NEC

Application Note

NEC V850 Microcontroller and IAR Compiler: Recommendation for Code Optimisation

Document No. U17741EE2V0AN00
Date Published April 2007

NOTES FOR CMOS DEVICES

① VOLTAGE APPLICATION WAVEFORM AT INPUT PIN

Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between V_{IL} (MAX) and V_{IH} (MIN) due to noise, etc., the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between V_{IL} (MAX) and V_{IH} (MIN).

② HANDLING OF UNUSED INPUT PINS

Unconnected CMOS device inputs can be cause of malfunction. If an input pin is unconnected, it is possible that an internal input level may be generated due to noise, etc., causing malfunction. CMOS devices behave differently than Bipolar or NMOS devices. Input levels of CMOS devices must be fixed high or low by using pull-up or pull-down circuitry. Each unused pin should be connected to V_{DD} or GND via a resistor if there is a possibility that it will be an output pin. All handling related to unused pins must be judged separately for each device and according to related specifications governing the device.

③ PRECAUTION AGAINST ESD

A strong electric field, when exposed to a MOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop generation of static electricity as much as possible, and quickly dissipate it when it has occurred. Environmental control must be adequate. When it is dry, a humidifier should be used. It is recommended to avoid using insulators that easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors should be grounded. The operator should be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions need to be taken for PW boards with mounted semiconductor devices.

④ STATUS BEFORE INITIALIZATION

Power-on does not necessarily define the initial status of a MOS device. Immediately after the power source is turned ON, devices with reset functions have not yet been initialized. Hence, power-on does not guarantee output pin levels, I/O settings or contents of registers. A device is not initialized until the reset signal is received. A reset operation must be executed immediately after power-on for devices with reset functions.

⑤ POWER ON/OFF SEQUENCE

In the case of a device that uses different power supplies for the internal operation and external interface, as a rule, switch on the external power supply after switching on the internal power supply. When switching the power supply off, as a rule, switch off the external power supply and then the internal power supply. Use of the reverse power on/off sequences may result in the application of an overvoltage to the internal elements of the device, causing malfunction and degradation of internal elements due to the passage of an abnormal current.

The correct power on/off sequence must be judged separately for each device and according to related specifications governing the device.

⑥ INPUT OF SIGNAL DURING POWER OFF STATE

Do not input signals or an I/O pull-up power supply while the device is not powered. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements.

Input of signals during the power off state must be judged separately for each device and according to related specifications governing the device.

- **The information in this document is current as of April, 2007. The information is subject to change without notice. For actual design-in, refer to the latest publications of NEC Electronics data sheets or data books, etc., for the most up-to-date specifications of NEC Electronics products. Not all products and/or types are available in every country. Please check with an NEC Electronics sales representative for availability and additional information.**

- No part of this document may be copied or reproduced in any form or by any means without the prior written consent of NEC Electronics. NEC Electronics assumes no responsibility for any errors that may appear in this document.

- NEC Electronics does not assume any liability for infringement of patents, copyrights or other intellectual property rights of third parties by or arising from the use of NEC Electronics products listed in this document or any other liability arising from the use of such products. No license, express, implied or otherwise, is granted under any patents, copyrights or other intellectual property rights of NEC Electronics or others.

- Descriptions of circuits, software and other related information in this document are provided for illustrative purposes in semiconductor product operation and application examples. The incorporation of these circuits, software and information in the design of a customer's equipment shall be done under the full responsibility of the customer. NEC Electronics assumes no responsibility for any losses incurred by customers or third parties arising from the use of these circuits, software and information.

- While NEC Electronics endeavors to enhance the quality, reliability and safety of NEC Electronics products, customers agree and acknowledge that the possibility of defects thereof cannot be eliminated entirely. To minimize risks of damage to property or injury (including death) to persons arising from defects in NEC Electronics products, customers must incorporate sufficient safety measures in their design, such as redundancy, fire-containment and anti-failure features.

- NEC Electronics products are classified into the following three quality grades: "Standard", "Special" and "Specific".

- The "Specific" quality grade applies only to NEC Electronics products developed based on a customerdesignated "quality assurance program" for a specific application. The recommended applications of an NEC Electronics product depend on its quality grade, as indicated below. Customers must check the quality grade of each NEC Electronics product before using it in a particular application.
  "Standard":Computers, office equipment, communications equipment, test and measurement equipment, audio and visual equipment, home electronic appliances, machine tools, personal electronic equipment and industrial robots.
  "Special": Transportation equipment (automobiles, trains, ships, etc.), traffic control systems, anti-disaster systems, anti-crime systems, safety equipment and medical equipment (not specifically designed for life support).
  "Specific": Aircraft, aerospace equipment, submersible repeaters, nuclear reactor control systems, life support systems and medical equipment for life support, etc.

The quality grade of NEC Electronics products is "Standard" unless otherwise expressly specified in NEC Electronics data sheets or data books, etc. If customers wish to use NEC Electronics products in applications not intended by NEC Electronics, they must contact an NEC Electronics sales representative in advance to determine NEC Electronics' willingness to support a given application.

(Note)
(1) "NEC Electronics" as used in this statement means NEC Electronics Corporation and also includes its majority-owned subsidiaries.
(2) "NEC Electronics products" means any product developed or manufactured by or for NEC Electronics (as defined above).

M8E 02. 11-1

Regional Information

Some information contained in this document may vary from country to country. Before using any NEC product in your application, please contact the NEC office in your country to obtain a list of authorized representatives and distributors. They will verify:

- Device availability
- Ordering information
- Product release schedule
- Availability of related technical literature
- Development environment specifications (for example, specifications for third-party tools and components, host computers, power plugs, AC supply voltages, and so forth)
- Network requirements

In addition, trademarks, registered trademarks, export restrictions, and other legal issues may also vary from country to country.

NEC Electronics America Inc.
Santa Clara, California
Tel: 408-588-6000
800-366-9782
Fax: 408-588-6130
800-729-9288

NEC Electronics (Europe) GmbH
Duesseldorf, Germany
Tel: 0211-65 030
Fax: 0211-65 03 1327

Sucursal en España
Madrid, Spain
Tel: 091- 504 27 87
Fax: 091- 504 28 60

Succursale Française
Vélizy-Villacoublay, France
Tel: 01-30-67 58 00
Fax: 01-30-67 58 99

Filiale Italiana
Milano, Italy
Tel: 02-66 75 41
Fax: 02-66 75 42 99

Branch The Netherlands
Eindhoven, The Netherlands
Tel: 040-265 40 10
Fax: 040-244 45 80

Tyskland Filial
Taeby, Sweden
Tel: 08-638 7200
Fax: 08-638 7222

United Kingdom Branch
Milton Keynes, UK
Tel: 01908-691-133
Fax: 01908-670-290

NEC Electronics Hong Kong Ltd.
Hong Kong
Tel: 2886-9318
Fax: 2886-9022/9044

NEC Electronics Korea Ltd.
Seoul, Korea
Tel: 02-558-3737
Fax: 02-558-5141

NEC Electronics Singapore Pte. Ltd.
Singapore
Tel: 65-6253-8311
Fax: 65-6250-3583

NEC Electronics Taiwan Ltd.
Taipei, Taiwan
Tel: 02-8175-9600
Fax: 02-8175-9670

Chapter 1 Introduction

When programming a microcontroller, code size, data size, real time behaviour, and execution speed have to be taken into account. In consequence it is a must for C compilers for embedded systems to offer target specific extensions like keywords and pragmas; but also to support fully the features of the microcontroller i.e.: memory model, addressing modes, assembler instructions.
That's why, it is essential for the user to understand how compilers handle data structure, parameters, and variables and how they optimise the code.
The purpose of this document is to give recommendations on code and RAM optimisation for the NEC V850 microcontroller using the IAR compiler. First of all, general optimisation strategies are described, code optimisation (size and speed), RAM optimisation and general hint given. The second part lists IAR compiler/linker options to use in order to obtain a good optimised code according to the chosen strategy. But the optimiser is not the only way to optimise the code. It is also important that the user write efficient C code for his optimisation strategy. The document's last part gives the user hints on how to write efficient C code. Some recommendations are general and some specific to the V850 or IAR compiler.

Note that this application note is specific to the NEC V850 microcontroller and the IAR Workbench. It doesn't claim to be complete and each example should be checked before being used in an application.

Chapter 2 General Optimisation

There are two types of code optimisation strategies: speed and size. Depending on which strategy is chosen, different optimisations are recommended.
IAR compiler has an optimiser which decides on the best optimisations methods in function of the strategy chosen. The list of options to be set is given in the next chapter.
But despite the software optimiser, the following coding methods are recommended when using one or the other optimisation strategy.
In both case the used of the V850 memory model is highly recommended.

2.1 Size Optimisation

- Use general functions, no specific one
- Replace repeated code sequence by a function
- Avoid macros

2.2 Speed Optimisation

- Use specific functions instead of general ones
- Loop unrolling
- In-lining small functions

2.3 V850 Memory Model

By default when coding the compiler will access data using the full 32-bit address:

```
movhi     hi(_x),r0,r17
movea     lo(_x),r17,r17
ld.b      0[r17],r17
```

Memory models are used to optimise data accesses. There are three memory models available with the V850. Two of them are taking advantage of the fact that the load and store assembly instructions accept a 16-bit offset. The last one used the short load and store assembler instruction specific to the V850.

2.3.1 The Tiny Memory Model (NEAR)

The Tiny Memory Model takes as base address 0. This means that all variables and constants within +/- 16-bit offset from the address 0 can be accessed with one instruction (4 bytes).

```
ld.b      16bitoff(_x)[zero],r17
```

2.3.2 The Small Memory Model (BREL)

The Small Memory Model takes as base address an address defined by the user at link stage.
GP (or R4) is the register containing the base address for all global variables. This means, that all variables within 16-bit offset from this address can be accessed with one instruction (4 bytes).

```
ld.b      16bitoff(_x)[gp],r17
```

2.3.3 The Short Memory Model (saddr)

The tiny memory model is using the short load and short store assembly instructions (2 bytes) which accept only 8-bit offset. The base address is defined by the user at link stage, and saved in the register EP (or R30). This memory model can be used in parallel with the small and tiny memory model.

```
sld.b     7/8bitoff(_x)[ep],r17
```

One memory model must be chosen as a default. It is recommended to use the zero or the small memory model as default. It is still possible to manually change the memory model inside the code with keywords. For more information about memory model please refers to the compiler documentation given with the software tool.

Chapter 3 Recommended Options when Using the IAR Compiler for the V850

3.1 General Code Optimisation

3.1.1 Use of the memory model:

Command line option: `-m{t,T,s,S,l,L}`

```
t/T   tiny/tiny with saddr (t default)
s/S   small/small with saddr
l/L   large/large with saddr
```

This option will set the default memory model for the whole project. It is still possible to specify the access type and the placement of specific variables directly in the code by using keywords.
e.g.:
A variable in the default memory:

```
int a;
```

A variable in near memory (tiny memory model):

```
int __near a;
```

A variable in base-relative memory (small memory model):

```
int __brel a:
```

3.1.2 Use of mask registers

Command line option: `--reg_const`

Registers r20 and r21 have respectively the value 0xFF and 0xFFFF. The compiler then used those registers every time it meets their value in the user code. It is useful to use those masks when the code contains a lot of comparison.
The compiler then doesn't have to load each time the value 0xFF or 0xFFFF in a register before using it. This generally decreases the code size.

Note that in a code that has few comparisons but a lot of local variables the code size might increase as 2 general purpose registers are blocked.

3.2 Size / Speed Optimisation

Command line Options:

- Size optimisation: -zn (n = 9, 6, 3).
- Speed optimisation: -sn (n = 9, 6, 3).

Those options switch on respectively all options that will reduce the code size and all options that will improve the performance.

n specifies the level of optimisation:

- 9 = high
- 6= Medium
- 3 = low

There are some options that the user can switch on or off when compiling with Medium or high:

- common expression elimination
- loop unrolling
- Function in-lining
- Code motion
- Type base analysis

3.3 RAM Optimisation

The V850 microcontroller has data alignment. This means that an integer (32-bits) must be placed on a 4-byte boundary address, a short (16-bits) on a 2-byte boundary address. A character doesn't have any alignment requirement.
To respect this alignment requirement some padding bytes might be introduced, causing the RAM usage to grow.
This paragraph explains how the IAR compiler handles this alignment.

3.3.1 Global variable

Global variables are sorted at linker time in order to use the memory in the best way possible. This means that the linker will reorder the global variables to have a minimum of padding bytes.
e.g.:

C-Code	MAP File

`int i0;`
`char c0;`
`int i1;`
`char c1;`
	`0x0 i0`
	`0x4 i1`
	`0x8 c0`
	`0x9 c1`

3.3.2 Unions / Structures

- Alignment of a variable of type struct/union

A variable of type struct or union inherits the alignment requirement of its fields. This means that, a structure with only characters has no alignment requirement. A structure whose biggest field is a short, will be 2-byte aligned, if the biggest field is an integer, it will then be 4-byte aligned.

- Alignment of the structures' or union's fields

Fields are allocated in the order of their declaration. This means that the compiler might introduce some padding bytes to respect the alignment of members. In consequence, when possible, always **have the fields of a structure ordered in function of their size: first integer (32 bits), then short (16 bits), then chars (8 bits).**

- Packing Option

IAR compiler supports a **#pragma pack(n)** to suppress padding bytes between fields in structures or unions. This obviously implies that members can be misaligned.
The disadvantage is that the compiler will generate code to access those misaligned fields correctly, thus increasing the code size.
One solution is to enable misaligned access. By default the compiler doesn't use this feature as misaligned access needs more bus cycles.
The use of misaligned access can be enabled with the option:

--allow_misaligned_data_access

Chapter 4 Hints for Code Optimisation

Even if the IAR optimiser is used, efficient C code can save some code or make your program faster. This chapter lists some hints on how to write efficient C code. Some of these hints are general and some specific to the V850 or IAR compiler.

4.1 Local Variables Should be of Type Integer

Except for arrays, structures and unions which are placed on the stack, normal local variables are placed in general purpose 32-bit registers. If a local variable is declared as a character or as a half-word and placed in a 32-bit register, then for every operation modifying this variable, the signed bit will be set. It is in consequence using one more instruction (2 bytes) than if it was declared as an integer. The following example is a simple illustration of what happens:

C Code & ASM code

```
void read_stuct_int(void)
{
   int i0;
   for(i0 =0;i0 < 100;i0++)
         MOV     zero,r1
         BR      (??read_stuct_int_0)
   {
      S1.my_i++;
   ??read_stuct_int_1:
         LD.W    (S1+4)[zero],r5
         ADD     1,r5
         ST.W    r5,(S1+4)[zero]
   }
         ADD     1,r1
   ??read_stuct_int_0:
         ADDI    -100,r1,zero
         BLT     (??read_stuct_int_1)
}
         JMP     [lp]

void read_stuct_char(void)
{
   char i0;
   for(i0 =0;i0 < 100;i0++)
         MOV     zero,r1
         BR      (??read_stuct_char_0)
   {
      S1.my_i++;
   ??read_stuct_char_1:
         LD.W    (S1+4)[zero],r5
         ADD     1,r5
         ST.W    r5,(S1+4)[zero]
   }
         ADD     1,r1
   ??read_stuct_char_0:
         ZXB     r1
         ADDI    -100,r1,zero
         BNC     (??read_stuct_char_1)
}
         JMP     [lp]
```

4.2 Use of Temporary Variables

When using complex structures, it is code saving to create a local variable to store one of the fields or sub-fields which is often used in the function. In the following example the structure has 3 layers, and only the 3rd layer is used in the test functions. By using a temporary variable, 32 bytes are saved.
e.g.:

C Code

```
//Structure definition
typedef struct P1
{
    char c;
    int i,j,k;
} S_P1;

typedef struct P2
{
    int dummy;
    S_P1 *p;
} S_P2;

typedef struct P3
{
    short dummy;
    S_P2 *q;
} S_P3;

// Function
void test_temp_var(S_P3 *s)
{
  S_P1 *temp = s->q->p;

  if(temp->i)
      temp->i= temp->j;
  if(temp->j)
      temp->i= temp->k;
  if(temp->k)
      temp->c= 0x1;

  return;
}

void test_notemp_var(S_P3 *s)
{
  if(s->q->p->i)
      s->q->p->i= s->q->p->j;
  if(s->q->p->j)
      s->q->p->i= s->q->p->k;
  if(s->q->p->k)
      s->q->p->c= 0x1;

  return;
}
```

IAR ASM Code

```
//void test_temp_var(S_P3 *s) = 52 bytes
test_temp_var:
        LD.W    (+4)[r1],r1
        LD.W    (+4)[r1],r1
        LD.W    (+4)[r1],r5
        CMP     zero,r5
        BE      (??test_temp_var_0)
        LD.W    (+8)[r1],r5
        ST.W    r5,(+4)[r1]
  ??test_temp_var_0:
        LD.W    (+12)[r1],r5
        LD.W    (+8)[r1],r6
        CMP     zero,r6
        BE      (??test_temp_var_1)
        ST.W    r5,(+4)[r1]
  ??test_temp_var_1:
        CMP     zero,r5
        BE      (??test_temp_var_2)
        MOV     1,r5
        ST.B    r5,(+0)[r1]
  ??test_temp_var_2:
        JMP     [lp]

//void test_notemp_var(S_P3 *s) = 84 bytes
test_notemp_var:
        LD.W    (+4)[r1],r1
        LD.W    (+4)[r1],r5
        LD.W    (+4)[r5],r5
        CMP     zero,r5
        BE      (??test_notemp_var_0)
        LD.W    (+4)[r1],r5
        LD.W    (+4)[r1],r6
        LD.W    (+8)[r6],r6
        ST.W    r6,(+4)[r5]
  ??test_notemp_var_0:
        LD.W    (+4)[r1],r5
        LD.W    (+8)[r5],r5
        CMP     zero,r5
        BE      (??test_notemp_var_1)
        LD.W    (+4)[r1],r5
        LD.W    (+4)[r1],r6
        LD.W    (+12)[r6],r6
        ST.W    r6,(+4)[r5]
  ??test_notemp_var_1:
        LD.W    (+4)[r1],r5
        LD.W    (+12)[r5],r5
        CMP     zero,r5
        BE      (??test_notemp_var_2)
        LD.W    (+4)[r1],r1
        MOV     1,r5
        ST.B    r5,(+0)[r1]
  ??test_notemp_var_2:
        JMP     [lp]
```

4.3 Logical Shift

When multiplying or dividing a variable with a 2exp (n) value, the use of logical shift instead of multiplication or division is recommended.

4.4 For Loop

4.4.1 If statement in a For-loop

When optimising for speed, avoid having an if-statement in a for-loop. It is better to move it outside the loop. Less test and branch instructions are then executed.
e.g.:

C Code:

```
void test_if_in_loop(void)
{
  int j;
  for (j = 0; j<100; j++)
   {
        if(inv==0)
        {
          array[j] = 100-j;
        }
        else
          array[j] = j;
   }
}

void test_if_out_loop(void)
{
  int j;
  if(inv==0)
  {
   for (j = 0; j<100; j++)
     {
        array[j] = 100-j;
     }
  }
  else
  {
   for (j = 0; j<100; j++)
      {
        array[j] = j;
      }
} }
```

IAR ASM Code:

```
//void test_if_in_loop(void) = 46 bytes

test_if_in_loop:
         MOV     zero,r1
         BR      (??test_if_in_loop_0)
   ??test_if_in_loop_1:
         MOV     r1,r5
         SHL     2,r5
         ST.W    r1,(array)[r5]
   ??test_if_in_loop_2:
         ADD     1,r1
   ??test_if_in_loop_0:
         ADDI    -100,r1,zero
         BGE     (??test_if_in_loop_3)
         LD.BU   (inv)[zero],r5
         CMP     zero,r5
         BNE     (??test_if_in_loop_1)
         MOV     r1,r5
         SHL     2,r5
         MOVEA   100,zero,r6
         SUB     r1,r6
         ST.W    r6,(array)[r5]
         BR      (??test_if_in_loop_2)
   ??test_if_in_loop_3:
         JMP     [lp]

//void test_if_out_loop(void) = 52 bytes

test_if_out_loop:
         LD.BU   (inv)[zero],r1
         CMP     zero,r1
         MOV     zero,r1
         BNE     (??test_if_out_loop_0)
   ??test_if_out_loop_1:
         MOV     r1,r5
         SHL     2,r5
         MOVEA   100,zero,r6
         SUB     r1,r6
         ST.W    r6,(array)[r5]
         ADD     1,r1
         ADDI    -100,r1,zero
         BGE     (??test_if_out_loop_2)
         BR      (??test_if_out_loop_1)
   ??test_if_out_loop_0:
         MOV     r1,r5
         SHL     2,r5
         ST.W    r1,(array)[r5]
         ADD     1,r1
         ADDI    -100,r1,zero
         BLT     (??test_if_out_loop_0)
   ??test_if_out_loop_2:
         JMP     [lp]
```

4.4.2 End event of a For-loop

Avoid having in the end of loop test and expression to evaluate. If this expression's result doesn't change during the for-loop lifetime, the result should be saved in a temporary variable. Thus the expression will not be evaluated at each iteration of the loop, reducing the execution time.

```
int a,b;
void end_of_loop(void) // number of bytes: 30 bytes
{
   for(i=0; i<(a*b); i++)
      MOV     zero,r29
      BR      (??end_of_loop_0)
   {
      init_struct(rr);
      ??end_of_loop_1:
            LD.W    (rr)[zero],r1
            JARL    (init_struct),lp
   }
            ADD     1,r29
      ??end_of_loop_0:
            LD.W    (a)[zero],r1
            LD.W    (`b`)[zero],r5
            MUL     r5,r1,zero
            CMP     r1,r29
            BLT     (??end_of_loop_1)
}


int a,b;
void end_of_loop_tmp(void) // number of bytes: 30 bytes
{
   int tmp = a*b;
            LD.W    (a)[zero],r29
            LD.W    (`b`)[zero],r1
            MUL     r1,r29,zero
   for(i=0; i<tmp; i++)
            MOV     zero,r28
            BR      (??end_of_loop_tmp_0)
   {
      init_struct(rr);
      ??end_of_loop_tmp_1:
            LD.W    (rr)[zero],r1
            JARL    (init_struct),lp
   }
            ADD     1,r28
      ??end_of_loop_tmp_0:
            CMP     r29,r28
            BLT     (??end_of_loop_tmp_1)
}
```

4.5 Modulo

For modulo operation x % (n) the assembly instruction DIV is generally used. This instruction costs several cycles and increases the execution time. In case n = 2^m, the expression x % (n) can be replaced by (x & (n-1)). The code generated by the compiler doesn't use then the DIV instruction, reducing the execution time.
e.g.:

C & ASM Code:

```
int test_modulo(int a, int b) // 8 bytes
   test_modulo:
{
   return (a%b);
         DIV     r5,r1,r6
         MOV     r6,r1
         JMP     [lp]
}

int test_modulo_and(int a, int b) // 6 bytes
   test_modulo_and:
{
   return (a&(b-1));
         ADD     -1,r5
         AND     r5,r1
         JMP     [lp]
}
```

4.6 Look-up Tables

When having a long list of tests in an if-statement or several if-statements, it is sometime an advantage for both speed and size to use a lookup table.
The disadvantage is that, depending on the table declaration (global, const, local…), RAM,ROM or stack is needed e.g.:

C Code:

```
int init_noLUT(int in)
{
    int out;
    in &=(8);

    if((in == 0) ||(in == 2)||(in == 4)||(in == 6)||(in == 8))
         out = 0xFA;
    else
         out = -1;

    return out;
}

char lookup_table[9] = {1,0,1,0,1,0,1,0,1};
int init_LUT(int in)
{
    int out;
    in &=(8);

    if(lookup_table[in])
         out = 0xFA;
    else
         out = -1;

    return out;
}
```

IAR ASM Code:

```
//int init_noLUT(int in) = 34 bytes

init_noLUT:
         ANDI    0x00000008,r1,r1
         CMP     zero,r1
         BE      (??init_noLUT_0)
         CMP     2,r1
         BE      (??init_noLUT_0)
         CMP     4,r1
         BE      (??init_noLUT_0)
         CMP     6,r1
         BE      (??init_noLUT_0)
         CMP     8,r1
         BNE     (??init_noLUT_1)
   ??init_noLUT_0:
         MOVEA   250,zero,r1
         JMP     [lp]
   ??init_noLUT_1:
         MOV     -1,r1
         JMP     [lp]

//int init_LUT(int in) = 22 bytes + 9 bytes in RAM

init_LUT:
         ANDI    0x00000008,r1,r1
         LD.BU   (lookup_table)[r1],r1
         CMP     zero,r1
         BE      (??init_LUT_0)
         MOVEA   250,zero,r1
         JMP     [lp]
         MOV     -1,r1
         JMP     [lp]
```

4.7 Switch Instruction

The code generated when using an if/else or a switch statement is approximately the same, as long as the case number is less than 3.
If the number of cases is bigger than 3, the compiler uses the V850 assembler instruction switch in relation with a lookup table.
The code is then smaller and faster. The difference in code size between switch and if/else increases with the number of cases.
In consequence, the switch statement brings an advantage over the if/else statement when the number of case is greater than 3:
e.g.:

C-Code

```
int init_switch(int in)
{
    int out;
    in &=(4);

    switch (in)
    {
         case 0:
             out = 0xff;
             break;
         case 1:
             out = 0xFe;
             break;
         case 2:
             out = 0xFD;
             break;
         case 3:
             out = 0xFC;
             break;
         case 4:
             out = 0xFB;
             break;
         case 5:
             out = 0xFA;
             break;

         default:
             out = -1;
    }
    return out;
}
```

```
int init_if_else(int in)
{
    int out;
    in &=(4);

    if(in == 0)
         out = 0xff;
    else
    {
         if(in == 1)
             out = 0xFe;
         else
         {
             if(in == 2)
               out = 0xFD;
             else
             {
               if(in == 3)
                   out = 0xFC;
               else
               {
                   if(in == 4)
                     out = 0xFB;
                   else
                   {
                     if(in == 5)
                         out = 0xFA;
                     else
                         out = -1;
                   }
               }
             }
         }
    }
    return out;
}
```

IAR ASM Code

```
//int init_switch(int in) = 62 bytes

init_switch:
         ANDI    0x00000004,r1,r1
         CMP     5,r1
         BH      (??init_switch_0)
         SWITCH  r1
   `?<Jump table for init_switch>_0`:
         DH      ((??init_switch_1)-($-0))>>1
         DH      ((??init_switch_2)-($-2))>>1
         DH      ((??init_switch_3)-($-4))>>1
         DH      ((??init_switch_4)-($-6))>>1
         DH      ((??init_switch_5)-($-8))>>1
         DH      ((??init_switch_6)-($-10))>>1
   ??init_switch_1:
         MOVEA   255,zero,r1
         JMP     [lp]
   ??init_switch_2:
         MOVEA   254,zero,r1
         JMP     [lp]
   ??init_switch_3:
         MOVEA   253,zero,r1
         JMP     [lp]
   ??init_switch_4:
         MOVEA   252,zero,r1
         JMP     [lp]
   ??init_switch_5:
         MOVEA   251,zero,r1
         JMP     [lp]
   ??init_switch_6:
         MOVEA   250,zero,r1
         JMP     [lp]
   ??init_switch_0:
         MOV     -1,r1
         JMP     [lp]
```

```
//int init_if_else(int in) = 68 bytes

init_if_else:
         ANDI    0x00000004,r1,r1
         CMP     zero,r1
         BNE     (??init_if_else_0)
         MOVEA   255,zero,r1
         JMP     [lp]
   ??init_if_else_0:
         CMP     1,r1
         BNE     (??init_if_else_1)
         MOVEA   254,zero,r1
         JMP     [lp]
   ??init_if_else_1:
         CMP     2,r1
         BNE     (??init_if_else_2)
         MOVEA   253,zero,r1
         JMP     [lp]
   ??init_if_else_2:
         CMP     3,r1
         BNE     (??init_if_else_3)
         MOVEA   252,zero,r1
         JMP     [lp]
   ??init_if_else_3:
         CMP     4,r1
         BNE     (??init_if_else_4)
         MOVEA   251,zero,r1
         JMP     [lp]
   ??init_if_else_4:
         CMP     5,r1
         BNE     (??init_if_else_5)
         MOVEA   250,zero,r1
         JMP     [lp]
   ??init_if_else_5:
         MOV     -1,r1
         JMP     [lp]
```

A switch statement is also always more efficient than an if/else, if some cases are common, e.g.:

C Code:

```
int init_switch(int in)
{
    int out;
    in &=(4);

    switch (in)
    {
         case 0:
             out = 0xff;
             break;
         case 1:
             out = 0xFe;
             break;
         case 2:
         case 3:
         case 4:
             out = 0xFB;
             break;
         case 5:
             out = 0xFA;
             break;
         default:
             out = -1;
    }
    return out;
}
```

```
int init_if_else(int in)
{
    int out;
    in &=(4);

    if(in == 0)
         out = 0xff;
    else
   {
         if(in == 1)
             out = 0xFe;
         else
         {
               if((in == 2) || (in == 3) || (in == 4))
                   out = 0xFB;
               else
               {
                   if(in == 5)
                     out = 0xFA;
                   else
                   {
                     out = -1;
                   }
               }
         }
    }
    return out;
}
```

IAR ASM Code

```
//int init_switch(int in) = 50 bytes
init_switch:
        ANDI    0x00000004,r1,r1
        CMP     5,r1
        BH      (??init_switch_0)
        SWITCH  r1
   `?<Jump table for init_switch>_0`:
        DH      ((??init_switch_1)-($-0))>>1
        DH      ((??init_switch_2)-($-2))>>1
        DH      ((??init_switch_3)-($-4))>>1
        DH      ((??init_switch_3)-($-6))>>1
        DH      ((??init_switch_3)-($-8))>>1
        DH      ((??init_switch_4)-($-10))>>1
   ??init_switch_1:
        MOVEA   255,zero,r1
        JMP     [lp]
   ??init_switch_2:
        MOVEA   254,zero,r1
        JMP     [lp]
   ??init_switch_3:
        MOVEA   251,zero,r1
        JMP     [lp]
   ??init_switch_4:
        MOVEA   250,zero,r1
        JMP     [lp]
   ??init_switch_0:
        MOV     -1,r1
        JMP     [lp]
```

```
//int init_if_else(int in) = 54 bytes

init_if_else:
         ANDI    0x00000004,r1,r1
         CMP     zero,r1
         BNE     (??init_if_else_0)
         MOVEA   255,zero,r1
         JMP     [lp]
   ??init_if_else_0:
         CMP     1,r1
         BNE     (??init_if_else_1)
         MOVEA   254,zero,r1
         JMP     [lp]
   ??init_if_else_1:
         CMP     2,r1
         BE      (??init_if_else_2)
         CMP     3,r1
         BE      (??init_if_else_2)
         CMP     4,r1
         BNE     (??init_if_else_3)
   ??init_if_else_2:
         MOVEA   251,zero,r1
         JMP     [lp]
   ??init_if_else_3:
         CMP     5,r1
         BNE     (??init_if_else_4)
         MOVEA   250,zero,r1
         JMP     [lp]
   ??init_if_else_4:
         MOV     -1,r1
         JMP     [lp]
```

4.8 If statement

It is recommended, when possible to put the assignment in the test. One instruction less is needed.

C Code:

```
void test_if_assignment_out(void)
{
  if(glob0+glob1 >10)
    glob10= glob0+glob1;
  else
    glob10 =0;
}

void test_if_assignment_in(void)
{
  if((glob10 = glob0+glob1) <10)
      glob10 =0;
}
```

IAR ASM Code:

```
//void test_if_assignment_out(void) = 26 bytes

test_if_assignment_out:
         LD.W    (glob0)[zero],r5
         LD.H    (glob1)[zero],r1
         ADD     r5,r1
         CMP     11,r1
         BLT     (??test_if_assignment_out_0)
         ST.W    r1,(glob10)[zero]
         JMP     [lp]
   ??test_if_assignment_out_0:
         ST.W    zero,(glob10)[zero]
         JMP     [lp]

//void test_if_assignment_in(void) = 24 bytes

test_if_assignment_in:
         LD.W    (glob0)[zero],r1
         LD.H    (glob1)[zero],r5
         ADD     r1,r5
         ST.W    r5,(glob10)[zero]
         CMP     10,r5
         BGE     (??test_if_assignment_in_0)
         ST.W    zero,(glob10)[zero]
   ??test_if_assignment_in_0:
```

4.9 Bit Fields

It is RAM and sometimes code saving to declare bit fields as character. No alignment is then taken into account.

C Code.

```
typedef struct // 9 bytes + 1 padding byte
{
   int a;
   char c,
   struct {
           short var0:4;
           short var1:4;
           short var2:4;
           short var4:4;
   }S_t:
}S_16;
S_16 s1;

void init(void)
{
   s1.S_t.var2 = 3; // 18 bytes
           LD.H    (s1)[zero],r1
           MOVEA   -449,zero,r5
           AND     r5,r1
           ORI     0x000000c0,r1,r1
           ST.H    r1,(s1)[zero]
```

This implies that:

C-Code	MAP File
int a char c S_t	0x0 a 0x4 c **0x6 S_t**

As the following code:

```
typedef struct // 9 bytes, no padding bytes
{
   int a;
   char c,
   struct {
           char var0:4;
           char var1:4;
           char var2:4;
           char var4:4;
   }S_t:
}S_16;
S_16 c1;

void init(void)
{
   c1.S_t.var2 = 1; // 16 bytes
           LD.BU   (c1)[zero],r1
           ANDI    0x000000c7,r1,r1
           ORI     0x00000010,r1,r1
           ST.B    r1,(c1)[zero]
}
```

This implies that:

C-Code	MAP File
int a char c S_t	0x0 a 0x4 c **0x5 S_t**

4.10 Increment / Decrement

When using IAR Compiler, a pre-increment/decrement on pointer is one instruction less than a post increment/decrement.

C Code.

```
void post_increment (void)
{
    c = *(ptr++);
}

void pre_increment (void)
{
    c = *(++ptr);

}
```

IAR ASM Code:

```
post_increment: // 24 bytes
   LD.W    (ptr)[zero],r1
   LD.BU   (+0)[r1],r1
   ST.B    r1,(`c`)[zero]
   LD.W    (ptr)[zero],r1
   ADD     1,r1
   ST.W    r1,(ptr)[zero]
   JMP     [lp]

pre_increment: // 20 bytes
   LD.W    (ptr)[zero],r1
   ADD     1,r1
   ST.W    r1,(ptr)[zero]
   LD.BU   (+0)[r1],r1
   ST.B    r1,(`c`)[zero]
   JMP     [lp]
```

Chapter 5 Conclusion

Whichever the optimisation strategy it is essential for the user to have a good knowledge of the micro-controller features as well as the features offered by the compiler.
Only with this knowledge can the best optimisation compromise be found.

[MEMO]

[MEMO]