eb72 end 0000ec7f length    269 section .text  
start 00000108 end 00000148 length      64 section .bss  
start 0000125a end 000015e9 length    911 section .debug  
  
Tools.o:  
start 0000ec7f end 0000ecf7 length    120 section .text  
start 00000148 end 00000151 length      9 section .bss  
start 000015e9 end 000017e2 length    505 section .debug  
...
```

**57 bytes in  
short range**

**81 bytes in  
long range**


↓

**74 bytes in  
short range**

**64 bytes in  
long range**

**+modm**

`@near BYTE gVMem[64];`

ST7 Code Optimization20

Looking at the module information in the map file and at the source code, we see that there is an array of 64 bytes that can be forced in the long range with a very simple modification (shown in the slide). With this array out of the way, we should be able to build in the Memory Short model, which is the most efficient, as everything is in page zero.

## Real-world example : memory model selection

### Model Selection

**+modm**


```
Segments
-----
start 0000e800 end 0000ecc2 length 1218 segment .text
start 0000e000 end 0000e0e1 length 225 segment .const
start
```

**+modms**

```
Segments
-----
start 0000e800 end 0000ed70 length 1392 segment .text
start 0000e000 end 0000e0e1 length 225 segment .const
start
start
```

**+modsl**

```
Segments
-----
start 0000e800 end 0000ef1b length 1819 segment .text
start 0000e000 end 0000e0e1 length 225 segment .const
start 00000080 end 00000080 length 0 segment .bsct
start 00000080 end 00000084 length 4 segment .iram
start 00000100 end 00000103 length 3 segment .data, initialized
start 0000ef23 end 0000ef26 length 3 segment .data, from
start 00000084 end 00000084 length 0 segment .share
start 00000000 end 00001ba1 length 7073 segment .debug
start 00000103 end 00000151 length 78 segment .bss
start 0000ffe0 end 00010000 length 32 segment .const
start 0000ef1b end 0000ef23 length 8 segment .init
```

ST7 Code Optimization21

Building with +modm shows a further decrease in code size, down to 1218 bytes. The slide shows the results for the three models tested.

## Array efficiency

**Array Efficiency**

`i = tab[j];`

extern char tab[10];	extern char tab[];
<pre>ld  x,_j ld  a,(_tab,x) ld  _i,a</pre>	<pre>ld  a,_j add a,#low(_tab) ld  c_x+1,a clr a adc a,#high(_tab) ld  c_x,a ld  a,[c_x.w] ld  _i,a</pre>

**COSMIC** Software **ST7 Code Optimization** 22

Here is another example of how to write C code in order to help the compiler generate the most efficient assembler: if you know the array dimension, let the compiler know about it as well; if the dimension is small, some optimizations will be done automatically.

## IO Registers

**IO Registers**

IO Registers can be accessed without assignment

```
c = I2CDR;
```

```
ld a, _I2CDR  
ld _c, a
```

```
I2CDR;
```

```
ld a, _I2CDR
```

**COSMIC** Software  
**ST7 Code Optimization** 23

Some special function registers need to be read to perform some action (for example to set or clear a bit): in this case (that is, when you need to read, but you don't care about the content), you can simply write the name of the register followed by a semicolon; you will save one assembler instruction

## Unused Functions

**Unused Functions**

- compile with **+split** option
- link vectors segment with **-k** flag

```
+seg .const -b 0xFFE0 -k  
vector.o
```

- unused functions are not linked
- marked as **\*\*\* removed \*\*\*** in the map file

**COSMIC** Software **ST7 Code Optimization** 24

If you are not sure that all the functions in your application are ever called (for example because you are working on an application written by someone else), you can tell the compiler to check this for you by using the `+split` option. This option will produce the code for each function in a separate section and the linker will not link in the functions that are never called.

Using the `+split` option requires marking the segment that contains the reset vector with `-k` in the linker file, otherwise the linker will remove everything (because the reset vector is not called by any other function).