APPLICATION NOTE

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.

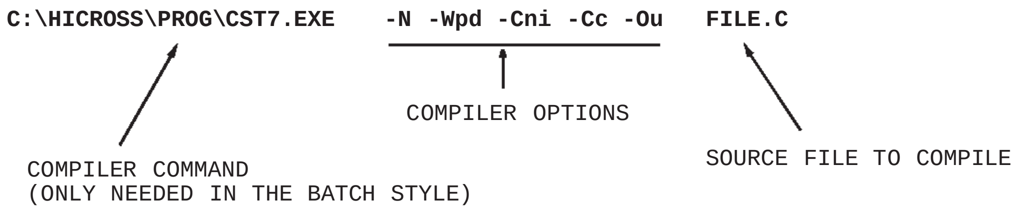
This application note is based on the ST7 Hiware C compiler but some examples can be applied to other ST7 C compilers. It is not exhaustive and for more details concerning the ST7 Hiware C compiler please refer to the “HICROSS for ST7” manual.

Table of Contents

Table of Contents

1 COMPILER OVERVIEW AND SYNTAX

The ST7 Hiware C compiler is an application which can be launched either through a batch command (from the make file for instance) or an interactive window interface. In both cases the command line has the following structure:



```
C:\HICROSS\PROG\CST7.EXE    -N -Wpd -Cni -Cc -Ou   FILE.C
```

Note: If in the active “default.env” file the COMPOPTIONS variable is filled with default compiler options, these options are taken into account for each compilation.

2 WRITING C FOR ST7

2.1 BASIC TYPE SIZE

As the ANSI C specifies that the size of the basic types is given by the architecture of the target microcontroller, the following table gives the default format for the ST7.

Type	**Default format**	**Default value range**		**Format available with -T option**
		min	**max**	
(unsigned) char	8bit	0	255	8bit, 16bit, 32bit see the “Hicross for ST7” manual for more details option
signed char		-128	127	
unsigned int	16bit	0	65535	
(signed) int		-32768	32767	
unsigned short		0	65535	
(signed) short		-32768	32767	
enum		-32768	32767	
unsigned long	32bit	0	4294967295	
(signed) long		-2147483648	2147483647	
float, double	IEEE32	*1.17...E-38F*	*3.40...E+38F*	IEEE32

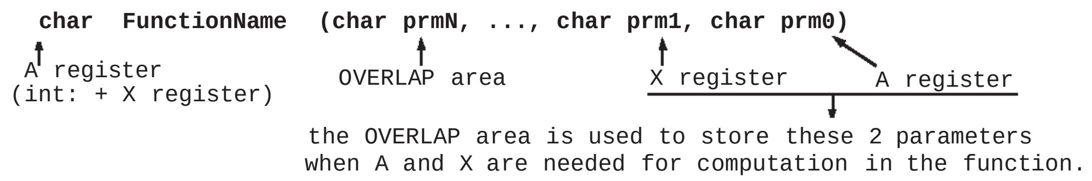
2.2 LOCAL VARIABLES AND PARAMETERS STRATEGY

Usually in most compilers the local variables and parameters are stored in the stack. To generate the most efficient code for the ST7 architecture, the Hiware C compiler does not use the stack for this purpose. A dedicated RAM area (called OVERLAP) where the data are overlapped as much as possible with reference to the execution structure is preferred.

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

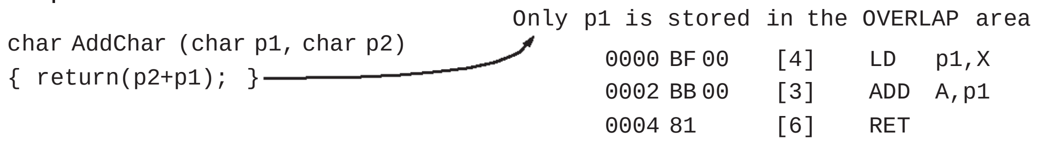
As it is described in the previous paragraph, by default the function parameters are stored in the dedicated OVERLAP memory area. But, the last two bytes in the parameter list are also always passed through ST7 hardware registers (A and X). The two least significant bytes of the return value are also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



```
char  FunctionName  (char prmN, ..., char prm1, char prm0)
```

A register
(int: + X register)
OVERLAP area
X register
A register
the OVERLAP area is used to store these 2 parameters
when A and X are needed for computation in the function.

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); }
```

```
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
-Ml	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 constant 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) {...}            PLACEMENT DECLARED IN THE PRM FILE
...
#pragma CODE_SEG DEFAULT
void FctName2 (void) {...}
```

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.

```
char * near FunctionName (char * near ptr)
{ return ((char * near) ptr +1); }
```

```
With near keyword:
   0000 4C     [3]  INC  A
   0001 81     [6]  RET
```

```
Without near keyword:
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
```

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.

```
char * far FunctionName (char * far ptr)
{ return ((char * far) ptr +1); }
```

```
Without far keyword:
   0000 4C     [3]  INC  A
   0001 81     [6]  RET
```

```
With far keyword:
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
```

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.

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

```
Generated code: ...
                IRET
```

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              | 90 89   [4]   PUSH  Y
void InterruptFct (void) {...}           | B6 00   [3]   LD    A,_R_Z
                                         | 88      [3]   PUSH  A
                                         | B6 01   [3]   LD    A,_LEX:0x1
            INTERRUPT PROLOG  ------->   | 88      [3]   PUSH  A
            + 17 bytes                   | B6 00   [3]   LD    A,_LEX
            + 34 cycles                  | 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
            INTERRUPT EPILOG  ------->   | 84      [4]   POP   A
            + 17 bytes                   | B7 00   [4]   LD    _LEX,A
            + 45 cycles                  | 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
```

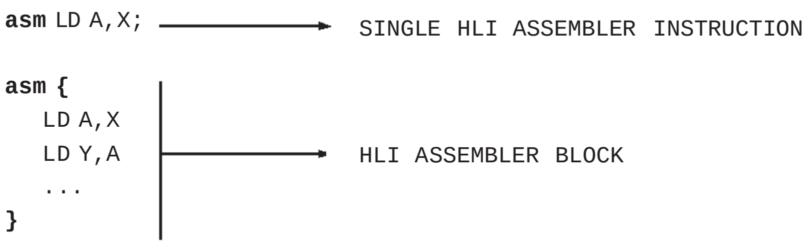
Note: The drawback of using the SAVE_REGS pragma is that the code size is increased by 34 bytes, the interrupt latency increased by 34 cycles (4.25µs with f_{CPU}=8MHz) and the total duration of the interrupt routine execution increased by 79 cycles (9.875µs with f_{CPU}=8MHz).

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.



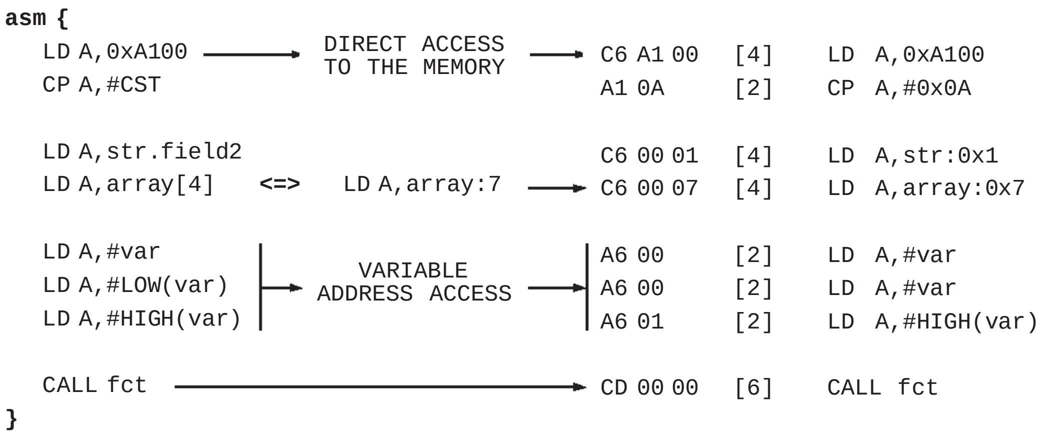
```
asm LD A,X;                  SINGLE HLI ASSEMBLER INSTRUCTION

asm {
   LD A,X
   LD Y,A                    HLI ASSEMBLER BLOCK
   ...
}
```

2.7.2 HLI ASSEMBLER AND C LANGUAGE

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

```
#define CST  10
struct {char field1; char field2;} str;
char var, array[10];
...
```



```
asm {
   LD A,0xA100          DIRECT ACCESS        C6 A1 00   [4]    LD  A,0xA100
   CP A,#CST            TO THE MEMORY        A1 0A      [2]    CP  A,#0x0A

   LD A,str.field2                           C6 00 01   [4]    LD  A,str:0x1
   LD A,array[4]   <=>   LD A,array:7        C6 00 07   [4]    LD  A,array:0x7

   LD A,#var                                 A6 00      [2]    LD  A,#var
   LD A,#LOW(var)         VARIABLE           A6 00      [2]    LD  A,#var
   LD A,#HIGH(var)     ADDRESS ACCESS        A6 01      [2]    LD  A,#HIGH(var)

   CALL fct                                  CD 00 00   [6]    CALL fct
}
```

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
...                                  A6 0A   [2]   LD  A,#0x0A
asm LDA10;
```

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 `-Wpd` option to prevent extern definition omissions.

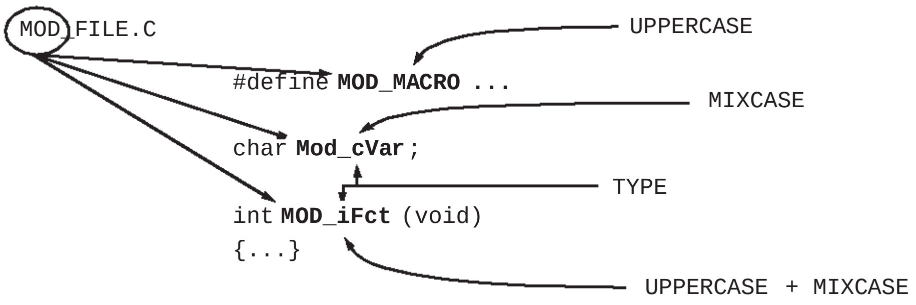
To prevent `#include` recursions, all header files have to be preprocessed. To avoid the duplication of declarations in several files (*.h and *.c), macro definitions and variable declarations have to be defined in the header file with `extern` preprocessing.

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

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:

```
                                              | C6 00 00  [4]    LD   A,c1
int i1; char c1, c2;                          | CB 00 00  [4]    ADD  A,c2
...                     THE BYTE ADDITION     | C7 00 01  [5]    LD   i1:0x1,A
i1 = c1 + c2;  -------  CARRY IS IGNORED  --> | 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.

```
                                          | AD 00   [6]    CALLR fctcall
void function (char c1)                   | A6 01   [2]    LD    A,#0x01
{   fctcall();                     -----> | B7 00   [4]    LD    c1,A
    c1 =1;                                | 81      [6]    RET
}
                               -Ou -----> 
                                            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.

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

Note: The `-Ol` 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).

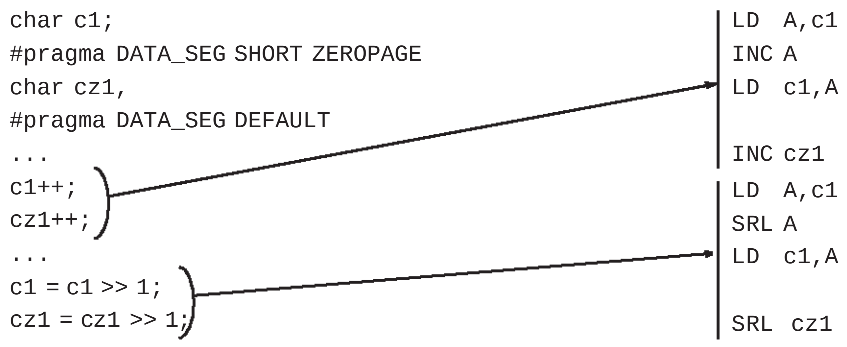
As the ST7 instruction set provides optimum bit handling instructions (BRES, BSET, BTJT, BTJF and BCP), the software has to use this structure as much as possible for binary data (this data has always to be located in the page 0). For other data types, `char` is the preferred data type to get the optimum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



```
char c1;
#pragma DATA_SEG SHORT ZEROPAGE
char cz1,
#pragma DATA_SEG DEFAULT
...
c1++;
cz1++;
...
c1 = c1 >> 1;
cz1 = cz1 >> 1;
```

```
LD  A,c1
INC A
LD  c1,A

INC cz1
```

```
LD  A,c1
SRL A
LD  c1,A

SRL  cz1
```

- 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 PADR;
~~volatile~~ char PADDR;
```

PADR 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.

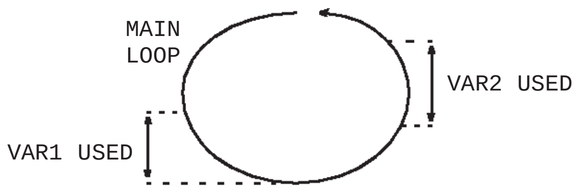
In the case of local variable definitions, this qualifier has to be used only for permanent data (not to be overlapped). Do not use this `static` qualifier when it is not needed to avoid removing the overlap optimization for local variables and increasing the used RAM size.

```
void fct1 (void)
{   static char var;
    var++;
    ...                        PERMANENT
}                                LIFE
                               (PROGRAM)
```

```
void fct2 (void)
{   ~~static~~ char var;
    var=0;
    ...                        TEMPORARY
}                                LIFE
                              (FUNCTION)
```

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).

```
enum {TRUE, FALSE} binary;
if (binary == TRUE)...
```

```
char varbin;
#define BINARY 0x01 /* Select bit 0.*/
if (varbin & BINARY)...
```

```
enum {ENUM1, ENUM2, ... ENUMn} list;
list = ENUM2;
```

```
char list;
#define ENUM1  1
#define ENUM2  2
list = ENUM2;
```

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.

```
char c1, c2;
char tc1[20], tc2[10];
struct SType {  char f1;
                int f2;
                char f3[8];
                char f4[25];
                char f5, f6; };
struct SType s1, ts1[3];
#pragma DATA_SEG SHORT ZEROPAGE
struct SType *ps1;
#pragma DATA_SEG DEFAULT

...
tc1[c2] = tc2[c2];
...
c1 = ps1->f6;
...
c2 = s1.f4[c2];
...
ps1 = &ts1[2];
...
ts1[c1].f1 = 1;
```

```
LD   X,c2
LD   A,(tc2,X)
LD   (tc1,X),A
```

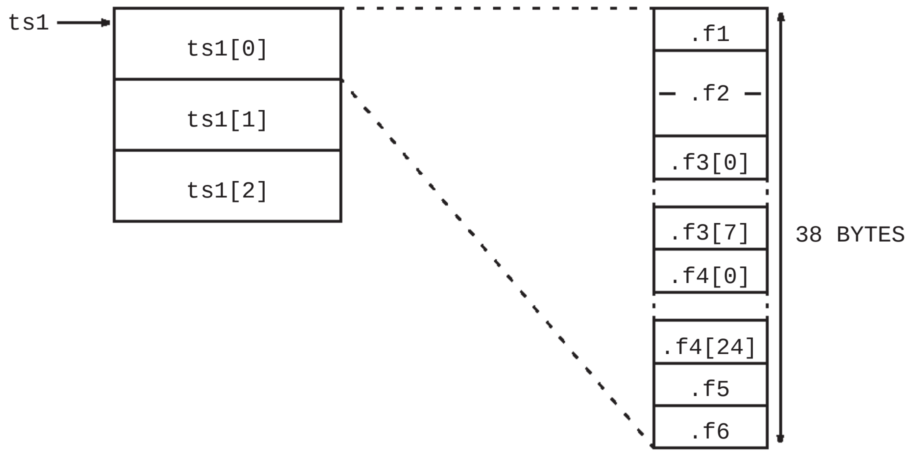
```
LD   X,#0x25
LD   A,([ps1.W],X)
LD   c1,A
```

```
LD   X,c2
LD   A,(s1:0x0b,X)
LD   c2,A
```

```
LD   A,#ts1:0x4c
LD   ps1:0x1,A
LD   A,#HIGH(ts1:76)
LD   ps1,A
```

```
LD   X,c1
LD   A,#0x26
MUL   X,A
LD   X,A
LD   A,#0x01
LD   (ts1,X),A
```

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)

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                       ~~1A 00   [5]   BSET vb,#5~~
char var;
#pragma DATA_SEG DEFAULT                             | 17 00   [5]   BRES vb,#3
...                                                  |
var |= 0x20;                      CODE OPTIMIZED     |
var &= 0xF7;                  WITHOUT THE -Onbf      | B6 00   [3]   LD   A,vb
var |= 0x22;                   COMPILER OPTION       | 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
#define SetBit(VAR, PLACE)	(VAR |= (1 << PLACE))	BSET VAR,#PLACE
#define ClrBit(VAR, PLACE)	(VAR &= ((1 << PLACE) ^ 255)	BRES VAR,#PLACE
#define ValBit(VAR, PLACE)	(VAR & (1 << PLACE))	BTJT VAR,#PLACE, label
		BTJF VAR,#PLACE, label

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
#define BitSet(BIT)	(*((unsigned char*) (BIT/8)) |= (~(1<<(7-BIT%8))))	BSET ...,#...
#define BitClr(BIT)	(*((unsigned char*) (BIT/8)) &= (1<<(7-BIT%8)))	BRES ...,#...
#define BitVal(BIT)	(*((unsigned char*) (BIT/8)) & (1<<(7-BIT%8)))	BTJT ...,#...,label
		BTJF ...,#...,label

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.

```
char c1;
int i1, i2, ti1[10];
...
i1 = i2;
...
i1 = ti1[c1];
```

```
LD  A,i2:0x1
LD  i1:0x1,A
LD  X,i2
LD  i1,X
```

```
LD  X,c1
SLL X
LD  A,(ti1:0x1,X)
LD  i1:0x1,A
LD  A,(ti1,X)
LD  i1,A
```

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.

```
char c1, c2, *p1;
#pragma DATA_SEG SHORT ZEROPAGE
char *p2;
#pragma DATA_SEG DEFAULT
...
c1 = *p1;
...
c2 = *p2;
```

```
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
```

p2 in PAGE 0: OPTIMUM

```
92 C6 00[6]   LD    A,[p2.W]
C7 00 00[5]   LD    c2,A
```

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:

```
#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);
```

```
LD  A,word1:0x1
LD  lsb,A
LD  X,A

LD  A,word1
LD  msb,A

LD  word2:0x1,X
LD  word2,A
```

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

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.

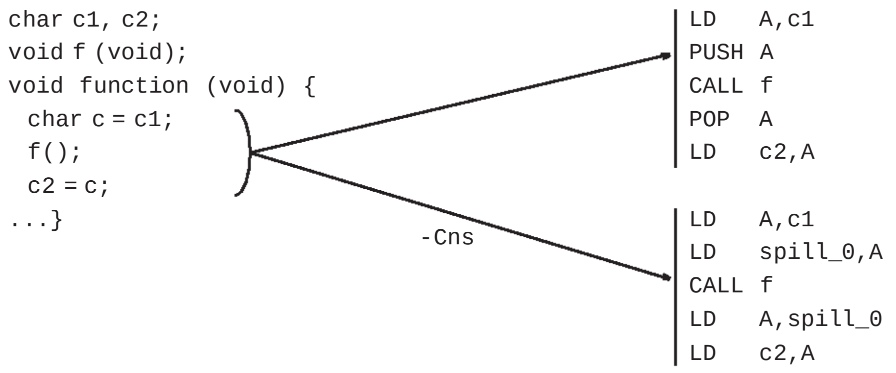
4.3 TEMPORARY COMPILER VARIABLES

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.



```
char c1, c2;
void f (void);
void function (void) {
  char c = c1;
  f();
  c2 = c;
...}
```

```
LD   A,c1
PUSH A
CALL f
POP  A
LD   c2,A
```

```
LD   A,c1
LD   spill_0,A
CALL f
LD   A,spill_0
LD   c2,A
```

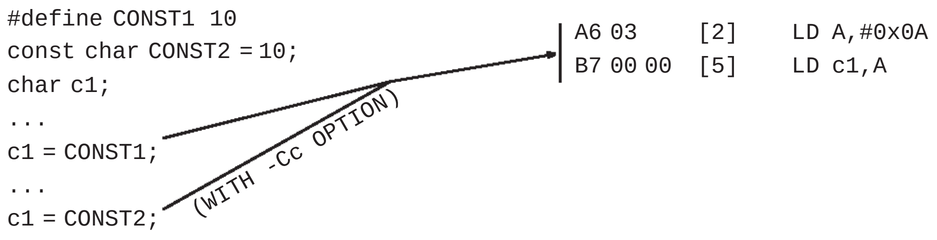
4.4 OPTIMUM CONSTANT DEFINITIONS

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...)

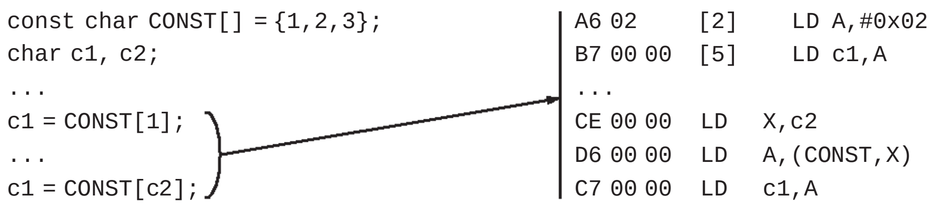
Both of these constants have to be allocated to the ROM area (see Section 2.4).

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.



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

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.



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

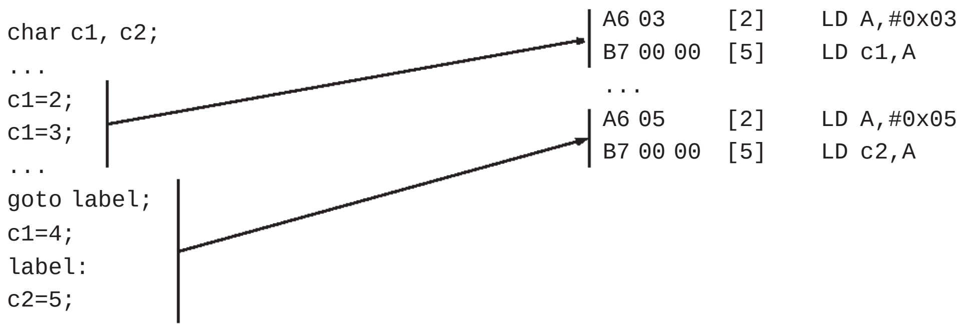
Note: the ST7 C compiler generates the most optimum code either using the `#define` command or a `const` variable. When a `const` variable is used, it is allocated as data only when it is mandatory (indexed access...) else the compiler replaces the constant value directly in the code.

4.5 OPTIMUM C STATEMENT MANAGEMENT

4.5.1 NON-RELEVANT C SOURCE CODE

4.5.1.1 UNREACHABLE CODE

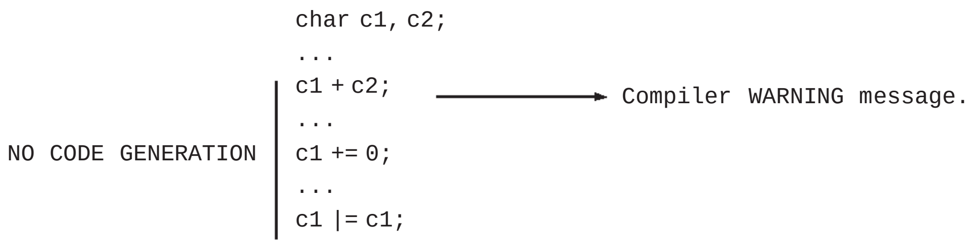
The ST7 C compiler does some automatic optimization for unreachable code which is usually a sign of a programmer error.



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

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.



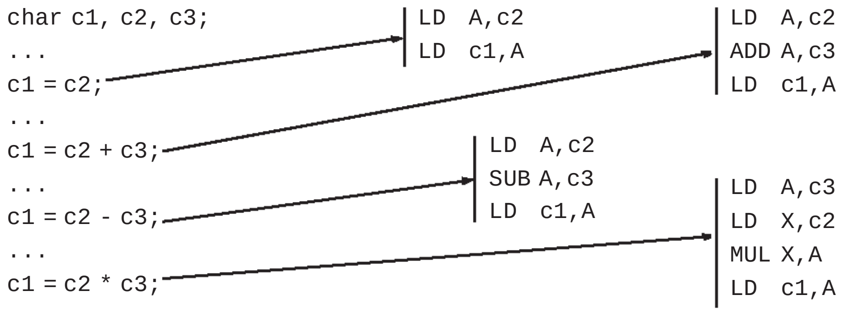
```
                    char c1, c2;
                    ...
                    c1 + c2;        ------>  Compiler WARNING message.
                    ...
NO CODE GENERATION  c1 += 0;
                    ...
                    c1 |= c1;
```

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

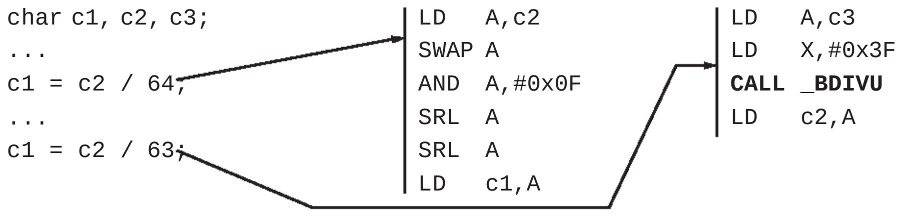
The following example illustrates the efficiency of the byte object used in basic C expressions.



```
char c1, c2, c3;
...
c1 = c2;          LD  A,c2
                  LD  c1,A
...
c1 = c2 + c3;     LD  A,c2
                  ADD A,c3
                  LD  c1,A
...
c1 = c2 - c3;     LD  A,c2
                  SUB A,c3
                  LD  c1,A
...
c1 = c2 * c3;     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.

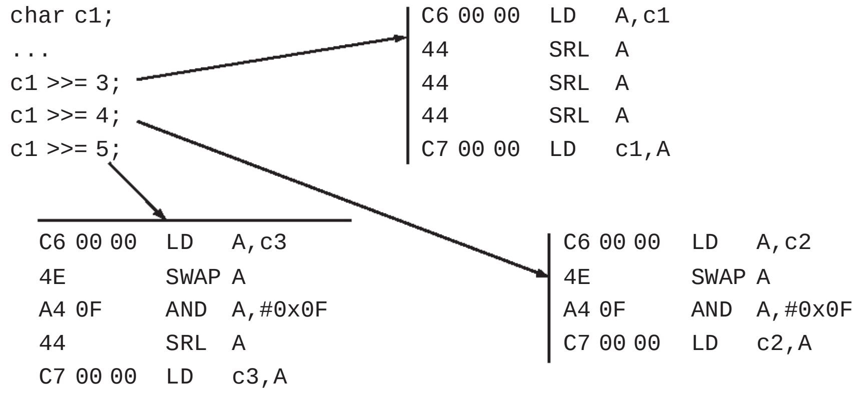


```
char c1, c2, c3;
...
c1 = c2 / 64;     LD   A,c2
                  SWAP A
                  AND  A,#0x0F
                  SRL  A
                  SRL  A
                  LD   c1,A
...
c1 = c2 / 63;     LD   A,c3
                  LD   X,#0x3F
                  CALL _BDIVU
                  LD   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.



```
char c1;
...
c1 >>= 3;
c1 >>= 4;
c1 >>= 5;
```

```
C6 00 00  LD   A,c1
44        SRL  A
44        SRL  A
44        SRL  A
C7 00 00  LD   c1,A
```

```
C6 00 00  LD   A,c3
4E        SWAP A
A4 0F     AND  A,#0x0F
44        SRL  A
C7 00 00  LD   c3,A
```

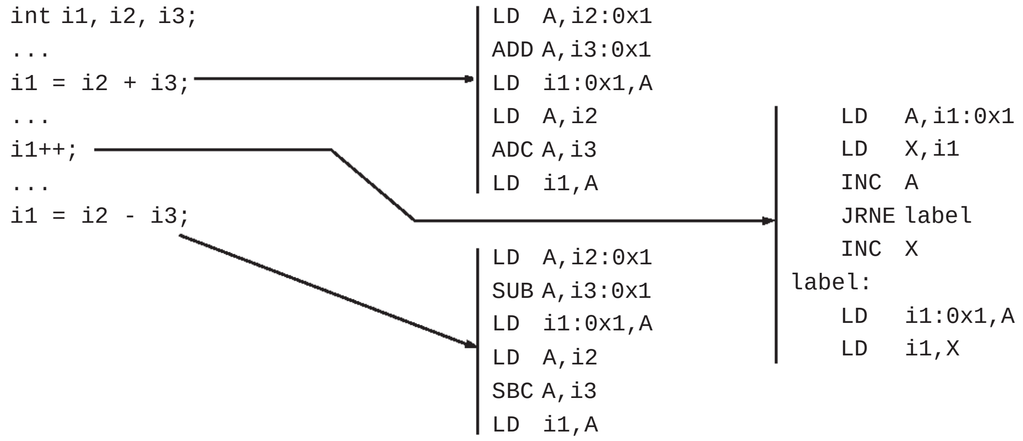
```
C6 00 00  LD   A,c2
4E        SWAP A
A4 0F     AND  A,#0x0F
C7 00 00  LD   c2,A
```

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.



```
int i1, i2, i3;
...
i1 = i2 + i3;
...
i1++;
...
i1 = i2 - i3;
```

```
LD  A,i2:0x1
ADD A,i3:0x1
LD  i1:0x1,A
LD  A,i2
ADC A,i3
LD  i1,A
```

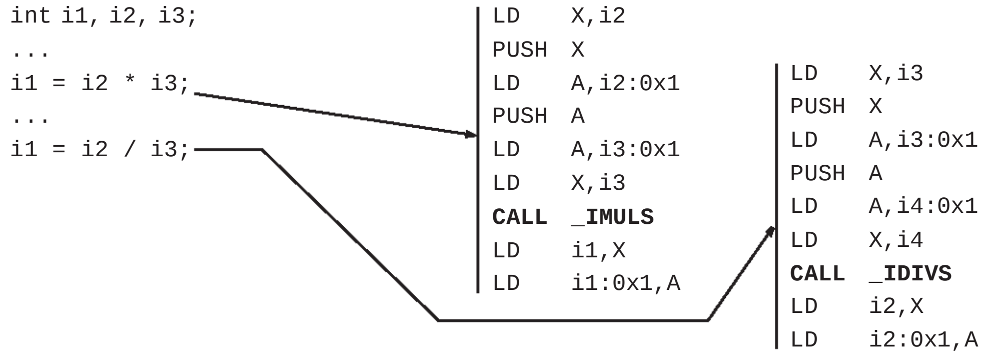
```
    LD   A,i1:0x1
    LD   X,i1
    INC  A
    JRNE label
    INC  X
label:
    LD   i1:0x1,A
    LD   i1,X
```

```
LD  A,i2:0x1
SUB A,i3:0x1
LD  i1:0x1,A
LD  A,i2
SBC A,i3
LD  i1,A
```

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.



```
int i1, i2, i3;
...
i1 = i2 * i3;
...
i1 = i2 / i3;
```

```
LD    X,i2
PUSH  X
LD    A,i2:0x1
PUSH  A
LD    A,i3:0x1
LD    X,i3
CALL  _IMULS
LD    i1,X
LD    i1:0x1,A
```

```
LD    X,i3
PUSH  X
LD    A,i3:0x1
PUSH  A
LD    A,i4:0x1
LD    X,i4
CALL  _IDIVS
LD    i2,X
LD    i2:0x1,A
```

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.

```
     if (c1==0)  c1=1;
else if (c1==10) c1=2;
else if (c1==22) c1=3;
else if (c1==13) c1=4;
else if (c1==45) c1=5;
else             c1=6;
```

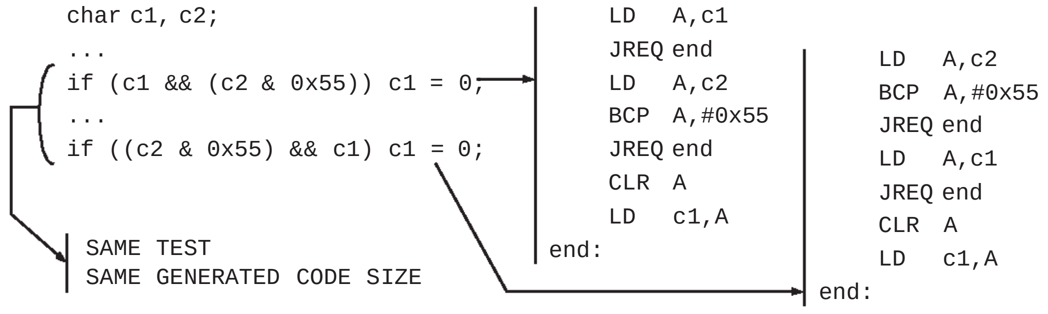
46 bytes < 53 bytes

```
switch (c1) {
   case 0:  c1=1; break;
   case 10: c1=2; break;
   case 22: c1=3; break;
   case 13: c1=4; break;
   case 45: c1=5; break;
   default: c1=6; break;
}
```

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.



```
char c1, c2;
...
if (c1 && (c2 & 0x55)) c1 = 0;
...
if ((c2 & 0x55) && c1) c1 = 0;

SAME TEST
SAME GENERATED CODE SIZE
```

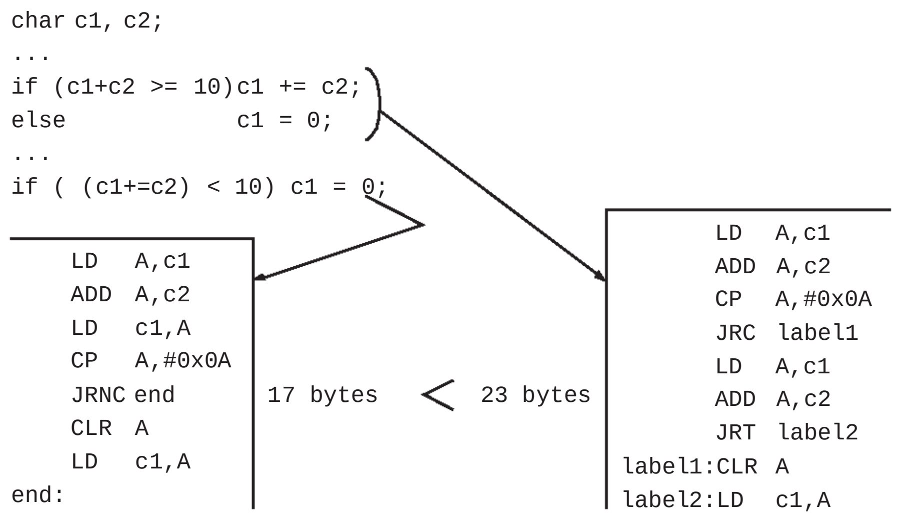
```
     LD   A,c1
     JREQ end
     LD   A,c2
     BCP  A,#0x55
     JREQ end
     CLR  A
     LD   c1,A
end:
```

```
     LD   A,c2
     BCP  A,#0x55
     JREQ end
     LD   A,c1
     JREQ end
     CLR  A
     LD   c1,A
end:
```

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.



```
char c1, c2;
...
if (c1+c2 >= 10)c1 += c2;
else            c1 = 0;
...
if ( (c1+=c2) < 10) c1 = 0;
```

```
     LD   A,c1
     ADD  A,c2
     LD   c1,A
     CP   A,#0x0A
     JRNC end
     CLR  A
     LD   c1,A
end:
```

17 bytes < 23 bytes

```
        LD   A,c1
        ADD  A,c2
        CP   A,#0x0A
        JRC  label1
        LD   A,c1
        ADD  A,c2
        JRT  label2
label1:CLR  A
label2:LD   c1,A
```

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
```

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.