



WRITING OPTIMIZED HIWARE C LANGUAGE FOR ST7

by Microcontroller Division Application Team

INTRODUCTION

The C language originally created for UNIX system computers is used now in a wide number of application areas especially in embedded 8-bit microcontroller systems. This choice to extend the C language to micro applications is mainly due to:

- Its structured language based on data types and enhanced control structures.
- Its portability to different microcontroller target machines.

The purpose of this note is to present how to write an optimized C software application for ST7 embedded system programming. The main topics focus on how to write C source code that generates the smallest code and data size. To reach this goal some specific C language extensions have to be used like compiler options and pragmas.

This application note is based on the ST7 Hiware C compiler but some examples can be applied to other ST7 C compilers. It is not exhaustive and for more details concerning the ST7 Hiware C compiler please refer to the "HICROSS for ST7" manual.

Table of Contents

INTRODUCTION	1
1 COMPILER OVERVIEW AND SYNTAX	4
2 WRITING C FOR ST7	4
2.1 BASIC TYPE SIZE	4
2.2 LOCAL VARIABLES AND PARAMETERS STRATEGY	4
2.3 PARAMETER PASSING AND FUNCTION RETURN	5
2.4 MEMORY ALLOCATION	5
2.4.1 CONSTANT MEMORY ALLOCATION	6
2.4.2 VARIABLE DATA MEMORY ALLOCATION	6
2.4.3 CODE MEMORY ALLOCATION	7
2.5 MEMORY ACCESS VIA POINTER	8
2.6 INTERRUPT HANDLING	8
2.6.1 FUNCTION AS INTERRUPT SERVICE ROUTINE	8
2.6.2 INTERRUPT FUNCTIONS AND HIWARE INTERNAL VARIABLES	9
2.7 HLI ASSEMBLER	10
2.7.1 SYNTAX	10
2.7.2 HLI ASSEMBLER AND C LANGUAGE	10
2.7.3 SPECIFIC HLI ASSEMBLER FEATURES	11
3 GENERAL C PROGRAMMING RULES	12
3.1 SOURCE AND HEADER FILES	12
3.2 MEANINGFUL NAMING STRATEGY	12
3.3 ANSI C LANGUAGE	13
3.4 GENERIC FUNCTIONS VS MACROS	13
3.5 AVOIDING GOTO STATEMENTS	13
3.6 STATIC GLOBAL/LOCAL VARIABLES	13
3.7 VOLATILE QUALIFIERS	13
3.8 BITFIELD STRUCTURE	14
4 OPTIMIZING CODE IN ST7 C SOURCE MODULES	15

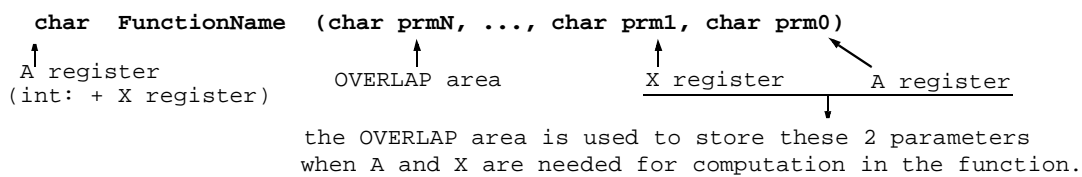
Table of Contents

4.1 GENERAL APPROACH	15
4.2 OPTIMUM VARIABLE DEFINITIONS	16
4.2.1 SMALLEST DATA TYPE	16
4.2.2 ST7 SHORT ADDRESSING MODE	16
4.2.3 VOLATILE QUALIFIER	17
4.2.4 STATIC LOCAL VARIABLE	17
4.2.5 UNION TYPE	17
4.2.6 AVOID ENUM TYPE	18
4.2.7 OPTIMUM TABLE AND STRUCTURE ACCESS	18
4.2.8 MASKED BYTE VARIABLES INSTEAD OF BITFIELDS	19
4.2.9 OPTIMIZED INT TYPE	21
4.2.10 OPTIMUM POINTER ACCESS	21
4.2.11 BASIC TYPE CONVERSION	21
4.2.12 NO OPTIMIZATION: REGISTER, AUTO	22
4.3 TEMPORARY COMPILER VARIABLES	22
4.4 OPTIMUM CONSTANT DEFINITIONS	22
4.5 OPTIMUM C STATEMENT MANAGEMENT	23
4.5.1 NON-RELEVANT C SOURCE CODE	23
4.5.1.1 UNREACHABLE CODE	23
4.5.1.2 REDUNDANT CODE	24
4.5.2 BYTE SIZE OBJECT OPERATIONS	24
4.5.2.1 OPTIMUM BASIC OPERATIONS	24
4.5.2.2 BYTE DIVISION	24
4.5.2.3 OPTIMUM BYTE SHIFT	25
4.5.3 WORD SIZE OBJECT OPERATION	25
4.5.3.1 OPTIMUM BASIC OPERATIONS	25
4.5.3.2 WORD MULTIPLICATION AND DIVISION	26
4.5.4 CONDITIONAL STATEMENTS	26
4.5.4.1 IF STATEMENT INSTEAD OF SWITCH	26
4.5.4.2 TEST CONDITION ORDER	27
4.5.4.3 IF STATEMENTS AND ASSIGNMENTS	27
4.6 OPTIMUM FUNCTION MANAGEMENT	28
4.6.1 FUNCTIONS VS MACROS	28
4.6.2 FUNCTION PARAMETERS AND RETURN VALUE	28
4.6.3 FUNCTION PARAMETER DECLARATION ORDER	29
5 CONCLUSION	30

In fact, this strategy is due to fact that the ST7 instruction set does not allow simple access to the data stored in the stack (like `LD A,[SP,#-n]`). The only possibility would have been a two instruction solution for each stack data access (`LD X,S` followed by `LD A,([#0x0100+n,X])` where `0x0100` is the stack bottom address and `n` the data place offset).

2.3 PARAMETER PASSING AND FUNCTION RETURN

As it is described in the previous paragraph, by default the function parameters are stored in the dedicated OVERLAP memory area. But, the last two bytes in the parameter list are also always passed through ST7 hardware registers (A and X). The two least significant bytes of the return value are also transferred through these two CPU registers.



This technique does not avoid the use of the OVERLAP area but when it is possible, it does some optimization of RAM use.

```

char AddChar (char p1, char p2)
{ return(p2+p1); }
    
```

Only p1 is stored in the OVERLAP area

```

0000 BF 00 [4] LD p1,X
0002 BB 00 [3] ADD A,p1
0004 81 [6] RET
    
```

2.4 MEMORY ALLOCATION

First of all the compiler memory model has to be chosen based on the ST7 application type. The following table gives some guidelines for choosing the correct compiler memory model option.

Compiler Option	Memory Model	Local Data Parameters	Global Data	ST7 Application Type
-Ms	SMALL	Page 0	Page 0	Application with all used RAM in page 0
-MI	LARGE	Page 0	Page 1..N*	Most efficient code generation for a standard ST7 application (local variables and parameters used more often than global data).
-Mx	LARGE Extended	Page 1..N	Page 1..N*	Very large ST7 application with a lot of local variables and parameters which can not be stored in page 0

***Note:** Global variables can be forced in the page 0 through the pragma `DATA_SEG SHORT`.

2.4.1 CONSTANT MEMORY ALLOCATION

C language constants are declared with a `const` keyword. By default, the Hiware C compiler allocates constants in the `DEFAULT_RAM` area. To force the constant memory allocation in the `ROM_VAR` area the `-Cc` compiler option has to be activated.

```
const char table[] = {...};  
const char *table[] = {...};
```

Note: the `ROM_VAR` keyword has always to be defined in the linker parameter file.

When a constant has to be allocated in a specific ROM area (not in the default one), the pragma `CONST_SEG` has to be used (this pragma is valid up to the next `CONST_SEG` pragma).

```
#pragma CONST_SEG MY_ROM  
const char table[] = {...};
```

→ PLACEMENT DECLARED IN
THE LINKER PARAMETER FILE

2.4.2 VARIABLE DATA MEMORY ALLOCATION

By default variables are allocated in the `DEFAULT_RAM` area. In ST7 applications some variables must have fixed addresses (hardware register definitions or variables forced in page 0). In ANSI style, the only way to do this is to directly use the address and the preprocessor capability.

```
#define VAR (*(char*) (0x0010))
```

The drawback of this method is that real C variables are not used which makes the debugging task harder. To resolve this, the ST7 Hiware C compiler provides the `DATA_SEG` pragma. With this pragma variables can be allocated to a specific memory area.

```
#pragma DATA_SEG MY_RAM  
char var;
```

→ PLACEMENT DECLARED IN
THE LINKER PARAMETER FILE

When variables are allocated to page 0 (from `0x0000` to `0x00FF`), the compiler has to know this information to minimize the code when they are used. This allocation is done through the pragma qualifier `SHORT`.

```
#pragma DATA_SEG SHORT MY_PAGE0  
char page0var;
```

→ PLACEMENT DECLARED IN
THE LINKER PARAMETER FILE

The ST7 Hiware linker allocates memory sequentially in the order of declaration and optimizes the data memory space when a variable is not used in the application.

Due to this optimization and as each pragma `DATA_SEG` is valid up to the next, some care has to be taken when declaring hardware registers. In fact, to avoid hardware register address

shifting, a "+" symbol has to be added to each object file name in the linker parameter file where there is such a declaration.

```

FILE.C                                FILE.PRM
#pragma DATA_SEG SHORT REG_AREA      NAMES
volatile char ADCDR;                  FILE.O+
volatile char ADCCSR;                 ...
#pragma DATA_SEG DEFAULT              PLACEMENT
                                        REG_AREA in NO_INIT 0x0070 TO 0x0072
                                        ...

```

Without the + and if the ADCDR register is not used, the ADCCSR is allocated to address 0x0070 instead of 0x0071.

Note: As the pragma DATA_SEG is valid up to the next one, when using it in a header file, always remember to reselect the DEFAULT_RAM area at the end.

```

FILE.H                                FILE.C
#pragma DATA_SEG SHORT REG_AREA      #include "FILE.H"
...                                    char var;
#pragma DATA_SEG DEFAULT              ...
/* END OF FILE */

```

Without the default pragma, "var" is allocated to REG_AREA.

2.4.3 CODE MEMORY ALLOCATION

In some circumstances, applications need to allocate part of generated code in specific ROM area. To allow this, the ST7 Hiware C compiler provides a pragma called CODE_SEG.

```

#pragma CODE_SEG MY_ROM
void FctName1 (void) {...}
...
#pragma CODE_SEG DEFAULT
void FctName2 (void) {...}

```

PLACEMENT DECLARED IN THE PRM FILE

This pragma is applied to the code in the same way as the DATA_SEG is applied to the variables. This means that it is valid up to the next pragma statement.

2.5 MEMORY ACCESS VIA POINTER

When LARGE memory models are selected (-Ml or -Mx compiler options), the optimum access to a data located in page 0 through a pointer can be reached with the use of the “near” keyword.

<pre>char * near FunctionName (char * near ptr) { return ((char * near) ptr +1); }</pre>		<p>Without near keyword:</p> <pre>0000 AB 01 [2] ADD A,#0x01 0002 97 [2] LD X,A 0003 4F [3] CLR A 0004 A9 00 [2] ADC A,#0x00 0006 90 97 [3] LD Y,A 0008 9F [2] LD A,X 0009 93 [2] LD X,Y 000A 81 [6] RET</pre>
<p>With near keyword:</p> <pre>0000 4C [3] INC A 0001 81 [6] RET</pre>		

When the SMALL memory model is selected (-Ms compiler option), the only access to a data located outside page 0 through a pointer can only be accessed with the use of the “far” keyword.

<pre>char * far FunctionName (char * far ptr) { return ((char * far) ptr +1); }</pre>		<p>With far keyword:</p> <pre>0000 AB 01 [2] ADD A,#0x01 0002 97 [2] LD X,A 0003 4F [3] CLR A 0004 A9 00 [2] ADC A,#0x00 0006 90 97 [3] LD Y,A 0008 9F [2] LD A,X 0009 93 [2] LD X,Y 000A 81 [6] RET</pre>
<p>Without far keyword:</p> <pre>0000 4C [3] INC A 0001 81 [6] RET</pre>		

2.6 INTERRUPT HANDLING

As the ST7 has interrupt management capabilities which are not included in the ANSI C standard, specific pragmas are implemented in the Hiware compiler.

2.6.1 FUNCTION AS INTERRUPT SERVICE ROUTINE

Writing a TRAP_PROC pragma before a function definition will produce the replacement of the RET instruction by an IRET one at the exit of this function.

<pre>#pragma TRAP_PROC void InterruptFct (void) {...}</pre>		<p>Generated code: ... IRET</p>
---	--	-------------------------------------

Note: To secure the unused interrupt vectors, write an empty dummy interrupt function.

2.6.2 INTERRUPT FUNCTIONS AND HIWARE INTERNAL VARIABLES

By default the ST7 Hiware C compiler allocates 3 storage variables for extended operations. These variables called `_SEX` (for indirect storing), `_LEX` (for indirect loading) and `_R_Z` (temporary storage) are located in page 0.

```

char *ptr;           ───────────→ 0000 C6 00 01 [4] LD A,ptr:0x1
*ptr = 0xFF;        0003 B7 01 [4] LD _SEX:0x1,A
                   0005 A6 FF [2] LD A,#0xFF
                   0007 CE 00 00 [4] LD X,ptr
                   000A BF 00 [4] LD _SEX,X
                   000C 92 C7 00 [7] LD [_SEX.W],A
                   000F 81 [6] RET
    
```

These variables can be used at any time (not controlled by programmer) in the application, either in the main program or in the interrupt routines. For this reason, as a first development step, the pragma `SAVE_REGS` has to be used to save these software registers. As a second step, after checking the generated code of the interrupt routine, this pragma can be removed if possible.

```

#pragma TRAP_PROC SAVE_REGS
void InterruptFct (void) {...}

    INTERRUP T PROLOG ─────────→
    + 17 bytes
    + 34 cycles

    INTERRUP T EPILOG ─────────→
    + 17 bytes
    + 45 cycles

    90 89 [4] PUSH Y
    B6 00 [3] LD A,_R_Z
    88 [3] PUSH A
    B6 01 [3] LD A,_LEX:0x1
    88 [3] PUSH A
    B6 00 [3] LD A,_LEX
    88 [3] PUSH A
    B6 01 [3] LD A,_SEX:0x1
    88 [3] PUSH A
    B6 00 [3] LD A,_SEX
    88 [3] PUSH A
    ...
    84 [4] POP A
    B7 00 [4] LD _SEX,A
    84 [4] POP A
    B7 01 [4] LD _SEX:0x1,A
    84 [4] POP A
    B7 00 [4] LD _LEX,A
    84 [4] POP A
    B7 01 [4] LD _LEX:0x1,A
    84 [4] POP A
    B7 00 [4] LD _R_Z,A
    90 85 [5] POP Y
    80 [9] IRET
    
```

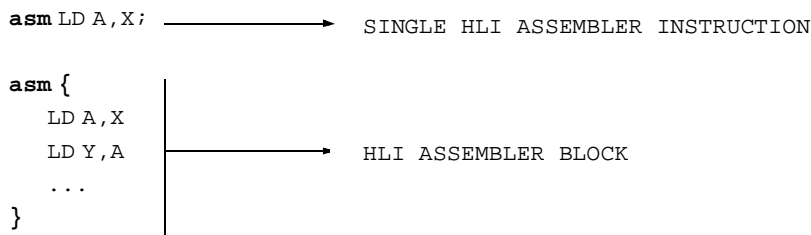
Note: The drawback of using the `SAVE_REGS` pragma is that the code size is increased by 34 bytes, the interrupt latency increased by 34 cycles (4.25µs with $f_{CPU}=8MHz$) and the total duration of the interrupt routine execution increased by 79 cycles (9.875µs with $f_{CPU}=8MHz$).

2.7 HLI ASSEMBLER

For critical source code specifically written for the ST7 microcontroller (mandatory write sequence, critical software timing...), the C language is not appropriate. To provide a solution Hiware C compiler provides a powerful HLI (high level inline) assembler.

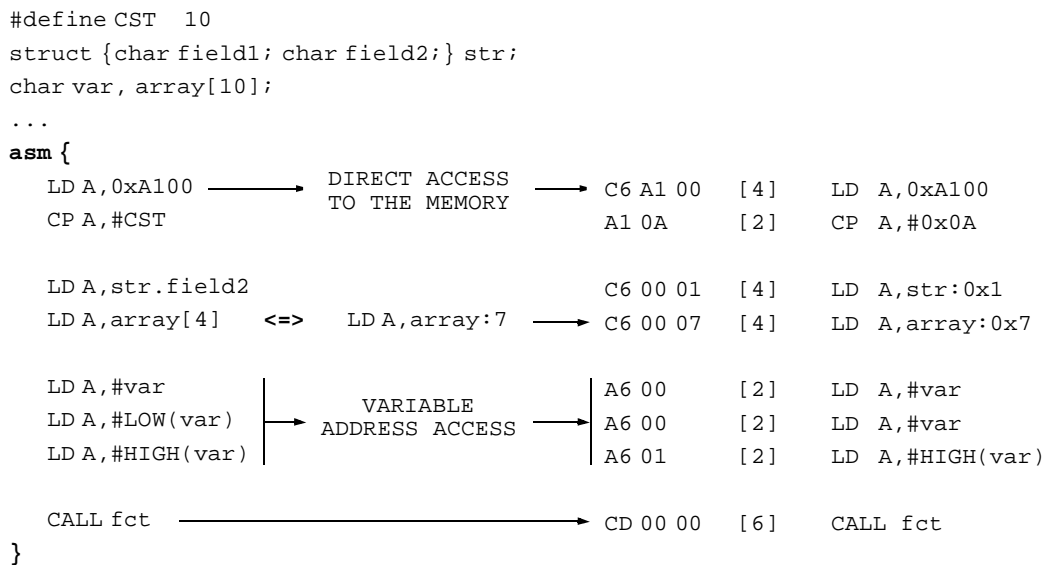
2.7.1 SYNTAX

The Hiware HLI assembler is based on the ST7 instruction set (cf. ST7 Programming Manual) and the `asm` keyword.



2.7.2 HLI ASSEMBLER AND C LANGUAGE

All C source data objects are directly accessible with the HLI assembler (variables, functions, macros...).



Note: As in the ANSI C preprocessor, the # symbol is used for string concatenation, and as the ST7 instruction set uses this symbol for immediate values, the Hiware C compiler provides a

dedicated `NO_STRING_CONSTR` pragma which disables this preprocessor feature. This pragma is valid for all the source modules.

```
#pragma NO_STRING_CONSTR
#define LDA10 LD A,#10
...
asm LDA10;
_____→ A6 0A [2] LD A,#0x0A
```

2.7.3 SPECIFIC HLI ASSEMBLER FEATURES

In the HLI syntax, the ST7 branches and data insertion in the code are possible. The following example shows some possible combinations:

```
asm {
    JP clear          0000 20 01 [3]  JRT *1 abs=0003
    DC 0xFF          0002 FF [4]    LD (X),X
clear: CLR X         0003 5F [3]    CLR X
    SKIP            0004 21 [3]    SKIP <JRF>
loop:  INC X        0005 5C [3]    INC X
    JRNE loop      0006 26 FD [3]  JRNE *-3 abs=0005
}
_____→ INC X INSTRUCTION
          IS SKIPPED IN THE
          1ST LOOP RLL
```

Notes:

- 1) The compiler optimizes absolute branches into relative ones whenever possible.
- 2) In the same way as ANSI C labels, the HLI assembler labels are valid only in the current function.

In Hiware HLI assembler syntax, only a few operators are available for constant expressions. These are: + - * / ().

```
asm ADD A,#12+(2-4)+4*5/2; _____→ AB 14 [2] ADD A,#0x14
```

3 GENERAL C PROGRAMMING RULES

The goal of this chapter is to give some C programming rules to improve the legibility, the portability and the robustness of the application C source code. In some circumstances, these rules can help to optimize the generated code size.

3.1 SOURCE AND HEADER FILES

To get a structured software application, whenever possible associate one header file “.h” with each “.c” source file. This header file should contain all the `extern` definitions and macros of the corresponding source file (all data and functions which can be used in other modules).

Note: The ST7 C compiler provides a `-wped` option to prevent extern definition omissions.

To prevent `#include` recursions, all header files have to be preprocessed. To avoid the duplication of declarations in several files (*.h and *.c), macro definitions and variable declarations have to be defined in the header file with `extern` preprocessing.

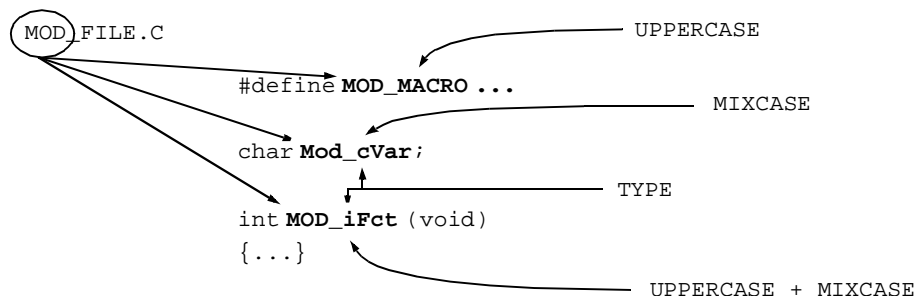
```
FILE.H      #ifndef FILE_H
             #define FILE_H
             #ifdef FILE_C
               #define EXT
             #else
               #define EXT extern
             #endif
             #define MACRO ...
             EXT char myvar; ...
             #endif

FILE.C      #define FILE_C
             #include "FILE.H"
             ...
             myvar = MACRO;
             ...
```

Note: in the ST7 Hiware C compiler a dedicated pragma (called ONCE) can replace the header file preprocessing (FILE_H).

3.2 MEANINGFUL NAMING STRATEGY

For reusable and legible C software modules, try as much as possible to implement a function, variable, preprocessor symbol and module naming strategy.



3.3 ANSI C LANGUAGE

The most important rule to get the best code portability is to write C program as much as possible in ANSI C like language. Sometimes, it is better to use `pragma` instead of specific non-ANSI instructions as the unknown pragmas have to be ignored by a compiler.

3.4 GENERIC FUNCTIONS vs MACROS

Use generic functions as much as possible instead of macros or specific functions.

Keep only very simple macros which generates optimum code (less instruction bytes than a `CALL` instruction).

Note: All repeated instruction blocks in the source code have to be shrunk to a single function whenever possible.

3.5 AVOIDING GOTO STATEMENTS

To get the most legible source code the `goto` statement as to be avoided each time there is an alternative.

3.6 STATIC GLOBAL/LOCAL VARIABLES

For good legibility and a well structured application, the following C language objects have to be declared as `static`:

- A global variable used in the module where it is declared.
- A function used only in the module where it is declared.
- A permanent local variable in a function.

A temporary local variable in a function has always to be declared as a local variable to save RAM area using the OVERLAP strategy.

```
char exported_var;
static char module_var;
static void module_fct (void)
{ ... }
void exported_fct (void)
{   char tmp_local_var;
    static char perm_local_var;
    ...}
```

3.7 VOLATILE QUALIFIERS

For all microcontroller hardware registers which can have their value modified directly by the hardware (status registers...), the `volatile` qualifier has to be added to their definition to allow the compiler to disable the register value memory optimization.

3.8 BITFIELD STRUCTURE

As ANSI C does not specify the bit allocation order in a bitfield, the use of this structure should be avoided for portability reasons. Consequently, bitfield structures should not be used for microcontroller hardware register definitions.

4 OPTIMIZING CODE IN ST7 C SOURCE MODULES

4.1 GENERAL APPROACH

To get the optimum generated code from the ST7 Hiware C compiler, the following conditions have to be met:

- Choose the memory model that suits the application, bearing in mind that the optimum is the SMALL one, then the LARGE one and finally the LARGE Extended one (to be avoided if it is possible). When Linking with a LARGE memory model, use the `near` keyword as much as possible when accessing short addressing data with pointers.

- Choose the best compiler options:

`-Cni`: by default the ANSI C operations with data type smaller than `int` have to be promoted to `int` (integral promotion). With this option, this integral promotion is omitted. Be aware that this option can generate some wrong code like the carry mistake in the following example:

<pre>int i1; char c1, c2; ... i1 = c1 + c2;</pre>	<p>THE BYTE ADDITION CARRY IS IGNORED</p>	<table border="0"> <tr><td>C6 00 00 [4]</td><td>LD</td><td>A, c1</td></tr> <tr><td>CB 00 00 [4]</td><td>ADD</td><td>A, c2</td></tr> <tr><td>C7 00 01 [5]</td><td>LD</td><td>i1:0x1, A</td></tr> <tr><td>5F [3]</td><td>CLR</td><td>X</td></tr> <tr><td>CF 00 00 [5]</td><td>LD</td><td>i1, X</td></tr> </table>	C6 00 00 [4]	LD	A, c1	CB 00 00 [4]	ADD	A, c2	C7 00 01 [5]	LD	i1:0x1, A	5F [3]	CLR	X	CF 00 00 [5]	LD	i1, X
C6 00 00 [4]	LD	A, c1															
CB 00 00 [4]	ADD	A, c2															
C7 00 01 [5]	LD	i1:0x1, A															
5F [3]	CLR	X															
CF 00 00 [5]	LD	i1, X															

`-Ou`: this optimization option removes all assignments to local variables not referenced later on in the code.

<pre>void function (char c1) { fctcall(); c1 = 1; }</pre>	<p>-Ou</p>	<table border="0"> <tr><td>AD 00 [6]</td><td>CALLR</td><td>fctcall</td></tr> <tr><td>A6 01 [2]</td><td>LD</td><td>A, #0x01</td></tr> <tr><td>B7 00 [4]</td><td>LD</td><td>c1, A</td></tr> <tr><td>81 [6]</td><td>RET</td><td></td></tr> </table>	AD 00 [6]	CALLR	fctcall	A6 01 [2]	LD	A, #0x01	B7 00 [4]	LD	c1, A	81 [6]	RET	
AD 00 [6]	CALLR	fctcall												
A6 01 [2]	LD	A, #0x01												
B7 00 [4]	LD	c1, A												
81 [6]	RET													
		<table border="0"> <tr><td>20 00 [3]</td><td>JRT</td><td>fctcall</td></tr> </table>	20 00 [3]	JRT	fctcall									
20 00 [3]	JRT	fctcall												

`-Os/-Ot`: by default the compiler optimizes the generated code size wise (`-Os`). This default option is based mainly on the use of library `CALL` statements instead of single level instruction sequences (when `-Ot` option is selected). The following example illustrates this compiler option choice.

<pre>int i1; signed char sc1; void function (void) { i1 = sc1; ...}</pre>	<p>-Ot</p>	<table border="0"> <tr><td>C6 00 00 [4]</td><td>LD</td><td>A, sc1</td></tr> <tr><td>5F [3]</td><td>CLR</td><td>X</td></tr> <tr><td>4D [3]</td><td>TNZ</td><td>A</td></tr> <tr><td>2A 01 [3]</td><td>JRPL</td><td>label</td></tr> <tr><td>5A [3]</td><td>DEC</td><td>X</td></tr> <tr><td>C7 00 01 [5]</td><td>label: LD</td><td>i1:0x1, A</td></tr> <tr><td>CF 00 00 [5]</td><td>LD</td><td>i1, X</td></tr> </table>	C6 00 00 [4]	LD	A, sc1	5F [3]	CLR	X	4D [3]	TNZ	A	2A 01 [3]	JRPL	label	5A [3]	DEC	X	C7 00 01 [5]	label: LD	i1:0x1, A	CF 00 00 [5]	LD	i1, X
C6 00 00 [4]	LD	A, sc1																					
5F [3]	CLR	X																					
4D [3]	TNZ	A																					
2A 01 [3]	JRPL	label																					
5A [3]	DEC	X																					
C7 00 01 [5]	label: LD	i1:0x1, A																					
CF 00 00 [5]	LD	i1, X																					
<table border="0"> <tr><td>C6 00 00[4]</td><td>LD</td><td>A, sc1</td></tr> <tr><td>CD 00 00[6]</td><td>CALL</td><td>_SEXT16</td></tr> <tr><td>C7 00 01[5]</td><td>LD</td><td>i1:0x1, A</td></tr> <tr><td>CF 00 00[5]</td><td>LD</td><td>i1, X</td></tr> </table>	C6 00 00[4]	LD	A, sc1	CD 00 00[6]	CALL	_SEXT16	C7 00 01[5]	LD	i1:0x1, A	CF 00 00[5]	LD	i1, X	<p>-Os</p>										
C6 00 00[4]	LD	A, sc1																					
CD 00 00[6]	CALL	_SEXT16																					
C7 00 01[5]	LD	i1:0x1, A																					
CF 00 00[5]	LD	i1, X																					

Note: The `-O1` or `-Or` optimization options have no effect in the ST7 Hiware C compiler.

4.2 OPTIMUM VARIABLE DEFINITIONS

4.2.1 SMALLEST DATA TYPE

To get the optimum code with a 8-bit microcontroller, the use of the smallest data type size is the best way. In the ANSI C language the smallest types are the bit (bitfield) and the `char` type (8 bits).

As the ST7 instruction set provides optimum bit handling instructions (BRES, BSET, BTJT, BTJF and BCP), the software has to use this structure as much as possible for binary data (this data has always to be located in the page 0). For other data types, `char` is the preferred data type to get the optimum generated code.

Using 32-bit variables (floating point arithmetic and long types) on a 8-bit microcontroller is inherently inefficient and character or integer arithmetic should be used whenever possible.

4.2.2 ST7 SHORT ADDRESSING MODE

The ST7 instruction set provides three main advantages to variables allocated in page 0 (from address 0x0000 to 0x00FF):

- short memory addressing access to get a shorter code. For instance a "LD A, var" instruction needs two bytes when the variable `var` is allocated to the page 0 while it needs three bytes when it is allocated outside (30% code size gain).
- ability to be directly handled by ST7 instructions usually dedicated to CPU registers (INC, DEC, CLR, CPL, NEG, RLC, RRC, SLA, SLL, SRA, SRL, SWAP, TNZ)..

<pre>char c1; #pragma DATA_SEG SHORT ZEROPAGE char cz1, #pragma DATA_SEG DEFAULT ... c1++; cz1++; ... c1 = c1 >> 1; cz1 = cz1 >> 1;</pre>	<pre>LD A, c1 INC A LD c1, A INC cz1 LD A, c1 SRL A LD c1, A SRL cz1</pre>
---	---

- bit handling capability (BRES, BSET, BTJT, BTJF and BCP instructions).

For these three reasons the following variables must be allocated whenever possible in short address memory space (page 0):

- All variables which have to be used as bitfields,
- The most frequently used standard variables.

4.2.3 VOLATILE QUALIFIER

To get the optimum generated code, the `volatile` qualifier has to be used only when it is needed and not systematically for each hardware register. A control register which can be modified only by the software or hardware microcontroller reset does not need to be a `volatile` variable.

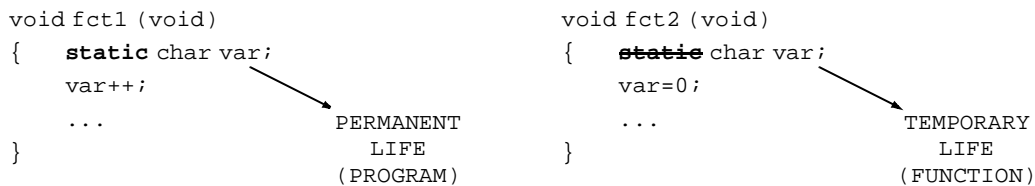
```
volatile char PADDR;
volatile char PADDR;
```

PADDR is the data register of the port A and can be modified in input mode by the hardware while the PADDR can only be modified by software or by a hardware RESET.

4.2.4 STATIC LOCAL VARIABLE

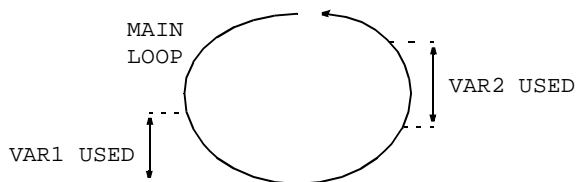
From a legibility point of view the local `static` definitions are to be used in preference to global ones whenever possible. This guarantees the robustness of the program against wrong static local variable use.

In the case of local variable definitions, this qualifier has to be used only for permanent data (not to be overlapped). Do not use this `static` qualifier when it is not needed to avoid removing the overlap optimization for local variables and increasing the used RAM size.



4.2.5 UNION TYPE

All global variables which do not have permanent validity during application execution and which are not able to be defined as local variables in a function, have to be grouped together and optimized into `union` type structures. The use of this method allows optimum RAM allocation for global variables as it is automatically done with the OVERLAP segment for the parameters and local variables.

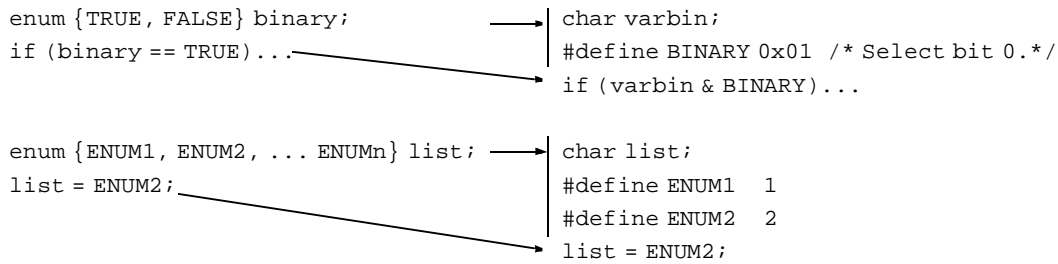


```
union { char var1;
        char var2; } varunion;
void main (void)
{ ... varunion.var1 = 10;
  ... varunion.var2 = 23; ... }
```

Note: the `var1` value is overwritten during the application execution by the `var2` one.

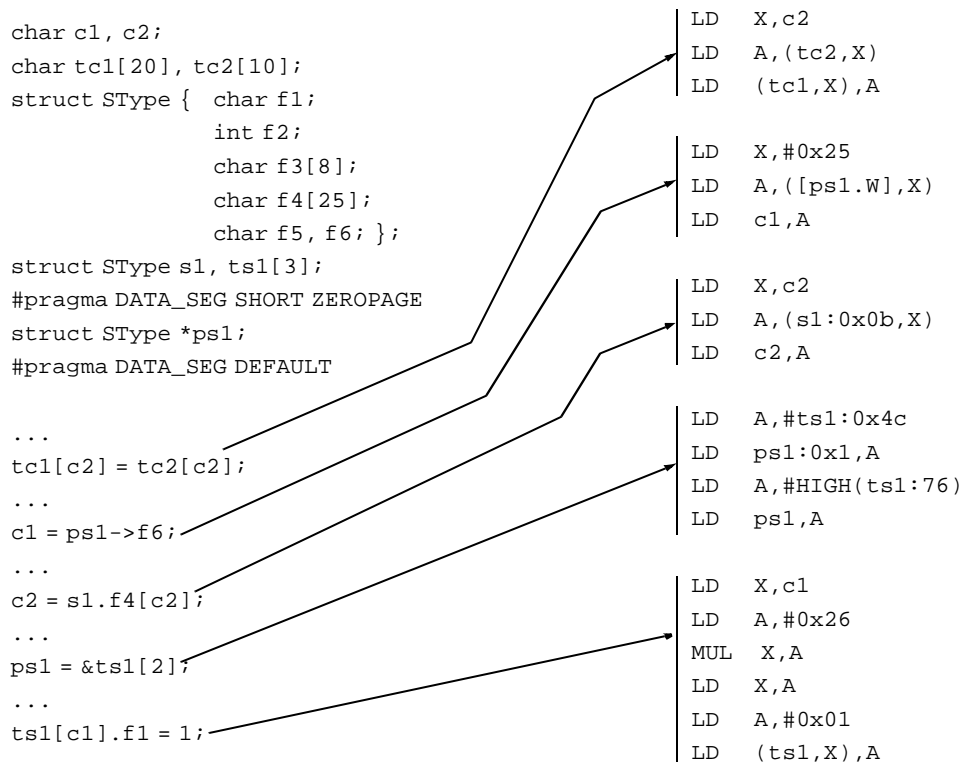
4.2.6 AVOID ENUM TYPE

In the ANSI C standard the `enum` type is based on the `int` type. This means that all enumerated elements are defined by two bytes for an ST7 target. For this reason and in order to generate optimum code, this `enum` type has to be avoided and replaced by preprocessor `#define` command (bit or byte format).

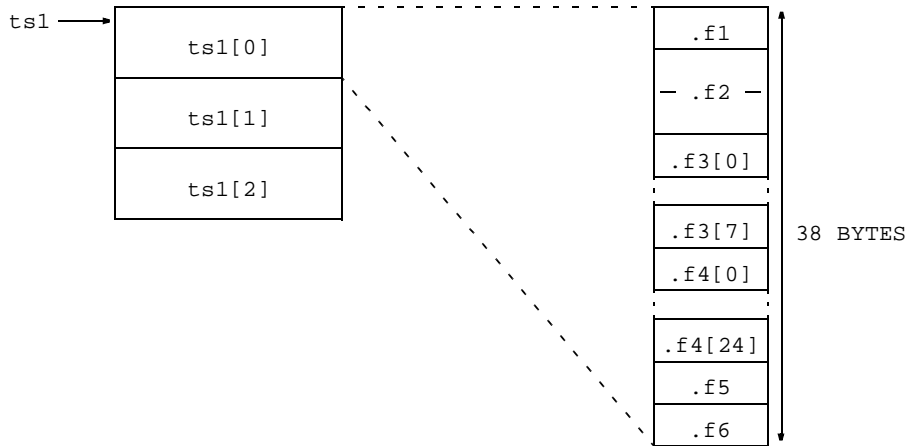


4.2.7 OPTIMUM TABLE AND STRUCTURE ACCESS

To get the optimum code size for this table and structure data, each data has to fit as much as possible into a 256-byte memory space to allow the compiler to use the powerful indexed memory access of the ST7 instruction set.



The structure and table memory alignment allocation is done according to the defined complex type without creating memory holes. Based on the previous example, the memory implementation of the `ts1` table is the following:



4.2.8 MASKED BYTE VARIABLES INSTEAD OF BITFIELDS

The ANSI C language provides a bitfield structure definition with direct access to the bit data through a syntactic combination. The ST7 C compiler optimizes the code and generates BRES, BSET, BTJT, BTJF and BCP instructions when using this bitfield structure (when it is allocated to page 0).

```
#pragma DATA_SEG SHORT ZEROPAGE
struct { unsigned intb0:1;
        unsigned intb1:1;
        unsigned intb2:1; } vbf;
#pragma DATA_SEG DEFAULT
...
vbf.b2 = 1;
if (vbf.b0)
    vbf.b1 = 0;
```

```
BSET vbf,#2
BTJF vbf,#0,label
BRES vbf,#1
label:
```

Note: for more details concerning the bit order assignment in the bitfield please refer to the “Hicross for ST7” manual.

But, as the ANSI C does not specify the bit assignment order in a bitfield structure, this structure is not suitable for generating portable microcontroller code especially when used for hardware register definition.

Then facing this limitation, the use of basic type such as `char` combined with mask expression is preferable. The main advantages are:

- optimum bit instruction code (bit instructions whenever possible and optimum)
- optimum byte instruction code (byte assignment when optimum)
- portability (bit assignment order defined by the programmer)

OPTIMIZING CODE IN ST7 C SOURCE MODULES

The following example illustrates these advantages and shows the purpose of the `-Onbf` compiler option. This option is mainly needed when handling hardware register bit where the setting and clearing sequences are critical.

```

#pragma DATA_SEG SHORT ZEROPAGE
char var;
#pragma DATA_SEG DEFAULT
...
var |= 0x20;
var &= 0xF7;
var |= 0x22;

```

CODE OPTIMIZED
WITHOUT THE `-Onbf`
COMPILER OPTION

1A 00	[5]	BSET vb,#5
17 00	[5]	BRES vb,#3
B6 00	[3]	LD A,vb
AA F2	[2]	OR A,#0xF2
B7 00	[4]	LD vb,A

The following examples describe two masking methods for bit handling. These two methods are based on preprocessor macros which generate optimum code when the data is located in page 0.

The first example is based on two parameter macros like the bit instruction set in the ST7 assembler. To get the optimum code each `VAR` variable has to be located in page 0.

MACRO NAME	MACRO C SOURCE CODE	GENERATED CODE
<code>#define SetBit(VAR, PLACE)</code>	<code>(VAR = (1 << PLACE))</code>	<code>BSET VAR,#PLACE</code>
<code>#define ClrBit(VAR, PLACE)</code>	<code>(VAR &= ((1 << PLACE) ^ 255)</code>	<code>BRES VAR,#PLACE</code>
<code>#define ValBit(VAR, PLACE)</code>	<code>(VAR & (1 << PLACE))</code>	<code>BTJT VAR,#PLACE, label</code>
		<code>BTJF VAR,#PLACE, label</code>

The second method is based on one parameter macro. This single parameter is the “bit address” in page 0 of the data to be manipulated. This bit address can be calculated using the following expression: `bit@ = (byte@ x 8) + (7 - bit_place)`

MACRO NAME	MACRO C SOURCE CODE	GENERATED CODE
<code>#define BitSet(BIT)</code>	<code>(*((unsigned char*) (BIT/8)) = ~(1<<(7-BIT%8))))</code>	<code>BSET ...,#...</code>
<code>#define BitClr(BIT)</code>	<code>(*((unsigned char*) (BIT/8)) &= (1<<(7-BIT%8)))</code>	<code>BRES ...,#...</code>
<code>#define BitVal(BIT)</code>	<code>(*((unsigned char*) (BIT/8)) & (1<<(7-BIT%8)))</code>	<code>BTJT ...,#...,label</code>
		<code>BTJF ...,#...,label</code>

4.2.9 OPTIMIZED INT TYPE

The ST7 is an 8 bit microcontroller so it is not optimized for `int` type variable handling (2 bytes). Nevertheless, all simple accesses to two byte variables are reduced to the minimum code size.

<pre>char c1; int i1, i2, ti1[10]; ... i1 = i2; ... i1 = ti1[c1];</pre>	<pre>LD A,i2:0x1 LD i1:0x1,A LD X,i2 LD i1,X</pre>	<pre>LD X,c1 SLL X LD A,(ti1:0x1,X) LD i1:0x1,A LD A,(ti1,X) LD i1,A</pre>
---	--	--

4.2.10 OPTIMUM POINTER ACCESS

As the ST7 instruction set allows indirect memory access only on pointers located in the page 0, these pointers have to be allocated as much as possible in the short access memory area to get the optimum generated code.

<pre>char c1, c2, *p1; #pragma DATA_SEG SHORT ZEROPAGE char *p2; #pragma DATA_SEG DEFAULT ... c1 = *p1; ... c2 = *p2;</pre>	<pre>C6 00 01[4] LD A,p1:0x1 B7 01 [4] LD _LEX:0x1,A CE 00 00[4] LD X,p1 BF 00 [4] LD _LEX,X 92 C6 00[6] LD A,[_LEX.W] C7 00 00[5] LD c1,A</pre>
<p style="text-align: center;">p2 in PAGE 0: OPTIMUM</p>	<pre>92 C6 00[6] LD A,[p2.W] C7 00 00[5] LD c2,A</pre>

4.2.11 BASIC TYPE CONVERSION

In ST7 embedded applications, it is usual to convert basic types such as `int` to `char` and conversely. Here are some ways of converting these basic types in C language:

<pre>#define LSB(WORD)WORD #define MSB(WORD)WORD>>8 #define WORD(MSB,LSB)(int) ((int) MSB << 8) + LSB char msb, lsb; int word1, word2; ... lsb = LSB(word1); msb = MSB(word1); word2 = WORD(msb, lsb);</pre>	<pre>LD A,word1:0x1 LD lsb,A LD X,A ... LD A,word1 LD msb,A ... LD word2:0x1,X LD word2,A</pre>
--	---

Note: The same conversion can be applied to 32 bit types keeping an optimum generated code.

OPTIMIZING CODE IN ST7 C SOURCE MODULES

4.2.12 NO OPTIMIZATION: REGISTER, AUTO.

As the ST7 is an accumulator-based machine, the ANSI C `register` and `auto` qualifiers have no meaning. In fact, the compiler optimizes the code using the A, X and Y CPU registers as much as possible.

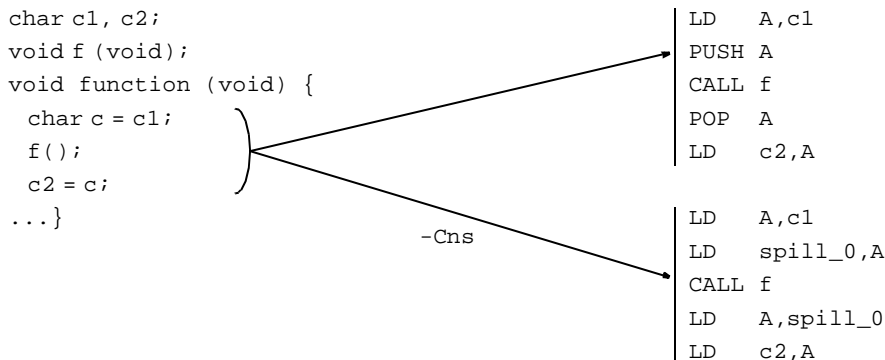
4.3 TEMPORARY COMPILER VARIABLES

For some complex C expressions, the compiler has to use temporary variables to solve the operations. These temporary variables can be:

- ST7 CPU registers (A, X or Y)
- Hiware software registers (`_SEX`, `_LEX` or `_R_Z`)
- stack locations (using PUSH and POP instructions)
- `spills` temporary variables (`spill_0`, `spill_1`...)

The `spills` temporary variables are managed by the compiler like the local variables and the parameters, which means that they are allocated to the OVERLAP memory segment.

Note: For specific applications where stack management is critical, the compiler `-Cns` option has to be used to avoid the stack location being used for temporary storage.



4.4 OPTIMUM CONSTANT DEFINITIONS

In an embedded microcontroller application two kinds of constant definitions exist:

- the simple constant values
- the data constant structure (fixed data lists, function pointer tables...)

Both of these constants have to be allocated to the ROM area (see Section 2.4).

When a simple constant value has to be used in an expression, it always has to be defined through the preprocessor `#define` command and not through a `const` variable to avoid ROM space being unnecessarily allocated to constant values.

<pre>#define CONST1 10 const char CONST2 = 10; char c1; ... c1 = CONST1; ... c1 = CONST2;</pre>		<pre>A6 03 [2] LD A,#0x0A B7 00 00 [5] LD c1,A</pre>
---	--	--

On the other hand, for data constants which have to be accessed like tables or for strings which have to be used several time during an application, using the `const` variable is the optimum solution.

<pre>const char CONST[] = {1,2,3}; char c1, c2; ... c1 = CONST[1]; ... c1 = CONST[c2];</pre>		<pre>A6 02 [2] LD A,#0x02 B7 00 00 [5] LD c1,A ... CE 00 00 LD X,c2 D6 00 00 LD A,(CONST,X) C7 00 00 LD c1,A</pre>
--	--	---

Note: the ST7 C compiler generates the most optimum code either using the `#define` command or a `const` variable. When a `const` variable is used, it is allocated as data only when it is mandatory (indexed access...) else the compiler replaces the constant value directly in the code.

4.5 OPTIMUM C STATEMENT MANAGEMENT

4.5.1 NON-RELEVANT C SOURCE CODE

4.5.1.1 UNREACHABLE CODE

The ST7 C compiler does some automatic optimization for unreachable code which is usually a sign of a programmer error.

<pre>char c1, c2; ... c1=2; c1=3; ... goto label; c1=4; label: c2=5;</pre>		<pre>A6 03 [2] LD A,#0x03 B7 00 00 [5] LD c1,A ... A6 05 [2] LD A,#0x05 B7 00 00 [5] LD c2,A</pre>
--	--	--

4.5.1.2 REDUNDANT CODE

The ST7 C compiler does some automatic optimization for redundant code which is usually a sign of a programmer error.

```

char c1, c2;
...
c1 + c2;
...
c1 += 0;
...
c1 |= c1;

```

NO CODE GENERATION

Compiler WARNING message.

4.5.2 BYTE SIZE OBJECT OPERATIONS

The main byte size object in the C language for ST7 is the basic standard `char` type.

4.5.2.1 OPTIMUM BASIC OPERATIONS

The following example illustrates the efficiency of the byte object used in basic C expressions.

```

char c1, c2, c3;
...
c1 = c2;
...
c1 = c2 + c3;
...
c1 = c2 - c3;
...
c1 = c2 * c3;

```

```

LD A,c2
LD c1,A
LD A,c2
ADD A,c3
LD c1,A
LD A,c2
SUB A,c3
LD c1,A
LD A,c3
LD X,c2
MUL X,A
LD c1,A

```

4.5.2.2 BYTE DIVISION

As the ST7 instruction set does not provide a division capability, the application has to be written as much as possible with power of 2 divisions in order to get the optimum generated code with shift instructions. Otherwise the generated code has to call the Hicross library and with a resulting increase in ROM size needs.

```

char c1, c2, c3;
...
c1 = c2 / 64;
...
c1 = c2 / 63;

```

```

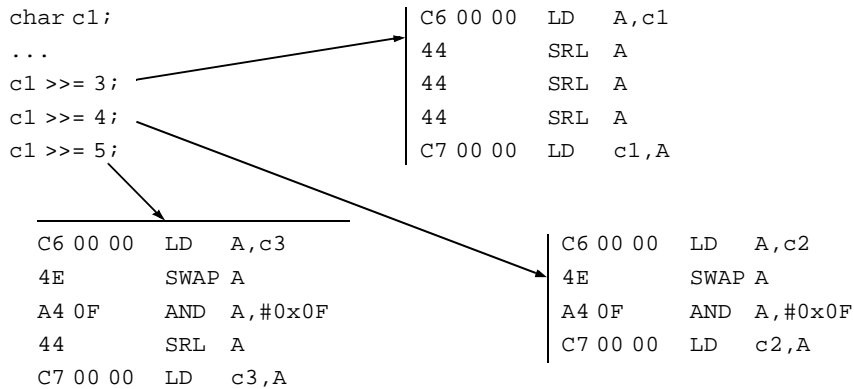
LD A,c2
SWAP A
AND A,#0x0F
SRL A
SRL A
LD c1,A
LD A,c3
LD X,#0x3F
CALL _BDIVU
LD c2,A

```

Note: For more details on the Hicross library, please refer to the “Hicross for ST7” manual.

4.5.2.3 OPTIMUM BYTE SHIFT

Inside an ST7 embedded application, it is usual to use shift operations. The ST7 C compiler provides optimum code generation using `SWAP` and `AND` instructions instead of sequential `SRL` and `SLL` ones when it is the best choice.

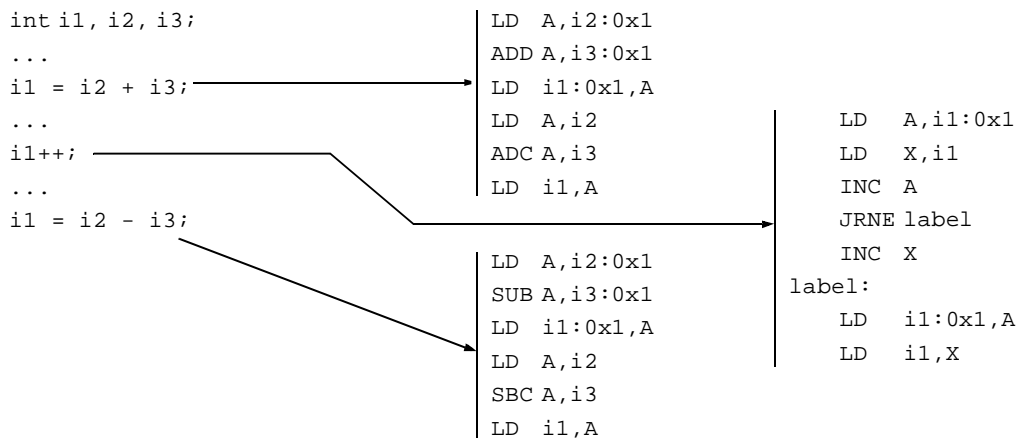


4.5.3 WORD SIZE OBJECT OPERATION

The basic standard `int` type is the main word size object (two bytes) in the ST7 C language.

4.5.3.1 OPTIMUM BASIC OPERATIONS

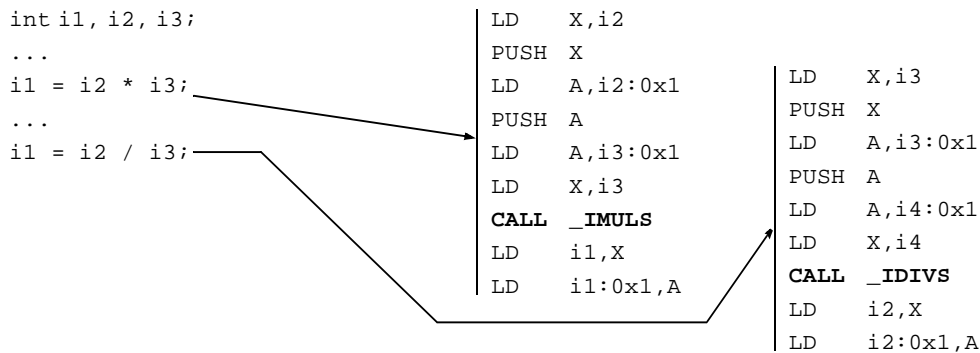
For basic operations on `int` types like additions and subtractions, the corresponding resources in the ST7 (`ADD`, `ADC`, `SUB`, `SUBC` instructions) provide an optimum generated code.



Note: Either for integer addition or subtraction the least significant byte is always affected first. This can cause some problems in some critical hardware register settings (16-bit timer counter registers).

4.5.3.2 WORD MULTIPLICATION AND DIVISION

Complex word operations such as multiplication and division need to use the Hicross library to solve the requested expression as the ST7 CPU does not provide hardware resources for these purposes.



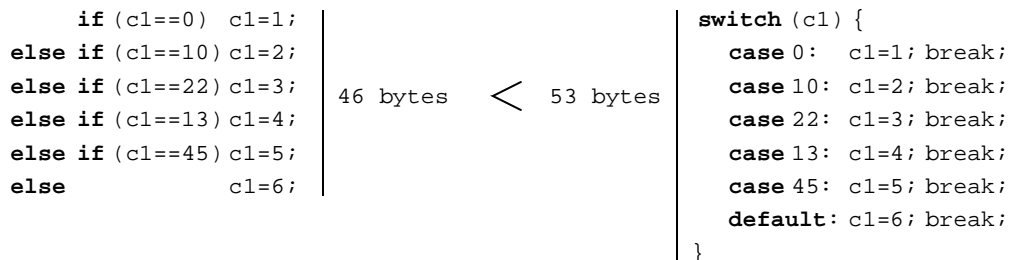
Note: For more details on the Hicross library, please refer to the “Hicross for ST7” manual.

4.5.4 CONDITIONAL STATEMENTS

4.5.4.1 IF STATEMENT INSTEAD OF SWITCH

The ANSI C language provides two control statements which can be used for the same sequential test purpose: IF/ELSE and SWITCH/CASE.

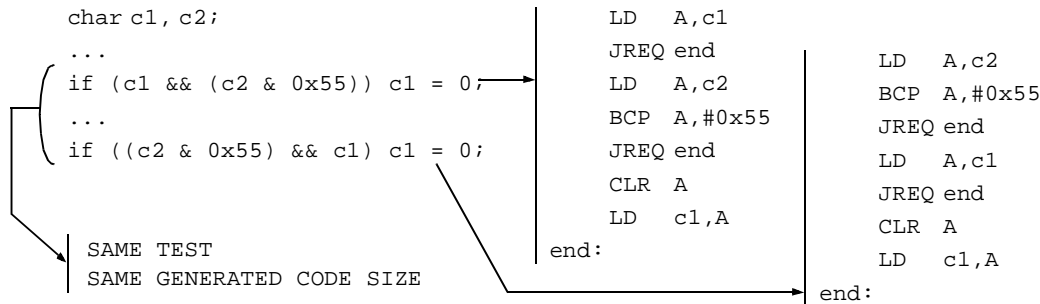
The Hiware C compiler is better optimized for IF/ELSE statements than for SWITCH/CASE statements so use IF/ELSE as much as possible, also for sequential testing.



Note: Using the IF/ELSE statement also has the advantage of specifying range conditions.

4.5.4.2 TEST CONDITION ORDER

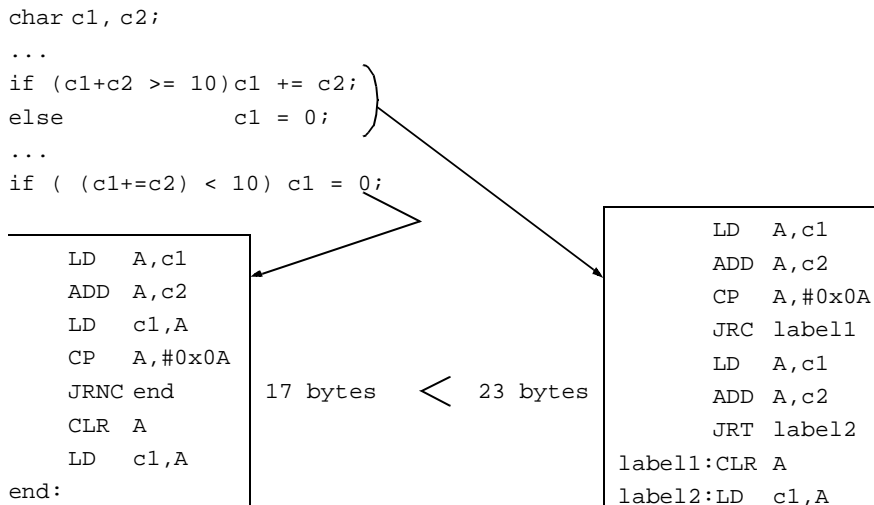
In terms of execution speed optimization, the test condition with the highest probability to be false has always to be considered first. The second rule to take into account is to place the fastest test conditions first.



Note: this rule has to be applied for IF/ELSE statements but also for all statements conditioned by a test (FOR, WHILE...).

4.5.4.3 IF STATEMENTS AND ASSIGNMENTS

To get the optimum generated code, try as much as possible to include the assignment in the condition.



Note: This rule has to be applied for IF/ELSE statements but also for all statements conditioned by a test (FOR, WHILE...).

4.6 OPTIMUM FUNCTION MANAGEMENT

4.6.1 FUNCTIONS vs MACROS

In term of generated code size for short expression sequences, it is usually better to define a preprocessor macro instead of a function (such as variable get/clear functions). Given this condition a compromise between generated code size and legibility has to be selected according to priorities depending on the application.

In the first example that follows, the “macro” solution has been chosen. The major advantage is that the generated code size is minimized each time the `var` variable is accessed. The drawback is that there is no security concerning the access of the `var` variable from an other module.

```
FILE.H | extern char var;           FILE.C | char var;
        | #define GET_Value  var   |      | ...
```

On the contrary, in the second example, the “function” solution has been chosen. The major advantage is that the `var` variable is protected against unwanted direct access from another module (declared `static` in the `file.c` module). The drawback is related to the `LD`, `CALL` and `RET` instructions (code size and execution time increased):

- In total the generated code size is increased by 4 bytes (`LD`, `RET`)
- Each access to the `var` variable is increased by 12 cycles (`CALL`, `RET`).

The above data assumes that the `var` variable is allocated outside page 0.

```
FILE.H | extern char Get_Value(void); FILE.C | static char var;
        |                               |      | ...
        |                               |      | char Get_Value(void)
        |                               |      | { return(var); }
        |                               |      | ...
```

4.6.2 FUNCTION PARAMETERS AND RETURN VALUE

As the A and X registers of the ST7 are used for parameter passing and the function return value, the C source which generates the optimum code has to contain as many functions as possible with a maximum of two byte size parameters and two byte size return value.

```
char fct1(int p0) {...}
int fct2(char p1, char p0) {...}
```

4.6.3 FUNCTION PARAMETER DECLARATION ORDER

To optimize the use of the A and X registers for parameter passing, the order of the parameters in the parameter definition list has to be reversed compared to the order in which they are used in the function.

The following two examples illustrate the importance of the parameter declaration order.

- In the first example, the code is optimum with parameter passing through A and X registers for p1 and p0.

char fctparam1 (char p2, char p1, char p0)	42	MUL X,A
{ c1 = p0 * p1;	C7 00 00	LD c1,A
return(p2); }	B6 00	LD A,p2

- In the second example, the source code is the same as the previous one, only the parameter declaration order has changed. The result of this move is a drastic increase of the generated code size (more than 130%) and of the OVERLAP segment usage (2 more bytes).

char fctparam2 (char p1, char p0, char p2)	BF 00	LD p0,X
{ c1 = p0 * p1;	B7 00	LD p2,A
return(p2); }	B6 00	LD A,p0
	BE 00	LD X,p1
	42	MUL X,A
	C7 00 00	LD c1,A
	B6 00	LD A,p2

CONCLUSION

5 CONCLUSION

The ST7 Hiware C compiler is a powerful tool for developing ST7 embedded application in C language. Following the rules described in this document provides a major added value to generated the optimum C code for an ST7 microcontroller.

THE PRESENT NOTE WHICH IS FOR GUIDANCE ONLY AIMS AT PROVIDING CUSTOMERS WITH INFORMATION REGARDING THEIR PRODUCTS IN ORDER FOR THEM TO SAVE TIME. AS A RESULT, STMICROELECTRONICS SHALL NOT BE HELD LIABLE FOR ANY DIRECT, INDIRECT OR CONSEQUENTIAL DAMAGES WITH RESPECT TO ANY CLAIMS ARISING FROM THE CONTENT OF SUCH A NOTE AND/OR THE USE MADE BY CUSTOMERS OF THE INFORMATION CONTAINED HEREIN IN CONNEXION WITH THEIR PRODUCTS.

Information furnished is believed to be accurate and reliable. However, STMicroelectronics assumes no responsibility for the consequences of use of such information nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of STMicroelectronics. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. STMicroelectronics products are not authorized for use as critical components in life support devices or systems without the express written approval of STMicroelectronics.

The ST logo is a registered trademark of STMicroelectronics

©1999 STMicroelectronics - All Rights Reserved.

Purchase of I²C Components by STMicroelectronics conveys a license under the Philips I²C Patent. Rights to use these components in an I²C system is granted provided that the system conforms to the I²C Standard Specification as defined by Philips.

STMicroelectronics Group of Companies

Australia - Brazil - Canada - China - France - Germany - Italy - Japan - Korea - Malaysia - Malta - Mexico - Morocco - The Netherlands - Singapore - Spain - Sweden - Switzerland - Taiwan - Thailand - United Kingdom - U.S.A.

<http://www.st.com>