To our customers,

Old Company Name in Catalogs and Other Documents

On April 1st, 2010, NEC Electronics Corporation merged with Renesas Technology Corporation, and Renesas Electronics Corporation took over all the business of both companies. Therefore, although the old company name remains in this document, it is a valid Renesas Electronics document. We appreciate your understanding.

Renesas Electronics website: http://www.renesas.com

April 1st, 2010
Renesas Electronics Corporation

Issued by: Renesas Electronics Corporation (http://www.renesas.com)

Send any inquiries to http://www.renesas.com/inquiry.

Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas 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 Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: "Standard", "High Quality", and "Specific". The recommended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as "Specific" without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as "Specific" or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is "Standard" unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.

   "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.

   "High Quality": 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; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.
8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

(Note 1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

HEW

Embedded C Programming III - Optimization (ECProgramIII_opt)

Introduction

This application note, Embedded C Programming III, will cover the various topics on optimization.

Two main areas are elaborated:

- Embedded C Programming Techniques and use of #pragma Directives
- High-performance Embedded Workshop (HEW) Optimizer Setting

The examples used will be based on HEW H8 Toolchain. These concepts will also be applicable to other toolchains and MCU series. (Note: SH series will have a different set of HEW optimization handling as cache and pipeline issues must be considered)

Target

All

Contents

1. Recapitulate Part 1 & 2 5

2. Basic Concept of Optimization 6
2.1 Definition 6
2.2 Methods 6
2.3 Basic Illustration 6
2.4 Evaluation 7

3. Further understanding of optimization 9
3.1 Do not optimize 9
3.2 Bound conditions 9
3.3 Optimize hot spot 9
3.4 Confusion to source level debugger 9
3.5 Dead code elimination 9
3.6 Test & evaluate 9

4. Optimization Jargon 10

5. How to optimize – Programming Technique 11
5.1 Data Handling 11
5.1.1 RAM usage 11
5.1.2 Data Type Usage 11
5.1.3 Use Variables of the Same Type for processing 12
5.1.4 Use of Unsigned Type 12
5.1.5 Float and Double 12
5.1.6 Data Declaration - Constant 12
5.1.7 Data Declaration - Volatile 12
5.1.8 Data Initialization at Declaration 12
5.1.9 Data Definition - Arrangement and Packing 13
5.1.10 Global and Local Variables 13
5.1.11 Passing Parameter Registers and Working Registers 14
5.1.12 Global Register variables 14
5.1.13 Passing Reference as Parameters 15
5.1.14 Return Value 15
5.1.15 Register Save and Restore 15
5.1.16 Use of short addressing to access variables 16
5.1.17 Pointer and Array 17
5.1.18 Better Data Structure and Representation 17
5.1.19 Accessing Structure 17
5.1.20 Array and Structure Initialization 17

5.2 Flow Control Handling 18
5.2.1 Switch 18
5.2.2 Jumps 19
5.2.3 Inline function 19
5.2.4 Function Calls and Addressing Modes 19
5.2.5 Tail Recursions 20
5.2.6 Loop Unrolling 20
5.2.7 Loop invariant IF code floating 20
5.2.8 Do-While statement 20
5.2.9 Loop Hoisting 21
5.2.10 Common expression shall be calculated once or earlier. 21
5.2.11 Else clause removal 21
5.2.12 Loop Overhead 21
5.3 Other Handling 22
5.3.1 Native Instruction and Data 22
5.3.2 Hand coded assembly 22
5.3.3 Lookup Table and Calculation 23
5.3.4 Polling and Interrupt 23
5.3.5 Fixed-point and floating-point arithmetic 23
5.3.6 Standard Library Routines 23
5.3.7 Input and Output Access 24
5.3.8 Specify Optimization Type for Each Module 24
5.3.9 Horner's Rule of Polynomial Evaluation 24
5.3.10 Factorization 24
5.3.11 Use Finite Differences to Avoid Multiplies 24
5.3.12 Condition Determination Using Substitution Values 25
5.3.13 Modula 25
5.3.14 Division & Multiplication 25
5.3.15 Constant Folding and Propagation 25
5.3.16 Constant in Shift Operations 25
5.3.17 Use Formula 26
5.3.18 Simplify Condition 26
5.3.19 Absolute Value 26

6. How to Optimize – HEW Setting 27
6.1 C/C++ - Optimize Category Setting 29
6.1.1 Register 29
6.1.2 Shift to multiple 29
6.1.3 Struct Assignment 30
6.1.4 Expression 30
6.1.5 Loop optimization 30
6.1.6 Loop unrolling 30
6.1.7 Inline function 31
6.1.8 Switch Statement 31
6.1.9 Function Call 31
6.1.10 Data Access 32

6.2 C/C++ - Other Category Setting 33
6.2.1 Avoid optimizing external symbols treating them as volatile 33
6.2.2 Treat enum as char if it is in the range of char 33
6.2.3 Increase a register for register variable 34
6.2.4 Put common subexpression on a register temporarily 34
6.2.5 Use EEPMOV in block copy 34
6.2.6 Group data by alignment 34
6.3 Standard Library Optimize and Other Category Setting 35
6.4 CPU setting 36
6.4.1 Change number of parameter registers from 2(default) to 3 36
6.4.2 Treat double as float 36
6.4.3 Pass struct parameter via register 36
6.4.4 Pass 4-byte parameter/return value via register 37
6.4.5 Pack struct, union and class 37
6.5 Link/ Library - Optimize Category Setting 38
6.5.1 Unify strings 38
6.5.2 Eliminate dead code 39
6.5.3 Use short addressing 39
6.5.4 Relocate registers 39
6.5.5 Eliminate same code 39
6.5.6 Use indirect call/jump 39
6.5.7 Optimize branches 39
6.6 Inter module optimization 40
6.6.1 Unifies Constant / Literal Strings 41
6.6.2 Delete No-referenced Symbols 42
6.6.3 Short Absolute Addressing 42
6.6.4 Indirect addressing 42
6.6.5 Register save/restore 43
6.6.6 Unifies same codes 43
6.6.7 Uses better branch instruction 43

7. Suggested Optimization Steps 44

8. Conclusion 45

Reference 45

1. Recapitulate Part 1 & 2

The topics covered in “Embedded C Programming I” are:

- Generated C files and sections of HEW.
- Effect of C initialization on each variable characteristics and storage areas
- Utilization of Stack and Heap.
- Usage of preprocessor directives (macro, conditional compilation and etc).
- Usage of extended functions (pragma, intrinsic functions and etc).
- Usage of available library.
- Effect of a function call on the stack and registers.
- Management of section by HEW
- Comparison of similar operation.
- Information on flow of project compilation, linking and debugging.
- Suggested programming techniques.

“Embedded C Programming II” illustrated the software control techniques on:

- Peripherals and ports
- External memory.

In this third part of Embedded C Programming, optimization is the key topics.

2. Basic Concept of Optimization

2.1 Definition

Optimization is a process of improving efficiency of a program in time (speed) or space (size).

2.2 Methods

Generally, optimization can be achieved by four methods:

- Choice of Compiler
- Compiler Setting
- Programming Algorithm and Techniques
- Rewrite program in assembly

2.3 Basic Illustration

A simple illustration of optimization by speed and size is as follow:

```
// Faster Speed

main()
{           ...
            ...
            XXXX
            YYYY
            ZZZZZ
            ...
            ...
            XXXX
            YYYY
            ZZZZZ
            ...
            ...
            XXXX
            YYYY
            ZZZZZ
            ...
            ...
            XXXX
            YYYY
            ZZZZZ
            ...
            ...
}
```

```
// Smaller Size

main()
{           ...
            ...
            call_routine();
            ...
            ...
            call_routine();
            ...
            ...
            call_routine();
            ...
            ...
            call_routine();
            ...
            ...
}

void call_routine(void)
{
            XXXX
            YYYY
            ZZZZZ
}
```

It can be observed that optimization with one method may affect the other.

A general phenomenon is faster operating code will have a bigger code size, whereas smaller code size will have a slower execution speed. However this may not be always true.

2.4 Evaluation

A program's optimization level can be evaluated based on the measurement of:

- Total code size,
- Total number of execution cycles (time taken).

These are determined by the basic component of a program, which is the assembly code (Instruction set / Opcode / Mnemonic).

In the MCU manual, these assembly codes characteristic are detailed:

- Instruction length → Determine Code Size
- Number of execution states → Determine Execution Speed

Example:

Mnemonic	Instruction length (bytes)	Number of execution states	Remarks
MOV.B Rs, Rd	2	2	Register usage has faster execution
MOV.W Rs, @Rd	2	6	
JSR @aa:16	4	8	8-bit absolute address jump take up smaller space
JSR @@aa:8	2	8	

Based on the instruction information, programmer can calculate the size and execution states of a module or project.

However it is almost impossible to make such calculation for large programs. The simpler methods are:

- Code size → The number of bytes allocated to each function and section are detailed in the generated MAP file.

Example:

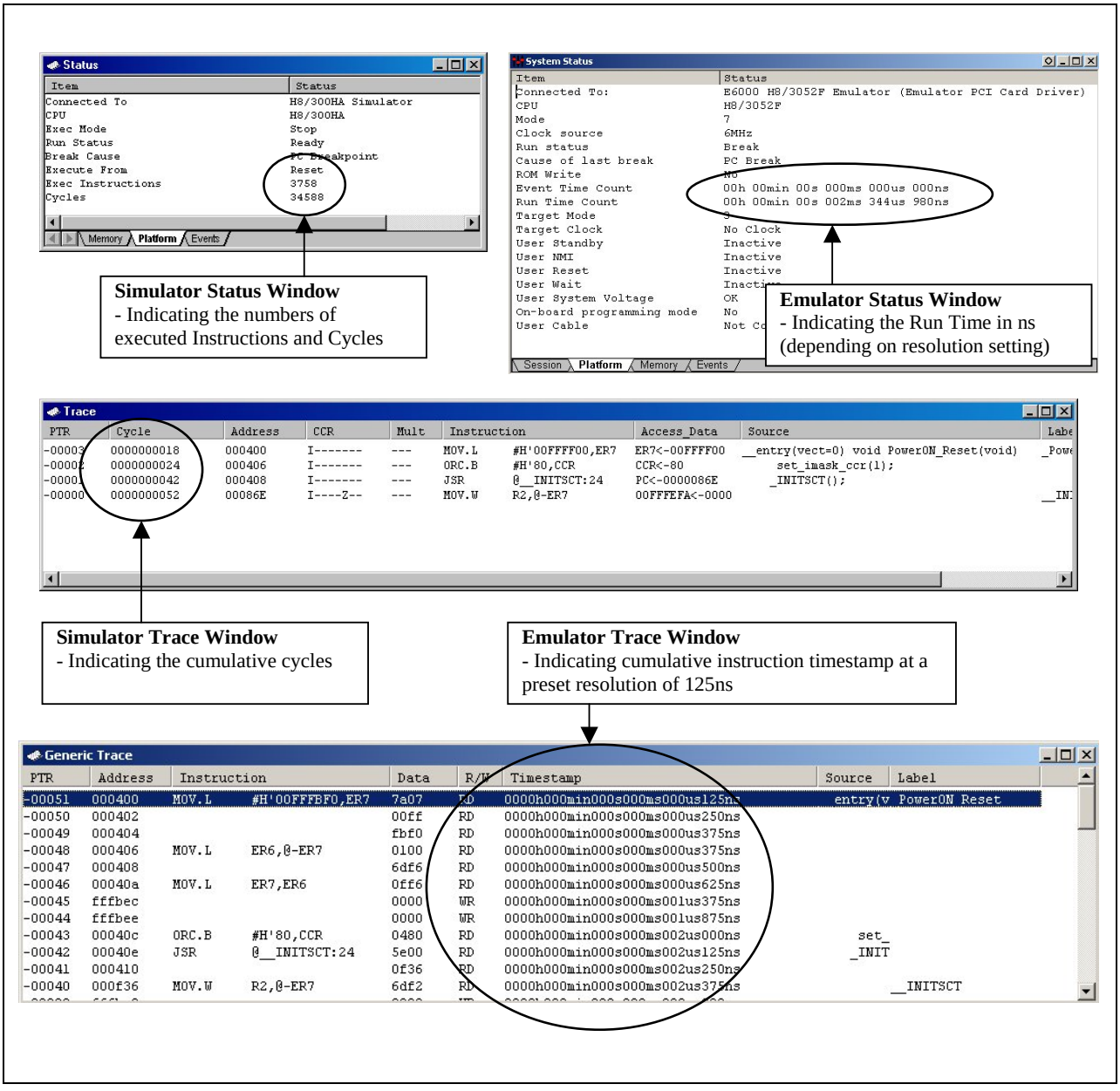
```
*** Mapping List ***
SECTION              START     END       SIZE   ALIGN
$VECT0               00000000  00000003     4    0
P                    000017f6  000049a9    31b4  2
D                    000049aa  000049aa     0    2
C                    000049aa  00006fcb    2622  2
B                    00ffe3b2  00ffe3b2     0    2
R                    00ffe3b6  00ffe3b6     0    2
S                    00ffee00  00ffefff     200  2
…
FILE=C:\ …\ release\mcu.obj
                     00003a3a  00003b69     130
 _initio             00003a3a     0   none ,g      1
 L96                 00003ad0     0   none ,l      0
 _enable_usb_irq     00003ada     0   none ,g      1
 _disable_usb_irq    00003aea     0   none ,g      0
 _start_wdog         00003afa     0   none ,g      0
…
```

Program Section takes up H'31b4 bytes

Enable_usb_irq takes up H'(3aea-3ada) bytes

- Execution speed → The time taken can be obtained by
  - Manual measurement through a hardware mean (such as using stop watch or scope)
  - Emulator / Simulator run-time counter (In HEW Status window)
  - Emulator / Simulator Trace Window - time stamping function.

Example:



3. Further Understanding of Optimization

3.1 Do Not Optimize

It is suggested that unless necessary, optimization shall be omitted. This process should be planned for, and not done at the end of the development cycle, whereby most scenarios had been tested. This may cause changes to the initial design and introduce more wastage of time and resources in debugging.

3.2 Bound Conditions

i. I/O

ii. Memory

iii. CPU

Generally these are the three main bound conditions that will slow the system performance. Excessive activity shall be avoided on I/O access, as it has the slowest events. There are different types of memory. Access to these memories must be managed efficiently to take proper utilization. If neither the I/O nor Memory is the performance deterrence factor, the CPU processing must be the main bottleneck.

3.3 Optimize Hot Spot

It is important to identify the objective of the intended improvement. It may be speed or size. For the speed improvement, the hot spot must be identified. Otherwise time may be wasted on non-critical area.

Pointers:

- In General, 80% of a program's execution time is spent executing 20% of the code.
- The most redundant area is the initialization code, which is used only once.
- Optimize the hot spot even at the cost of making the other area slower.

3.4 Confusion to Source Level Debugger

C Source level debugging is done with reference to the original source code. Since optimization may change the interpretation of the original source code, the debugger may not able to relate the assembly code directly to the C source file accurately.

3.5 Dead Code Elimination

This is just an example of how optimization works. The optimizer will remove code that it sees as redundant. Thus, programmers must define their routines and variables with the correct keyword. Otherwise, useful components may be eliminated by the optimizer.

3.6 Test and Evaluate

It is important to document and keep all test results. Comparison and evaluation of the results, before and after optimization, will help to maintain the integrity of the software.

4. Optimization Jargon

The following are list of commonly used optimization terms/ techniques:

Loop unrolling:	Means repeating lines of code inside a loop.
Loop flipping:	Allow the elimination of the initial conditional jump.
Loop invariant:	Code within a loop, which deals with data values that remain constant as the loop repeats.
Constant folding:	Process of detecting operations on constants, which could be done at compiler time rather than run time
Constant propagation:	Constants used in an expression are combined, and new ones are generated. Some implicit conversions between integer and floating-point types are drawn.
Copy propagation:	The use (copy) of similar values.
Strength Reduction:	Replacing expensive calculation with one that takes less time.
Algebraic transformations:	Use of algebraic properties such as commutativity, associativity and distributive.
Induction variable simplification:	Induction variables change linearly with the loop count. The process includes strength reduction, and simplifies calculations for variables whose value would otherwise be dependent upon the loop index.
Tail recursions:	Placement of function call at the end of the calling function, to reduce the returning process.
Dead store elimination:	Eliminate code that cannot be reached or code whose results are not subsequently used.
Inlining:	Replace function calls with actual program code.
Common subexpression elimination:	Parts of the expressions that appear in several places are computed in temporary variables.

5. How to Optimize – Programming Techniques

The following sections will focus on:

- Programming techniques
- Usage of #pragma directives (for H8 toolchain)

These techniques will enable a better optimization control over each function and module.

However the intelligent HEW optimizer can perform some of these techniques automatically. This will be detailed in section 6, whereby the controls over the whole project or individual files by HEW will be explained.

5.1 Data Handling

5.1.1 RAM Usage

Shortage of RAM space is a common concern. The nominal RAM size for most 8-bit MCU is a mere 1 to 4K bytes size.

Three main components of RAM are:

- Stack
- Heap
- Global data

The reduction in one component will enable the increase in the others. Unlike stack and heap, which are dynamic in nature, global data is fixed in size. Programmers may like to consider the reduction of global data usage, and place more emphasis on local variables control in stack and registers.

- Stack Depth can be estimated using the HEW Call Walker and Profiler. *Please refer to Application Note on "Stack Analysis using Call Walker"*
- Global Data allocation can be viewed in the MAP file (Generated via HEW option → H8 Toolchain → Link/Library → List)

5.1.2 Data Type Usage

The use of correct data type is important in a recursive calculation or large array processing. The extra size, which is not required, is taking up much space and processing time.

Example

- Speed concern:- Byte multiplication - MULXU .B R1L,R2L - take up 12 cycles
  - Word multiplication - MULXU.W R1,ER2 - take up 20 cycles
- Size concern: - char data_collect[100];
  - long data_collect[100]; -take up 4 times more spaces

Other considerations:

- Programming algorithm
- Instead of accessing external bus for two times, it may be better to read the data as a word (16-bit external data bus), and process it as a byte.

5.1.3 Use Variables of the Same Type for Processing

Programmers should plan to use the same type of variables for processing. Type conversion must be avoided. Otherwise, precious cycles will be waste to convert one type to another (Unsigned and signed variables are considered as different types).

5.1.4 Use of Unsigned Type

All variables must be defined as “unsigned” unless mathematical calculation for the signed bit is necessary. The “signed-bit” may create complication, unwanted failure, slower processing and extra ROM size.

5.1.5 Float and Double

Maximum value of Float = 0x7F7F FFFF

Maximum value of Double = 0x7F7F FFFF FFFF FFFF

To avoid the unnecessary type conversion or confusion, programmers can assign the letter “f” following the numeric value.

```
x = y + 0.2f;
```

To further limit the use of double, programmers can set the option “Treat double as float” in the HEW Option → H8 Toolchain → CPU window.

5.1.6 Data Declaration - Constant

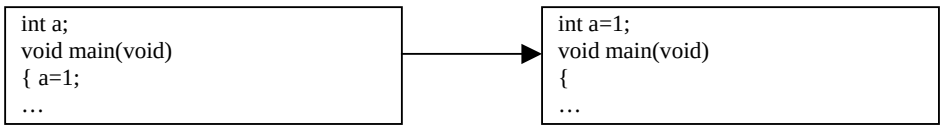
The “const” keyword is to define the data as a constant, which will allocate it in the ROM space (section C). Otherwise a RAM space will also be reserved for this data. This is unnecessary as the constant data is supposed to be read-only.

5.1.7 Data Declaration - Volatile

“Volatile” keyword will forbid the compiler from performing any optimization on the variable. This is usually used on IO registers and variables that will be altered by interrupts. This is necessary as the value of these variables can be asynchronously accessed.

5.1.8 Data Initialization at Declaration

Data should be initialized at declaration.

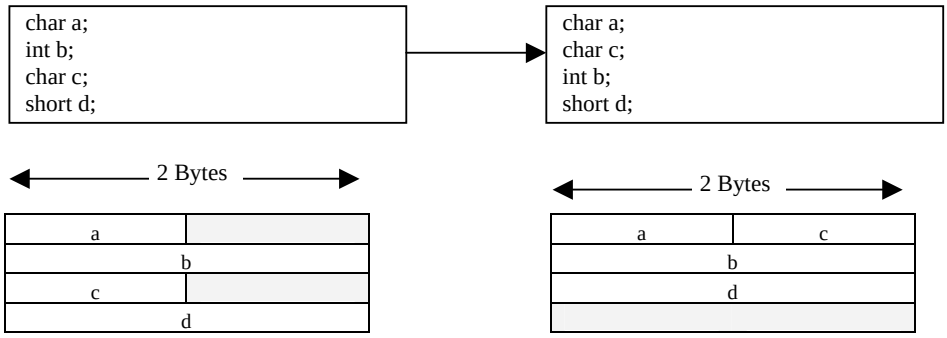


In the first case, ‘a’ being an uninitialized data (B section) will require the program to perform assignment instructions when main routines is entered (Taking up of P section). However if data is initialized during declaration, the compiler will treat the data as an initialized data (D section). These data will have their initial values loaded at the startup stage, whereby D section (ROM) is copied to the R section (RAM).

In comparison, the second method (data initialization at declaration) will be more efficient, as the whole section of data is copied instead of variables by variables.

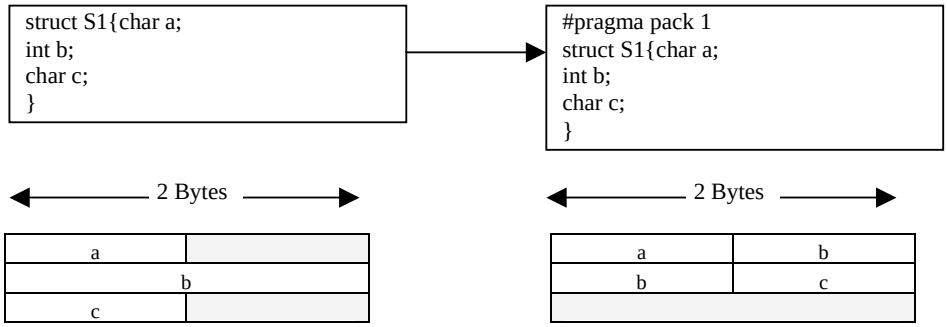
5.1.9 Data Definition - Arrangement and Packing

The declaration of the components in a structure will determine how the components are being stored. Due to the memory alignment, it is possible to have dummy area within the structure. It is advised to place all similar size variables in the same group.



Alternatively programmers can set the setting in HEW Options → H8 Toolchain → C/C++ → Other → "Group data by alignment", and the compiler will "rearrange" to group similar type of variables together.

The similar issues will happen when declaring elements for a structure. In this case, the setting to "compress" the structure is at Options → H8 Toolchain → CPU tab → "Pack struct, union and class"



As the structure is packed, integer b will not be aligned. This will improve the RAM size but operational speed will be degraded, as the access of 'b' will take up two cycles.

5.1.10 Global and Local Variables

Local variable is preferred over the global variable in a function. Generally, global variables are stored in the memory, whereas, local variables are stored in the register. Since register access is faster than the memory access, implementing local variables will improve speed operation. Moreover, code portability also encourages the use of local variables.

However if there are more local variables than the available registers, the local variables will be temporary stored in the stack.

5.1.11 Passing Parameter Registers and Working Registers

In a function call, the parameters will be stored in the passing parameter registers (ER0, ER1 and/or ER2), whereas the working registers (ER3, ER4, ER5) will be used for any data manipulation within the function. In general, the increase in the number of registers being utilized will improve the operation speed of the modules. However due to the limited number of MCU registers, process, global allocated register, number of local variables assigned and etc… this increase in number of passing parameter registers and working registers will not guarantee an improvement. Thus, programmers must be careful when using this feature.

- Do not unnecessary declare any bigger size variables.
- Limit the number of variables (about 6x byte size variables).

HEW allows programmers to have control on the allocated registers.

- To increase the number of passing parameter registers:
  - #pragma regparam 2/3 directives
  - Options→H8 Toolchain →CPU tab → "Change number of parameter registers from 2(default) to 3"

- To increase the number of working registers:
  - Options→H8 Toolchain →C/C++ Tab →Other Category →"Increase a register for register variable"

5.1.12 Global Register Variables

The declaration of "register" can be used if the variable is accessed very frequently. Operation speed will be greatly improved. However, there is only limited registers in a MCU. Moreover this usage will also limit the number of working registers for other processing functions.

Example of identifying frequently accessed variable:

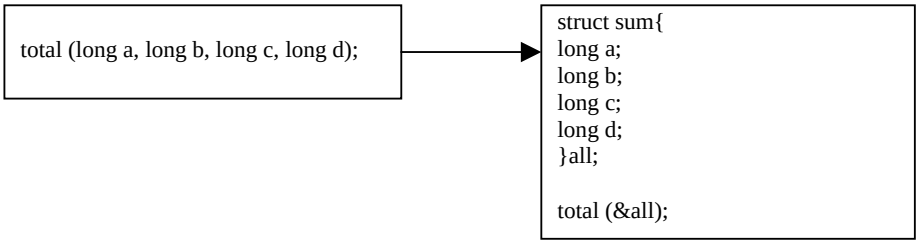
- Generate the project map file at Options→ H8 Toolchain→ Link/Library→ List Category
- Check for the most frequently [Counts] accessed variables in the map file
- Assign the variables to the registers. Programmers have to match the size.
- Generally ER4 and ER5 can be used. This makes up 8 bytes of data.
- #pragma global_register (global_variable R4)

Note: Global registers cannot be used when a library is specified as an object for inter-module optimization.

5.1.13 Passing Reference as Parameters

Larger numbers of parameters may be costly due to the number of pushing and popping actions on each function call. It is more efficient to pass structure reference as parameters to reduce this overhead.

Example:



5.1.14 Return Value

The return value of a function will be stored in a register. If this return data has no intended usage, time and space are wasted in storing this information. Programmer should define the function as “void” to minimize the extra handling in the function.

5.1.15 Register Save and Restore

It is a usual practice for function to save registers during entry, and restore registers upon exit. However if the caller function saves and restores all registers, the called functions do not need to save and restore any registers.

Example:

- #pragma noregsave / regsave

```
#pragma noregsave ( fun1,fun2, fun3)
#pragma regsave (fun)
fun()
{
     fun1();
     fun2();
     fun3();
}
```

- Options→ H8 Toolchain → Link/ Library → Optimize Category → “Reallocate registers”

The above demonstrates the possibility of removing the whole register saving and restoring process.

Another possible optimization method is to improve on the register save & restore process.

The two implementations of saving and restoring registers are:

- PUSH and POP the required registers (Take up larger ROM Space but it is faster)
- Call a runtime routine to save and restore all register (Slow execution but smaller ROM size)

This can be controlled via

- Options→ H8 Toolchain → C/C++ Tab → Optimize Category → “Register”

5.1.16 Use of Short Addressing to Access Variables

This is to make use of the native instruction to access frequently-used variables. These instructions take lesser ROM space than the absolute addressing type.

There are three possible settings:

i. Options→ H8 Toolchain → C/C++ tab → Data Access → "@aa [default], @aa:8 or @aa:16"

- This allows the control to limit a C/C++ file, or globally to a project.
- Section $ABS 8/16 must be defined.

ii. Options→ H8 Toolchain → Link/ Library → Optimize Category → "Use short addressing"

- This allows HEW Linker to judge and control the whole projects.

iii. #pragma section $ABS8/16

- Programmers can make use of this directive to control the location (short addressing space/section) of the desired variables.

Due to the nature of "short addressing mode" (limited space to store the variables), it is not feasible to allocate all variables within the 8 and 16 bit absolute address space.

Example:

- 8 bit absolute address area (<$ABS8> sections) available in the advanced mode range from H'FF FF00 to H'FF FFFF.
- 16 bit absolute address area (<$ABS16> sections) available in the advanced mode range from H'FF 0000 to H'FF FFFF.

Therefore it is necessary to identify frequently accessed variables, to be placed within the sections. This can be determined in the map file. Example

```
…
…
SECTION=B
FILE=C:\… \.xxx.obj                00ffdf20  00ffdf23       4
  _count1                          00ffdf20       2   data ,g       3
  _count2                          00ffdf22       2   data ,g       9
…
…

*** Variable Accessible with Abs8 ***
SYMBOL                       SIZE    COUNTS  OPTIMIZE
…
…
*** Variable Accessible with Abs16 ***
SYMBOL                       SIZE    COUNTS  OPTIMIZE
_count3                  2           1
_count4                  2           2
_count5                  2           16
_count6                  2           4
…
…
```

Frequent usage

5.1.17 Pointer and Array

A pointer will be more efficient than using an array. This is due to the use of register addressing modes (@Rn, @Rn+, @-Rn).

5.1.18 Better Data Structure and Representation

Proper data structure consideration can improve the program.

Example

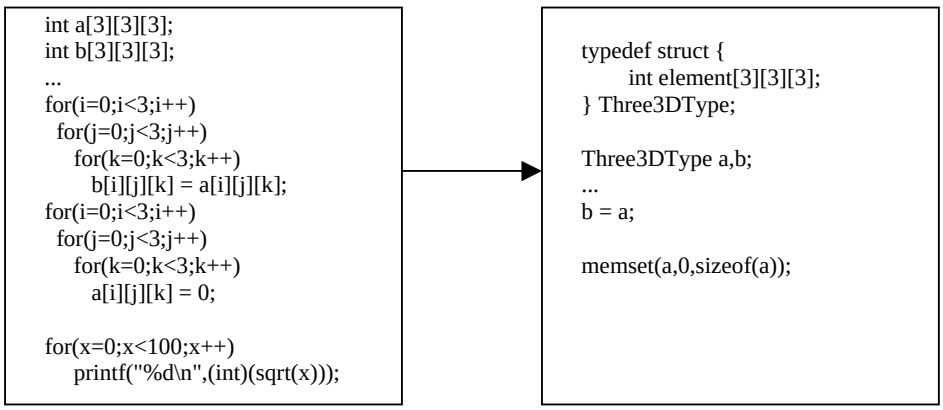
- Use computation to regenerate a large junk of data (compression, technique), this will reduce the space usage. However, the computation process may slow down the operation. (Instead of having a array of [0,0,0,0,0,0,1,1,2,2,2,3,3,3,3…], this can be replaced with [6, 2, 3, 4…], which signifies 6x'0', 2x'1', 3x'2', 4x'3'…)

5.1.19 Accessing Structure

Structure enables efficient access of variables. The explicit address of the structure is encoded only once, when being loaded into a register. Thereafter, all members of the structure are referenced in relative register mode.

5.1.20 Array and Structure Initialization

A simple illustration of implementation:



5.2 Flow Control Handling

5.2.1 Switch

There are two possible techniques in implementing a Switch statement: If–then and Table

This can only be controlled via Option → H8 toolchain → C/C++ → Optimize Category → Switch Statement “Auto/ If then/ Table”

```
switch (test)
{
        case 1:         P_IO.PDR9.BYTE = 0x11;
                break;
        case 2:         P_IO.PDR9.BYTE = 0x22;
                break;
        case 3:         P_IO.PDR9.BYTE = 0x44;
                break;
        case 4:         P_IO.PDR9.BYTE = 0x11;
                break;
        case 5:         P_IO.PDR9.BYTE = 0x22;
                break;
        case 6:         P_IO.PDR9.BYTE = 0x44;
                break;
        other:          P_IO.PDR9.BYTE = 0x88;
                break;
}
```

The compiler generated assembly code based on

“If-Then” setting

```
MOV.W    @(-4:16,R6),R0
MOV.B    R0H,R0H
BNE      L83
CMP.B    #1,R0L
BEQ      L72
CMP.B    #2,R0L
BEQ      L73
CMP.B    #3,R0L
BEQ      L74
CMP.B    #55,R0L
BEQ      L75
CMP.B    #99,R0L
BEQ      L76
CMP.B    #77,R0L
BEQ      L77
BRA      L83
```

“Table” setting:

```
MOV.W    @(-6:16,R6),R0
SUBS.W   #1,R0
MOV.W    #5,R5
CMP.W    R5,R0
BHI      L83
MOV.B    @(L84:16,R0),R0L
SUB.B    R0H,R0H
ADD.B    #LOW L72,R0L
ADDX.B   #HIGH L72,R0H
JMP      @R0
```

Table

- Table implementation will be preferred if there are many “cases”. However if the “cases” conditions are not in sequential number, the table will not be able to be generated.
- Table implementation has same execution speed for all cases.

If-Then

- It has the overall efficiency if there are lesser cases.
- Place cases of high occurrences (or events in needs of fast response) in the earlier order. This will improve the hit rate and thus the speed of operation.

5.2.2 Jumps

A sequential execution will be faster than a program flow that has many jumps condition.

Examples:

- Inline function
- "Else clause removal"

5.2.3 Inline Function

The technique will cause the compiler to replace all calls to the function, with a copy of the function's code. This will eliminate the runtime overhead associated with the function call. This is most effective if the function is called frequently, but contains only a few lines of code.

Example:

```
#pragma inline (sum)
…
int sum(int a, int b)
{           return (a+b);
}
…
routine()
{           …
            total  = sum (x,y);
            …
            sub_total = sum (cost_a, cost_b)
            …
}
```

5.2.4 Function Calls and Addressing Modes

There are numerous methods to reach a location (function calls): absolute, relative and indirect.

The relative addressing mode and indirect mode are more compact (smaller size), as it does not specify the full address (absolute addressing mode).

Examples:

i. Absolute (JSR @aa:16 – 4 bytes and 8 cycles) and Relative Access (BSR d:8 – 2 bytes and 6 cycles)

- A simple mean to achieve a relative addressing is to place all related functions within a file. (Note a BSR d:8 can access to address at –128 to 127 ranges).
- Options → H8 Toolchain → Link/Library → Optimizer Category → Optimize branches (dealing with BSR and JSR)

ii. Indirect Access (JSR @@aa:8 – 2 bytes & 8 cycles)

- Options → H8 Toolchain → C/C++ → Optimizer Category → Function call – "@aa[default] or @@aa:8"
  - In this case, a section <$INDIRECT> must be declared within the address 0-0xFF
  - If <indirect.h> is specified, all run-time routines to be used are called in the indirect accessing format.
- #pragma indirect

```
#pragma indirect fun
void fun (void);
void main(void){
            fun();
}
```

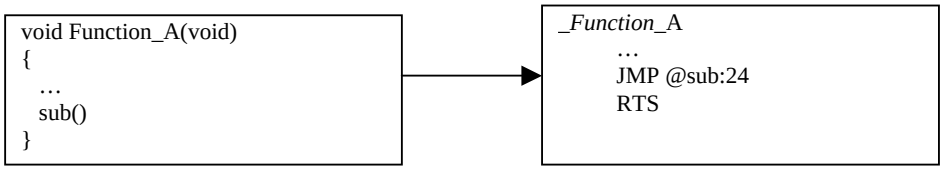
Function "fun" called by "JSR @@$fun"

5.2.5 Tail Recursions

Place the function call instruction at the end of the routine to improve operation speed and size.

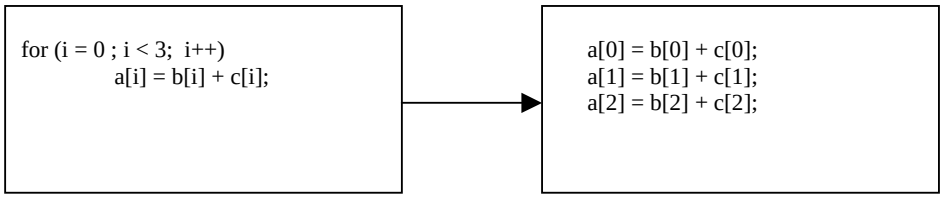
Provided:

- The calling function does not place its parameter or return value address on the stack
- The function call is followed by the RTS instruction.

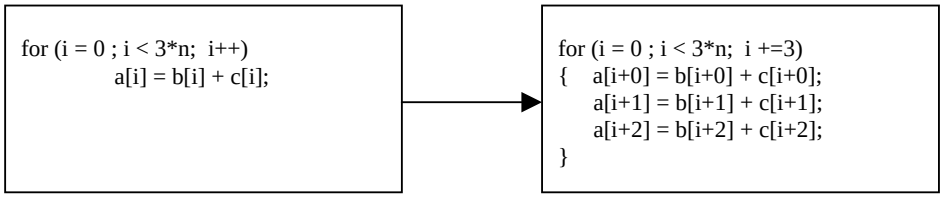


5.2.6 Loop Unrolling

If the loop is small, overhead will be higher.

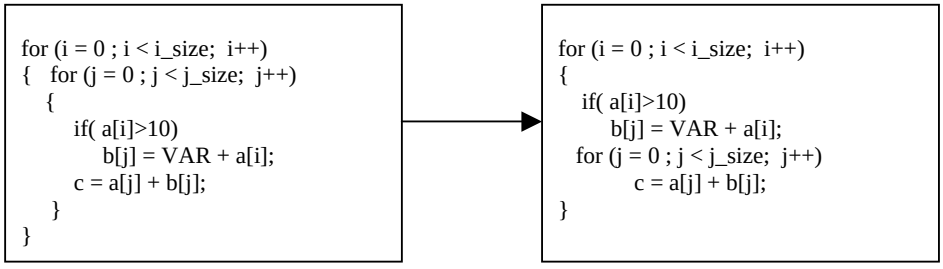


If the loop is larger, the overhead can also be reduced by:



5.2.7 Loop Invariant IF Code Floating

'IF' statements that do not change from iteration to iteration may be moved out of the loop



5.2.8 Do-While Statement

The Do-While statement will have 1 comparing iteration lesser than the while loop. This will improve the operation if the loop must be performed at least once (Loop Flipping).

5.2.9 Loop Hoisting

```
for (i = 0 ; i < i_size;  i++)
{   if (Condition_A)
        Dothis(i);
    else if (Condition_B)
        Dothat(i);
    else
        Doall(i);
}
```

→

```
if (Condition_A)
{    for (i = 0 ; i < i_size;  i++)
        Dothis(i);
}
else if (Condition_B)
{    for (i = 0 ; i < i_size;  i++)
        Dothat(i);
}
else
{    for (i = 0 ; i < i_size;  i++)
        Doall(i);
}
```

For the first case, Condition_B will have an overhead of the “if(Condition_A)”.

5.2.10 Common Expression Should be Calculated Once or Earlier

Parameter can be calculated at earlier stages, such as the power up initialization stage instead of actual execution stage. This will help to speed up the processing.

```
for (i = 0 ; i < end;  i++)
{           c = sqrt(a_type, b_type);
            d = c + i;
            …
}
```

→

```
c = sqrt(a_type, b_type);

for (i = 0 ; i < end;  i++)
{           d = c + i;
            …
}
```

5.2.11 Else Clause Removal

```
if (Condition_A)
  DoThis();
else
  DoThat();
```

→

```
Dothat();
if (Condition_A)
{  Undo_DoThat();
   DoThis();
}
```

A jump condition is inevitable in the first case, whereas the second case gives the higher possibility process (Condition_A) to have a sequential execution. This technique is possible only if Undo “DoThat()” process is possible.

5.2.12 Loop Overhead

The MCU have a conditional branch mechanism that works well when counting down from positive number to zero.

```
for (i=0; i<10;i++)
{
…
}
        …
        DEC R1
        CMP 10
        BNE  L1
        …
```

→

```
for (i=10; i>0; i--)
{
…
}
        …
        DEC R1
        BNZ  L1
        …
```

It is easier to detect “Zero” (Branch non zero – single instruction) than a “predefined number” (Compare and perform jump – two instructions)

5.3 Other Handling

5.3.1 Native Instruction and Data

It is wise to understand the CPU architecture and instruction sets. This will enable a better use of program to achieve a faster execution and smaller program size.

Example:

1. For H8, since bit manipulation is possible (BSET,BCLR …) ,

   Thus usage of if(P_IO.PDR3.BIT.P30) instruction will be more efficient

   Than if(P_IO.PDR3.BYTE & 0x01) instruction

2. Although there is a Multiplication instruction/opcode (MUL), it may be better to use the shift instruction instead.

   MUL will take up 12 or 20 cycles, whereas SHILL takes up 2 cycles

5.3.2 Hand Coded Assembly

Further optimization may be obtained by coding in assembly language.

There are two methods in HEW:

i. #pragma asm

```
motor_control()
{
        #pragma asm
                CLRMAC
        #pragma endasm
}
```

ii. #pragma inline_asm

```
#prama inlin_asm(shlu)
extern unsigned int x;
static unsigned int shil(unsigned int a)
{
        SHLL.W      R0
        BCC         ?L1
        SUB.W       R0,R0
       ?L1:                    /* Local label starts with ? */
}
void main(void)
{
        x = shlu(x)             /* Inline expansion is performed */
                                /* within the main function */
}
```

5.3.3 Lookup Table and Calculation

In lower operation frequency of MCU, lookup table may be faster than recalculation methods. However, programmers must make their judgment on the complexity and speed requirement.

Example:

A function like $y = ax + bx^2$ will already take up significant CPU processing time.

However, the function y = 2x can be implemented with a shift instruction (2 cycles). Thus this function is preferred to be implemented with re-calculation method than a lookup table method.

5.3.4 Polling and Interrupt

Interrupt latency may be a bigger overhead than implementing polling method.

5.3.5 Fixed-point and Floating-point Arithmetic

It takes up much processing power to perform floating-point arithmetic in a non-floating point processor. If accuracy is not a requirement, programmers should use fixed-point calculation instead. Otherwise, the calculation can be re-implemented in a cheaper mean.

Example:

- 123.45 + 678.89 is equivalent to (12345 + 67889) with a decimal point placed at the correct place.

5.3.6 Standard Library Routines

Most standard library routines are written to cater for all possible conditions. Thus, it may not be efficient for specific operation.

However, if the library function is specifically written for the application, it shall be wise to use it as optimization should be taken care of.

Example:

- The printf function takes up a huge space as it is written to cater for floating point arithmetic.
- However, HEW can disable this floating-point facility by “#include <no_float.h>”.
- Further optimization is possible if printf is custom written.
- *Please refer to Application Note: “Writing a printf function to LCD & serial port”*

5.3.7 Input and Output Access

The following are some guides on dealing with the worst bound condition - Input and Output access:

- Unnecessary I/O access should be avoided.
- I/O access within a loop should be avoided, unless necessary
- Use unformatted (binary) I/O whenever possible.
- Access data from memory. If possible, data is to be read and stored in memory for processing (non-volatile)

5.3.8 Specify Optimization Type for Each Module

#pragma option can be used to limit and control the optimization regions. It will have the priority over the HEW Option window setting

```
#pragma option speed                    // From this point, code will be optimized based on Speed
void function_A(void)
{
…
}
#pragma option size                     // From this point, code will be optimized based on Size
void function_A(void)
{
…
}
```

5.3.9 Horner's Rule of Polynomial Evaluation

The rules state that a polynomial can be rewritten as a nested factorization. The reduced arithmetic operations will have better ROM efficiency and execution speed.

$Ax^5 + Bx^4 + Cx^3 + Dx^2 + Ex + F$ → ((((Ax + B) * x + C) * x + D) * x + E) * x + F

5.3.10 Factorization

The compiler may be able to perform better when the formula

Z = X* A + X* B + X* C + X* D → Z = X* (A + B + C + D)

5.3.11 Use Finite Differences to Avoid Multiplies

```
for (i = 0 ; i < 10;  i++)
    printf(" %d\n", i*10);
```
→
```
for (i = 0 ; i < 100;  i+=10)
    printf(" %d\n", i);
```

5.3.12 Condition Determination Using Substitution Values

The assignment statement (MOV instruction) will affect the CCR Zero flag, and thus enable a more efficient loop control.

```
While (*p)
{   *q++ = *p++;
}
*q++  = *p++;
```

→

```
While (*q++ = *p++);
```

5.3.13 Modula

```
x = y %32 ;
```

→

```
x = y &31;
```

5.3.14 Division and Multiplication

```
x = y * 8 ;
x= y / 8 ;
```

→

```
x = y <<3;
x = y >>3;
```

5.3.15 Constant Folding and Propagation

If data is to be calculated during a task, the calculating process may slow down the task execution. These data can be prepared at an earlier stage:

- At compiling stage → as constant data
- At initialization stage

```
y= 3 * 5 * x;
```

→

```
y = 15 * x;

or

z=15;
…
y = z * x;
```

5.3.16 Constant in Shift Operations

For shift operations, if the shift count is a variable, the compiler calls a runtime routine to process the operation. If the shift count is a constant, the compiler does not call the routine, which give significant speed improvement.

```
int shift=8;

data = data <<shift;
```

→

```
#define SHIFT 8

data = data << SHIFT;
```

5.3.17 Use Formula

Example: (Sum of 1 through 100)

```
n=100;
for (x=0, y=1; y<=n; y++)
        x +=y;
```

→

```
n=100;
x = n* (n >>1);  //n^2 /2
```

5.3.18 Simplify Condition

```
If ( a==b && c==d && e==f )
  {…}
```

→

```
If( (( a-b) | (c-d) | (e-f)) ==0 )
   {…}
```

```
if( x>=0 && x<8 && y>=0 &&y<8 )
    {...}
```

→

```
if( ((unsigned)(x|y))<8 )
    {...}
```

```
if( (x==1) || (x==2) || (x==4) || (x==8)
|| ... )
```

→

```
if( x&(x-1)==0 &&x!=0 )
```

5.3.19 Absolute Value

```
#define abs(x) (((x)>0)?(x):-(x))
```

→

```
static long abs(long x)
{    long y;
     y = x>>31;    /* Not portable */
     return (x^y)-y;
 }
```

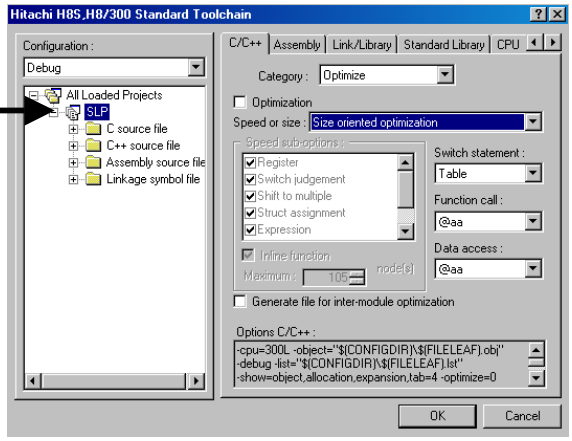
6. How to Optimize – HEW Setting

In this section, HEW optimization techniques will be highlighted. The optimization settings can be applied to the whole project or individual files.

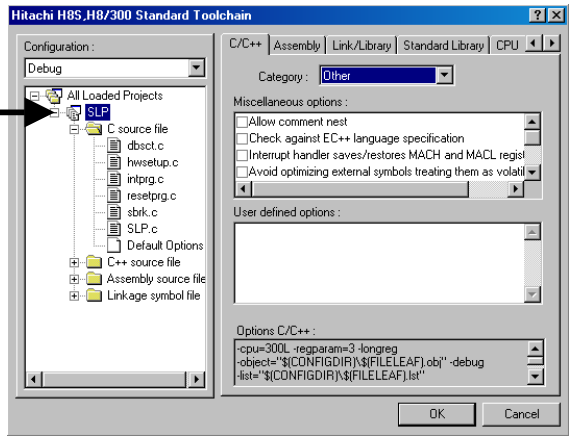
All related setting for Optimization can be set in the HEW Option → H8 Standard Toolchain window.

There are five main areas of interest (Assembly setting is not discussed) for the user:

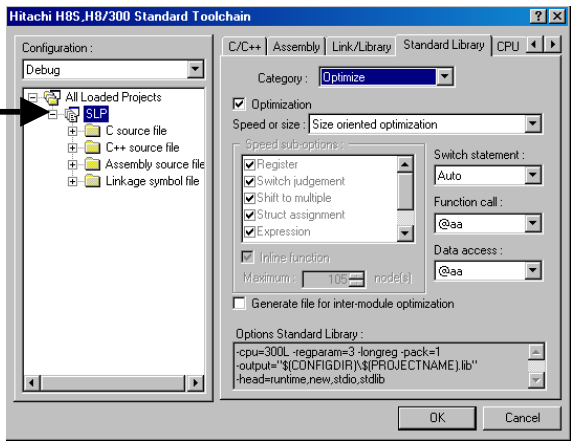
i. C/C++ - Optimize Category



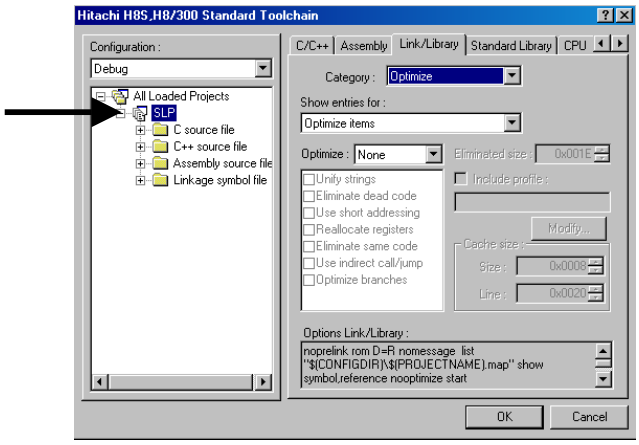
ii. C/C++ - Other Category



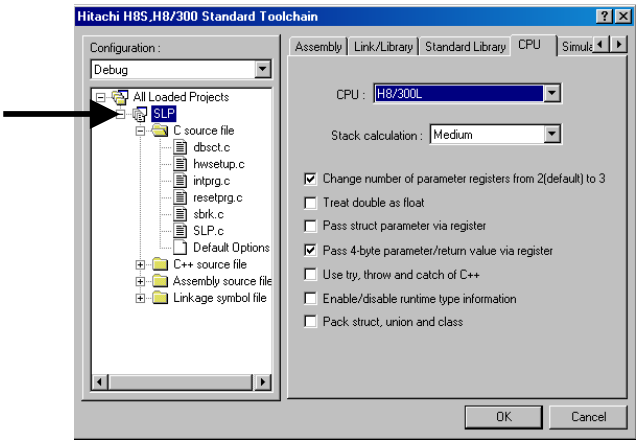
iii. Standard Library



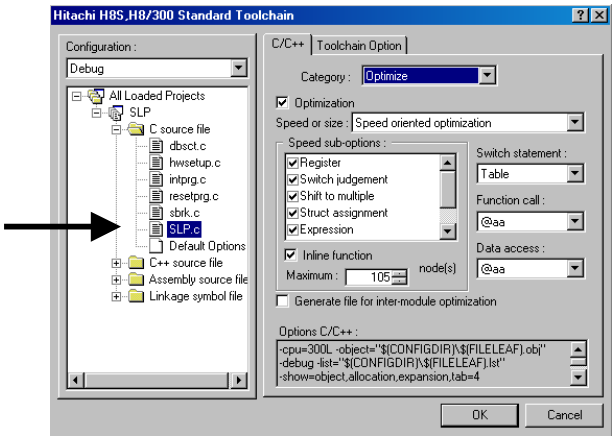
iv. Link/ Library - Optimize Category



v. CPU Setting



NOTE: 1. The previous five selections apply globally to all files in the project. Setting for individual file is possible, if individual file is clicked.

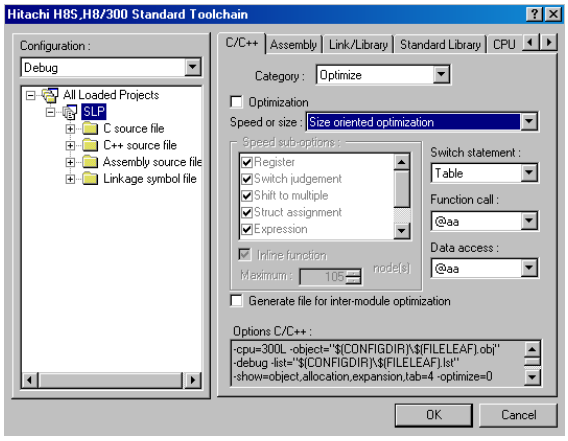


2. This section provides an overview of HEW setting for a single file or whole project optimization. The details (including example) of each settings are illustrated in HEW Application Note for H8 Toolchain.- “Section 5 Using the Optimization Functions” (Available in Renesas web site - Download →Crosstool→ Documents.)

3. Some of the settings’ concepts have been explained in the earlier sections.

4. The #pragma directives have a higher priority level than the HEW option window settings.

6.1 C/C++ - Optimize Category Setting



Two main setting are available:

i. Speed oriented optimization

ii. Size oriented optimization

The setting will select the various optimization techniques:

i. Register

ii. Switch judgment

iii. Shift to multiple

iv. Struct assignment

v. Expression

vi. Loop optimization

vii. Loop unrolling

Other settings are:

i. Inline function: maximum node

ii. Switch statement

iii. Function call

iv. Data access

6.1.1 Register

[Enable] Perform register save and restore by push and pop expansion.

[Disable] Perform register save and restore by using the runtime routine library (if the number of registers to be save and restored is three or more)

[Remarks] This has no effect for H8S series, as STM/LDM or PUSH/PULL will be used

[Refer to Section] 5.1.15 Register Save and Restore

[Speed Improvement] O (Improvement attained)

[Size Reduction] X (Efficiency reduced)

6.1.2 Shift to Multiple

[Enable] The execution time of shift operation can be enhanced.

[Remarks] Multiple shift instructions can be enhanced by performing some looping operations.

[Speed Improvement] O (Improvement attained)

[Size Reduction] X (Efficiency reduced)

6.1.3 Struct Assignment

[Enable] The execution time of structure assignment expression can be enhanced.

[Remarks] Run-time routines are normally called during access to (bigger size) structure or double type data. This will be removed if the option is enabled to improve operational speed.

[Speed Improvement] O (Improvement attained)

[Size Reduction] V (Improvements achieved in some programs)

6.1.4 Expression

[Enable] Execution time of the arithmetic operations, comparison expression, and assignment expression are enhanced.

[Remarks] The operations are expanded with codes that do not access run-time routine.

[Speed Improvement] O (Improvement attained)

[Size Reduction] X (Efficiency reduced)

6.1.5 Loop Optimization

[Enable] Induction variable in a loop statement is eliminated and loop is expanded.

[Remarks] This is possible if

- The initial value for the loop is a constant
- The final judgment of the loop is a constant
- The number of repetition for the loop is either a multiple of 3 or an even number.
- No goto labels are included in the loop

[Speed Improvement] O (Improvement attained)

[Size Reduction] V (Improvements achieved in some programs)

6.1.6 Loop Unrolling

[Enable] Induction variable in a loop statement is eliminated and loop is expanded.

[Refer to Section] 5.2.6 Loop Unrolling

[Speed Improvement] O (Improvement attained)

[Size Reduction] V (Improvements achieved in some programs)

6.1.7 Inline Function

[Enable] Functions called which are within the specified Nodes size will have its function replaced by its code.

[Selection] Maximum Nodes: This selection limits the maximum size of the target function to be inline. Number of nodes signifies the number of units of compiler internal processing. The larger the size indicates the greater the node numbers. Default is 105 nodes.

[Related Command] #pragma inline

[Refer to Section] 5.2.3 Inline Function

[Remarks] Inline expansion will not be performed if

- Functions including variable parameter
- Functions referencing addresses of parameters.
- Functions in which the type of a real parameter and that of a dummy parameter do not match.
- Functions calling inline expanded function
- Functions that exceed the size limitation of the inline expansion.

[Speed Improvement] O (Improvement attained)

[Size Reduction] X (Efficiency reduced)

6.1.8 Switch Statement

[Enable] The compiler will determine the best method to perform the case statement.

[Selection] Auto[default], “If-Then or “Table”

[Refer to Section] 5.2.1 Switch

[Remarks] Execution speed for all cases in the “Table” implementation will be constant, whereas in the “If-Then” implementation, execution speed will vary for all cases. Thus the highest hit rate case or timing-critical case should be placed in the early stage.

[Speed Improvement] O (Improvement attained)

[Size Reduction] V (Improvements achieved in some programs)

6.1.9 Function Call

[Selection] @aa[default] or @@aa:8

[Enable] @@aa:8 is selected - Function calls will be done in indirect addressing mode. The storage area for the function call will be located in <$INDIRECT> section.

[Related Command] #pragma indirect

[Refer to Section] 5.2.4 Function Calls and Addressing Modes

[Remarks] If <indirect.h> is specified, all run-time routines to be used are called in the indirect addressing format.

[Speed Improvement] X (Efficiency reduced)

[Size Reduction] O (Improvement attained)

6.1.10 Data Access

[Selection] @aa [default], @aa:8 or @aa:16

[Enable] @aa:8 is selected -.Variables will be stored in $ABS8 section and accessed via the 8-bit absolute addressing mode.

(e.g. advanced mode memory range H'FF FF00 - H'FF FFFF for 8 bit absolute address area <$ABS8> sections)

[Enable] @aa:16 is selected - Variables will be stored in $ABS16 section and accessed via 16 bit absolute addressing mode.

(e.g. advanced mode memory range H'FF 0000 - H'FF FFFF for 16-bit absolute address area,).

[Related Command] #pragma abs8/ab16

[Refer to Section] 5.1.16 Use of Short Addressing to Access Variables

[Remarks]

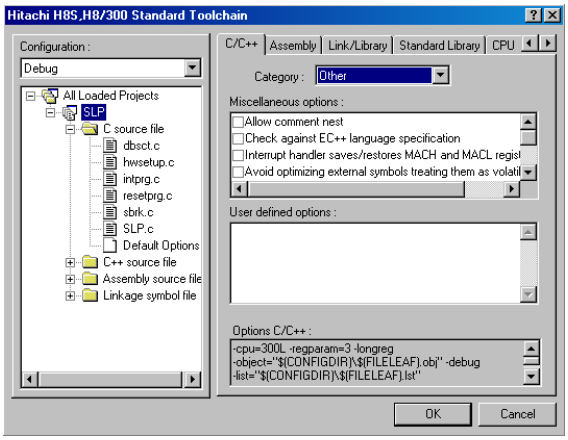
- <$ABS8> section is used to store 8 bit data (char), and <$ABS16> section is used to stored 16 bit data (integer)
- Due to limited memory space, this option may not be feasible to apply to the whole project. Frequently accessed variables must be identified and allocated.

[Speed Improvement] O (Improvement attained)

[Size Reduction] O (Improvement attained)

6.2 C/C++ - Other Category Setting

The available settings in this tab are:



i. Allow comment nest

ii. Check against EC++ language specification

iii. Interrupt handler save/restores MACH & MACL register if used

iv. Avoid optimizing external symbols treating them as volatile

v. Treat enum as char if it is in the range of char

vi. Increase a register for register variable

vii. Put common subexpression on a register temporarily

viii. Use EEPMOV in block copy

ix. Group data by alignment

The option related to optimization will be explained as follow:

6.2.1 Avoid Optimizing External Symbols Treating Them as Volatile

[Enable] All external variables will be treated as volatile. Thus there will be no optimization for all these external variables.

[Disable] The compiler will treat the external variables as what the declaration is.

[Recommend] Disable

[Speed Improvement] -

[Size Reduction] -

6.2.2 Treat Enum as Char if it is in the Range of Char

[Enable] The numeration data declared by enum will be treated as char (in byte form) if they are within –128 to 127 ranges.

[Disable] All data are treated as integer (in word [2x bytes] form)

[Recommend] Enable

[Speed Improvement] O (Improvement attained)

[Size Reduction] O (Improvement attained)

6.2.3 Increase a register for register variable

[Enable] Four registers [(E)R3 to (E)R6] will be used for variable manipulation.

[Disable] Three registers [(E)R4 to (E)R6] are used.

[Recommend] Enable

[Refer to Section] 5.1.11 Passing Parameter Registers and Working Registers

[Remark]: Disable if complicated expression are used in the program

[Speed Improvement] V (Improvements achieved in some programs)

[Size Reduction] V (Improvements achieved in some programs)

6.2.4 Put Common Subexpression on a Register Temporarily

[Enable] Common subexpression will be eliminated when optimized

[Speed Improvement] -

[Size Reduction] O (Improvement attained)

6.2.5 Use EEPMOV in Block Copy

[Enable] Structure assignment is using “EEPMOV” block move instruction.

[Disable] “MOV” instruction or run time routine are to be used.

[Remarks] If an NMI interrupts occurs during EEPMOV operation, control moves to the next instruction after the interrupt processing, and therefore, EEPMOV operation cannot be guaranteed. Precautions must be taken against NMI interrupts when this option is used.

[Speed Improvement] O (Improvement attained)

[Size Reduction] X (Efficiency reduced)

6.2.6 Group Data by Alignment

[Enable] Data of the same type is grouped together.

[Recommend] Enable

[Refer to section] 5.1.9 Data Definition - Arrangement and Packing

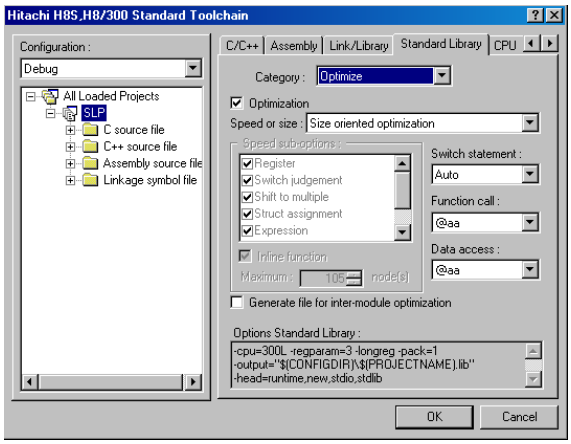
[Speed Improvement] O (Improvement attained)

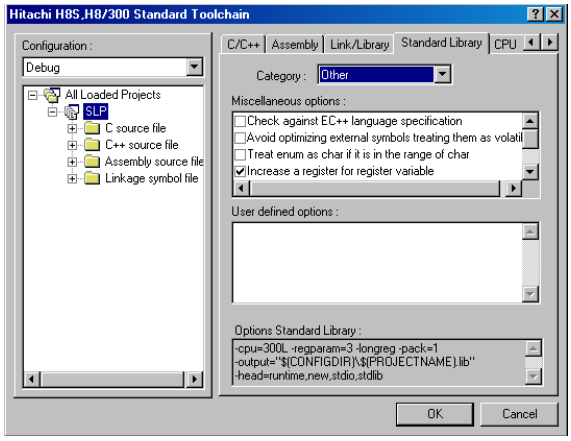
[Size Reduction] O (Improvement attained)

6.3 Standard Library Optimize and Other Category Setting

The available settings in these tabs are similar to C/C++ tabs.

Please refer to the previous C/C++ sections for the explanation.





6.4 CPU setting

The available settings are:

i. Change number of parameter registers from 2(default) to 3

ii. Treat double as float

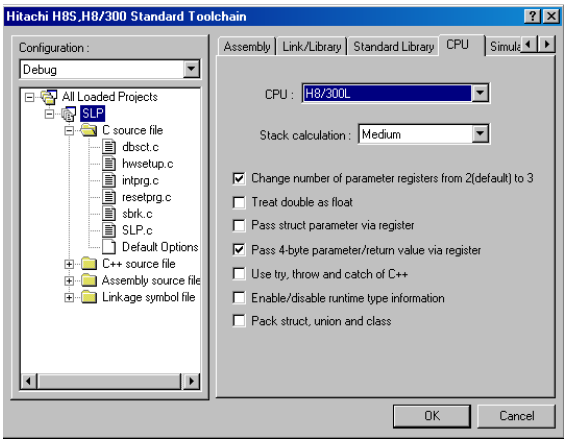
iii. Pass struct parameter via register

iv. Pass 4-byte parameter/return value via register

v. Use try, throw, and catch of C++

vi. Enable/ Disable runtime type information

vii. Pack struct, union and class



6.4.1 Change Number of Parameter Registers from 2(default) to 3

[Enable] (E)R0 , (E)R1 and (E)R2 are used for parameter passing in a function calls.

[Disable] (E)R0 and (E)R1 are used

[Related Command] #pragma repram 2/3

[Refer to Section] 5.1.11 Passing Parameter Registers and Working Registers

[Remarks] This feature is applied to all files and libraries. It cannot be specified individually to each file.

[Speed Improvement] V (Improvements achieved in some programs)

[Size Reduction] V (Improvements achieved in some programs)

6.4.2 Treat Double as Float

[Enable] Both double and float are 4 bytes in length

[Disable] Treats double as 8 bytes and float as 4 bytes

[Refer to Section] 5.1.5 Float and Double

[Speed Improvement] -

[Size Reduction] -

6.4.3 Pass Struct Parameter via Register

[Enable] Parameters are passed through registers

[Disable] Parameters are passed through memory

[Speed Improvement] V (Improvements achieved in some programs)

[Size Reduction] V (Improvements achieved in some programs)

6.4.4 Pass 4-byte Parameter/Return Value via Register

[Enable] Allocates 4 byte parameters to register

[Disable] Allocate 4 byte parameters to memory

[Remarks] The above condition applies to H8/300 only. For other series, 4 byte data is always assigned.

[Speed Improvement] V (Improvements achieved in some programs)

[Size Reduction] V (Improvements achieved in some programs)

6.4.5 Pack Struct, Union and Class

[Enable] Struct, union and class will be packed

[Disable] Struct, union and class will remain as it is declared.

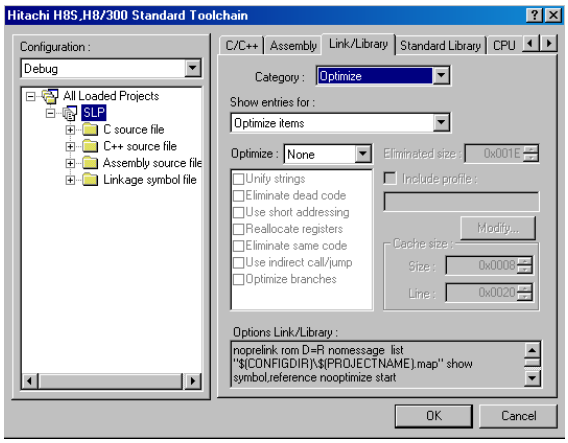
[Refer to Section] 5.1.9 Data Definition - Arrangement and Packing

[Speed Improvement] X (Efficiency reduced)

[Size Reduction] O (Improvement attained)

6.5 Link/ Library - Optimize Category Setting

HEW has enabled a user-friendly approach in selecting the optimization setting:



i. All

ii. Speed

iii. Safe

iv. Custom

v. None

The above setting determine the optimization techniques used:

i. Unify strings

ii. Eliminate dead code

iii. Use short addressing

iv. Reallocate registers

v. Eliminate same code

vi. Use indirect call/jump

vii. Optimize branches

Eliminated size - Specify the minimum size to unify same code (Used for “eliminate same code”)

Profile - Load a profile generated by Debugger.

- This dynamic information will enable the inter-module optimization to be performed.

Relation Between Profile Information and Optimization (for H8 C/C++ and assembly)

i. Variable access - Allocate frequently accessed variables in the first stage

ii. Function call - Lowers the optimized order for frequently accessed functions

iii. Branch - Allocate the frequently accessed function, to a nearby location to the calling function.

6.5.1 Unify Strings

[Enable] Unify similar value constants having the const attributes. Constants having the const attributes are:

- Variable defined as const in C program
- Initial value of character string data
- Literal constant

[Size Reduction] O (Improvement attained)

6.5.2 Eliminate Dead Code

[Enable] Variables and functions that are not referenced and executed in all application will be eliminated

[Size Reduction] O (Improvement attained)

6.5.3 Use Short Addressing

[Enable] Frequently accessed variables are allocated to the 8/16 bit absolute addressing area.

[Refer to Section] 5.1.16 Use of Short Addressing to Access Variables

[Speed Improvement] O (Improvement attained)

[Size Reduction] O (Improvement attained)

6.5.4 Reallocate Registers

[Enable] Function calls relations are investigated so as to reallocate register, and delete redundant register save or restore.

[Refer to Section] 5.1.15 Register Save and Restore

[Remarks] PUSH and POP instructions in a subroutine may be removed, as registers used in the routine are not destroyed.

[Speed Improvement] O (Improvement attained)

[Size Reduction] O (Improvement attained)

6.5.5 Eliminate Same Code

[Enable] Create a subroutine for the same instruction.

[Speed Improvement] X (Efficiency reduced)

[Size Reduction] O (Improvement attained)

6.5.6 Use Indirect Call/Jump

[Enable] Frequently accessed functions are called via memory indirect addressing. Memory address is allocated to the range of 0 to 0xFF if there is a space.

[Refer to Solution] 5.2.4 Function Calls and Addressing Modes

[Size Reduction] O (Improvement attained)

6.5.7 Optimize Branches

[Enable] Optimize branch instruction size according to program allocation information.

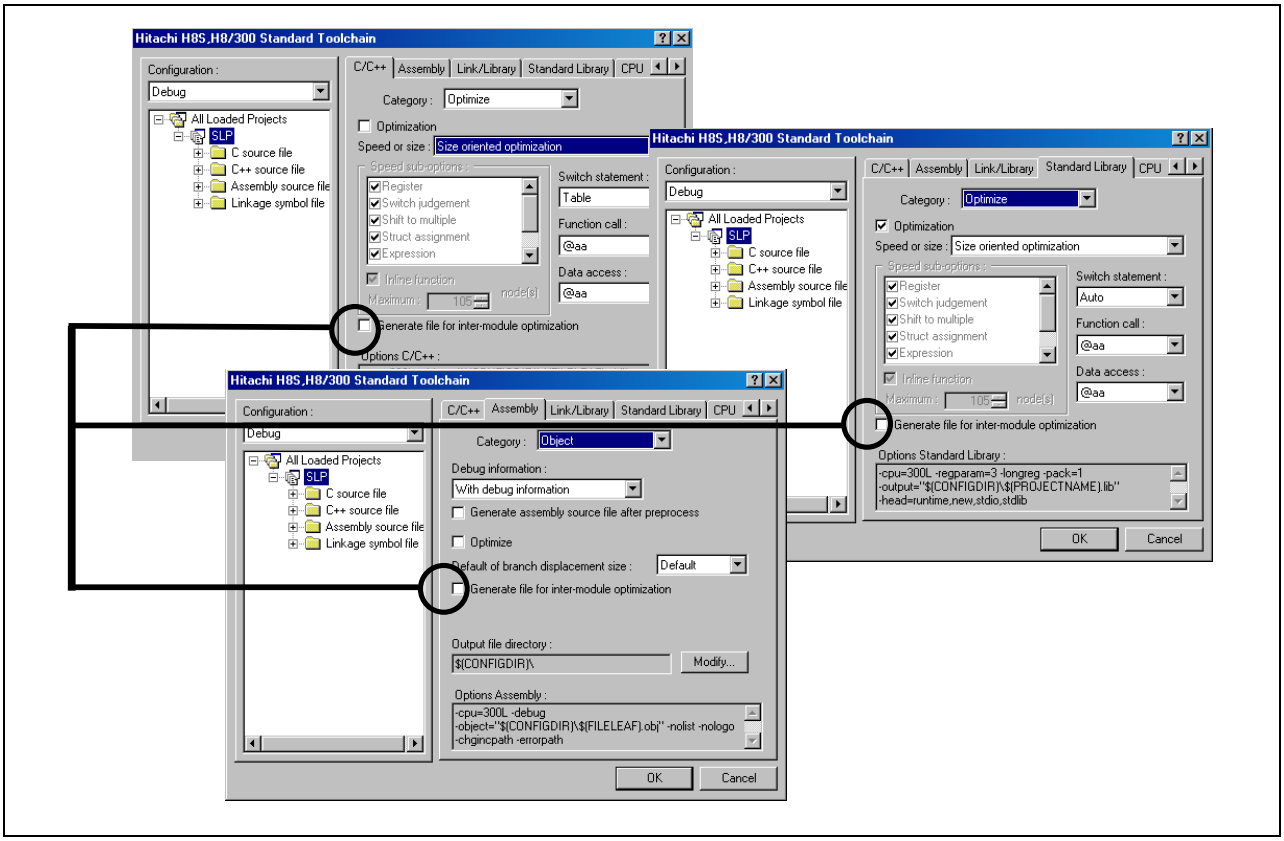
[Remarks] Usage of JSR or BSR

[Refer to Solution] 5.2.4 Function Calls and Addressing Modes

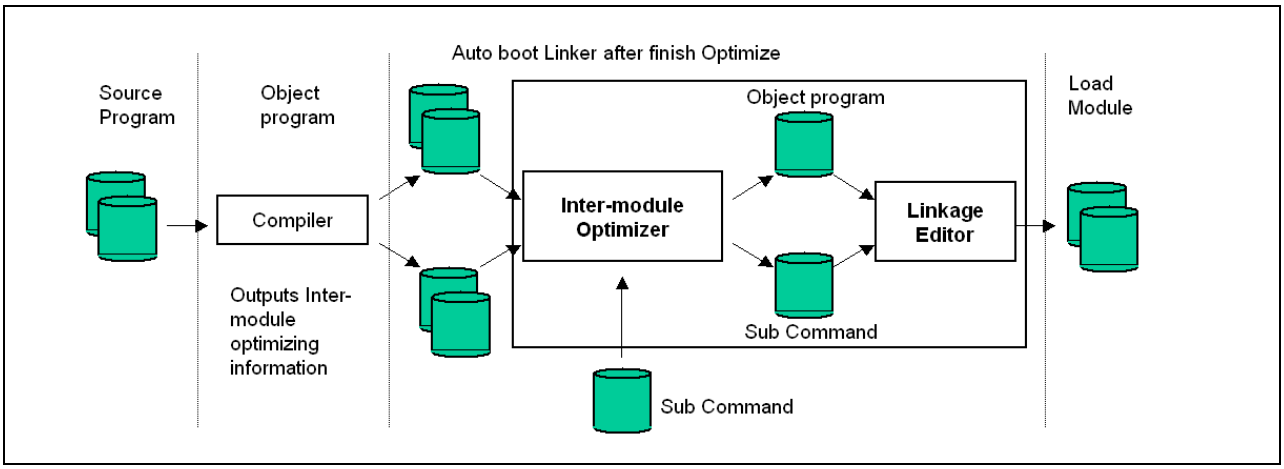
[Size Reduction] O (Improvement attained)

6.6 Inter Module Optimization

Inter-module optimization can be set in the C/C++, Assembly and Standard Library tabs



This will enable inter –module optimizing information to be generated for the linker to perform a better optimization.



The optimize items for inter-modules are similar to the Link/Library tab:

i. Unifies constant/literal strings

ii. Delete no-referenced symbols

iii. Short Absolute addressing

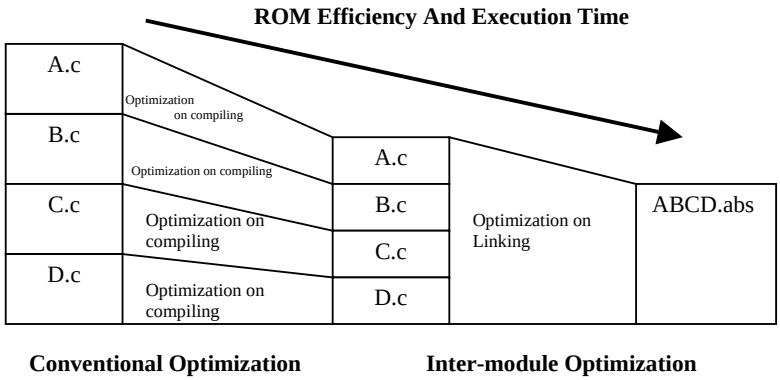
iv. Indirect addressing

v. Register save/restore

vi. Unifies same codes

vii. Uses better branch instruction

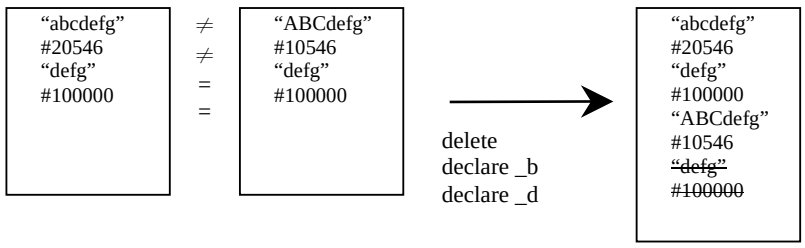
The exception is that it can have better optimization, as condition for all modules (global) are considered, instead of performing the adjustment only for one module (local).



A simplified explanation of the inter-module optimization is illustrated as follow:

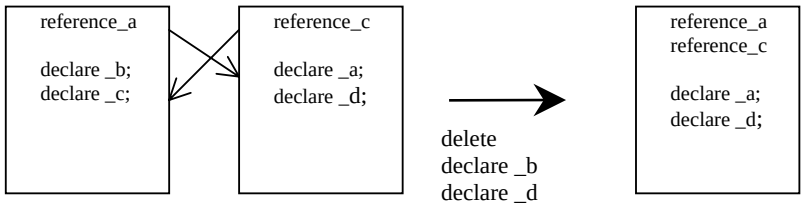
6.6.1 Unifies Constant / Literal Strings

Same strings, constants are searched and eliminated in modules.



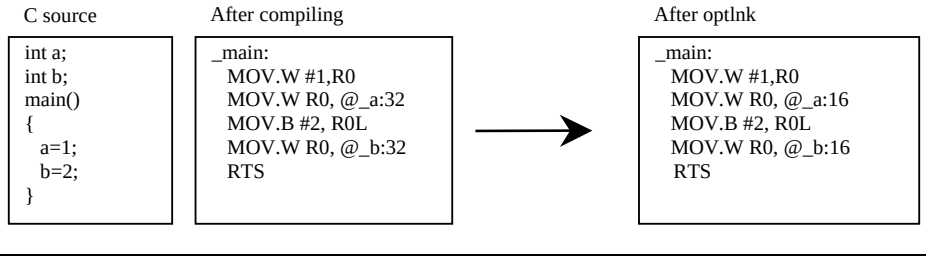
6.6.2 Delete No-referenced Symbols

Deletes variables and functions that are not referenced.



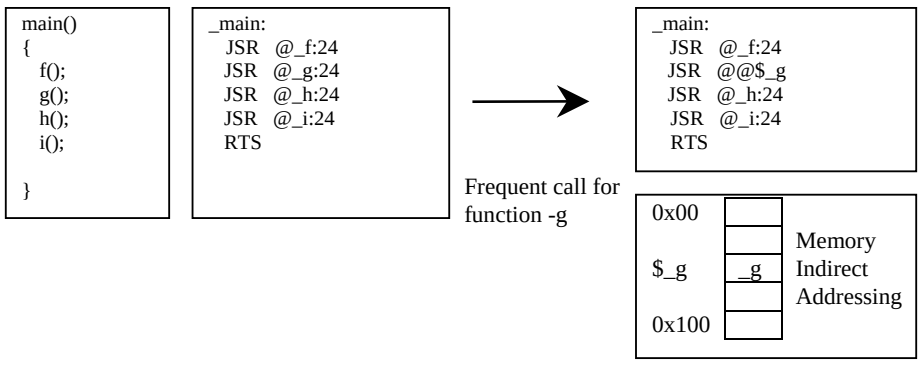
6.6.3 Short Absolute Addressing

Allocate frequently-accessed variables to the area accessible in the 8/16 bit absolute addressing mode.



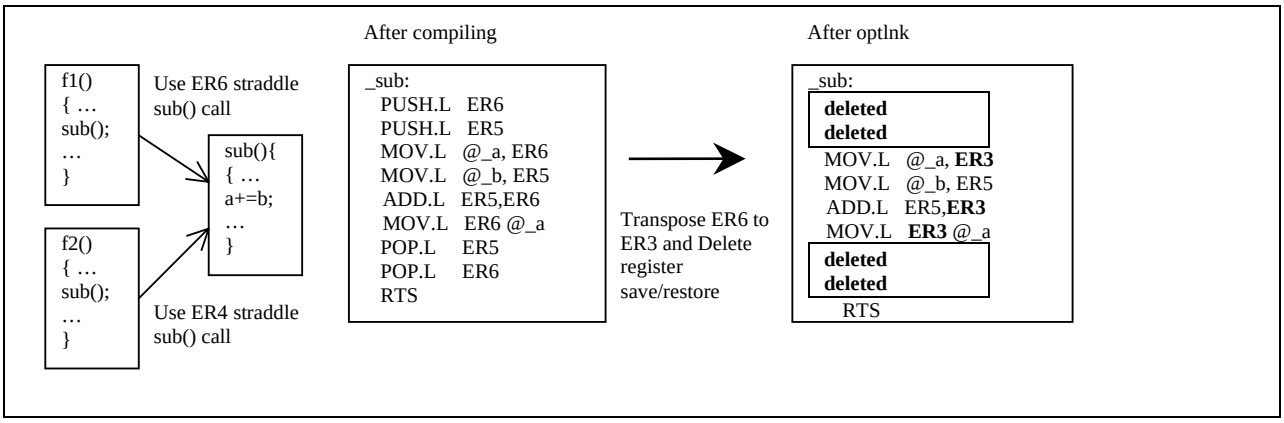
6.6.4 Indirect Addressing

Allocate addresses of frequently accessed functions to the range 0 to 0xFF if there is space.



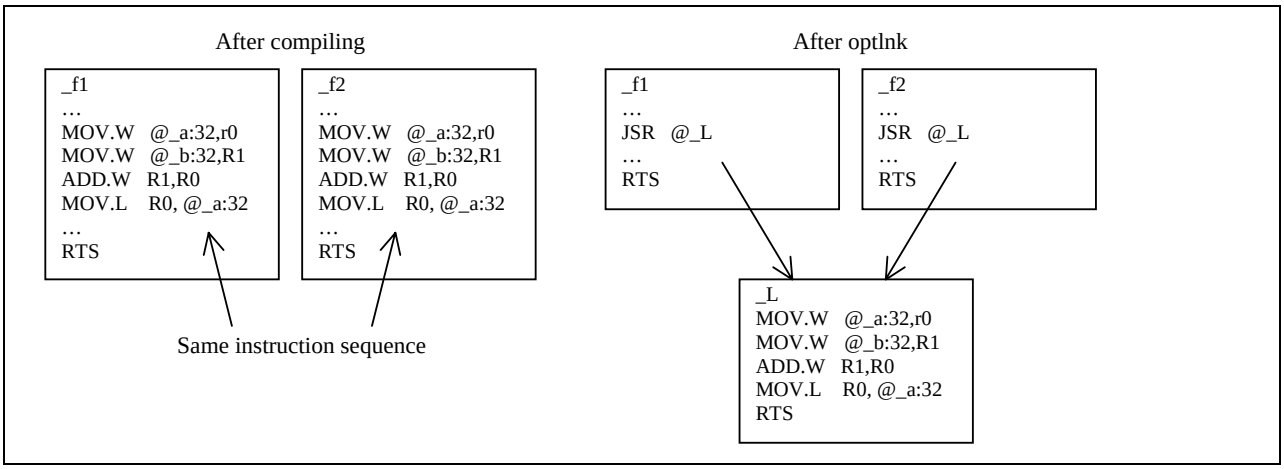
6.6.5 Register Save/Restore

Investigates function calls, relocates registers and deletes redundant register save or restore codes.



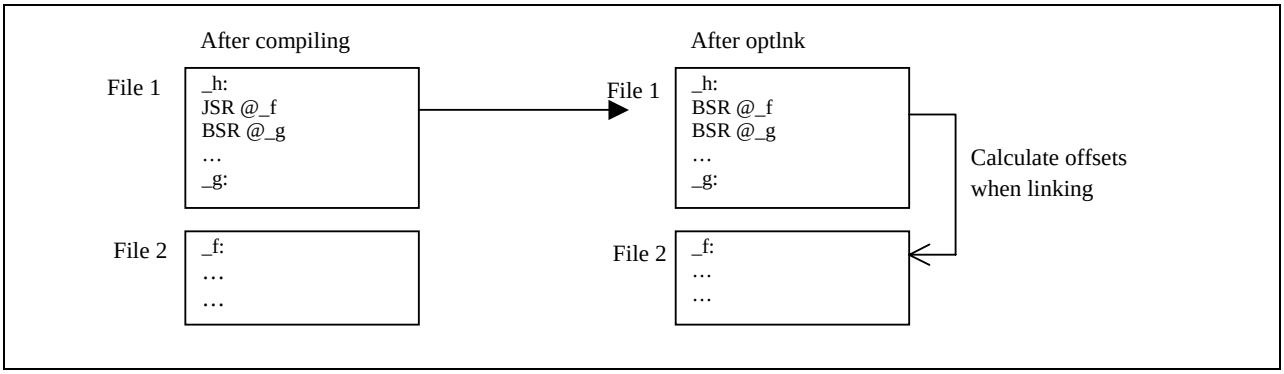
6.6.6 Unifies Same Codes

Create a subroutine for the same instruction sequence.



6.6.7 Uses Better Branch Instruction

Optimizes branch instruction size according to program allocation information



7. Suggested Optimization Steps

The following optimization steps are suggested as a basic guide:

i. Plan for the project – Focus on the areas which need faster execution and smaller code size

ii. Pay extra optimizing effort in coding these critical areas.

iii. Compile (without optimization turn on)

iv. Check, Test and Qualify the system operations.

v. If not necessary, retain current settings.

vi. Otherwise, optimize for speed or size for the actual requirement.

- Make use of utilities such as HEW performance analyzer and profiler to identify the hot spot.
- Make use of the HEW session – Debug, Release and etc, so as to make comparison between session output
- If ROM or RAM space is insufficient, consider the need to optimize the project as a whole, or purely work on certain modules.
- If response is slow, consider the need to optimize for speed in the critical routines, instead of the whole project.

vii. After optimization, perform check, test and qualify the whole system operation against the earlier test results.

8. Conclusion

There are no fixed rules in optimization.

To achieve good optimization, programmers have to plan and implement optimization early. Other than having a good algorithm and techniques, programmers must also have a clear understanding of the controller's architectures and compiler behavior.

This document did not take into the consideration of cache and pipeline. When these topics are involved, programmers will have to make sure that their program will not create too much cache missed and disruptive pipeline operation.

Optimization can be done either through programming or via HEW optimizer setting. There are some HEW optimizer setting that can be implemented via the use of #pragam directives or programming. This is provided for better control. With the right combination of programming techniques and optimizer settings, programmer will be able to perform better optimization.

Reference

- *HEW Application Note for H8 Toolchain [Chapter5, 6 & 7] (Renesas)*
- *HEW C/C++ Compiler, Assembler, Optimizing Linkage Editor manual [Chapter 2, 4, 9 & 10] (Renesas)*
- *A book on C by Al Kelley Ira Pohl (Addison –Wesley)*
- *The Practice of Programming by Brain W.Kernighan & Rob Pike (Addison –Wesley)*
- *Fundamentals of Embedded Software where C and Assembly Meet by Daniel W.Lewis (Prentice Hall)*
- *Programming Embedded Systems in C and C++ by Michael Barr (O'REILLY)*
- *Writing Solid Code by Steve Maguire (Microsoft Press)*

Revision Record

Rev.	Date	Description	
		Page	Summary
1.00	March 04	—	First edition issued

Keep safety first in your circuit designs!

1. Renesas Technology Corporation puts the maximum effort into making semiconductor products better and more reliable, but there is always the possibility that trouble may occur with them. Trouble with semiconductors may lead to personal injury, fire or property damage.
   Remember to give due consideration to safety when making your circuit designs, with appropriate measures such as (i) placement of substitutive, auxiliary circuits, (ii) use of nonflammable material or (iii) prevention against any malfunction or mishap.

Notes regarding these materials

1. These materials are intended as a reference to assist our customers in the selection of the Renesas Technology Corporation product best suited to the customer's application; they do not convey any license under any intellectual property rights, or any other rights, belonging to Renesas Technology Corporation or a third party.
2. Renesas Technology Corporation assumes no responsibility for any damage, or infringement of any third-party's rights, originating in the use of any product data, diagrams, charts, programs, algorithms, or circuit application examples contained in these materials.
3. All information contained in these materials, including product data, diagrams, charts, programs and algorithms represents information on products at the time of publication of these materials, and are subject to change by Renesas Technology Corporation without notice due to product improvements or other reasons. It is therefore recommended that customers contact Renesas Technology Corporation or an authorized Renesas Technology Corporation product distributor for the latest product information before purchasing a product listed herein.
   The information described here may contain technical inaccuracies or typographical errors.
   Renesas Technology Corporation assumes no responsibility for any damage, liability, or other loss rising from these inaccuracies or errors.
   Please also pay attention to information published by Renesas Technology Corporation by various means, including the Renesas Technology Corporation Semiconductor home page (http://www.renesas.com).
4. When using any or all of the information contained in these materials, including product data, diagrams, charts, programs, and algorithms, please be sure to evaluate all information as a total system before making a final decision on the applicability of the information and products. Renesas Technology Corporation assumes no responsibility for any damage, liability or other loss resulting from the information contained herein.
5. Renesas Technology Corporation semiconductors are not designed or manufactured for use in a device or system that is used under circumstances in which human life is potentially at stake. Please contact Renesas Technology Corporation or an authorized Renesas Technology Corporation product distributor when considering the use of a product contained herein for any specific purposes, such as apparatus or systems for transportation, vehicular, medical, aerospace, nuclear, or undersea repeater use.
6. The prior written approval of Renesas Technology Corporation is necessary to reprint or reproduce in whole or in part these materials.
7. If these products or technologies are subject to the Japanese export control restrictions, they must be exported under a license from the Japanese government and cannot be imported into a country other than the approved destination.
   Any diversion or reexport contrary to the export control laws and regulations of Japan and/or the country of destination is prohibited.
8. Please contact Renesas Technology Corporation for further details on these materials or the products contained therein.