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 I (ECProgramI)

Introduction

This Application Note covers the fundamental theory of embedded C Programming, with the objectives of generating small & efficient code, which are easy to maintain, read and debug.

It will highlight:

- How an embedded system control flows
- How C instructions behave in an embedded system
- What is being handled during the power up sequence
- What is uniqueness in embedded C programming
- How ROM and RAM space are being used
- Guides for writing C program

The examples used will be based on High-performance Embedded Workshop (HEW) Version 2 (SLP/TINY C Compiler version 5.0) and H8 SLP is used as the target for explanation. However this explained fundamental concept would be applicable to other compilers and MCU series.

In this document, the emphasis is on the lower control level of the embedded system. i.e the kernel or driver level. Thus some of the discussions may not be applicable to programmer of the application level, as the controls have been taken care by their kernel. However the discussion of various topics will help the programmers, to understand the underlying fundamentals of embedded system programming.

This document will not cover the details of the general C program and the all the setting for the compiler. Reader can obtain many C programming references over the Internet. For compiler detail, please refer to Chapter 9 & 10 of HEW compiler user manual.

Target

All

Contents

1. Uniqueness of Embedded C Programming

In a simplified classification, there are two main systems that C will run.

- The larger system, such as the personal computer, and
- The smaller system that is laid around everywhere, such as the coffee maker, telephone, …etc

Generally these smaller systems operate on an embedded controller, which is coded with either an assembly or C language. As these embedded systems have a dissimilar characteristic as the personal computer, the C language written for these systems must be treated differently → Embedded C.

The embedded controller (or MCU, short for Micro-Controller Unit) will normally have limited ROM and RAM space. Although some systems may have slightly bigger memory space, and run on complex system such as a Real Time Operating System (RTOS). Most embedded systems may be powered on & off at any instant. However some of these systems may be operating in a vigorous environment, in which failure is undesirable, such as car controlling systems and medical appliances.

Due to various needs for the embedded system, writing programs for embedded system have to put in much more consideration. There is a pool of engineers that will still insist on writing assembly language, as embedded C will not be as optimized as the assembly code. However embedded systems have been changing very rapidly. In order to keep up the pace and shorten development time, embedded C programming is inevitable. Embedded system written in C will be much easier to read, debug, maintain and port to another system. Moreover the intelligence of the compiler's optimizer has been improved tremendously over the years.

Unlike the PC platform, embedded system does not have a common standard. Moreover due to the underlying difference in different MCU vendors, compilers for the various MCU vendors behave differently too. Therefore the embedded programmer has to be equipped with the fundamental working principle of embedded C compiler, in order to generate a reliable and optimized system.

Embedded C Programmer has to know and understand the detail of how the C language can control the MCU.

- Unlike a PC, each embedded system has it unique parts, example the standard input/output. For a PC, a printf will send the data string to the monitor, whereas embedded programmer will have to define the output. This may be a serial port or a LCD panel.
- A PC programmer may simply create a huge array. However there may not have enough resource for this memory in an embedded system, as some of the smaller MCU have only 128 bytes of RAM!
- Embedded programmer must be very careful with their functions called and variables usage. Due to limited RAM area, stack overflow may happen if the there are too much levels of function calls.
- Embedded C come with certain intrinsic function such as set interrupt mask, which may be created differently for different MCU family and compiler.
- Embedded programmer needs to know the detailed mapping of a MCU.
- Embedded programmer has to optimize their program in term of speed or size to suit to their application.

- Embedded programmer needs to be worried of system hang/ crash due to unforeseen circumstances, such as external noise to system, low battery supply, enormous key press by users…etc.

- Embedded programmer should build debugging facilities into their system to assist troubleshooting. Unlike PC applications, which run in a control Window environment, it is difficult to trap bugs in an embedded system even with the state-of-the-art in-circuit emulator.

- Every compiler conforms to its own syntax. Thus the embedded system do not understand some additional keywords of Borland C such as far, huge, pascal…

- In any embedded MCU, programmers have to be ascertained what a float or double is. As it may be 4 bytes long or 8 bytes long.

- Embedded system is commonly used for high precision application that needs extreme reliability, such as medical and automobile applications. This system is usually run on a real time operating system.

- Embedded programmer needs to know the detailed characteristic of the MCU so as to fully utilize all its available functions, such as its low power modes, peripherals, multipliers and etc.

- Embedded programmer has to understand that the actual execution of the program is still in assembly level!

- Embedded programmer has to be aware of byte & word access, small & big endian, 2 & 3 states access, understanding of peripherals behaviors, (e.g. timer is counting, ADC is converting)…

2. HEW Generated Code

Renesas provides High-performance Embedded Workshop (HEW) as its development platform, which is an integrated window-based Editor, Compiler, Assembler, Linker and Debugger. In order to has quick coding startups, HEW generates a basic framework of code based on programmer selection of MCU, operating mode, stack and etc.

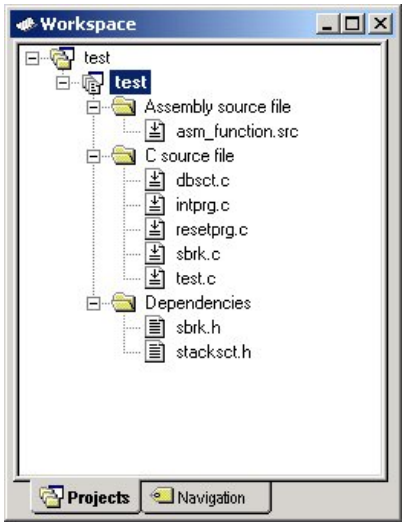


Figure 1 Generated workspace

The generated section, which defined in [Option/…Toolchain/ Link/ Section] provides the boundary, which specified where the programs & data codes should reside.

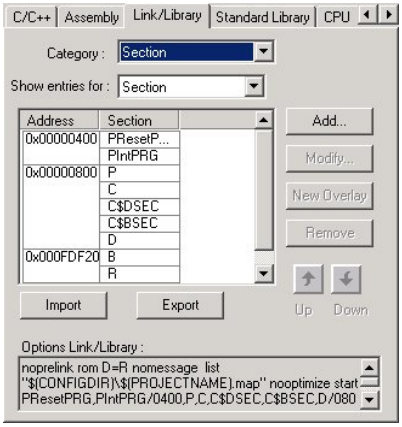


Figure 2 Generated Section

The generated code formed the followings:

- (intprg.c) Entry point to interrupt vector,
- (resetprg.c) Reset entry point vector
- (hwsetup.c) Initialization routine for hardware setup
- (dbsct.c) Initialization routine for sections.
- (sbrk.c) Initialization of heap
- (c_programming.c) Main function
- (iodefine.h) Definition of hardware (peripherals) access structure & address
- (sbrk.h) Definition of heap size
- (stacksct.h) Definition of stack size

Generally the generated code/files, such as iodefine.h, hwsetup.c and intprg.c are different for different selected devices. Embedded programmer or firmware programmer must have a total understanding of the generated code, and thus able to edit the generated template to suit to their applications.

A quick look at the resetprg.c (Figure 3) will shows that the power up sequence of a embedded system

- Initialized the stack pointer (not displayed)
- Disabled all interrupt
- Copy all initialized data from ROM to RAM
- Reset all uninitialized data to zero
- Create and initialized the heap
- Initialized the hardware peripherals
- Enabled the interrupt
- Call main

NOTE: For more information on HEW, please refer HEW user manual, or Application note: Code generation.

3. An example on embedded C program

The following C & Header files are extracted from the project generated based on HEW 2 SLP / TINY tool-chain compiler version 5.0. The generated code is based on the SLP device- H8/38024F.

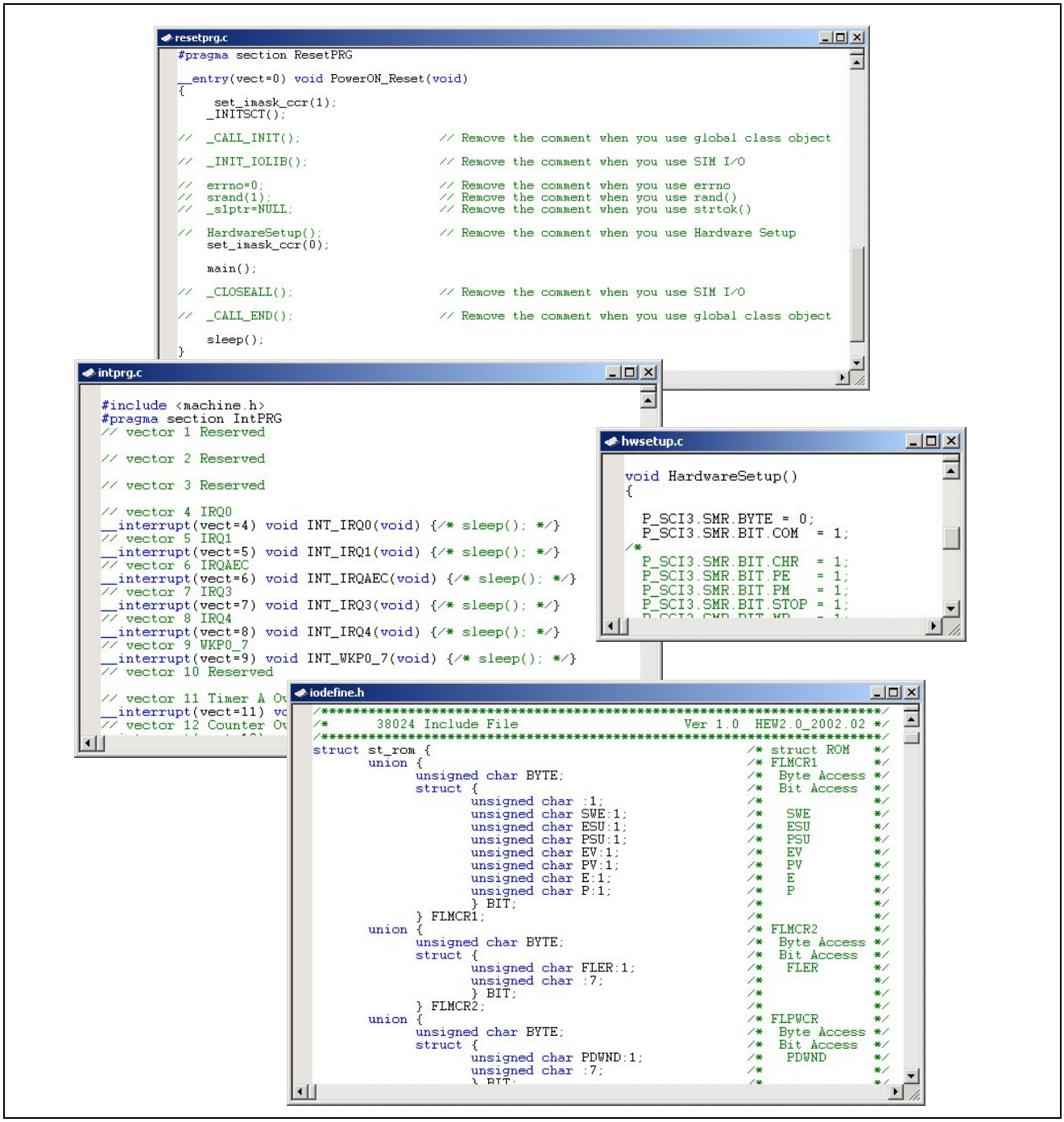


Figure 3 Auto generated files

Based on the generated code, a basic C program is written as follow:

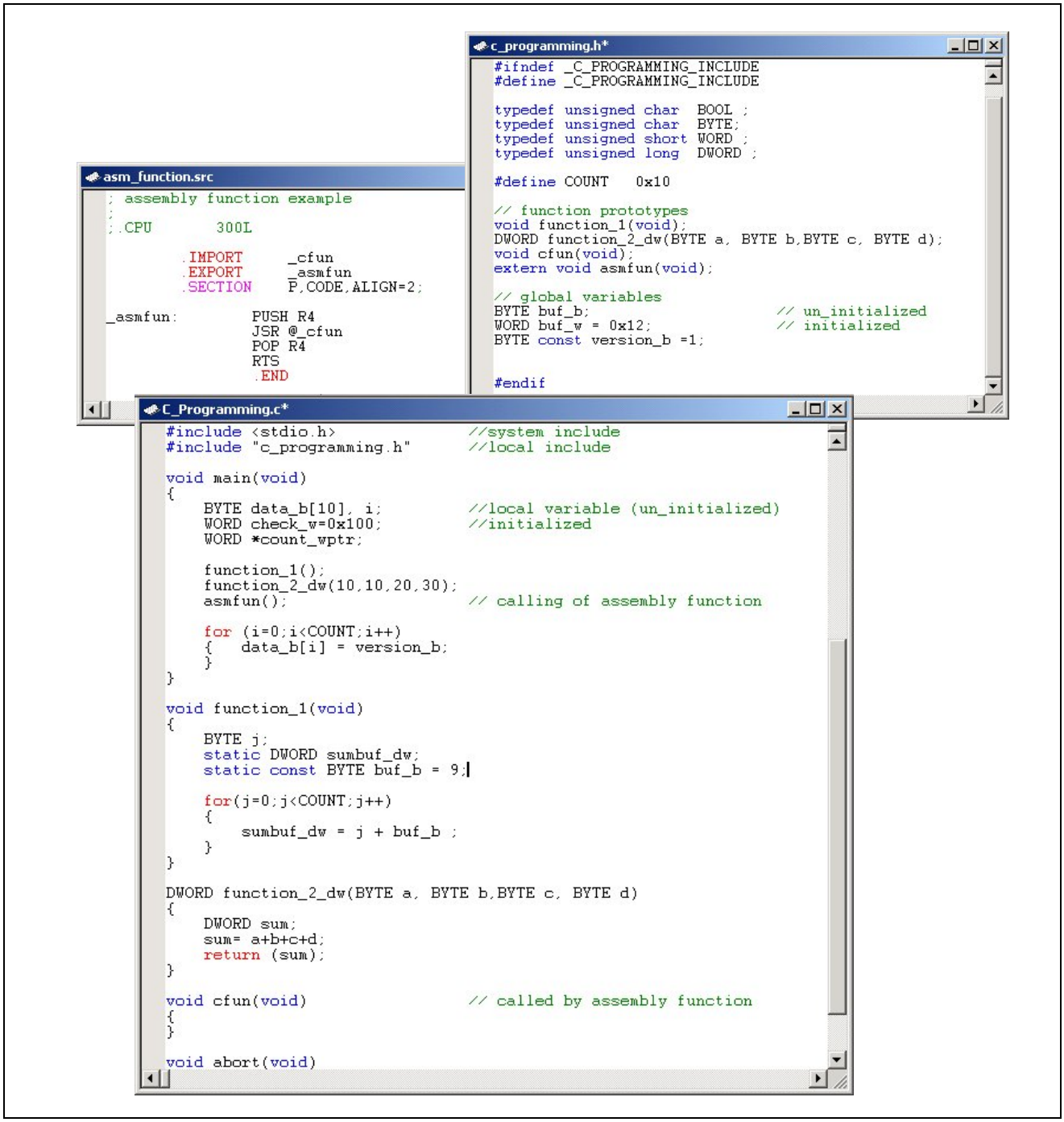


Figure 4 Edited C programs

4. C Variable Initialization

In C variable initialization, programmer will be using keywords such as:

- Automatic
- Static
- External
- Register
- Constant
- Volatile
- Struct
- Union
- Enumeration

All these can be declared either locally or globally.

Generally these declarations have the similar meanings and functions for both embedded & PC based C program.

In C programming, variable's declaration will determine the variable:

- Storage class
- Scope of access

However, questions to be raised in terms of embedded control will be:

- Which types of memory (ROM / RAM) do these variables reside?
- Where (address) are these variables stored?
- How are these variables being accessed?

A simple means to find out the details is to generate a list and map files. Programmer will be able to identify where do the linker stored these variables to. [Option\ …ToolChain\ C, /C++, Assembly, Link/Library \ List]

4.1 Declaration of variable sizes & types

HEW treats the data types as stated in the following table:

Table 1 Variable Size

Data Type	Size (Bytes)	Remark
Char	1	
Short	2	
Int	2	
Long	4	
Enum	1 or 2	
Bool	1	C++ only
Float	4	
Double, long double	8	
Pointer (normal mode)	2	
Pointer (Advanced mode)	4	
Pointer [Function] (normal mode)	6	
Pointer [Function] (Advanced mode)	8	

For some compilers, treatment of the above declaration may be different. Some examples are:

Variable Type	char	short	int	long	Float	double
	1	2	4	4	4	8
No of Bytes	1	2	4	8	4	8
	1	2	2	4	4	8

The above declaration can be modified with the keyword signed or unsigned, to further limit its data range and operation. i.e.

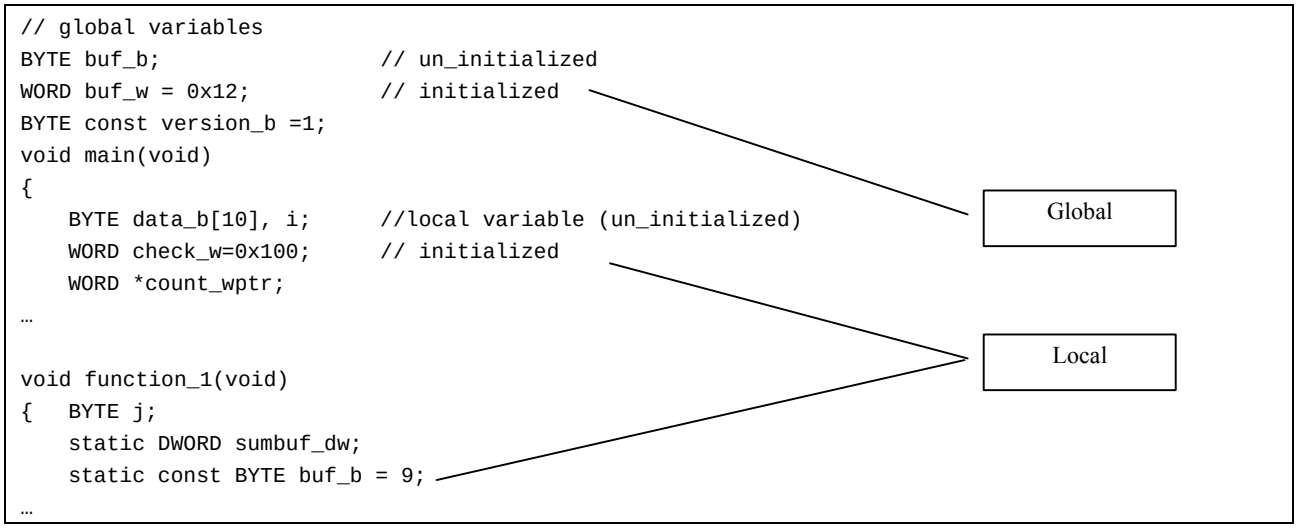
```
integer count            // range from –32768 to 32768
whereas
unsigned integer count   // range from 0 to 65535
```

This will also determine how the C is being converted to assembly code, in order to perform necessary operations, such as addition, and bit manipulation.

4.2 Global & local variable

In C programming, the difference between global & local is mainly the scope of access. A global variable can be accessed anywhere. It is the data permanence and data sharing nature that it is being employed. In contrast, local variable is used locally only, the value will be lost when the local routine is returned. In other words, local variables have a narrow scope & they are temporary in nature.

In another viewpoint, local variable encourage modularity & reentrancy, whereas global variables do not.



Since global variable is permanent, it must occupy a memory space permanently. In HEW, the global variables are put in the section C, D, B and R depending on its nature.

Table 2 Memory space allocations for global declaration

Section	Name	Location	Variable name	Description
C	Constant Area	ROM	version_b	Can be read Globally
D	Initialized data area	ROM	(buf_w=0x12)	ROM(D) space to store the initial value. Thus a routine will be called to copy these value to RAM(R) space at startup
B	Unitialized Data area	RAM	buf_b	Global data that has no initial value.
R	Initialized data area	RAM	buf_w	Global data that has initial value.

When a function is entered, memory must be allocated to the local variable. Two temporary storage spaces are:

i. Register

ii. Stack area.

Generally, ER0 & ER1 are used (ER2 will be used if the option is set). If the numbers of local variables/parameters exceed the available register, the stack area will be used for the temporary processing storage space. Please refer to Figure 7 for the basic illustrations, and HEW compiler manual Chapter 9 .3 for the detail explanations.

Uninitialized data in section B will be initialized to ‘zero’ at startup.

4.3 Automatic & Static Variables

All variables are generated as auto in default. Automatic variables are dynamically allocated when the assigned function is entered. Automatic variables are discarded when the function exits. All local variables are automatic in nature, unless specified.

Automatic variables have the following properties:

- Dynamically allocated & released, which allow memory reuse
- Limited scope of access, which provide data protection
- Can be made reentrant
- Code is relocatable as absolute addressing is not being used
- Number of variables are limited by the stack size

When a variable is declared static, it signifies that its value will not change even when the routine had been exit. Its value is permanent in nature, just like a global variable. However it can only be accessed locally.

```
void function_1(void)
{   BYTE j;
    static DWORD sumbuf_dw;
    static DWORD sumdualbuf_dw =0x1234;
    static const BYTE buf_b = 9;
…
```

In an embedded system, this is possible if the variable occupies a fixed memory space. The following table shows how HEW deals with the static variables;

Table 3 Memory space allocations for local declaration

Section	Name	Location	Variable name	Description
C	Constant Area	ROM	buf_b	Can be read locally
D	Initialized data area	ROM		ROM(D) space to store the initial value. Thus a routine will be called to copy these value to RAM(R) space at startup
B	Unintialiszd Data area	RAM	sumbuf_dw	Local Static data that has no initial value.
R	Initialized data area	RAM	sumdualbuf_dw	Local Static data that has initial value.

Unintialized data in section B will be initialized to ‘zero’ at startup.

4.4 Constant & Volatile

Constant & volatile are type qualifiers as they restrict, or qualify the way an identifier of a given type can be used.

Constant is used to define a variable to be read-only. Thus it is stored in the ROM area. If programmer ignored the keyword, the program will still work normally, except that it will occupy an extra memory (RAM) space.

Volatile notify the compiler that the variable can change its value in some unspecified way by the hardware. This can happen in two ways

- Interrupt (for shared variable)
- Input/output port (for peripherals)

If volatile is not defined, the intelligence of the compiler will reduce (if optimization is enabled) the code, if it assumed that by the nature of the continuous flow of code, the variable value will not change.

Example:

```
condition =TRUE;  // condition is not volatile
While (condition)
{
        action_1();
}
action_2()
```

The compiler may assume that the “condition” variable is always TRUE, thus need not perform the while loop check on the “condition” variable!

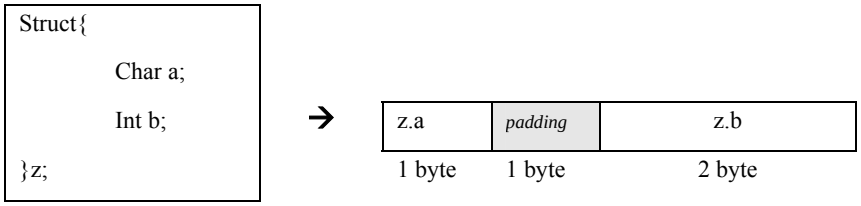
4.5 Register

Register declared variables forced the compiler to use the available registers (such as R4-R6[R3]), as a variable if the rules allow. In this case, speedy operation is possible. However compiler technology has improved tremendously since C was first introduced, and the need for the programmer-assisted optimization is no longer important

4.6 Structure, Union & Class

Due to alignment, the structure & Union Data allocation will tend to have unused area generated in between members. There is no requirement for the component to be continuous. This freedom allows compiler to align members on word boundary, in order to minimize access time.

However if contiguous data is required, in order to improve the syntax use to access memory mapped I/O device. HEW [Option\Toolchain\ CPU] <Pack struct, union, and class> can be set to pack the structure.



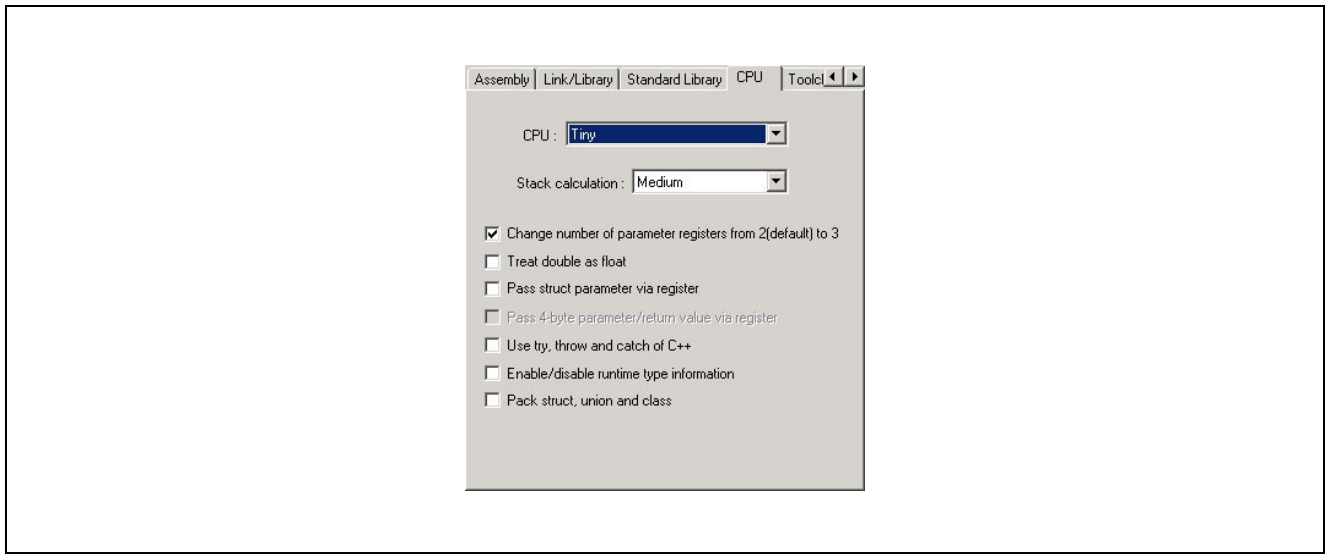


Figure 5 Pack struct, union and class

Refer to Chapter 10 of HEW compiler user manual for more details.

4.7 Bit definition & access

Bit manipulation for IO register is done using structure;

```
struct st_sci3 {                                    /* struct SCI3  */
       union {                                      /* SPCR         */
             unsigned char BYTE;                    /*  Byte Access */
             struct {                               /*  Bit Access  */
                    unsigned char :2;               /*              */
…
       union {                                      /* SMR          */
             unsigned char BYTE;                    /*  Byte Access */
             struct {                               /*  Bit Access  */
                    unsigned char COM:1;            /*   COM        */
                    unsigned char CHR:1;            /*   CHR        */
                    unsigned char PE:1;             /*   PE         */
                    unsigned char PM:1;             /*   PM         */
                    unsigned char STOP:1;           /*   STOP       */
                    unsigned char MP:1;             /*   MP         */
                    unsigned char CKS:2;            /*   CKS        */
                    } BIT;                          /*              */
             } SMR;
};
…
#define P_SCI3 (*(volatile struct st_sci3 *)0x0000FF91)/* SCI3 Address */
```

The C instructions:

```
P_SCI3.SMR.BYTE      = 0;
P_SCI3.SMR.BIT.COM   = 1;
```

Preprocessor replacement:

```
(*(volatile struct st_sci3 *)0x0000FF91).SMR.BYTE = 0;
 (*(volatile struct st_sci3 *)0x0000FF91).SMR.BIT.COM  = 1;
```

Generated assembly code

```
SUB.B    R0L,R0L
MOV.B    R0L,@65448:8
BSET.B   #7,@65448:8
RTS
```

5. Stack & Heap

Stack and Heap are dynamic memory used for temporary variable processing.

Stack is used whenever function is called or returned. It is used to store return address, data and temporary variables.

The project generation will load the stack size in stackact.h. In order to change the stack size and initial value, user is advised to edit the value in the [Edit Project Configuration] window.

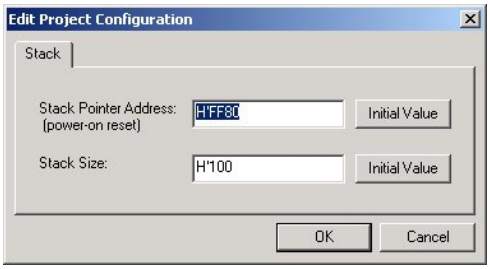


Figure 6 Change of Stack

Heap is used by some standard library functions, such as malloc() and free(). Heap is declared in sbrk.c and sbrk.h. The usefulness of heap is its persistence of static allocation and memory conservation through reuse. However the continuous allocation (malloc) and free of memory may cause fragmentation, which made smaller memory block cannot be reused. It is not advisable to use heap control in the less complicated and small embedded system.

6. Preprocessor Directives

Line begin with # are called preprocessor directives. As the name implied, the 'instruction' will be processed before the compiler begin operation.

6.1 File inclusion

This is used for

- Sharing of declarations &
- Reading clarity.

```
#include <file>
        "file" searched in current directory
        <file> searched in other place
```

These cause the preprocessor to replace the line with the file content before compiling.

6.2 Symbolic representation & Macro

```
#define   identifier    token_string
#define   SIZE          255
#define   SQ(x)         ((x)*(x))          // Macro
```

These cause the preprocessor to replace every occurrence of the identifier with the string before compiling, which improves program clarity and portability. The parentheses are to protect against the expansion of macro expression, from leading to an unanticipated order of evaluation.

Example:

```
For   #define   SQ(x)          x*x
      SQ(a+b)   will expand to   a+b * a+b          // Wrong

For   #define   SQ(x)          ((x)*(x))
      SQ(a+b)   will expand to   ((a+b) * (a+b))    // Correct
```

6.3 Conditional Compilation

The conditional compilation directives such as:

```
#if             constant_integral_expression
#ifdef          identifier
#ifndef         identifier
#endif
#undef          identifier
```

These are very useful for program development. It will helps to produce codes that are portable & easy for debugging.

Examples:

```
#ifndef  SIMULATOR
                // machine dependent code
#endif
```

```
#if DEBUG
        printf(" DATA SEND=%x\n",a);
#endif
```

Programmer has to direct the printf function to an output device, such as LCD or serial port

It is a good practice to have the following in a header file, to avoid double declaration.

System.h

```
#ifndef _SYSTEM_INCLUDE
#define _SYSTEM_INCLUDE
…
#endif
```

NOTE:

Avoid using double underscores. According to ANSI C standards, double underscores are reserved and should only be used for predefined preprocessor variables (e.g. __FILE__, __DATA__)

7. Extended Functions

The HEW compiler can support the following three extended specifications:

7.1 Pragma extension & keyword

The #pragma directives provide extra control for the programmer. (HEW compiler Chapter 10.2)

The following will highlight some common useful examples:

Table 4 Pragma extension and keyword

Specifier	Function
#pragma asm *NOP* #pragma endasm	Embeds assembly language
#pragma section *Ext_Code*	Control of section
#pragam regsave #pragam noregsave	If a caller function has taken care of all registers data integrity, the called function can be make “noregsave”(no save/retore of any register). This may improve code size & speed.
#pragma global_register	This will assign a register for a global variable which is frequently accessed in all the source files.
#pragma pack *1*	In H8S & H8, boundary control in compiler is 2 (even address) except 1 byte variable. But the boundary alignment of structure can be made 1 (odd address) by the #pragma pack 1directive. or [HEW- Option\ …Toolchain\CPU – pack struct, union, and class] **NOTE**: RAM size is decreased, but ROM size will be increased because all structure member is accessed by BYTE instruction
#pragma option *speed*	This will enable a better control in optimization. In HEW option window, control of optimization can be done to all files or individual file. In this case, control is extended to functions within a file. Other available options are: - abs8/16 - inline - inline-asm - interrupt - indirect - entry - stacksize - evenaccess - reparam2/3

7.2 Section address operator

The two operators are:

- __sectop
- __secend

These are used to declare the start address of <section name>, and end+1 address of section name. These have been used in the dbsct.c generated by HEW.

7.3 Intrinsic Function

The compiler provides functions that cannot be written in C/C++, as intrinsic functions. The following functions can be specified as intrinsic functions.

- Setting and referencing the conditional code register
- Setting and referencing the extend register
- Multiply and accumulate (MAC) instructions
- Rotation
- Special instructions (TRAPA, SLEEP, MOVFPE, MOVTPE, EEPMOV, TAS, and NOP)
- Overflow testing
- Decimal operation

Intrinsic functions can be written in the same calling method as regular functions. However, when using intrinsic functions, #include <machine.h> must be declared.

Example:

```
set_ccr(0);
i = rotrw(5,data) ;    // rotate data 5 bits to the right
trapa(0);
```

8. Library

The standard library contains many useful functions that most programmers need. Programmers need not to worry about how the function looks, as they are compiled codes, which is not readable. However the programmer must specify the function prototype, by including the appropriate header file. The compiler will know where to locate the standard library file.

General classification of library:

Table 5 Classification Library functions

Library type	Description	Include Files
Program diagnostics	Outputs program diagnostic information	<assert.h>
Character Handling	Handles and checks characters	<ctype.h>
Mathematics	Performs numerical calculations such as trigonometric functions	<math.h> <mathf.h>
Non-local jump	Supports transfer of control between functions	<setjmp.h>
Variable arguments	Support access to variable arguments for functions with such argument	<stdarg.h>
Input/output	Perform Input Output handling. By using <no_float.h>, I/O functions that do not support floating point number can be provided (smaller size)	<stdio.h> <no_float.h>
General Utilities	Perform C program standard processing such as storage area management	<stdlib.h>
String Handling	Perform string comparison, copying etc	<string.h>
others	Define macro names used by the stardard include files	<stddef.h>
	Defines various limit values relating to the internal representation of floating point numbers	<float.h>
	Define various limit values relating to compiler internal processing	<limit.h>
	Defines the value to set in errno when an error occurs in a library function	<errno.h>

Example

```
#include < math.h>


double d, data;

data = sin(d);
```

However, due to the fact that the library is written for general standard usage, it may not be optimized toward the programmer application. If further code size reduction is required, programmer may need to customize these functions.

An example of the printf() function is being explained in the Application Note - “Writing a printf function to LCD and serial port”

Refer to HEW compiler Chapter 10.3 for more library detail.

9. Function calling

Function must be declared and defined.

- A function declaration (Prototypes) specifies the syntax (name and input and output parameters)
- A function definition specifies the actual program to be executed.

Programmers should strictly follow the prototype as defined, instead of assuming compiler behaviors of promoting 'character' argument to 'integers'.

The stacking process when a C function is called is as follow:

- Prior to a function call, parameters are pushed onto the stack (if the registers are fully utilized)
- Return address are pushed onto stack automatically (hardware) when the function is called
- Previous Stack frame is saved to stack (ER6)
- Registers value are saved (other than ER0-1,2)
- Local variable are saved.

When the function is exiting, the reverse will happen.

- Popping of the data values to back to their source.
- "Return" data is stored in ER0 (if the function is not defined as void)

The following gives a basic illustration of the stack area when a function is being called and returned. Thus all values of local variables will be destroyed upon function exit. The point to note is that the stacking & unstacking process are dependent on the function declared. Example: if parameters of the function are able to be stored in ER0, ER1 & ER2 (depending on compiler setting), the stacking of parameters onto the stack will be unnecessary.

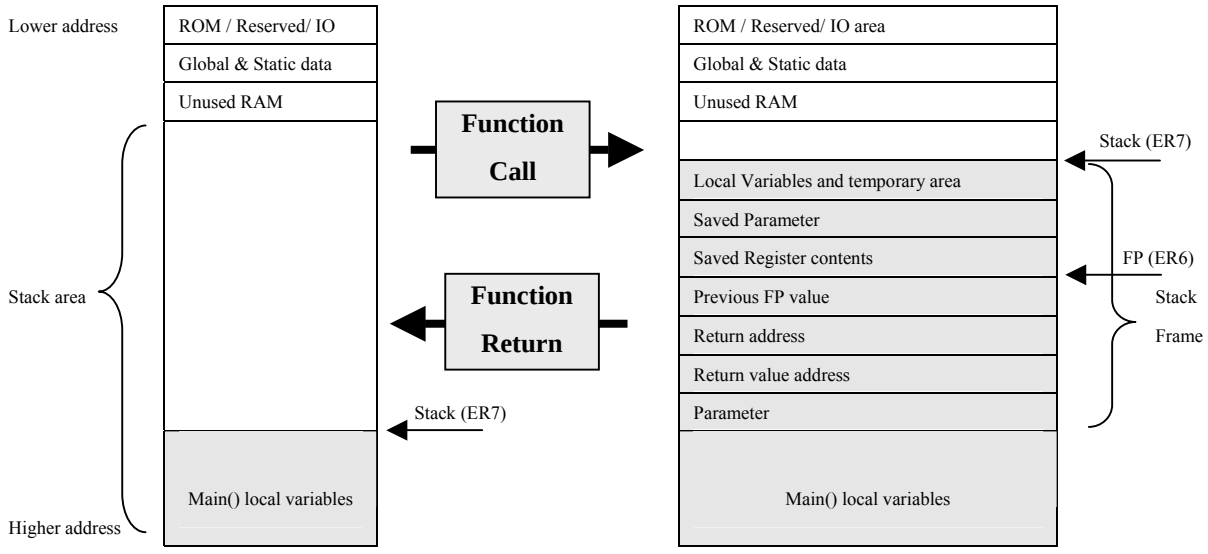


Figure 7 Stack & unstacking

10. Section

For a basic MCU with all ROM, RAM & IO built-in, there is nothing to worry when HEW generated the framework. However for a larger system (16M of address space) whereby there are several different external areas. Programmer will need to map their codes to the correct space. "#pragma section *name*" is used to identify where the code shall reside.

There are two points to note:

- Define the section at the beginning of the code using (#pragma section *name*), and set to the default at the end of the code *(*#pragma section)
- Define the section address in the HEW [option\… toolchain \linker\ section]

NOTE:

When section name is defined as [new_area], the code will be located in [Pnew_area], and constant will be located in [Cnew_area], and so on…

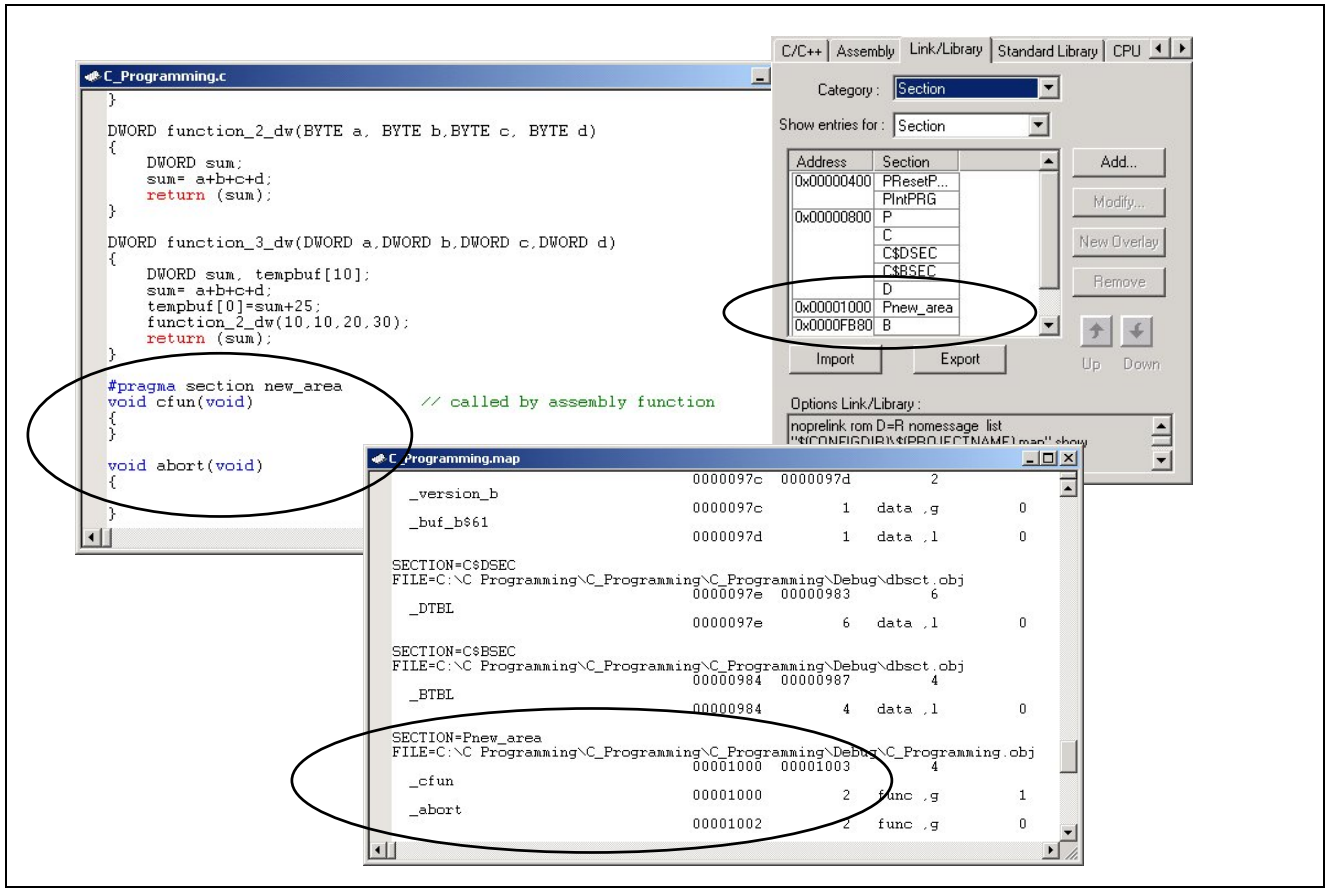


Figure 8 Section control

11. Highlights

11.1 Macro & Function

Macro is just a directive that happens before the compiler takes action. The preprocessor simply replace the symbol with the code (without checking the syntax), whereas function will cause the running program to be disrupted and jumped to the pre-located position for execution. In another view, Macro improves operational speed, whereas function improves code size.

Macro is usually represented as a statement or expression, whereas function will contains more complicated control.

As Macro is treated as a replacement of code, it is recommended to have its body & arguments parenthesized.

11.2 Pointer & Array

When an array is initialized, the compiler allocated the declared memory space for the array. Whereas, when a pointer is initialized, a memory space is put aside to store the address data. Programmer must be wary about the memory space usage, as the compiler can only check the correctness of the code syntax, and not how the pointer is being used.

Examples:

1. When a pointer is declared as integer (2 bytes), an incremental of 1 (or ++) will increase the address by 2.

2. When a pointer is made to point to an array, it can or may exceed the array boundary, thus causing corruption to other data.

3. When a block of memory is required to be used, it must be allocated (declared) before access, otherwise it may be used by other process or variables.

11.3 #Define & constant

The usage of [#define LIMIT 256] & [const int limit = 256] may be almost similar. However when preprocessor has replaced the constant variable before the compiler can do its work, the valuable debugging information is lost. Thus [LIMIT] will not be able to be watched over the debugger window.

11.4 Memory Management

From the earlier section, it can be discovered that:

i. ROM space has been occupied:
   a. Code
   b. Constant
   c. Initial value for variables.

ii. RAM space is occupied by:
   a. Stack
   b. Heap
   c. Global variables
   d. Static variables

Two points for review:

i. The local variables of a main function behave like “static global” variables. This is because after the main function is entered, it will never be returned. Thus its local variable will always occupy the stack memory (just like any static & global variables).

ii. A system designer may treat the global and static variables as similar in characteristics, since both occupy the RAM permanently. However a C programmer may not agreed, as both declaration are different and have a different scope of access.

11.5 C and Assembly

The actual execution of the program, be it written in C, is still in assembly level. Programmer has to anticipate the possible issues when interrupts are concerned. It is understood that the MCU will finish executing the current opcode before it can entertain the interrupt routine.

12. Compiler, Linker & Debug

With HEW integrated environment, programmer can

1. Write their embedded C

- Generate a basic embedded C project files based on click of button
- Edit the basic project files

2. Compile/Assemble the code

- The compiler will make syntax check on code
- Generate the object code, list files, …based on compiler option settings

3. Link the Project

- Put all the object files into a available boundary of space,
- Produce an absolute debug file (ABS- ELF/DWALF2 format)
- Produce stack information, map file…

4. Debug the code with an Emulator or a Simulator

- Check the operationally correctness of the code.

The process will continue until the product is tested fully.

NOTE:
Both assembler and compiler generate object codes. However assembler performs a simpler task, as it make one-to-one translation of readable assemblers code (mnemonics) into the equivalent machine code (opcode). On the other hand, compiler converts line of instructions into a set of opcodes.

13. Suggestion for Programmers

The recommendation is made based on the criteria of having:

- Readable code
- Easy code maintenance

Indentation, Brace & Parentheses style

- Use vertical & horizontal spacing generously. The spacing should reflect the block structure clearly.
- Use Brace & Parentheses generously to make codes easier to read, and process clearly defined

Examples:

```
if (a = = b + 2 && c = = f + 5 || d < r….)
➔
if  (       (a == b+2)                                  // Prefer
      &&  (c == f+5)
       ||   (d  <   r)
….)
```

```
for(I = 0 ; i < count ; i++ , j++ ) printf("data=%d",I);
➔
for( i=0 ; i < count ; i++)                             // Prefer
{           printf("data=%d",I);
            j++;
}
```

With the following arrangement, brace can be easily matched, and code can be read & debug easier.

```
while(check_ready)
{           for (j=0;j<count;j++)
            {           z = j+2;
            …
                        if(z=LIMIT)
                        {           …
                        }
                        else
                        {           …
                        }
            }// end for
}// end while
```

Variable Naming:

- All #define constant should be in CAPS
- Use understandable & descriptive name for global variables such as array_size, max_distance, … instead of i, j, k…
- Use short name for local variable such index, i. j…
- Do not use standard library keyword for variable or function name, such as atoi, printf, …
- Use additional character to make variable more readable, such as addition of "ptr" for pointer, "u" for unsigned variable, "b" for byte, "w" for word, 'i' for integer, 'l' for long….

  Examples: *record_wptr,

  temperature_ul

NOTE: Programmer may like to build their own style to make their code readable. A more formal style, which is adopted by Microsoft, is the Hungarian Notation. In Hungarian Notation, variables are to be prefixed with lowercase tags following with the variable name "words" each beginning with a capital letter

```
char cGain
```

Header files

- Use the same name as the C source file.
- Use directive to prevent re-definition
- Do not use name that will conflict with standard library
- Use relative path instead of absolute.

Comments

- Put essential comments only
- Comments are not necessary good, if they are not informative, misleading, or updated.
- If the source code have to be modified by many people, addition of date & programmer's name are essential.

```
// Aug 03 John – changed to pointer access
```

Type define

- It is a good practice for embedded system control to define the byte and word size. This enables easier understanding of control bit & bytes.

```
typedef unsigned char     BOOL;     // 8-bits
typedef unsigned char     BYTE;     // 8-bits
typedef unsigned short    WORD;     // 16-bits
typedef unsigned long     DWORD;    // 32-bits
```

Modular, Portable and Reuseable

- In order to reuse a code, the functions must be written with objectives in mind. A modular and portable code will have its specification clearly detailed.
- If the function depends on clock speed, update in the #define statement will make all the calculations done.
- Allow the compiler to have more control, instead of manual of performing manual manipulation, absolute address allocation…

Design for Debug

- The programmers must consider the sequence & step for debugging even though a state of the art emulator is used.
- Use #ifdef DEBUG to generate information for debugging.
    - Printf to LCD – monitor the sequence of program flow and data
    - Printf to serial port – monitor the sequence of program flow and data at PC hyper terminal
    - Blink LED – at the startup routine to signify that the program is running
    - Toggle Port Pin - to measure timing on the oscilloscope
    - Write data to memory – to monitor in emulator, data analysis

14. Conclusion

In order to be a good embedded C programmer, the understanding of controller's architectures and compiler behavior is indispensable.

Reference

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	September 03	—	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.