C Compiler V3 User Guide

Revision: V2.00 Date: March 22, 2024

www.holtek.com

Notice

This document may be not be the latest version. As Holtek's tools and documents will continue to be updated, some dialog boxes and tool descriptions in actual use may differ from the contents of this document. For the most up-to date information, visit the Holtek website at:

http://www.holtek.com.tw/en/mcu_tools_users_guide

Table of Contents

Introduction

This document describes the C programming language basic functions and the C compiler V3 syntax structure and its optimization, helping the users develop application programs quickly by using the C compiler V3.

The C compiler V3 is based on the GCC 4.6.2 version or above. Refer to the GCC user's guide except for the machine related chapters.

It is assumed that the reader already has the following basic qualities:

- Knows how to write C programs
- Has already read and understood the target MCU datasheet

Chapter 1 C language

This chapter introduces the C language syntax and structure from the simplest to the more complex, providing a basic knowledge for learning the C compiler V3. Due to the architecture of the Holtek microcontroller, the contents described in this chapter are all based on the C language, compatible with the C compiler V3 syntax.

This chapter covers the following topics:

- Data types, operators and expressions
- Functions
- Arrays and Pointers
- Structures, unions and enumeration
- Preprocessor directives
- Flow control
- Scope

1.1 Data types, operators and expressions

1.1.1 C Data types

Due to the necessity of using ROM or RAM, before using the variables, the data type must be defined. The data types consist of basic data types, Structure data types, Pointer and void. The basic data types contain integer, character and floating types. The structure data types contain structure, arrays, union and enumeration. Using these structure data types will be able to construct the required data structures.

The basic data types (C compiler V3) are shown in the following table 1-1-1:

Data Type	Size(bit)	Scope
bit[1]	1	0, 1
_Bool[2]	8	0, 1
char [3]	8	-128~127
unsigned char	8	0~255
short	16	-32 768~32 767
unsigned short	16	0~65535
int	16	-32 768~32 767
unsigned int	16	0~65535
long	32	-2147483648~2147483647
unsigned long	32	0~4294967295
long long	32	-2147483648~2147483647
unsigned long long	32	0~4294967295
float [4]	24	-3.4E+038~3.4E+038
double [5]	32	-3.4E+038~3.4E+038
long double [5]	32	-3.4E+038~3.4E+038

Table 1-1-1 Basic Data Types (C compiler V3)

Note: [1] Bit type, the least significant bit is valid. For example, if bit a = 4; then a=0.

[2] _Bool type, when the value is not 0, then the result is 1, when the value is 0, then the result is 0. For example: _Bool a = 4; then a=1.

[3] If no signed or unsigned is marked in front of the basic data type, it will be regarded as signed. It is the same for below.

[4] The floating type is 24-bit supported by C compiler V3, and only 4~5 digit precision is supported by V3.20 or above.

sign	exponent (e)	mantissa (m)
23	22~15	14~8 7~0

[5] Double and long double data types have IEEE 754 32-bit format, and 6~7 digit precision supported by V3.20 or above.

sign	exponent (e)	mantissa (m)
31	30~23	22~16 15~8 7~0

1.1.2 Constant and Variable

A constant value remains unchanged during program execution. A variable is a storage unit with specific attributes in memory, which is used to store data, depending upon the required data type for the defined variables.

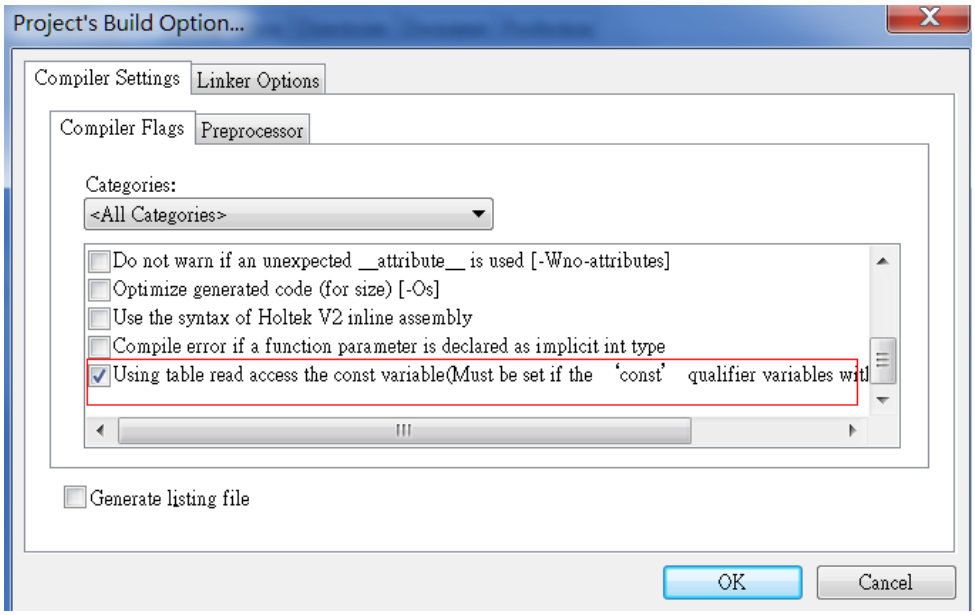
1.1.2.1 Numeric Constant, String Constant, Constant

- Numeric Constant: C Compiler V3 supports the specified Hexadecimal(0x) and Octal(0) prefix. The binary value begins with 0b. For example, the number 237 is represented in binary as 0b11101101.
- String Constant: Included within double quotation marks, ex. “abc”.
- Constant: C Compiler V3 supports up to 64 pages of const constants and also supports const array, the pointer pointed to const and so on.

Example:

```
const unsigned char TABLE[8]={1,2,3,4,5,6,7,8}
unsigned char Test[10];
void TEST_Const(unsigned char *data, unsigned char counter)
{
    unsigned char i, T_counter;
    T_counter = counter;
    for(i=0; i< T_counter; i++)
    {
      Test[i] = *data++;
    }
}
void main()
{
    TEST_Const(TABLE,8);
}
```

Regarding the extended instruction MCUs (such as the HT66F70A), the ROM width is 16 bits and with TBHP register (such as the HT66F60) need to selete the following option, V3 Compiler (V3.20 or above) supports the const constants which are designated address.



The statements are as follows:

```
const type  __attribute__ ((at(addr)))  var[] = {1,2,3,4,5};
```

for example:

```
const char __attribute__ ((at(0x123))) ci[5] = {1,2,3,4,5};
```

then, the results in ROM are as follows:

address	0x123	0x124	0x125	…
content	0x0201	0x0403	0x0005	…

1.1.2.2 Variable Definition

During program execution, the variable values will change. Every variable must have a name defined before it is used. This name is case-sensitive. The variable must be declared in advance. When making variable declarations, the data type must be designated so as to inform the compiler of the required memory size, such as int a;.

Identifiers are used to identify variables, constants, functions, types and a number of characters. Identifiers must consist of letters, digits and underscores and begin with letters and underscores.

1.1.2.3 Variable Storage Class

Each variable has two properties in C language, they are Data Type and Storage Class. The storage class has two kinds of types, Static Storage and Dynamic Storage, including auto, static, register and extern.

1. auto: Local variable, if not declared as static, it defaults to auto, writing “auto char a” or “char a” has the same effectiveness.
2. static: Can be divided into global static storage and local static storage. If static is added to the global variable, then the variable can only be referenced in this document. The local static storage is that the variable will retain its value after the whole program ends. When the function is called next time, the variable will already have a value.
3. register: The above two kinds of variables are stored in the memory, while this variable is stored in a register. As there are special case MCUs, the details will not be described here.

4. extern: This type of variable is defined in another document, which indicates that the variable has been defined externally. Therefor add an “extern” when using the variable. Details are described in the following sections.
5. volatile: A type specifier. Designed to qualify the variables which are accessed or modified by different operators. Variables defined using volatile cannot be omitted because of compiler optimization.

   Variables recommended to be defined with volatile: special register, variables used in the interrupt functions, variables defined for some certain function codes (such as a delay function).

   With regard to other variables, it is not recommended that they are defined using volatile, as this will reduce the compiler optimization.

code list 1.1:

```
File1.c
int cpv;

File2.c
extern cpv;                 // quotes the variable cpv in File1.c
int c;
int statictest()
{
    cpv = 5;
    static int k = 26;      // static variable
    auto int p = 0;         // auto may be omitted
    k++;
    p++;
    cpv++;
    return (k + p + cpv);
}

void main()
{
    c = statictest();       // c = 34
    c = statictest();       // c = 35
}
```

The result is 34, 35.

1.1.3 C Language Operation Introduction

C language has many kinds of operations, including arithmetic operation, logical operation, bitwise operation, assignment operation, conditional operation, comma operation, etc. The operation priority is listed in Appendix B.

1.1.3.1 Type conversion

Type conversion rules:

- Mix Data type Arithmetic Operation
  - if the conversion type is smaller than the integer, the result is integer
  - convert small type to large type (The conversion process is shown in fig. 2_1_1)
- assignment between different data types
  - the conversion type is the type to the left assignment statement
- Function Arguments/Passing Return values
  - the conversion type is the type of the parameter /returned value

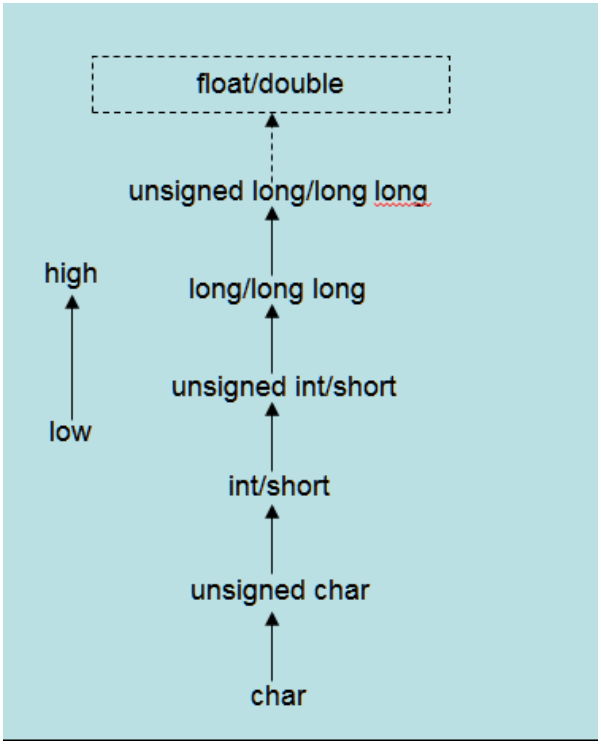


fig. 2_1_1

Type Conversion Example:

int a = 20000;
int b = 20000;
void main(void)
{
long c = a+b;
}
	char a = 100;
	char b = 100;
	void main(void)
	{
	int c = a+b;
	}

Result: C = -25536; Description: a and b are all int type, so should use int type to calculate, the result is 40000, but the result is beyond the range of int, therefore the result is -25536. Solution: long c = (long)a+b; Result = 40000;	Result = 200; Description: a and b are all smaller than int, therefore using int type to calculate, the result is 200.

1.1.3.2 Logical Operation

The results of both the Logical Operation and Relational Operation only have two values: TRUE or FALSE. The numeric value of a relational or logical expression is 1 if the relation is true, and 0 otherwise. Logical operations are AND, OR and NOT, and expressions are connected by && ,|| or !.

Code List 1.2:

```
char a, b, c, d, e, f, g, h;
void main()
{
    a = 0x41;
    b = 0x31;
    e, f;
    c = 0xaa, d = 0x55, g = 0x5a, h = 0xa5;
    if((c = a > b) || (d = a)){      // logic "OR" short circuiting
```

```
        e = 0x18;
    }
    else{
        e = 0x81;
    }

    if((g = a < b) && (h = a)){      // logic "AND" short circuiting
        f = 0x18;
    }
    else{
        f = 0x81;
    }
}
```

Operation Result: a = 0x41, b = 0x31, c = 0x01, d = 0x55, e = 0x18, f = 0x81, g = 0x00, h = 0xa5, both d and h are not given any value.

NOT(!) operation special application: !!(0xXX), when 0xXX is not 0, then the operation result is 1.

1.1.3.3 Bitwise operation

There are two very attractive points in the C lauguage, one is the pointer, and the other is the bitwise operation. There are six operators for manipulating bit-by-bit operations, they are bitwise AND(&), bitwise OR(|),bitwise XOR (^), bit complement (~), left shift (<<)and right shift (>>). The operation time will be reduced significantly by using the bitwise operation skillfully. The following are common applications and operations:

1. Change lower case to upper case, clear bit: 'a' & 0xDF, the result is 'A'
2. Change upper case to lower case, set bit: 'A' | 0x20, the result is 'a'
3. Take one's complement, in other words, Exclusive-OR to the bit and 1 (the first bit complement): 0xFF ^ 0x01, the result is 0xFE
4. Partial multiplication simplification, multiply with the n-th power of 2, which is equated to a variable left shift for n bits, such as 0x02 multiplies 4, 0x02 << 2, here, 2 is shown that 4 = the twice power of 2, the result is 8
5. Partial division simplification, divide with the n-th power of 2, which is equated to moving the variable with a right shift for n bits, such as 0x08 divided by 4, 0x08 >> 2, here, 2 is shown that 4 = twice the power of 2, the result is 2
6. Partial remainder simplification, remainder with the n-th power of 2, which is equated to 2n AND (2n-1), such as the remainder of 15 and 8, which is equated to 15 & 7. Here, 7 is shown that 8-1=7, the result is 8
7. Other Multiplication Simplification, such as 0x08 * 7 = 0x08 * (8 - 1) = (0x08 << 3) - 0x08
8. Rotate, rotate a 16-bit value left for n bits, 0xXX >> (16 - n) | 0xXX << n
9. Rotate, rotate a 16-bit value right for n bits,0xXX << (16 - n) | 0xXX >> n

1.2 Functions

The function is the same as a subprogram, where each function can implement one special function.

During the application program development, frequently used function modules can be written as a single function, so as to reduce the repetitive work. In fact, a function is an address, calling the program to jump to this address to excute the instructions.

1.2.1 Function declaration and definition

It is not necessary to define the function before being declared. The declared function must be an

existent function. If using a library function, then directly add the header file. If it is a user-defined function, then the function must be declared behind the document. Functions cannot be nested.

Function definitions include the return value type, function name, parameter list and the function body, form:

return-type function-name (arg1, arg2, ...)

```
{
    statements…        } function body
}
```

Example 2: Find the larger one of two numbers

code list 1.3:

```
int max(int, int);        // function declaration
int a;
void main()
{
    a = max(10, 20);      // function calling
    a++;
}
int max(int a, int b)     // return value type, function name (parameters list)
{
    return a > b ? a : b;
}
```

Operation result: a = 21

1.2.2 Parameter list and return value

Parameters are written in parentheses after the function name, and will be called into a function.

Formal and actual parameters:

Actual parameters can be constants, variables or expressions, but they must be definite values, which will be passed to the formal parameters of the function when the function is called.

When the function is defined, the formal parameter must be specified as a data type.

The formal and actual parameters must be compatible.

The most important characteristic of formal parameters is one-way passing. This means that any changes to the formal parameters will not affect the actual parameter values in the calling routine.

In the C compiler V3, the naming rules of function Local variables and parameters is _ funname_2[n]. For example, variables of fun function are named as _fun_2, n indicates the variables number.

If the return value size is one byte, then the value is stored in the ACC, if it is two bytes, then its low byte is stored in ra, the high byte is stored in rb, if it is four bytes, then the four bytes are stored in ra, rb, rc, rd from low byte to high byte.

Due to the MCU limitations (no stack), the parameters and internal variables will take up RAM space. The function variables which do not have a call relationship with each other can share the same RAM space.

Example 3: The value a is changeable?

Code list1.4:

```
int a;
void change(int b)
{
    b = 7;
```

```
}
void main()
{
    a = 15;
    change(a);
}
```

Operation result a = 15

A function can return a value to the calling routine, the return type can be void, but it is important to not write any statements after the return statement.

1.2.3 Function Call

1.2.3.1 Function Call method and Process

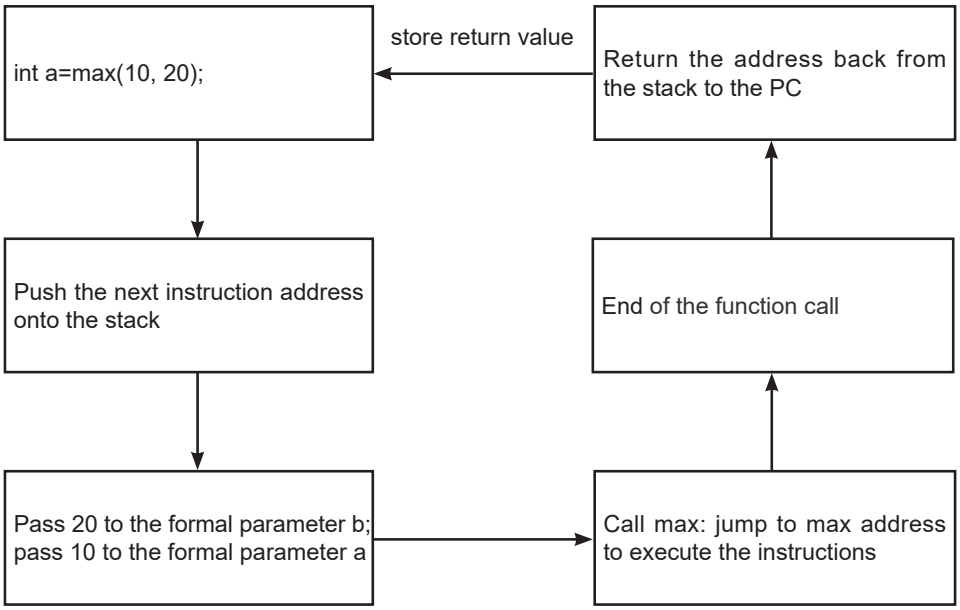
There are three methods to call the function:

1. Function statements: the function call can be used as a statement to achieve a function, such as change(a);
2. Function expressions: for example, int a = 3*max(10, 20);
3. Function parameters: for example, int a = max(max(20, 30), 20);

Function Call Process:

The function call process and the function parameters passing direction of the different compilers are different. In the C language, the formal parameter passing direction is from right to left. The following example describes the process of calling a function in the C compiler V3 (take Example 2 as an example):

Process 1:



Address assignment of function local variables:

1. There is no RAM stack for HT8 devices. All function local variables are allocated when compiling (link stage), which takes up RAM space.
2. Linker allocates local variables according to the relationship between function calls. If there is no calling relationship between the function1 and function2, they can share the local variable space, otherwise, they cannot.

3. Linker allocates independent local variable spaces for the main program and the interrupt, which will not be shared (the interrupts and the main program are also not allowed to call subfunction together).
4. In addition to local variables, the compiler will also use eight intermediate variables, ra~rh. Because there is no general register, so ra~rh are allocated in ordinary RAM. The main program and the interrup intermediate variables, ra~rh, are also allocated separately.

1.2.3.2 Nest Call

The function nest call means that the function can be called in another function. The function call can be nested, but the function definition can not be nested. The recursive call is a special nest call. It means that the function can be called directly or indirectly in this function itself.

1.2.3.3 External Function

There are two methods to use external functions, one is using the header files and include, the other is using the keyword extern.

1. Header file include method:

Example 4: Call the external addition and subtraction function.

Code List 1.5:

```
File1.h
#ifndef _File1_H
#define _File1_H
typedef unsigned char u8;
u8 add(u8 a, u8 b);
u8 sub(u8 a, u8 b);
#endif

File1.c
#include "FILE1.H"
u8 add(u8 a, u8 b)
{
   return a + b;
}
u8 sub(u8 a, u8 b)
{
   return a - b;
}

File2.c
#include "FILE1.H"
u8 c, d;
void main()
{
   u8 a = 10;
   u8 b = 10;
   c = add(a, b);
   d = sub(c, b);
}
```

Operation Result: a = 10, b = 10, c = 20, d = 10

2. extern method:

Example 5: call the external multiplication and division function

Code List 1.6:

```
File1.c
typedef unsigned char u8;
u8 mul(u8 a, u8 b)
{
    return a * b;
}
u8 div(u8 a, u8 b)
{
    return a / b;
}

File2.c
typedef unsigned char u8;
extern u8 mul(u8 a, u8 b);
extern u8 div (u8 a, u8 b);
u8 c, d;
void main()
{
    u8 a = 10,b = 10;
    c = mul(a, b);
    d = div(a, b);
}
```

Operation Result: a = 10, b = 10, c = 100, d = 1

1.2.3.4 Inline Function

Using the keyword inline to qualify functions, then the function is named inline function. Using inline function body statments instead of calling functions to achieve the function purpose will reduce time and stack overheads when calling functions. However the inline function of multiple calls may add much additional code.

Code List 1.7:

```
unsigned char max;
inline void getmax(unsigned char var1, unsigned char var2)
{
    max = var2 > var1 ? var2 : var1;
}

void main()
{
    getmax(23,32);
}
```

Here, using the codes max = var2 > var1 ? var2 : var1; in place of getmax(23,32); function.

1.2.4 Main Function

The Main function is a special function, which is the start of a program execution. After initializing the MCU environment, the Main function is executed. Every project must include a main function, however the main function does not require parameters and return values.

1.2.5 Standard Library

The C Compiler V3 (v3.20 or above)supports math.h, string.h and other standard librarys. For a more complete description refer to the IDE/doc--< standard library user's guide >.

1.3 Pointer and Array

An array is composed of elements of the same data type, using the array name and index to distinguish each element of the array. Every byte has one number called address in memery, while

the pointer itself is a variable that stores the address numbers.

1.3.1 Array Definition, Initialization and Usage

Array definition: data-type array-name [argument number], argument number must be a literal constant or a symbolic constant. Ex. unsigned char led_table[5] [1].

Array Initialization:

1. Define an array. Ex. unsigned char led_table[5];[2] then give values to the elements one by one.
2. Initialization when define an array. Ex. unsigned char led_table[5] = {0, 1, 2, 3, 4};
3. Initialization without argument number. Ex. unsigned char led_table[] = {0, 1, 2, 3, 4}; it shows the argument number is 5.
4. Partial initialization. Ex. unsigned char led_table[5] = {2, 1};[3]

Note: [1] The array index is a positive integer starting from zero up to the total number of elements minus one.

[2] when giving values to the array, the array index should not exceed the elements number, for example, when the elements number is 10, then the maximum index is 9.

[3] led_table[0] = 2, led_table[1] = 1 the values of the others which are not given any values are 0.

An array can be used in the following form: array_name[index]. Ex. led_table[2]. Because the C language doesn't check the array boundaries, it is important to prevent situations of array index overflow.

1.3.2 Multidimensional Array

An array which has a multiple index is not required to define the highest dimension. Ex.

led_table[][4] = {{1, 2, 3},{},{1}};

The second dimension is 3 in the above example. Each dimension must be specified when it is used.

1.3.3 String and End Flag

String constants are saved in ROM, ended with '\0'. For example, String char c[] = "chip", "chip" takes 5 characters.

If the last element of a character array is '\0', then it also can be used as string.

An array name is the array starting address in the memory, such as led_table[5]. Then led_table is the address of led_table[0].

1.3.4 Pointer

Variables defined in programs will be assigned to the memory unit, while the memory unit has a number which is called "address", the pointer content stores the address of the variables.

1.3.5 Pointer Data Type

The pointer points to the address of the different types of variables (RAM address), such as int, char and so on. Therefore, the data type of variables which is pointed to by the pointer must be indicated when defining the pointer. There are some special pointer data types such as the data types of the pointer which points to functions, points to pointers and points to multidimensional arrays. Pointer size is not relevant to the pointer data types, regardless of what the data type is, the pointer size is fixed and only depends on the microcontroller architecture. Ex. char *p; long *q. When using sizeof(p) and sizeof(q), the value is always 2(C compiler V3).

1.3.6 Pointer Operation

There are three steps to operate a pointer: pointer definition, pointer initialization and using a

pointer. It is important to be careful that the pointer is initialized when being used. If writing values to the address where the uninitialized pointer are pointed to, unpredictable errors will be created. There are two pointer operators: &(obtain the memory address) and *(obtain the content).

1. Operation of the pointer pointed to variables:

   Example 6: Is the value changeable?

   Code List 1.8:

```
char a;
void change(char *b)
{
   *b = 'b';
}
void main()
{
   a = 'a';
   change(&a);
}
```

   operation result: a = 'b', because the passed parameter is the address of a, changing the value of the address is equal to changing the argument.

2. Operation of the pointer pointed to pointers:

   Example 7: Is the value changeable?

   Code List 1.9:

```
char *a;
void change(char *b)
{
   b = (char *)0x81;            // modify the pointer itself
}
void main()
{
   a = (char *)0x80;
   change(a);
}
```

   Operation Result:a = 0x80

   Since the parameter passing is unidirectional, if it is desired to change the value a, the code is as follows:

   Code List 1.10:

```
char *a;
void change(char **b)
{
   *b = (char *)0x81;           // modify pointer content
}
void main()
{
   a = (char *)0x80;
   change(&a);
}
```

   Operation Result: a = 0x81

3. Operation of an array pointer and pointer array:

   An array pointer is the pointer which is pointed to in an array, defined as: char (*b)[5];

   The pointer array is the array which stores the data type of pointer. defined as: char* a[5].

   What is the difference? Here, a is the first address of the array, 5 pointer values are stored in the

array a, while b is a pointer, it points to the first address of the array that the data type is char and length is 5. If using sizeof to operate, we can know that the length of a is 10, the length of b is 2.

4. Index overflow:

Code List 1.11:

```
unsigned char a8[5] = {0,1,2,3,4};
unsigned char *p = a8;
*(p + 5) = 8;
```

Operation results: if a8 is stored in the start address 0x0085 of memory, then the value of address 0x008a is 8. If an important value is stored in 0x008a, it will affect the operation results of the entire programs.

5. Operation constant pointer and pointer constant:

constant pointer means that the contents of the pointer pointed to is constant, defined as the follows: const char *c_p1;

The pointer constant means that the value of the pointer itself cannot be changed, but the pointed contents can be changed, defined as follows: char *const p2_c;

code list 1.12:

```
void main()
{
   char a[5] = {0, 1, 2, 3, 4}, b[5];
   const char *p = "Hello World!";
   char * const q = a;
   *(p + 1) = 'o';          // try to modify the contents of the string constants,
error.
   p = "Hello! World!";     // the pointer pointed to the constant can be changed
   p = b;                   // can also point to the variable address

   *(q + 1) = 2;            // modify the contents of the pointer constant pointed to
   q = b;                   // try to modify the value of constant pointer, error.
}
```

6. Pointer operation:

The Pointer can execute add, subtract, increment, decrement and other operations.

code list 1.13:

```
unsigned int a[5] = {1,2,3,4,5};    // assume a = 0x84
unsigned int *p = a;                // p = 0x84
void main()
{
   p = p + 1;                       // p = 0x86, unsigned int occupies 2 bytes.
   *(++p) = 10;                     // p = 0x88
}
```

operation result: p = 0x88, a[2] = 10

1.3.7 Differences between array name and pointer

The array name is the first address of the array in memory, it is similar to the pointer, but there are also a few differences between them:

1. The array name is a constant, so is not allowed to be modified and is equal to a pointer constant.
2. The operation result is different when using sizeof to operate, the result of the array name shows the occupied space, while the result of the pointer is fixed 2.
3. The array name cannot be incremented and decremented.

1.4 Structure, Union and Enumeration

A structure is a collection of data, unlike an array, the elements of the array must be same types, while the structure can be different data types. A union is similar to the structure. The enumeration limits the value of variables in a scope of a specified range.

1.4.1 Using Structure, Union and Enumeration

Example 8: The definition of register PA and bit PA5 (using Structure and Union)

code list 1.14:

```
#define DEFINE_SFR(sfr_type, sfr, addr) \
static volatile sfr_type sfr __attribute__ ((at(addr)))
typedef unsigned char __sfr_byte;
typedef struct {                        // define structure
    unsigned char __pa0 : 1;
    unsigned char __pa1 : 1;
    unsigned char __pa2 : 1;
    unsigned char __pa3 : 1;
    unsigned char __pa4 : 1;
    unsigned char __pa5 : 1;
    unsigned char __pa6 : 1;
    unsigned char __pa7 : 1;
} __pa_bits;

typedef union {                         // define union
    __pa_bits bits;                     // using of structure
    __sfr_byte byte;
} __pa_type;
DEFINE_SFR(__pa_type, __pa, 0x1a);      // define register PA
#define _pa         __pa.byte           // using of union members
#define _pa5        __pa.bits.__pa5     // using of structure members
```

The first enumeration constant has the last specified value if no explicit value is specified. If all the enumeration constants are not specified, then the first enumeration constant has the value 0. Subsequent enumeration constants without explicit associations receive an integer value one greater than the value associated with the previous enumeration constant.

code list 1.15:

```
enum weekday{sun = 6, mon = 0, tue, wed, thu, fri, sat};
const unsigned char *dayLine;
const unsigned char *printDay[7] = {
    "HI, Monday ! ",
    "I am Tuesday ! ",
    "Today is Wednesday ! ",
    "Today is Thursday ! ",
    "Happy Friday ! ",
    "Today is Saturday ! ",
    "Today is Sunday ! ",
};

const unsigned char* getDay(enum weekday today)
{
    return printDay[today];
}
```

```
void main()
{
    enum weekday today = sat;
    dayLine = getDay(today);     //dayLine = "Today is Saturday !"
    unsigned char ch;
    while(*dayLine){
        ch = *dayLine;
        dayLine++;
    }
}
```

1.4.2 Differences between Structure and Union

The main difference between structure and union is the configuration of the memory space. A structure assigns memory space for every member while a union provides a way to manipulate different kinds of data in a single storage area. The union size is the largest data type size.

For example, in the example 8, if using sizeof (__pa_type)to operate the structure size, then the result is 1, if _pa5 is set to 1,then the 5th member of __pa_type is set to 1 as well.

1.5 Differences between Preprocessor, Macro definition and Inline function

There are three preprocessors in the C language: macro define (#define), include file (#include), conditional compiler (#if, etc.). If preprocessing was executed before compiling, such as in macro expansion, then it is performed when precompiling.

1. Macro definition: note that when using macro definition, the macro name is simply replaced by the macro defined content and does not execute any operations. For example, #define S(r) 3.1415926*r*r, if using S(6+6), then the result is 3.1415926*6+6*6+6, but it is not the desired result. Then by using #define S(r) 3.1415926*(r)*(r) the desired result can be obtained.
2. Include file: using #include to include the files, using “file.h” is different from using “<file.h>” to include files, “file.h” is used for user defined files, while “<>” is used for library files, in order to reduce the time spent in searching path when compiling. Pay attention to avoid duplication of header file including.
3. Conditional compiler (Compilation): choose one of the source codes to compile as there is no need to compile all the codes. This is used for debugging and code transplants between the different machines. There are 3 types: (the else directive is alternative):

   (1) #ifdef symbol

   source code1
   #else
   source code2
   #endif

   (2) #ifndef symbol

   source code1
   #else
   source code2
   #endif

   (3) #if symbol

   source code1
   #else
   source code2
   #endif

4. Differences between macro definition and inline function:

(1) Macro definition only does simple replacement without any operations.

(2) Macro is defined when precompiling, while the inline function is used when function calls occur.

(3) Macro definition parameters can not be specified with any data types, while the inline function must specify the data types.

(4) The compiler does not check the macro definition contents, if using #define error **8, there is no error to be found, while the inline function is different.

Example 9: preprocessor comprehensive example

code list 1.16:

```
FUNCTION.H
#ifndef _FUNCTION_H      //if removing #ifndef, then the redefinition error occurs
#define _FUNCTION_H

typedef unsigned char    u8;
typedef unsigned int     u16;
typedef unsigned long    u32;

#define BOOL   u8
#define TRUE   1
#define FALSE  0

#define LeftShift(val, times) (val) << (times)
#define Max(num1, num2)   (num1) > (num2) ? (num1) : (num2)

u8 getMax(u8 num1, u8 num2)
{
   return Max(num1, num2);
}

#endif


KEY.H
#ifndef _KEY_H
#define _KEY_H

#include "FUNCTION.H"
u8 GetKey(u8 num1, u8 num2)
{
   u8 key = getMax(num1, num2);
   return LeftShift(key, 2);
}

#endif
MAIN.C
#include "FUNCTION.H"
#include "KEY.H"
#define DEBUG 1              //using when debugging

u8 _debugval;
```

```
void main()
{
    u8 ch;
    const u8 *Led_String = "YOU";
    while(*Led_String){
        #if DEBUG        //if changing 1 to 0, then this statement can not be compiled.
        _debugval = *Led_String;
        #endif
        ch = *Led_String;
        ch = LeftShift(ch, 1);
        GetKey(ch, 0);
        Led_String++;
    }
}
```

If passing ch++ to num1, passing 0 to num2 of macro definition Max(num1, num2), then(ch++) > (0) ? (ch++) : (0), but the value of ch is not the expected value. Therefore, the macro definition should be noted.

1.6 Flow Control

1.6.1 Three Execution Flow

The execution flow of the C language is sequential execution, optional execution and loop execution.

1.6.2 Using of, if and switch

If and switch are conditional expressions. When many conditions exist, the if statement can be replaced with the switch statement. The switch variable must be enumerated and a basic data type except for float. The case statement variable must be a constant value (not including the constant with const), the break statement will be used after the case statement was executed. If no break statement exists, then the next statement will continue to be executed. Multiple case expressions can execute the same statements. The default statement shows the default cases and can execute without a break.

Code List 1.17:

```
unsigned char f;
……
switch(f){
case 12:
case 13:
    f += 1;
    break;
case 14:
    f += 2;
    break;
default:
    f += 3;
}
```

1.6.3 Loop and nested-looping

1. while(expression) statement

2. do…while(expression);

Differences: do…while(expression); the statement executes at least once.

code list 1.18:

int sum = 0;
void main()
{
  int i = 11;
  while(i < 11){
  sum += i;
  i++;
  }
}
	int sum = 0;
	void main()
	{
	  int i = 11;
	  do{
	  sum += i;
	  i++;
	  } while(i < 11);
	}

result:sum = 0	sum = 11

3. nested-looping

In loop statements, the loop can be executed more than once.

1.6.4 break and continue

Break is used for loop or switch case statement, while continue is only used for the loop statement, the differences between them are that the continue statement orders the program to skip to the end of the loop and begins the next iteration of the loop, while the break statement is to jump out from a loop and execute other statements outside this loop.

code list 1.19:

```
while(1)
{
    int j = 0;
    while(1)
    {
        j++;
        if(j == 5)
        {
        continue;            // execute the next iteration of the loop
        }
        if(j == 10)
        {
        break;               // jump out from the loop
        }
    }
}
```

1.6.5 goto statement

The goto statement can directly jump to a statement label to execute programs. It is generally not recommended to use goto statements.

Example 10: using goto correctly to optimizes code

code list 1.20:

```
typedef unsigned char u8
#define BOOL   u8
#define TRUE   1
#define FALSE  0
u8 result;
```

```
BOOL fun1()
{
   ……
}
```

```
BOOL fun2()
{
   ……
}
```

```
BOOL fun3()
{
   ……
}
```

Original version:

```
void main()
{
    BOOL b_result = FALSE;
    u8 b[3], a[3];
    u8 *p = a;
    b_result = fun1();
    if(!b_result){
        result = 0x55;
        p = b;
    }
    b_result = fun2();
    if(!b_result){
       result = 0x55;
       p = b;
    }
    b_result = fun3();
    if(!b_result){
       result = 0x55;
       p = b;
    }
}
```

goto version:

```
void main()
{
    BOOL b_result = FALSE;
    u8 b[3], a[3];
    u8 *p = a;
    b_result = fun1();
    if(!b_result) goto error;
    b_result = fun2();
    if(!b_result) goto error;
    b_result = fun3();
    if(!b_result) goto error;
error:
    result = 0x55;
    p = b;
}
```

example 11:using do…while(0)instead of goto

do…while(0)version:

```
void main()
{
    char b_result = 0;
    u8 b[3], a[3];
    u8 *p = a;
    do{
       b_result = fun1();
       if(!b_result) break;
       b_result = fun2();
       if(!b_result) break;
       b_result = fun3();
       if(!b_result) break;
    }while(0);
    result = 0x55;
    p = b;
}
```

1.7 Scope

Both the variable and the function has its scope. Global variable/function can be used in the whole project after being configured as extern. If a global variable is defined as static, then this variable can only be used in this file. In order to save ROM space, static is not recommend. The local variable can be used in the statements of the function after being declared. If the variable is declared in the if and switch of the loop, then the variable can not be used in the following statements after executing the if and switch statements. As seen with the j variable of code list 1.19, the external statements of the while loop can not be used.

Chapter 2 C compiler V3 Expansion and Limitation

2.1 Holtek C compiler V3 settings in the HT-IDE3000

2.1.1 Operation environment

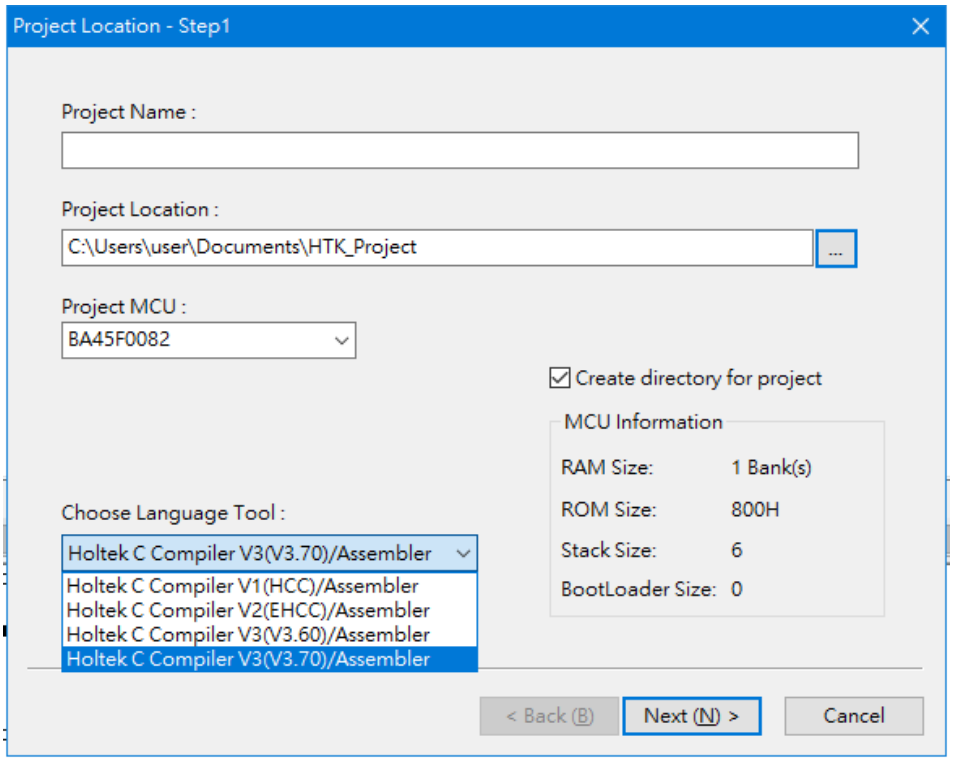
This chapter is written based on the HT-IDE3000 8.1.6 version.

Note: The Holtek C V3 must operate in the HT-IDE3000 7.71 or later version

2.1.2 Building a new project use Holtek C V3 compiler

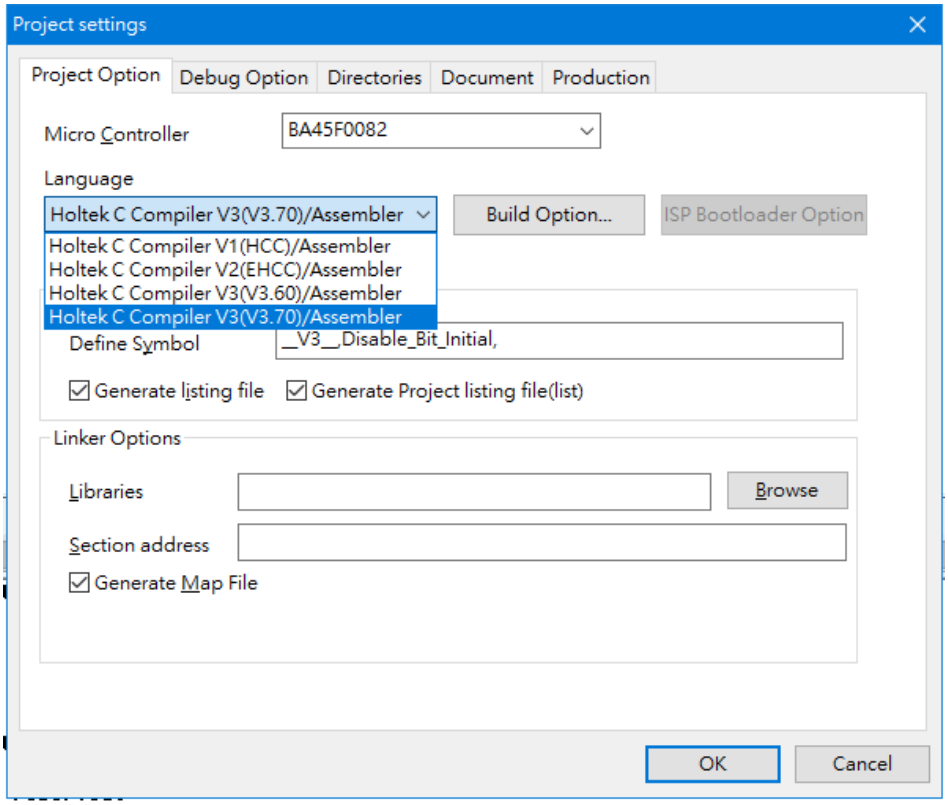
In the HT-IDE3000 development environment, build a new project using the following method:

- Move the mouse cursor to the Project menu, and click the left button
- Move the mouse cursor to the New command, and click the left button
- In the following window, select Holtek C Compiler V3(V3.xx)/Assembler in the Choose Language Tool



2.1.3 After opening the project, how to use the Holtek C V3 compiler

After opening the project, click Project Setting command in Option menu, and click Holtek C Compiler V3(V3.xx)/Assembler in the Language Tool to set Holtek C V3 compiler.



2.1.4 Project Compiler Option settings

1. Define Symbol

The setting is shown as the following red box. The HT-IDE3000 will pass the parameter to the compiler: -DGCC_COMPILER -DCOUNT=5

Be equivalent to macro definition:

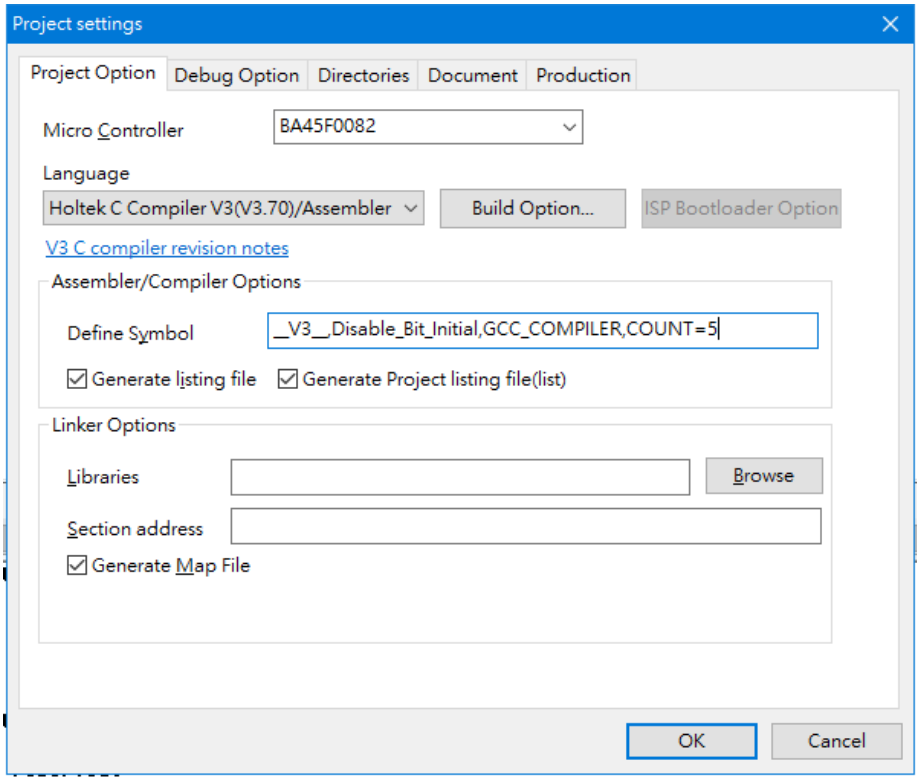
```
#define GCC_COMPILER
#define COUNT 5
```

Its effective scope is the whole project, the multiple macro definitions are separated by ‘,’.

The example of using -D command line option is conditional compilation. Ex:

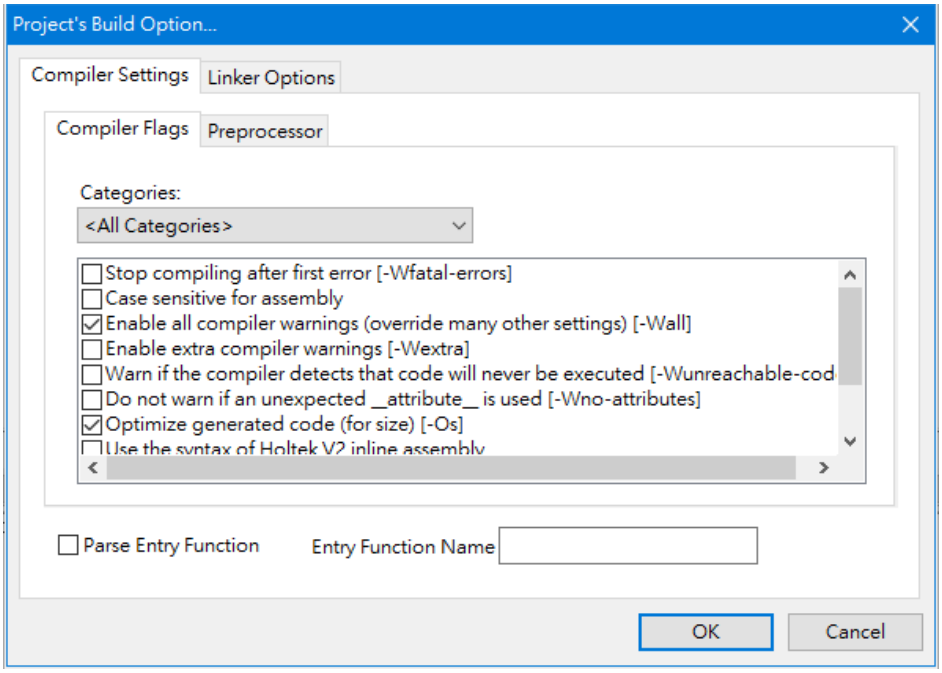
```
#if GCC_COMPILER
typedef unsigned int uint16;
#else
typedef unsigned int uint8;
#endif

#if COUNT==5
x = 5
#elif COUNT == 2
x = 6;
#else
x = 7;
#endif
```

Note: When using the C Compiler V3,HT-IDE3000 (7.72 or above version), the macro definition is "__V3__,Disable_Bit_Initial" by default.



2. Compile parameter

When using Holtek C V3, choose the compiler option, the optimizing parameter -Os is selected by default in the HT-IDE3000, click "Project's Build Option" in the project settings, and a dialog box as follows will appear:



Case sensitive for assembly: For mixed language projects. If a project includes either a c file or an asm file, then this option can be enabled. (Words in C are Case-sensitive, words in assembly language are Case-insensitive by default).

String alignment(eg. DW 'string1'). Two chars will be aligned in one program ROM word: For an asm file, when the selected MCU is 16 bits(such as HT66F50), this option can be enabled, then every two characters in the string will take one word. If disabled, then one character takes one word by default. If the selected MCU is 14 or 15 bits (such as the HT46R4AE), then this option can not be enabled.

Enable extra compiler warnings: such as lack of function return value, unsigned value comparing with zero

Optimize generated code (for size) [-Os]: Optimization option.

Use the syntax of Holtek V2 inline assembly: You can use the #asm/#endasm completed inline assembler syntax function when select this option

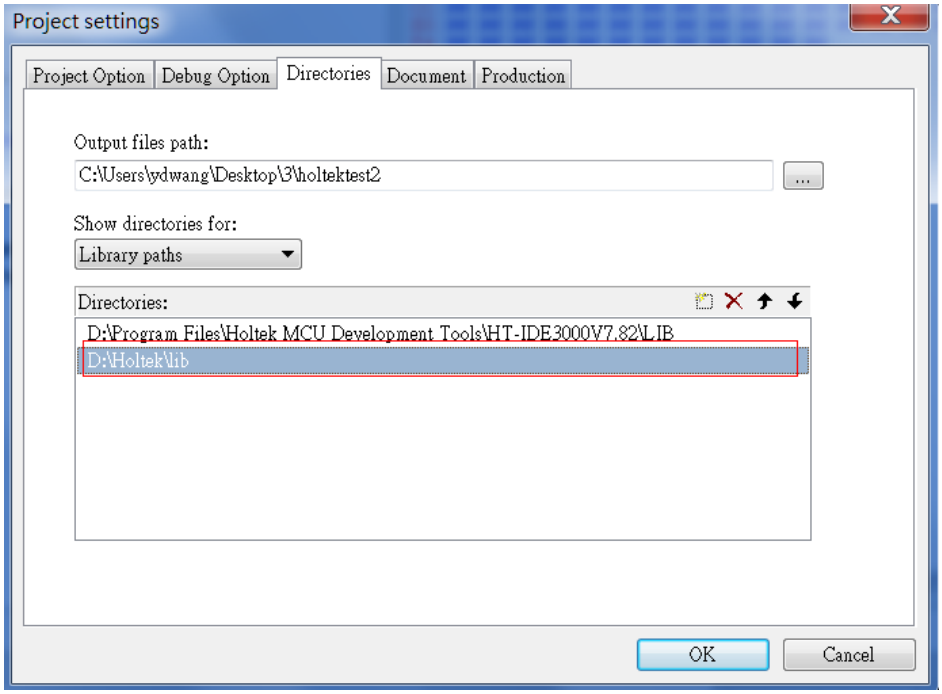
Using table read access the const variable: MCU (without extended instructions) with TBHP register and the PROM width is 16 bits has this option.The size of table will be halved when this option is selected.The option must be set if the 'const' qualifier variables with specified address exist.

Set the size of enumeration data as byte: The default type of enum compiler is int. When the defined enum value is not greater than 127, selecting this option can save space.

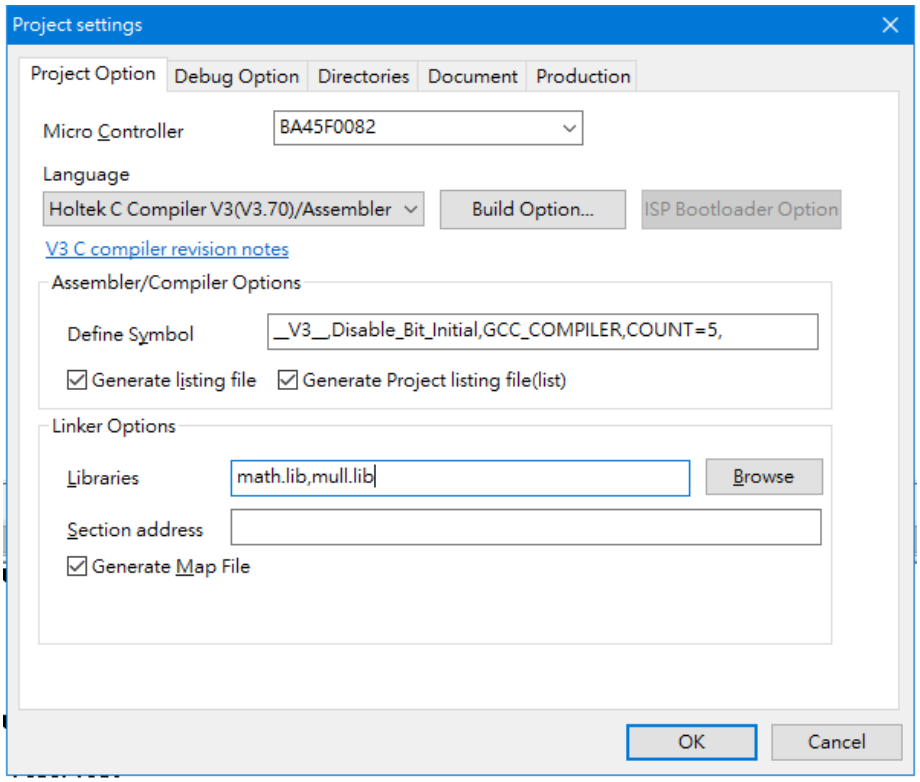
2.1.5 Linker options

2.1.5.1 Include The Library

1. Add the library paths to the directories:



2. Input the library name, as follows, math.lib, mull.lib, separated by ‘,’.



2.1.5.2 Set the function address

format: _fun_name=addr, _fun2_name=addr2, …

here, _fun_name is '_'+function name, addr is the address, it is hexadecimal, if there are multiple functions, then the address is separated by ','.

Note: This feature is not recommended because it will affect compiler optimization.

2.1.5.3 Generate Map File

when checked, a .map file will be generated.

2.1.5.4 Linker options

Optimize data memory: If not nesting interrupts occur in the project, select the option can save data memory.

Remove unused function: If a function is not called, then the system will not assign space for it, this option is enabled by default.

Uninitialized global/static varibles are automatically set to 0 (Exclude bit type): If a global has no initial value, the default initial value is 0, this value will be cleared when the program starts running. This operation does not contain bit variables.

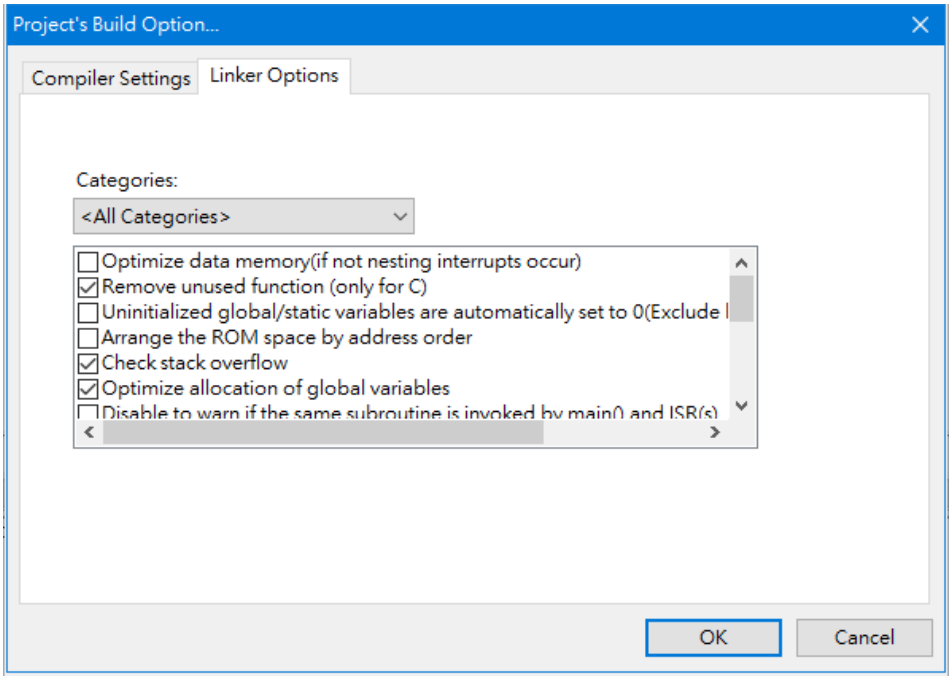
Arrange the ROM space by address order: For the MCUs with multiple ROM banks, in order to save BP switching instructions, the compiler may not necessarily assign programs in ROM bank order. Selecting the option can arrange the ROM space by address order.

Check stack overflow: When the main program stack call layer number exceeds the number of MCU stack, a warning will be generated.

Optimize allocation of global variables: For the MCUs with extended instructions, the global variables with many calls are preferentially allocated to RAM bank 0.

Disable to warn if the same subroutine is invoked by main() and ISR(s): The compiler does not allow the main program to call the same subfunction as the interrupt, but users can select this option if they are sure that the usage is no problem.

Warn if the absolute address variable is overlap: Check whether the absolute address variable is overlapping or not. If the address is overlapping, then a warning will be generated.



2.2 C compiler V3 Extended syntax and function

2.2.1 Interrupt Service Routines

If the MCU peripheral devices contain interrupt functions, the programs also need the interrupt functions to finish the task, then the interrupt service routine(ISR) should be defined in the following way:

```
void __attribute((interrupt(0x0c))) ISR_tmr0 (void)
{
tick++ ;
}
```

The interrupt service routine must obey the following rules:

→ The type of the return value must be void

→ No parameters included

→ Must use__attribute((interrupt(0x0c))) to setup the interrupt vector

→ The registers, ACC, BP, STATUS, MP, TBLP, TBHP, will be saved automatically when the system is outside the interrupt entrance.

→ The variables that can be accessed by both the interrupt service routine and other functions are defined as volatile.

Note: MP/TBLP will be saved when they are used in the interrupt function.

The following rules should be noted when using the interrupt service routine:

The Holtek C V3 supports an interrupt internal call function. The different interrupts and the main function can not call the same function, otherwise it will cause a RAM overlap. The linker will detect this phenomenon and issue a warning. If the called function is not declared and uses any local variables, this warning can be ignored. For example:

```
void fun1(){}
void fun2(){fun1();}
void main()
{
  fun1();
}
void __attribute((interrupt(0x04))) isr1(void)
{
  fun1();
}
void __attribute((interrupt(0x08))) isr2(void)
{
  fun2();
}
```

- The main and isr1 both call fun1. This is a common call.
- Although isr2 does not call fun1 directly, but it calls fun2, and fun2 calls fun1, so it is also a common call. The call diagram will be found in the map file.
- In the same way, the different ISRs can not call the same function, unless it can be guaranteed not to enter interrupt 2 during the execution of interrupt 1.

2.2.2 Absolute variable

A variable is designated to the fixed address, for example: _bp in the address[04H] can be described as follows:

```
static volatile unsigned char _bp __attribute__ ((at(0x04)));
```

The compiler assigns this variable to this address, but if the variable is not used throughout the program, the linker may assign other variables to this address. The compiler will translate it into an EQU instruction with assembly-language as follows: __bp EQU 4h

In the head file for ht48c70-1.h, define it:

```
#define DEFINE_SFR(sfr_type, sfr, addr)
static volatile sfr_type sfr __attribute__ ((at(addr)))
```

Description: Modify with a volatile keyword to prevent optimization.

So if the program appears include MCU.h, it also be written as follows:

```
DEFINE_SFR(unsigned char _bp, 0x04);
```

Array, pointer, structure variables can also be defined as absolute variables, which defined ways are the same with general variable except one additional address, for example:

```
static volatile unsigned char arr[10] __attribute__ ((at(0x140)));   // array
static unsigned int *volatile p __attribute__ ((at(0x040)));         // pointer
typedef struct
{
    int a;
    int b;
}my_type;
static volatile my_type ab  __attribute__ ((at(0x040)));             // struct
```

2.2.3 MCU header file introduction

The HT-IDE3000 has MCU header files, these header files contain the definitions of some registers and flags.

In the project, it is only necessary to write #include “MCU.h”, such as #include “ht66f60.h”, which will include the header file automatically.

The main contents of header files are:

1. Absolute addresses and interrupt syntax abbreviations:

```
#define DEFINE_ISR(isr_name, vector)\
void  __attribute((interrupt(vector))) isr_name(void)
```

here, isr_name is the interrupt service routine name, vector is the interrupt address, for example:

```
DEFINE_ISR(isr_timer,0x0c)
{}
#define DEFINE_SFR(sfr_type, sfr, addr)\
static volatile sfr_type sfr __attribute__ ((at(addr)))
```

Here, static shows that the special register is static. Each C file may include a MCU header file, therefore the special register must be defined as static, sfr_type is the data type and sfr is the special register name, addr is the address.

for example: DEFINE_SFR(unsigned, _bp, 0x03)

2. Special register definition:

```
#define _acc        __acc
```

here, __acc is the variable that has been defined by DEFINE_SFR. Users can use it in their programs directly.

3. Flag definition:

```
#define _c          __status.bits.__c
```

here, __status is the defined variable, if users want to define the flag, the follwing ways can be used as a reference:

a. Frst, define a struct (Collection of all the status register flags) and union (using status to operate on complete bytes or a single bit):

```
typedef struct {
   unsigned char __c : 1;
   unsigned char __ac : 1;
   unsigned char __z : 1;
   unsigned char __ov : 1;
   unsigned char __pdf : 1;
   unsigned char __to : 1;
   unsigned char __cz : 1;
   unsigned char __sc : 1;
} __status_bits;

typedef union {
   __status_bits bits;
   unsigned char byte;
} __status_type;
```

b. Specify status to the address 0x0a. If it is a general bit variable, then there is no need to specify the address

```
DEFINE_SFR(__status_type, __status, 0x0a);
```

c. Define _pa0 macro as a bit field for easy reference

```
#define _c          __status.bits.__c
```

4. Inline function definition:

Built-in Function	Description
GCC_RL(varname)[1]	varname rotates left without carry
GCC_RLC(varname) [1]	varname rotates left with carry
GCC_RR(varname) [1]	varname rotates right without carry
GCC_RRC(varname) [1]	varname rotates right with carry
GCC_NOP()/_nop()	execute a dummy instruction (NOP)
GCC_SWAP(varname) [1]	exchange the first four bits and the last four bits of varname

GCC_HALT()/_halt()	execute a HALT instruction
GCC_CLRWDT()/_clrwdt()	clear the watchdog timer (CLRWDT)
GCC_CLRWDT2()/_clrwdt2()	clear the watchdog timer (CLRWDT2)
GCC_CLRWDT1()/_clrwdt1()	clear the watchdog timer (CLRWDT1)
GCC_DELAY(n) [2]	Delay n instructions

Note: [1] varname must be a 8-bit variable

[2] 0 <= n < 263690

5. If MCU contains EEPROM, then define __EEPROM_DATA (a, b, c, d, e, f, g, h);

Using this statement to write a value to EEPROM:

```
#define ___mkstr(x)                 #x
#define __EEPROM_DATA(a, b, c, d, e, f, g, h) \
asm("eeprom_data .section 'eeprom'"); \
asm("db\t" ___mkstr(a));          \
asm("db\t" ___mkstr(b));          \
asm("db\t" ___mkstr(c));          \
asm("db\t" ___mkstr(d));          \
asm("db\t" ___mkstr(e));          \
asm("db\t" ___mkstr(f));          \
asm("db\t" ___mkstr(g));          \
asm("db\t" ___mkstr(h));
```

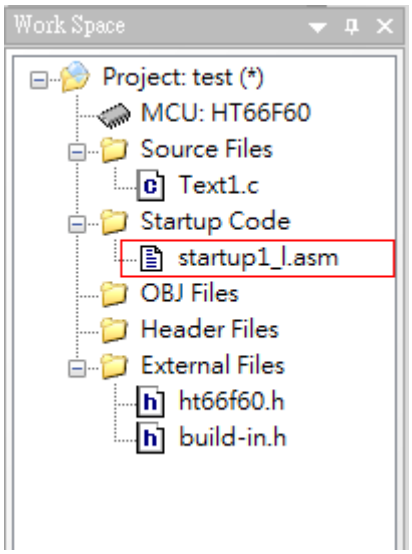
8 values can be written in order at once, such as:

__EEPROM_DATA(1, 2, 3, 4, 5, 6, 7, 8);

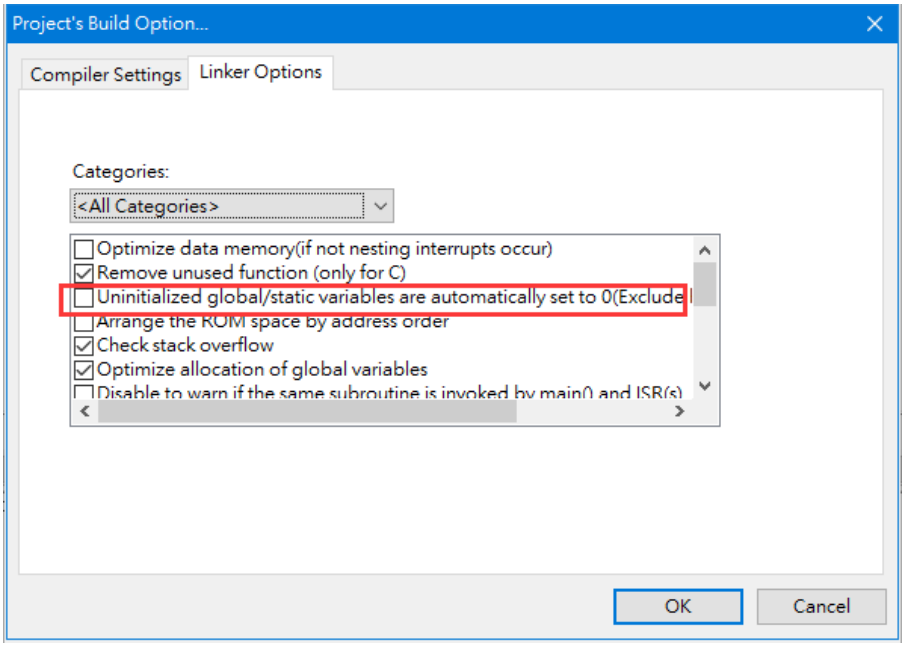
Note: __EEPROM_DATA can not be written into the function body.

2.2.4 Variable Initialization

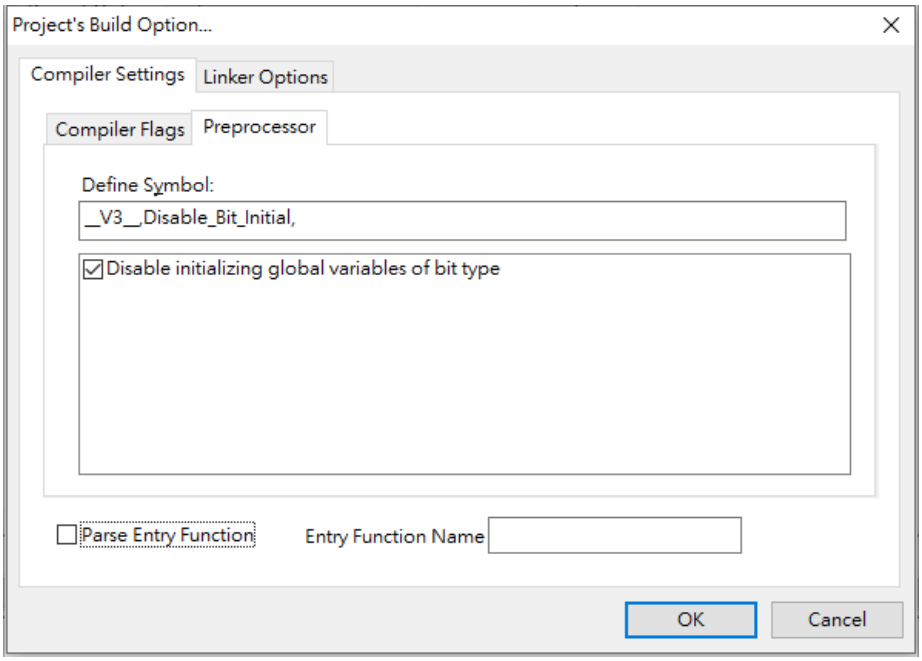
The C Compiler V3 supports variable initialization by calling a startup program at the beginning. The startup program is composed of startup0 and startup1:



1. When establishing a new project, startup1 will be added to the project automatically. If users need to change the variable value to 0 after resetting the followwing option need to be selected. This function is implemented in the 7.72 or above version of the HT-IDE3000.



2. Startup program code is opened (this function is implemented in the version of HT-IDE3000 7.72 or above)
3. If the the IDE version is updated, the startup file under the project folder must be deleted,and the IDE3000 will reload it.
4. The compiler does not initialize the global / static bit variables by default. If users want to modify this setting, it can be modified here in the version of HT-IDE3000 8.05 or above:
   a. Select this option: The compiler does not initialize the bit variables and ignore the initial value.
   b. Diselect this option: The compiler will initialize the bit variables. If there is no initial value, the default value is 0.



2.2.5 Inline Assembly Code

If simplified program code compilation and higher execution efficiency is required, then assembly-language instructions can be added into a C language program. The syntax is:

1. asm (“opcode [operands]”);

   Compiler outputs the instruction opcode [operands] directly

   e.g.

```
extern void FUN() ;
asm(“extern _FUN_PAR1:byte”);      // extern function parameter declaration
void main( )
{
  asm(“mov a,1”);
  asm(“mov _FUN_PAR1,a”);
  asm(“call _FUN”);                // function call
}
```

2. asm (“opcode %0” : “=m”(varname) : “m”(varname));

   While varname is a variable name. To avoid being optimized, add volatile.

   e.g.

```
#include “ht46ru25.h”
char i;
void main()
{
  char c;
  volatile asm(“rl %0” : “=m”(_pd): “m”(_pd));    // sfr
  volatile asm(“mov a, %0” : “=m”(c): “m”(c));    // Local variable can read a value
  volatile asm(“mov %0,a” : “=m”(i) : “m”(i));    // Global variable can write a value
  while(1);
}
```

3. #asm/#endasm

   When choose the option “Use the syntax of Holtek V2 inline assembly”, the following syntax can be use to edit a assembly segment.

   e.g.

```
void main( )
{
  #asm
  MOV A,1
  MOV _FUN_PAR1,A
  CALL _FUN
  #endasm
}
```

Note:

- Every statement only writes one instruction, it needs to use quotation marks.
- The second embedded assembly format can only occur within the function. Its variable size should not exceed 1byte.
- The first embedded assembly format can occur outside the function and can be used to define variables such as section etc. It will output the string between the quotes. The compiler will not deal with this.

2.2.6 Specify Function Address

C Compiler V3.20 or above version offers support to specify the function address, syntax:

```
unsigned char  __attribute__((at(addr))) fun (char parm){}
```

key word __attribute__((at(addr))) specifies the function address to addr, such as:

```
unsigned char  __attribute__((at(0x373))) foo (char parm){}
```

This shows a defined address at 0x373. The function specified address supports parameter and return values.

2.2.7 Specify Const Address

For the MCU whose ROM is 16-bits and contains the TBHP register, the C Compiler V3.20 or above supports specification of the address, syntax:

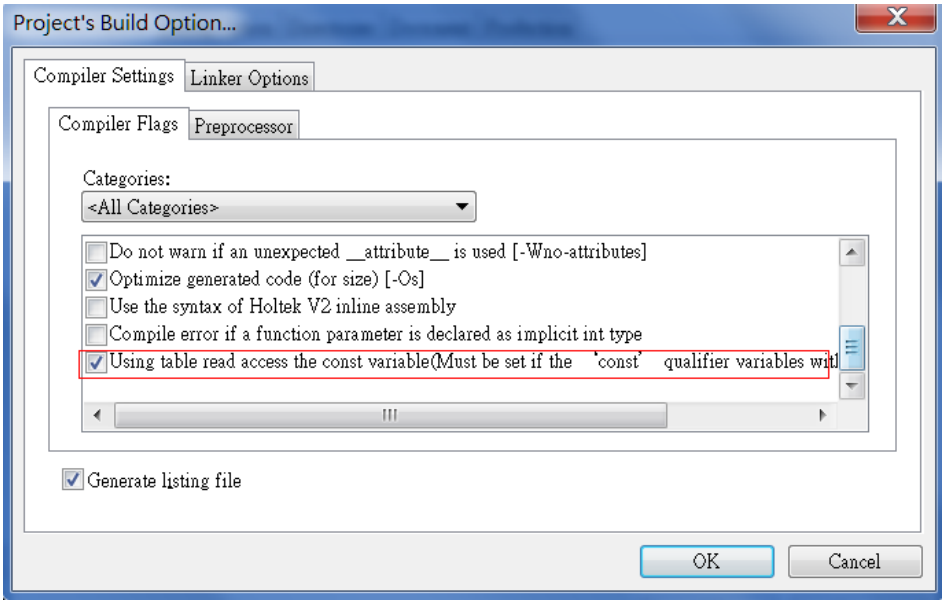
```
const char __attribute__ ((at(addr)))  cvar1 [3]={1,2,3};
```

key word __attribute__((at(addr))) specifies the function address to addr, such as:

```
const int __attribute__ ((at(0x123)))  a[3]={1,2,3};
int b;
int c = 9;
int fun(int *pa,int a)
{
    a+=*pa;
    retrun a;
}
void main()
{
    b=fun(a,c);
}
```

__attribute__ ((at(0x123))) means that the defined address of 0x123.

The option shown below must be set to use this function:



2.2.8 Variable Assignment

Holtek MCUs have two memory areas, ROM and RAM. The variable assignment rules are as follows:

1. The general variables will take up RAM space, the initial values are stored in the ROM space.
2. Const variables take up ROM space, but if using a volatile qualifier(no specify address), then this will take up RAM space.

3. If the static variable value is not changed in the program, and passes the optimized parameter, then the compiler will regard this as a const and take up ROM space.

2.2.9 __attribute__ Syntax

This section describes the syntax with which __attribute__ may be used, and the constructs to which __attribute__ specifiers bind, for the C language.

1. __attribute__((entry))(support above IDE V7.82)

   Specify a entry function.

   Syntax:

   ```
   __attribute__((entry))
   void entry_function_name (void){}
   ```

   Noted:

   a. entry has no parameters.

   b. return-type should be void.

   c. parameters type should be void.

   d. The main/isr function can be set to entry, but is not valid.

   Example:

   ```
   __attribute__((entry))
   void func (void)
   {}
   ```

2. __attribute__((at(addr)))

   Specify the address of function/variable

   Syntax:

   For function:

   ```
   __attribute__((at(addr)))
   return_type function_name (parameters, …){}
   ```

   for variable:

   ```
   __attribute__((at(addr)))type variable_name;
   ```

   Noted:

   a. ‘addr’ is the address of function/variable, can not missing

   b.The variable with __attribute__((at(addr))must be described as static.

   c. For the MCU whose ROM is 16-bits and contains the TBHP register, the C Compiler V3.20 or above supports specification of the address of constant.

   d. The main/isr function can be specified with address.

   Example:

   ```
   __attribute__((at(0x400)))
   void func (void)
   {}
   __attribute__((at(0x180)))
   int a;
   __attribute__((at(0x100)))
   const int array[4]={1,2,3,4};
   ```

3. __attribute__((interrupt(addr)))

   Define an Interrupt Service Runtine

   Syntax:

```
__attribute__((interrupt(addr)))
void isr_name (void){}
```

Noted:

a. 'addr' is the address of function/variable, can not missing and must be 4 times number.

Example:

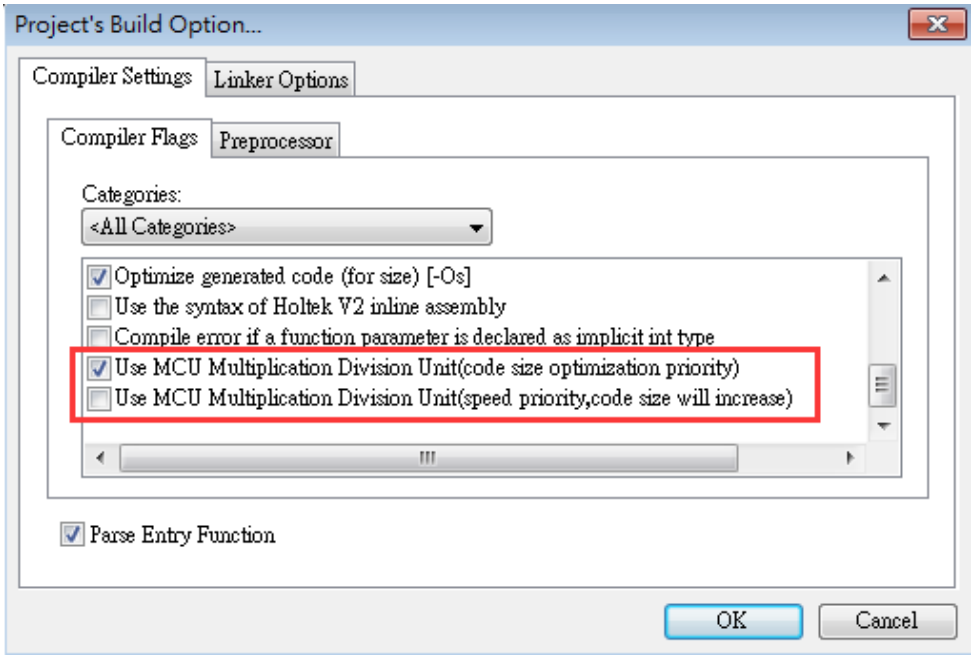
```
__attribute__((interrupt(0x04)))
void isr_timer (void){}
```

A function can be specified with a variety of attribute:

```
a. __attribute__((entry,at(addr)))
b. __attribute__((interrupt(addr),at(addr)))
```

2.2.10 Hardware Multiplier and Divider

1. The hardware Multiplier and Divider function is applied to MCUs which have an integrated MDU. The MDU-related registers, such as MDUWR0, MDUWR1, MDU0R0, etc., if they exist in a specific MCU datasheet indicate that an MDU function is contained in this MCU.
2. The IDE3000 V7.90 supports the MDU function.
3. For MCUs that contain an MDU function, there will be two parameters under the path of "Option Project → settings → Build OptionsàCompiler Settings" on IDE3000.



A. "Use MCU Multiplication Division Unit (code size optimization priority)": Enable the MDU function. This option is checked by default. Note that for programs transformed from the IDE3000 V7.89 or former versions, this option state should be confirmed.

B. "Use MCU Multiplication Division Unit (speed priority, code size optimization priority)": As most MDUs are 16 bits, if the MDU function is used for char/unsigned char type multiplication and division operations, the code size may not be reduced, however the operation speed will be faster. This option is suitable for users who priority is speed.

4. Most MCU hardware multiplier and dividers only supports three operations, namely 16bit*16bit, 32bit/16bit and 16bit/16bit, except for the HT66FM5440 which supports 8bit*8bit and 8bit/8bit

operations. Therefore the MDU function is not required for all the multiplication and division operations:

Multiplication	long	U long	int	U int	Char	U char
long	—	—	—	—	—	—
int	—	—	MDU	MDU	MDU	MDU
char	—	—	MDU	MDU	MDU	MDU
U long	—	—	—	—	—	—
U int	—	—	MDU	MDU	MDU	MDU
U char	—	—	MDU	MDU	MDU	MDU

Division	long	U long	int	U int	char	U char
long	—	—	—	—	—	—
int	—	—	—	MDU	—	—
char	—	—	—	MDU	—	—
U long	—	—	—	MDU	—	—
U int	—	—	MDU	MDU	MDU	MDU
U char	—	—	—	MDU	—	MDU

Note: a. For the char type operation, the MDU function can be used only when the "speed priority" option is selected.

b. For division operations, the MDU function can be used only when what is at the left of the equation is smaller than the long type.

c. For division operations, the left side is the dividend.

d. U means unsigned.

5. Benefit comparison between the ways of using the hardware MDU and the Multiplication/ Division Operation Library

	Multiplication/Division Operation Library		MDU	
	Size (word)	Execution Time (Instruction Cycle In the Worst Case)	Size (word)	Execution Time
8bit × 8bit	10	106	15	25
8bit / 8bit	25	133	15	25
16bit × 16bit	19	289	19	29
16bit / 16bit	37	330	19	29
32bit × 32bit	31	895	—	—
32bit / 32bit	67	1160	—	—
32bit/16bit	67	1160	19	29

2.2.11 Bit data type

1. C Compiler V3.5 and the later versions (HT-IDE3000 V7.93 and the later versions) start to support bit data type.
2. Bit variable occupies 1 bit, the least significant bit is valid.
3. Bit variable does not support stuct/union/const/register/array/pointer/function parameter and return value
4. The bit variable address can be specified using the following syntax:

```
static volatile __attribute__((at(addr),bitoffset(val))) bit flag1;
```

Here "addr" means the address unit and "val" means the position in this unit.

For example:

```
static volatile __attribute__((at(0x80),bitoffset(3))) bit flag1;
```

Means the flag1 will occupy the address of [80h].3.

5. Examples :

```
volatile bit flag1;
volatile bit flag2;
static volatile __attribute__((at(0x18),bitoffset(0))) bit pa0;
void main()
{
  while(1)
  {
      if(flag1&flag2)
      pa0 = 1;
  }
}
```

2.3 C compiler V3 limitation

2.3.1 Function Pointer

The C Compiler V3 does not support the function pointer, so the following instruction will display an error:

C:\Users\test\Text1.c:14:10: error: Holtek-gcc does not yet support function pointer.

```
void FileFunc(){}
void EditFunc(){}
void foo()
{
     typedef void (*funcp)(void);
     funcp pfun= FileFunc;
     pfun();
     pfun = EditFunc;
     pfun();
}
```

2.3.2 Recursive Function

Holtek MCUs do not support recursive calls, but the compiler can optimize a special recursive call, called Tail Recursive Call, and change it to a non – recursive function. Details are described in chapter 3.10.

2.3.3 7-bit MP – Memory Pointer

The MPs in some of the older Holtek MCU structures only have 7-bits. In order to achieve higher optimization efficiency, the C Compiler V3 does not support this kind of MCU. Refer to the file < Holtek C Compiler V3 FAQ > for the MCU types.

2.4 Compiler managed resources

Some special function registers in the Holtek MCUs are used by the C Compiler V3 so users should be careful when using them. These registers and the main functions to the complier are listed in the following table. When entering into interrupt service routines, the used registers will be saved automatically.

special registers used by complier	Main functions
MP/IAR	Access the value of RAM space, used with RAM BP
BP	RAM BP is used to access the value of RAM space, ROM BP is used to access ROM space
STATUS	For expression calculation
TBLP	For table read or storing RAM BP

special registers used by complier	Main functions
TBHP	For table read
TBLH	For table read
ACC	For storing function return values, etc.
PCL	For table read

Chapter 3 C compiler V3 optimization function

3.1 Optimization Contents Introduction

The C Compiler V3 is an optimized complier whose main objective in optimisation is to simplify the code and reduce the quantity of code required. By default, enable –Os will execute all the optimization functions. The following table shows each function as executed by the compiler, including whether each optimized function can affect debugging.

	whether affecting the debugging	save ROM	save RAM	save Stack
Algebraic Transformations	√	●		
copy propagation/value propagation	√	●		
Unreachable code Elimination	√	●		
Dead-code Elimination	√	●		
Constant Folding		●		
constant propagation		●		
Inline Procedure	√			●
Strength Reduction	√	●		
Tail Recursive Call				●
subexpression elimination		●		
Tail merging	√	●		
BP optimization		●		
Dead section elimination	√	●		

3.2 Algebraic Transformations

Algebraic transformations mean that the compiler will replace some expressions with simpler operations which have the same functions. For example:

```
-(-a) → a.
if(!a && !b) → if(!(a || b))
```

This kind of replacement will not affect debugging.

3.3 Copy Propagation/Value Propagation

Copy propagation is a kind of transformation, for the variables x and y, when y was assigned to x. If the values of x and y are not changed in the following code, then the value y can replace the value x. This kind of optimization can not reduce the number of instructions, but it can eliminate any dead-code. Refer to the section 3.5 “Dead-code Elimination” for more information. For example:

```
00 char c;
01 void foo (char a)
02 {
03   char b;
04   b = a;
05   c = b;
06 }
```

Description: Line 05 can be changed to c=a; then the code in line 04 has no purpose, therefore it can be deleted. Variable b also can be deleted.

Copy propagation will affect the debugging, because of the elimination of line 04 and variable b. The user can not set a breakpoint in line 04, and there will be no variable b in the watch window.

3.4 Unreachable Code Elimination

The Unreachable Code Elimination function is used to delete the code that will not be executed in

normal programs. For example:

```
if (1)
{
  x = 5;
}
else
{
  x = 6;
}
```

In the above codes, the else part will never be executed. After optimizing the code using this function, the new assembly code generated will not contain the else part. The Unreachable Code Elimination function will affect the breakpoint setting in some lines of the C source code.

3.5 Dead-code Elimination

In the function calculation, if the values are not used in the whole function, then these values are called dead values. The instructions which only calculate the dead values are called dead instructions. The values outside the function are used values (not dead values) because these values can not be sure of being used. Refer to the example in the copy propagation/value propagation section 3.3.3.

Dead-code Elimination will affect any breakpoint settings and variable monitoring in some lines of the C source codes.

3.6 Constant Folding

Constant Folding means that the compiler will calculate the expression internal values directly, but will not generate the assembly codes:

Example:

```
double a, b;
a = b + 2.0 / 3.0;
```

compiler will output the following assembly statements →

```
a = b + 0.666667;
```

Constant Folding will not affect debugging.

3.7 Constant Propagation

The variables of some expressions are not constants, but the compiler can operate on their values through simple computation. For example:

```
float a=5.0f, b;
b = a + 1.0f;
```

Here the compiler will output the following assembly statements →

```
float a=5.0f, b;
b = 6.0f;
```

Constant Propagation will not affect debugging.

3.8 Inline Procedure

When a subfunction is a simple function (as in the following conditions), the subfunction code will be directly written to the function which call the subfunction. Thus it can reduce the use of the stack:

The following rules should be obeyed when using an inline function:

1. The function can not contain complex statements. The complex statements include:switch statements, for statements, while and do while statements, goto statements.
2. The function can not be a recursive function.

3. Function statements in the function body cannot be more than 30 lines.

4. The subfunction and the generating function must be in the same C file.

Example:

```
float square (float a)
{
   return a * a;
}
float parabola (float x)
{
   return square(x) + 1.0f;
}
inline function:
float parabola (float x)
{
   return x * x + 1.0f;
}
```

Disadvantages of inline functions:

The program size will increase along with the expansion of the function.

To avoid using the inline function, subfunction and generating function can use the different files to program.

Advantages of the inline function:

1. After the function was expanded, it may have additional optimization.

2. Reduced stack usage.

3. After the subfunction was expanded, if not called by other functions, then the code will be as the dead section and deleted by the Linker (refer to 2.3.14)

The inline function will create a situation where the expanded subfunction can not set breakpoints.

3.9 Strength Reduction

Strength reduction means that some simple loops can be changed to some general statements. For example.

```
for(i=0; i<10; i++)
{
   j=i;
   k=j+i;
}
```

can be changed to:

```
i=10;j=9;k=18;
```

Strength reduction will create a situation where simplified code will not be able to set breakpoints.

Note: The compiler only cares about the operation results, therefore the special purpose codes (ex. implement delay function by loops) will have no meaning. In such cases, it is recommended to use inline assembly instead of loops or define the variables as volatile. For example:

Example 1:

```
for(i=0; i<10; i++)
{
   asm(”nop”);
}
```

Example 2:

```
volatile unsigned char count;
```

```
for(i=0; i<10; i++)
{
   count++;
}
```

3.10 Tail Recursive Call

Holtek MCU does not support recursive calls, but there is a special recursive call that can be implemented by the compiler. Here the compiler will change a recursive call to a non-recursive call. This is called the Tail Recursive Call.

for example:

```
int primes(int a, int b)
{
    if(a==0)return b==1;
    if(b==0)return a==1;
    return primes(b,a%b);
}
```

For the above code, the last lines of code which are calling the function itself is called a Tail Recursive Call. Here the compiler will change the recursive call to a loop:

```
int primes(int a, int b)
{
 L1:
    if(a==0)return b==1;
    if(b==0)return a==1;
    a = b;
    b = a%b;
    goto L1;
}
```

A Tail Recursive Call will not affect debugging.

3.11 Subexpression Elimination

If there are the same subexpressions in the expressions, then the compiler will calculate these subexpressions first and store them into a temporary variable. The variable can be operated directly. This is called subexpression elimination. For example, the following expression 'b*c' is a public subexpression and can be stored in the variable 'tmp' first, thus saving the b*c operation once.

```
    int a,b,c,d,g;
    a = b * c + g;
    d = b * c * d;
→
    tmp = b * c;
    a = tmp + g;
    d = tmp * d;
```

Subexpression Elimination will not affect debugging.

3.12 Tail Merging

The same instruction sequences can be merged into one instruction sequence, this operation is called Tail Merging. The following C source codes show how to use it:

```
00 if ( user_value )
01 {
02     PORTB = 0x55;
03     user_value=0;
04 }
05 else
```

```
06 {
07     PORTB = 0x80;
08     user_value=0;
09 }
```

Ln03 and Ln08 are the same code. The compiler will compile the Ln 08 code only and the Ln03 code will be deleted. If executing the if branch, then after executing the Ln02, the program will jump to Ln08 and not to the else branch which may create some misunderstanding. Because the multiple source codes may have the same assembly code sequence, this will make it difficult for the debugger to determine which code line is executing.

3.13 ROM BP Optimization

For the microcontroller with multiple ROM banks, it is important to set the BP to choose the bank before using Jmp and call. If the current BP is the BP after using Jmp and call, the linker will delete the following instruction mov BP, a.

If the ROM BP instruction can not be deleted, then using the set/clr BP.5 (,6,7) instruction will save using the ACC. For example:

```
void  fun1()
{
     fun2();
}
```

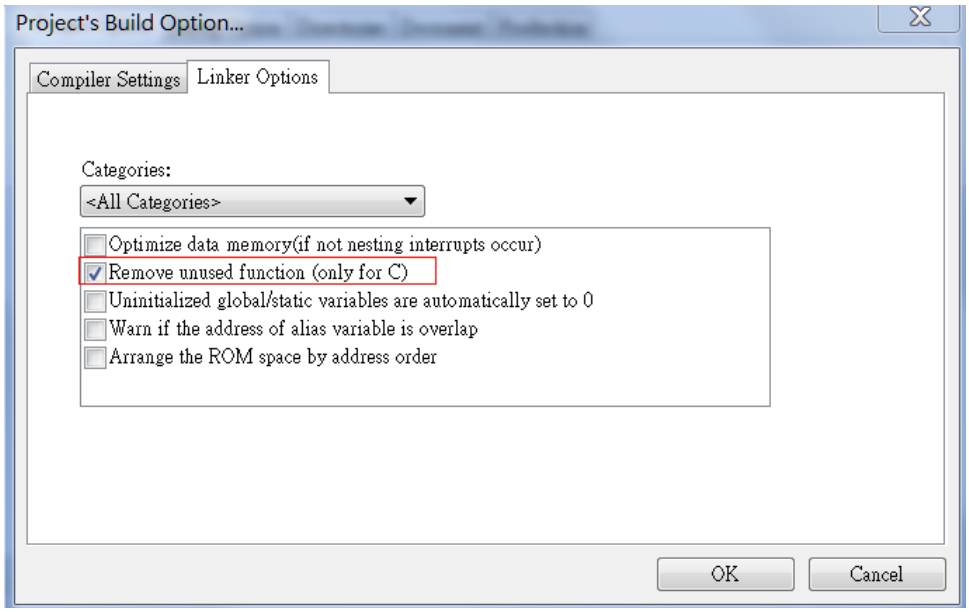
If fun1 and fun2 are assigned to the same bank, then the code will be call _fun2.

If fun1 is assigned to bank0, fun2 is assigned to bank1, then the code will be:

```
set BP.5
call _fun2
clr BP.5
```

3.14 Dead section Elimination

If a function was not called during the programs, then the linker will not assign space for this function, this character can be selected by the following option. A special condition is that when the compiler optimizes the function call to an inline expanded function, then the subfunction may be removed without being called. If this project contains ASM files, this function is invalid.



Chapter 4 Contrast of Holtek C V1, Holtek C V2, Holtek C V3, ANSI C

4.1 Data Type

Data type	Size (bit) C compiler V1	Size (bit) C compiler V2	Size (bit) C compiler V3	Size (bit) ANSI C
bit	1	1	1	N
char	8	8	8	8
signed char	8	8	8	8
unsigned char	8	8	8	8
short	8	16	16	16
unsigned sholt	8	16	16	16
Int	8	16	16	16
unsigned int	8	16	16	12
Long	16	32	32	32
unsigned long	16	32	32	32
long long	N	N	32	64
unsigned long long	N	N	32	64
float	N	32	24	32
double	N	32	32	64
long double	N	N	N	128

Holtek C V2 float/double all use IEEE754 32-bit format.

The floating type is 24-bit supported by C compiler V3, and only 4~5 digit precision is supported by V3.20 or above.

sign	exponent (e)	mantissa (m)
23	22~15	14~8 7~0

Double and long double data types have IEEE 754 32-bit format, and 6~7 digit precision supported by V3.20 or above.

sign	exponent (e)	mantissa (m)
31	30~23	22~16 15~8 7~0

The bit type cannot be used as pointer, also cannot be defined as const type.

4.2 Array

Dimension	C compiler V1 (the Longest Array)	C compiler V2 (the Longest Array)	C compiler V3 (the Longest Array)	ANSI C (the Longest Array)
One dimension array	256	①	②	No limits
Two dimension array	N	①	②	No limits
3 or more dimension array	N	N	②	No limits
Pointer array	N	①	②	No limits
Function array	N	Functional limitation	N	No limits
String array	Not supported	Not supported	②	No limits

Note: ① If it is a const array, it may apear bug when the total length is over 1 page, so is no recommended to use. If it is a general array, the length should not exceed one rambank.

② If it is a const array, the length should not exceed 32K. If it is a general array, the length should not exceed one rambank.

4.3 Identifier Reserved Words

Reserved Word	C compiler V1	C compiler V2	C compiler V3	ANSI C
auto	●	●	●	●
break	●	●	●	●
bit	●	●	●	
case	●	●	●	●
char	●	●	●	●
const	●	●	●	●
constant		●		
continue	●	●	●	●
default	●	●	●	●
do	●	●	●	●
double		●	●	●
else	●	●	●	●
enum	●	●	●	●
extern	●	●	●	●
float		●	●	●
for	●	●	●	●
goto	●	●	●	●
if	●	●	●	●
int	●	●	●	●
long	●	●	●	●
register		●	●	●
return	●	●	●	●
short	●	●	●	●
signed	●	●	●	●
sizeof	●	●	●	●
static	●	●	●	●
struct	●	●	●	●
switch	●	●	●	●
typedef	●	●	●	●
union	●	●	●	●
unsigned	●	●	●	●
void	●	●	●	●
volatile	●	●	●	●
while	●	●	●	●
vector	●	●		
attribute			● ①	
at			● ②	
interrupt			● ③	
entry			● ④	

Note: 1. The ① and ② are used to define absolute address variable, for example:

unsigned char sfr __attribute __((at(0x40))); it means the definition of sfr variable in the 0x40 address.

2. The ① and ③ are used to define interrupt variable, for example:

void __attribute((interrupt(0x04))) isr_name(void) {…}; it means the definition of interrupt isr_name in the 0x04 address.

3. The ④ is for entry function, please refer to 2.2.9.

4.4 Operator

Operators	C compiler V1	C compiler V2	C compiler V3	ANSI C
Arithmetic operators (+,-,*,/,%)	●	●	●	●
Relation operators (>,<,==,>=,<=,!=)	●	●	●	●
Logical operators (!,&&,||)	●	●	●	●
Bitwise operators (<<,>>,~,|,^,&)	●	●	●	●
Assignment operators (=,+=,-=,*=,/=,%=,>>=,<<=,&=,^=,|=)	●	●	●	●
Condition operators(? :)	●	●	●	●
Comma operators(,)	●	●	●	●
Pointeroperators(* and &)	●	●	●	●
Byte evaluation operators(sizeof)	●	●	●	●
Type conversion operators ((type))	●	●	●	●
Structure memberoperators(. ->)	●	●	●	●
Suffix operator([ ])	●	●	●	●
Function adjustment operators(( ))	●	●	●	●
Increment operators(++)	●	●	●	●
Decrement operators((--)	●	●	●	●
Minus sign operators(-)	●	●	●	●
Plus sign operators(+)	●	●	●	●
RAM address assignment operators(@)	●	●		

4.5 Preprocessor Instructions

Preprocessor instruction	C compiler V1	C compiler V2	C compiler V3	ANSI C
#asm	Y	Y	N ②	N
#define	Y	Y	Y	Y
#elif	Y	Y	Y	Y
#else	Y	Y	Y	Y
#endif	Y	Y	Y	Y
#error ①	Y	Y	N	N
#if	Y	Y	Y	Y
#ifdef	Y	Y	Y	Y
#ifndef	Y	Y	Y	Y
#include	Y	Y	Y	Y
#pragma	Y	Y	N	N
#undef	Y	Y	Y	Y

Note: ① Indicates error information: #error size too big

② C compiler V3 inline assembly code use asm (“”) format, please refer to 2.4.

4.6 Preprocessor directive #pragma

The Format:

```
#pragma   keyword   [option]
```

Some keywords may have options.

Keyword	C compiler V1	C compiler V2	C compiler V3	ANSI C
bp_free		●		
bp_nofree		●		
function		●		
nobp		●		
nolocal		●		
nomp0		●		
nomp1		●		
rambank0 norambank	●	●		
rombank0 norombank		●		
rombank		●		
vector	●	●		
novectornest		●		
inline			● ②	

Note: ② C compiler V3 supports inline function, and its format is the same as the standard C language.

4.7 Const Variable

Const Variable	C compiler V1	C compiler V2	C compiler V3	ANSI C
Data Type	bit excepted	bit excepted	bit excepted	any
Used directly by otherdocuments	N	N	Y (when quoting, add qualifier extern before const)	Y (when quoting, add qualifier extern before const)
Should to be declared as globle	Y	N	N	N
Initialization setting at announcement	Y	Y	Y	Y
Constant array size assignment	Y	Y	Y	Y
Addressing operating bit	N ①	N ①	Y	Y

4.8 Pre-defined Head Files

Pre-defined Head File	C compiler V1	C compiler V2	C compiler V3	ANSI C
HTxxxxx.H	Y	Y	Y	N
assert.h	N	N	N	Y
ctype.h	N	Y	Y	Y
errno.h	N	N	N	Y
float.h	N	N	N	Y
limits.h	N	N	N	Y
locale.h	N	N	N	Y
math.h	N	Y	Y	Y
setjmp.h	N	N	N	Y
signal.h	N	N	N	Y
stdarg.h	N	N	N	Y
stddef.h	N	N	N	Y
stdio.h	N	N	N	Y
stdlib.h	N	Y	Y	Y
string.h	N	Y	Y	Y
time.h	N	Y	Y	Y

4.9 Main Functions

Main Function Rule	C compiler V1	C compiler V2	C compiler V3	ANSI C
Number(piece)	1	1	1	1
Data return type	void	void	void	int
Parameter(piece)	None	None	None	2(one pointerarray)

4.10 Interrupt Functions

Interrupt Function Rule	C compiler V1	C compiler V2	C compiler V3	ANSI C
Setup interrupt vectorvalue	Y	Y	Y	No interupt functions
Number(piece)	Plurality acceptable	Plurality acceptable	Plurality acceptable	
Data return type	void	void	void	
Parameter	None	None	None	
Re-enterinterrupt	N	Y ①	Y ①	
Interrupt called in the program	N	N	N	
Interrupt calling assembly function	Y	Y	Y	
Interrupt calling C function	N	Y ②	Y ③	

Note: ① Although different interrupts can be nested at the same time, the same interrupt cannot be generated twice at the same time until the previous interrupt ISR has completed. For those controllers without interrupt nesting feature, the interrupt function cannot be enabled in the service routine.

② It is necessary to define the called function as #pragma nolocal, otherwise RAM overuse will occur which is not recommended.

③ The function called by interrupt can not call the same function with the main function, otherwise RAM overuse will occur. The different interrupt can not call the same function. For instance:
isr1→fun1→fun3
main→fun2→fun1
isr2→fun3
then the isr1 and main both call fun1, while the isr1 and isr2 both call fun3.

4.11 Built-in Functions

Function	C compiler V1 (Paremeter Type)	C compiler V2 (Paremeter Type)	C compiler V3	ANSI C
_clrwdt()	Y	Y	GCC_CLRWDT()	N
_clrwdt1()	Y	Y	GCC_CLRWDT1()	N
_clrwdt2()	Y	Y	GCC_CLRWDT2()	N
_halt()	Y	Y	GCC_HALT()	N
_nop()	Y	Y	GCC_NOP()	N
_rr(int8 *)	Y(int *)	Y(char*)	GCC_RR(int 8)	N
_rrc(int8 *)	Y(int *)	Y(char*)	GCC_RRC(int 8)	N
_lrr(int16*)	Y(long *)	Y(int *)	N	N
_lrrc(int16*)	Y (long*)	Y(int *)	N	N
_rl(int8 *)	Y(int *)	Y(char*)	GCC_RL(int 8)	N
_rlc(int8 *)	Y(int *)	Y(char*)	GCC_RLC(int 8)	N
_lrl(int16*)	Y(long *)	Y(int *)	N	N
_lrlc(int16*)	Y(long *)	Y(int *)	N	N
_swap(int8 *)	Y(int *)	Y(char*)	GCC_SWAP(int 8)	N
_delay(unsigned long tick)	Y(tick<=65535)	Y(tick<=263690)	GCC_DELAY(tick)	N

4.12 Other Functions

Function	C compiler V1	C compiler V2	C compiler V3	ANSI C
Inline assembly	Y	Y	Y ②	N
Static variable	Static variables and functions not supported	Static variables and functions not supported	Y	Y
Constant	Bit constant supported	Bit constant supported	Bit constant supported	Bit constant not supported
Structure and Union	bit field placed in the 8-bit unit. Two 8-bit units are now allowed, neitheris the definition for over 9-bit field	bit field placed in the 8-bit unit. Two 8-bit units are now allowed, neitheris the definition for over 9-bit field	Up to 32-bit field max	Up to 32-bit field max
Function	Recursion function not supported	Recursion function not supported	Recursion function not supported	Recursion function supported
Pointer	Cannot be applied to constant and bit variables. Function pointer not supported	Cannot be applied to constant and bit variables. For pointer function, it must be full domain and all function without parameters	Function pointer not supported	Y
Initial value	Initial value cannot be set when the global variable is being declared. While declaring the const,the initial value must be set first	The initial value cannot be set when the global variable is being declared. declaring the const,the initial value must be set first	Y	Y
Stack	Layers limited ①	Layers limited ①	Layers limited ①	Layers not limited

Note: ① There are limited numbers of stack levels for every MCU. When calling functions, the stack levels required should be taken into account. The numbers of layers occupied by operators or functions during the calling process are shown in the table below:

Operator/Function	Stack Needed	Operator/Function	Stack Needed
main()	0	_rl(int */ char*);	0
_clrwdt()	0	_rlc(int *);	0
_clrwdt1()	0	_lrl(long */ int *);	0
_clrwdt2()	0	_lrlc(long *);	0
_halt()	0	_delay(unsigned long)	1
_nop()	0	*	1
_rr(int */ char*);	0	/	1
_rrc(int *);	0	%	1
_lrr(long */ int *);	0	array/pointer	1
_lrrc(long *);	0	Integer and floating conversion	1

② refer to 2.3.3

Chapter 5 Command Line Mode

This chapter describes the C compiler V3 command line mode and guides programmers on how to use the command mode to compile an original file.

The main contents are as follows:

- entering into the command line environment
- using the command mode to generate an object file
- command line parameters

5.1 Setting Environment Variable

To setup the environment variable before using the command line environment.

Using the set command in the command line to add a bin file in the HT-IDE3000 installation path. For example,set PATH=%PATH%; XXX/bin.

Here, XXX/bin is the bin file in the HT-IDE3000 installation path. The HT-IDE3000 tools of the bin file can be used in the present CMD window.

5.2 Using the Command Mode to compile the original file

After setting the environment variables, then being to write the code. Taking example 5 as an example, compile file1.c and file2.c in example 5 to the corresponding assembly files.

command: hgcc32 [options] cfile –o asmfile

compile file1.c: hgcc32 -g -Os file1.c -o file1.asm

compile file2.c: hgcc32 -g -Os file1.c -o file2.asm

The asm file will then be generated by the above instructions.

5.3 Command line parameter

Parameter	Description
-g	Generate a debug message
-O0/-O1/-O2/-O3/-Os	Optimization parameter
-D<macro>[=<val>]	Macro definition
-I<path>	Set the header file path
-msingle-ram-bank	Single RAM
-mmulti-ram-bank	Multiple RAM (default)
-msingle-rom-bank	Single ROM
-mmulti-rom-bank	Multiple ROM (default)
-fno-builtin	Does not use gcc built in function
-mno-tbhp	Without TBHP
-mtbhp=addr	Specify the TBHP address addr (09H is default)
-mlong-instruction	Extended instruction MCU

There are five kinds of optimization parameters: -O0, -O1, -O2, -O3 and –Os. The compiler can only set one of them to compile.

Optimization level:

-O0: This level will disable all the optimization options. The default level is the level before setting –O. The codes can not be optimized.

-O1: This is the basic optimization level. The compiler will try to generate faster and smaller code with limited time overheads. These optimizations are very basic and can be completed successfully.

-O2: Advanced optimization for -O1, it is the optimized level, -O2 enables more signs than –O1. After setting –O2, the compiler will try to increase the code performance while not increase code size but will consume much more compilation time

-O3: This is the highest optimization level. Using this option, the code compilation time will become longer and the generated code may be different from the original codes. For this reason it is not generally used.

-Os: This level is used to optimize the code size. Using -O2, the code stored space will not increase. This is extremely useful for small storage capacity devices.

For a more detailed parameter description refer to the GCC user's guide.

Chapter 6 Multiple file programming

Multiple files may be used in a project. Here, similar functions and definitions are written into one file in order to manage them more easily. This chapter describes the related contents of multiple file programming.

The main contents are as follows:

- Header file
- Common variables
- Function call from other original files
- Using libraries

6.1 Header File

Head file used to declare variables and functions, define macros, specify the address variables and types and Can't define generic variables and functions.

```
#ifndef _XXX_H
#define _XXX_H
.....
#endif
```

6.2 Common Variables

If multiple original files access a common variable, then the variable needs to be defined as a global variable without static. Using extern to declare the variable as an external variable, after which the variable can be used. In the same way the variable can be used in the header file. The extern can be either in the function body or outside the function.

6.3 Calling a function from other original files

Using the header file and extern allows external functions to be referenced. The external functions may modify the global variables of the original files when calling the external functions.

Example 12: Using external functions to modify external variables

Code list 5.1:

```
FUNCTION.H
#ifndef _FUNCTION_H
#define _FUNCTION_H

typedef unsigned charu8;
typedef unsigned int u16;
typedef unsigned longu32;

#define BOOL   u8
#define TRUE   1
#define FALSE  0

u8 getMax(u8 num1, u8 num2);

#endif
FUN.C
#include "FUNCTION.H"
u8 g_var;
```

```
u8 getMax(u8 num1, u8 num2)
{
    if(num2 == 0){
        g_var = 0x55;
    }else if(num1 == 0){
        g_var = 0xaa;
    }
    return num1 > num2 ? num1 : num2;
}

MAIN.C
u8 sk = getMax(28, 0);
void main()
{
    while(1){
        GCC_NOP();
    }
}
```

Operation result: sk = 28, g_var = 0x55

6.4 Using libraries

6.4.1 Generate Libraries

Refer to the “Chapter 9 Library Manager” in the HT-IDE3000 User’s Guide.

6.4.2 Considerations

Special care should be taken with the library-shared in different MCUs:

1. For the same instruction type, the MCU with extended instructions cannot be shared library with the MCU without extended instructions.
2. The MCU with single ROM/RAM bank cannot be shared library with the MCU with multiple ROM/RAM banks.
3. The MCU without the TBHP register cannot be shared library with the MCU with the TBHP register.
4. The MCU with a 16-bit ROM cannot be shared library with the MCU with a 14/15-bit ROM.
5. The options selected in the project should be the same with the library.

6.4.3 Reference Libraries

Refer to section 2.1.5.

Chapter 7 Mixed Language

In order to improve the efficiency of programs and ROM utilization, it is necessary to write programs using mixed languages. This chapter describes how to program with mixed languages.

The following subjects are discussed:

- Data format
- Calling assembly function from a C program
- Calling a C function from an assembly program

7.1 Data format

The data format is Little Endian where the low byte occupies the lower address and the high byte occupies the higher address. For example:

```
static long ldata __attribute__ ((at(0x180)));
ldata = 0xAABBCCDD;
In the memory, the data format is as follow:
```

Address	0x180	0x181	0x182	0x183
Content	0xDD	0xCC	0xBB	0xAA

7.2 Variable and function naming rules

- When Holtek C V3 compiler is compiling global variable and function, it will add a underscore character in front of the old name. For example:
  The global variable count is compiled into _count.
  The function GetTotalSize is compiled into _GetTotalSize.
- Local variables, static variables and function parameters are named quite irregularly, so please refer to the debug information in the assembly files compiled. But it is noted that the compiled name may be different each time.
- After the assembler compiled the assembly program, it will change the names of variable and function for the upper case letters. For example:
  The variable count is compiled into COUNT.
  The function GetTotalSize is compiled into GETTOTALSIZE.

7.3 Calling an assembly function from the C program

Assembly Language Program function definition rules:

1. Add an underscore character prior to the function name and declare it as a public variable.
2. If the function includes parameters, declare the function parameters as public variables.
3. Define the variables as local if there are local variables.
4. Define functions with proc/endp.

Calling Rules for C:

1. Uses capital letters to define and declare the called function name.
2. If the function includes parameters, declare the function parameters externally.
3. Call the function.

Example 13: Assembly language subtraction function

Code list 6.2:

```
    CODE.ASM
    public _opera
public _opera_var1
public _opera_var2                ;; define the function and parameters as public

_opera .section page 'code'
_opera proc                       ;; define function with proc/endp
    local _opera_var1 db ?        ;; define parameters
    local _opera_var2 db ?
    local _result_local  db ?     ;; define local variables
    mov a, _opera_var1
    sub a, _opera_var2
    mov _result_local, a
    ret
_opera endp

MAIN.C
extern unsigned char OPERA();     // define the function namewith uppercase
asm("extern _OPERA_VAR1  : byte");
asm("extern _OPERA_VAR2  : byte");

void main()
{
    volatile unsigned char result;
    asm("mov a, 20h");
    asm("mov _OPERA_VAR1, a");

    asm("mov a, 10h");
    asm("mov _OPERA_VAR2, a");

    result = OPERA();             // function call
    while(1){
        asm("nop");
    }
}
```

Operation result: result = 0x10

7.4 Calling a C function from the assembly program

Calling Rule for C: Uses capital letters to define and declare the called function name.

Mixed lauguage program function definition rules:

1. Declare the function name starting with underscored letters to be external functions
2. Declare the call function with proc/endp
3. If the function includes parameters, declare all the corresponding variables to be external variables. The parameter assembly names may refer to the assembly files which the C function compiles. Note that each compile result may be different, so try not to take parameters.
4. After calling the C function, read the return value, if the return value is 1 byte, then it will be placed into ACC. If the return value is 2 bytes the low byte will be placed into ra and the high byte into rb. If the return value is 4 bytes the return value will be placed into ra, rb, rc and rd from low byte to high byte. Note that ra~rd are already defined, they just need to be declared before being used.

Code list 6.3:

```
FUN.C
int DISPLAY(char row, int col)  // define function, the function name must be uppercase.
{
  int retval;
  retval=(int)(row << 1) + col;
  return retval;
}

CODE.ASM
;;code.asm call function _DISPLAY
extern _DISPLAY : near          ;; declare function name as external name
extern _DISPLAY_2 : byte        ;; declare parameter variable name
extern ra : byte                ;; declare return value
extern rb : byte
CODE .section 'code'
_code proc
local _code_loc db 2 dup(?)     ;; define local variable
MOV A, 10h
MOV _DISPLAY_2, A               ;; save the value to the second parameter col
CLR _DISPLAY_2[1]               ;; set the high byte of the second parameter to 0
MOV A, 20h
MOV _DISPLAY_2[2], A            ;; save the value to the first parameter row
CALL _DISPLAY                   ;; call C function _DISPLAY
MOV A,ra
MOV _code_loc,A
MOV A,rb
MOV _code_loc[1],A              ;; get return value from ra,rb…, and store in the local
                                ;; variable _code_loc,the low byte is stored in ra, the
                                ;; high byte is stored in rb. RET
_code endp
```

Operation result: Variable code_loc = 0x0050

Note that if the following errors in fig. 6_3_1 are shown:

```
Error(L2001) : Unresolved external symbol 'RA'
Error(L2001) : Unresolved external symbol 'RB'
```

fig. 6_3_1

Then the option “Case sensitive for assembly” in the compiler flags option needs to be checked.

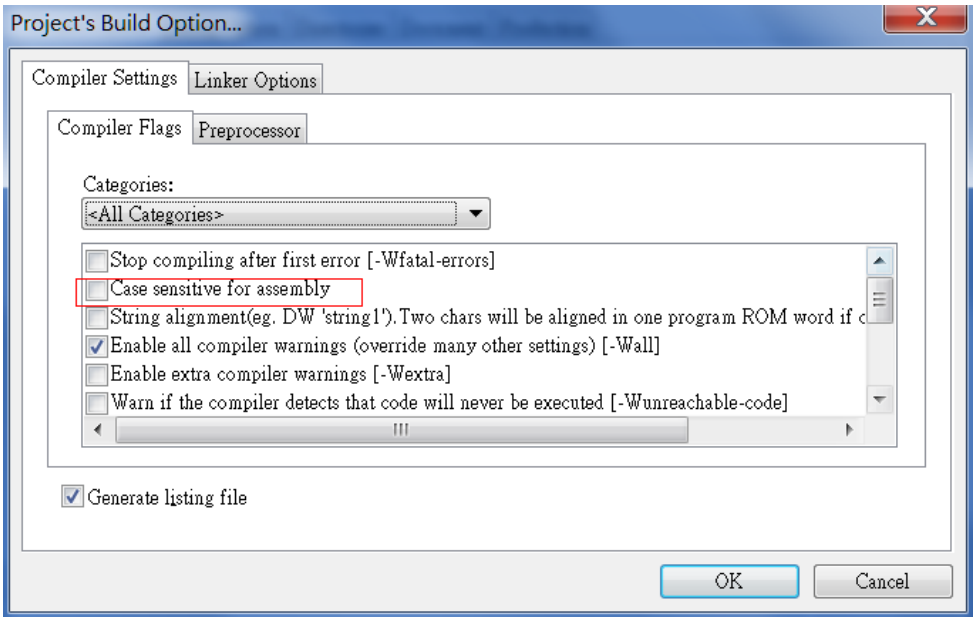


fig. 6_3_2

Chapter 8 Common Error Solutions

8.1 Internal Error

If error information includes an internal compiler error, then it is a compiler internal error Contact Holtek in this case. For example:

```
ting\ROM_Bank_Setting.c: In function 'main':
ting\ROM_Bank_Setting.c:72:22: internal compiler error: in emit_library_call_value_1, at calls.c:3929
```

8.2 RAM bank0 overflow

For those without extended instruction architecture MCU, the C Compiler will assign the variables to RAM bank0 (extended instruction MCU can assign the variables to any bank automatically) by default. When bank0 is full, RAM bank 0 overflows and the following massage will be shown:

```
Linking...
Error(L1038) : RAM (bank 0) overflow,memory allocation fails for section '_fun1'
Total 1 error(s), Total 0 Warning(s)
'ROM_Bank_Setting' - Total 1 error(s), 0 warning(s)
```

When the above message is shown, you should take the following measures:

- Check the data type is correct or not (especially the programs from V1 C Compiler)
- If it is multi RAM bank MCU, you can put the global variables to other banks manually, please refer to the section 2.2.2

8.3 ROM/RAM space overflow

When ROM or RAM space is not enough, the following message will be shown:

```
Error(L1038) : ROM (bank *) overflow,memory allocation fails for sec
Total 1 error(s), Total 43 Warning(s)
'eWriterProLCD_000013' - Total 1 error(s), 45 warning(s)
```

When the above message is shown, the following measures should be taken:

- Check the optimize parameter -Os is open or not, refer to the section2.1.4
- Check the map file to understand the RAM/ROM assignment and delete unnecessary programs.

8.4 Variable Overlap Warning

When absolute address variable positions overlap, the following messages will be generated:

```
Linking...
Warning(L3010) : (Absolute Adress:80H,length:8) is overlay with(Adress:80H,length:8)
Warning(L3010) : (Absolute Adress:88H,length:6) is overlay with(Adress:88H,length:6)
Warning(L3010) : (Absolute Adress:8eH,length:13) is overlay with(Adress:8eH,length:13)
```

There are two conditions which may cause the above warnings:

- The same absolute variables are defined many times in different files, such as the variable var is defined in a.h:
  static volatile unsigned char var __attribute__ ((at(0x180)));
  When t1.c and t2.c both include a.h, then the warning message will be popped up, in this case, this warning message can be ignored, or set the option to avoid the warning message. refer to section 2.1.5
- The defined addresses of different variables overlap, shown as the follows, the addresses of _b and _a overlap, _b needs to be defined in the address 0x0142

```
DEFINE_SFR(unsigned int _a, 0x0140);
DEFINE_SFR(unsigned char _b, 0x0141); //error
```

8.5 Variable Redefinition

If a variable (not absolute address variable) is defined in the header file, while the header file is quoted by multiple .c files, then variable redefinition will occur:

```
Linking...
Error(L1031) : Public symbols are duplicated
               Public symbol '_a' in the file C:\Documents and Settings\ydwang\My Documents\HTK_Project\t\T.OBJ
               Public symbol '_a' in the file C:\Documents and Settings\ydwang\My Documents\HTK_Project\t\T1.OBJ
```

Solution:

Do not define the variables in the header file. If t.c and t1.c both need to use a, then define a in a file and declare a with extern int in the header file:

```
//t.C
#include "t.h"
int a;
void main()
{
        a=2;
}
```

```
//t.h
extern int a;
```

```
//t1.c
#include "t.h"
void fun()
{
       a=3;
}
```

Chapter 9 Programming Examples

This chapter shows how to use the C compiler V3 to compile MCU programs.

The main contents are as follows:

- Using the interrupt function
- Using a mixed language program

9.1 LED flashing using the interrupt

This example uses the timer to control LED flashing. The time interval is 1s, which means the LED light is on for 1s and off for 1s.

Code list 7.1:

```
#include "HT66F50.h"
void main()
{
    _acerl=0x00;
    _cp0c=0x08;
    _pac = 0x00;                // set PAC as output
    _pa = 0xff;               // All SEG off
    _mf0e = 0x01;               // enable Multi-function 0 interrupt
    _t2ae = 0x01;               // enable T2A interrupt
    _tm2c0 = 0x30;              // set clk = f(sys)/64
    _tm2c1 = 0xc1;              // set Compared with CCRA
    _t2af = 0x00;               // clear T2A interrupt flag
    _mf0f = 0x00;               // clear Multi-function 0 flag
    _emi = 1;                   // enable interrupt
    _tm2al = 0x03;              // Matching value
    _tm2ah = 0x00;
    _t2on = 1;                  // start counting
    while(1);
}

DEFINE_ISR(ISR_ADC, 0x14)       // definition ISR
{
    _t2af = 0x00;               // clear T2A interrupt flag
    _pa = ~_pa;
    _tm2al = 0x24;              // Matching value
    _tm2ah = 0xf4;
}
```

9.2 7-segment LED display number using table

To display numbers using a 7-segment LED from the assembly table using mixed language.

```
code list 7.2:
Table.ASM
#INCLUDE HT66F50.INC

public _code

_code .SECTION 'CODE'
_code proc
```

```
TAB_7_SEG:
    DC 0F9C0H
    DC 0B0A4H
    DC 09299H
    DC 0F882H
    DC 09580H
    DC 08388H
    DC 0A1A7H
    DC 08E86H
_code endp
    END
```

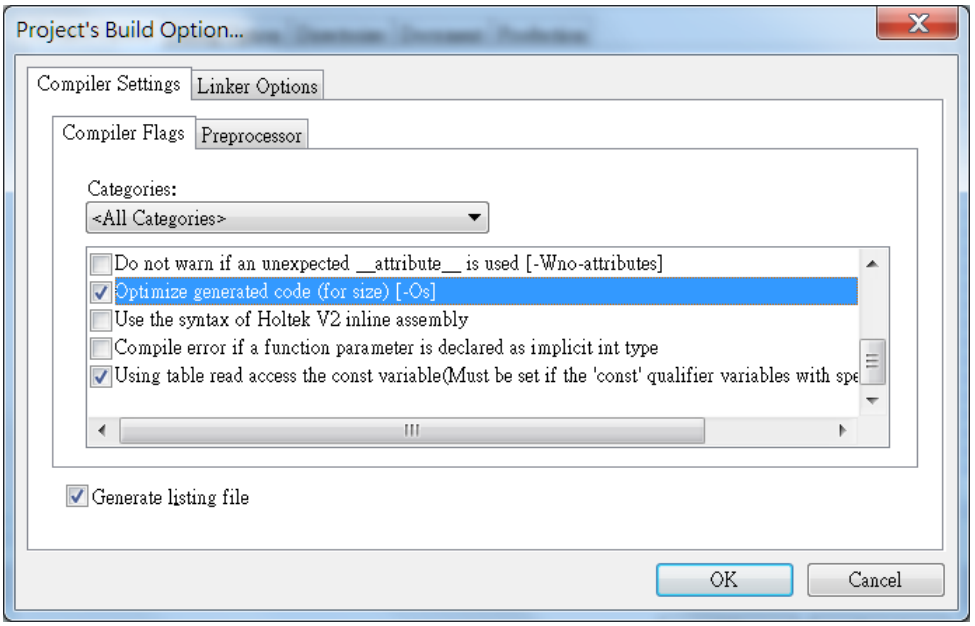
Main.c

```
#include "HT66F50.h"
void _delay(unsigned char times)
{
    volatile unsigned char t1, t2, t3;
    t1 = times;
    while(t1--){
        t2 = 3;
        while(t2--){
            t3 = 110;
            while(t3--);
        }
    }
}
asm("extern _CODE:near");
void main()
{
    unsigned char k;
    _acerl=0x00;
    _cp0c=0x08;
    _pac = 0x00;
    _pa = 0xff;
    asm("mov a, low _CODE");
    asm("mov %0, a" : "=m"(_tblp));

    asm("mov a, high _CODE");
    asm("mov %0, a" : "=m"(_tbhp));
    while(1)
    {
        for(k = 0; k < 8; k++)
        {
            asm("tabrdc %0" : "=m"(_pa));
            asm("inc %0" : "=m"(_tblp));
            _delay(50);
            asm("mov a, %0" : "=m"(_tblh));
            asm("mov %0, a" : "=m"(_pa));
            _delay(50);
        }
        asm("mov a, 0ffh");
        asm("mov %0, a" : "=m"(_pa));    //all SEG off
        _delay(50);
    }
}
```

Chapter 10 Program Optimization

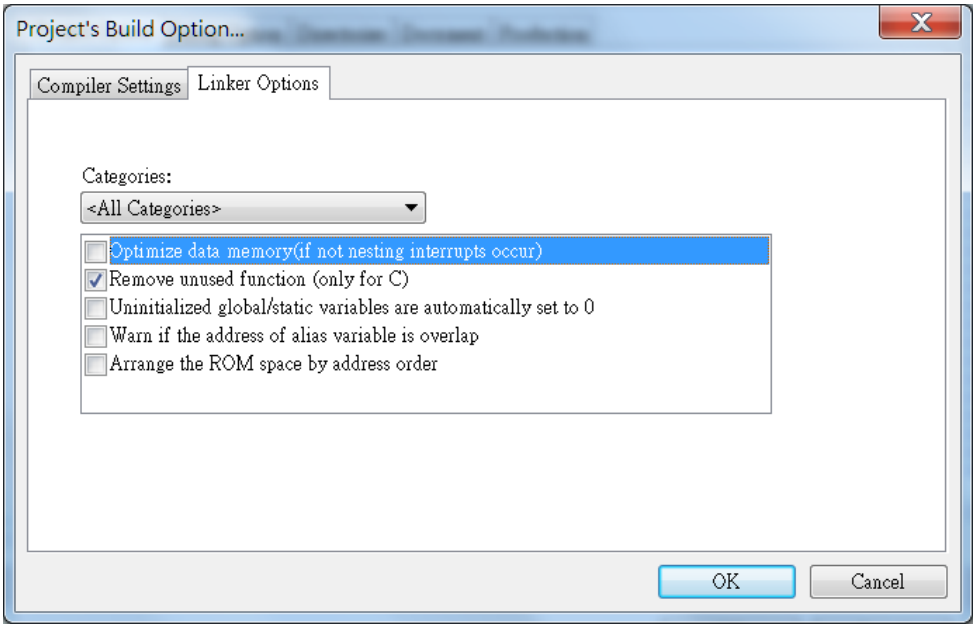
10.1 Optimization Options

Select “Optimize generated code”, i.e., -Os, as shown below. For more detailed optimization contents description refer to chapter 3.

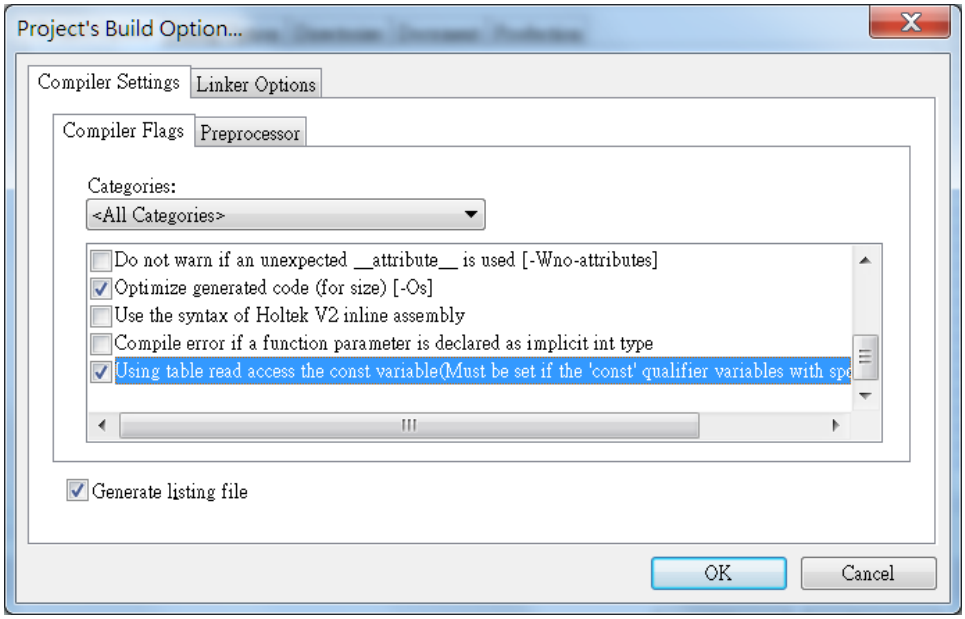


If no nested interrupts occur in the program, which means the program will not enter another interrupt during the execution of the current interrupt, then selecting the “Optimize data memory” option can save RAM space.

The interrupt program should be written as briefly as possible, otherwise it will take up too much RAM space.

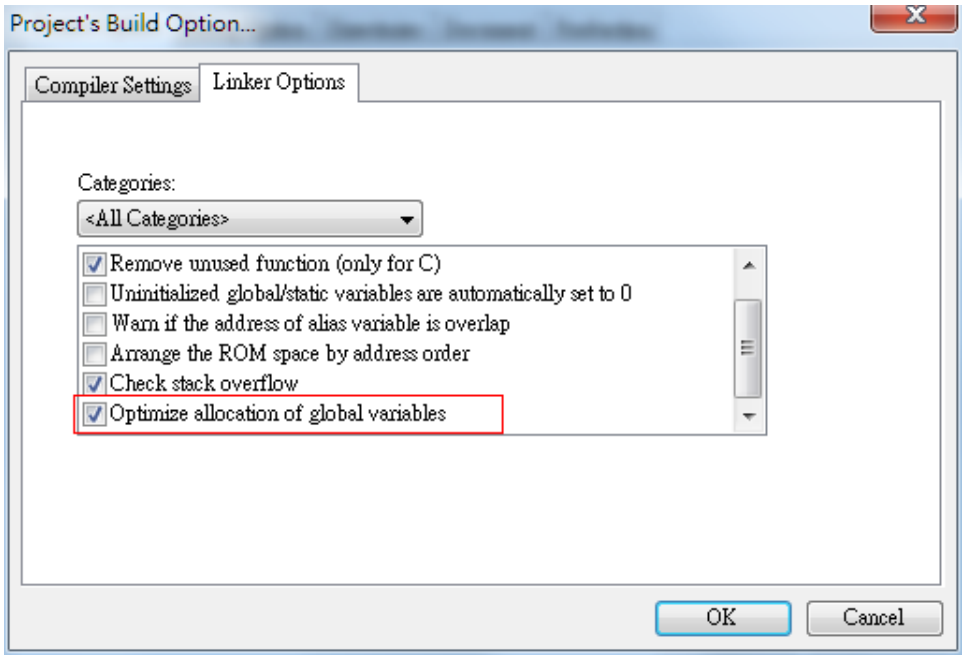


For the MCU whose ROM is 16-bits wide and contains the TBHP register and without extended instructions, the Compiler supports table look up instructions to access constant variables as shown by the following option which is checked by default. If the number of the const variables is large, checking it will save code, otherwise it will waste. Users can choose whether to check it according to the actual program.

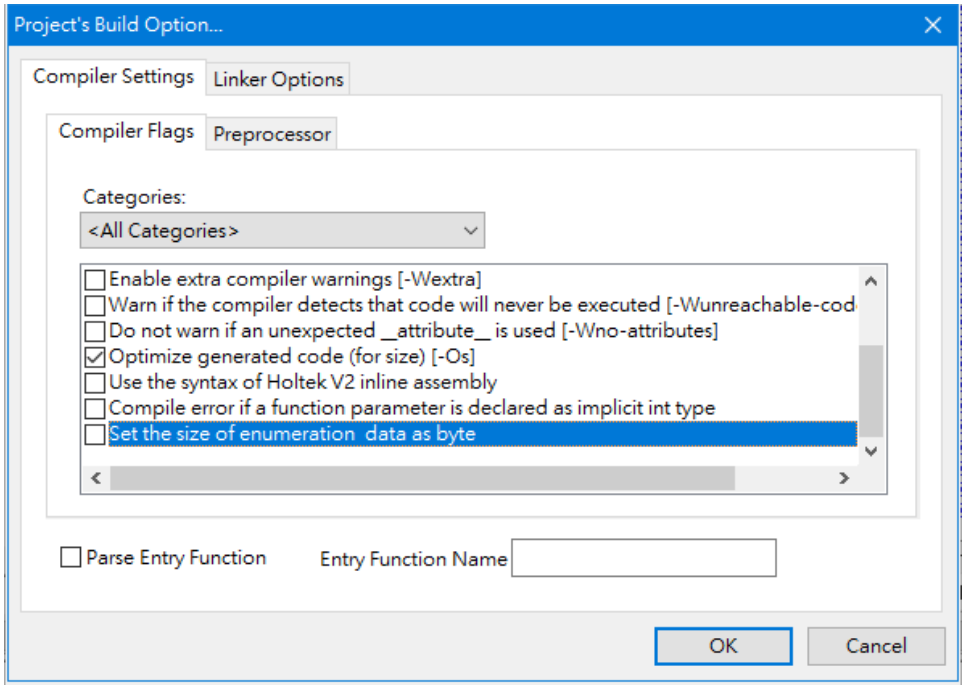


Select the following parameter “optimize allocation of global variables”:

For the MCU with extended instructions, it is not necessary to specify a variable address. The linker can automatically assign all variable addresses including banks other than bank0 in RAM. It can assign, according to the using frequency of the variables, and give priority to the more frequently used variables.



If the defined enum value does not exceed 127, or check the following options:



10.2 Variable Declaration

10.2.1 unsigned/signed

If there are no negative conditions, using unsigned will save code size.

char array[10];
void main(void)
{
  char i;
  for(i = 0;i<=9;i++)
  array[i] = 0;
}
	char array[10];
	void main(void)
	{
	  unsigned char i;
	  for(i = 0;i<=9;i++)
	  array[i] = 0;
	}

Code size:34	Code size:31

10.2.2 Date type

Select the data types with the appropriate scope to make the instruction more simple.

long i;
void main(void)
{
  if(i>=456)
  i = 2;
}
	unsigned int i;
	void main(void)
	{
	  if(i>=456)
	  i = 2;
	}

Code size:22	Code size:14

10.2.3 Floating Constant

The floating constant defaults to the double type. If the required precision in the calculation is not high, it can be forced to change to float. For example: (float)3.14 .

float s,r;
void main()
{
  s = r * r * 3.14;
}
	float s,r;
	void main()
	{
	  s = r * r * (float)3.14;
	}

Code size:343	Code size:177

The compiler will not carry out constant folding in floating operations. So if it is required to calculate two floating constants the result can be first calculated:

#define HALF (float)0.5
#define QUARTER (float)0.25
float l,r;
void main()
{
  r = l * HALF * HALF;
}
	#define HALF (float)0.5
	#define QUARTER (float)0.25
	float l,r;
	void main()
	{
	  r = l * QUARTER;
	}

Code size:171	Code size:152

10.2.4 Const Array

A const array defined as a global variable can save more RAM space than one which is defined as a local variable.

unsigned char sum;
unsigned char dx[7];
void main()
{
  const unsigned char tx[7] =
  {1,3,5,15,5,3,1};
  unsigned char i;
  for(i=0;i<7;i++)
  sum += dx[i]*tx[i];
}
	const unsigned char tx[7] = {1,3,5,15,5,3,1};
	unsigned char sum;
	unsigned char dx[7];
	void main()
	{
	  unsigned char i;
	  for(i=0;i<7;i++)
	  sum += dx[i]*tx[i];
	}

RAM size:23	RAM size:16

10.2.5 Define variables with similar functions into an array to use loop statements

unsigned int n1,n2,n3,n4;
void func()
{
  unsigned char i;
  for(i=0; i<10; i++)
  {
  n1 += (i * 201);
  n2 += (i * 202);
  n3 += (i * 203);
  n4 += (i * 204);
  }
}
	unsigned int n[4];
	void func()
	{
	  unsigned char i,j;
	  for(i=0; i<10; i++)
	  {
	  for(j=0; j<4; j++)
	  n[j] += (i * (j + 200));
	  }
	}

Code size:140	Code size:75

10.2.6 Except for the delay function, local variables cannot be defined using volatile

10.3 Program structure

10.3.1 Adjust statement order to apply tail merging optimization

The compiler will execute tail merging optimization, refer to section 3.12. For switch or if/else statements, write the same statement at the end of each branch to apply tail merging.

```
unsigned char n;
unsigned char array[4];
void func()
{
  switch(n)
  {
    case 1:
      array[1] = 0xff;
      array[0] = 4;
      array[2] = 2;
      array[3] = 1;
      break;
    case 2:
      array[2] = 0xff;
      array[0] = 4;
      array[1] = 3;
      array[3] = 1;
      break;
    case 3:
      array[3] = 0xff;
      array[0] = 4;
      array[1] = 3;
      array[2] = 2;
      break;
    default : break;
  }
}
```

Code size:33

```
unsigned char n;
unsigned char array[4];
void func()
{
  switch(n)
  {
    case 1:
      array[1] = 0xff;
      array[2] = 2;
      array[3] = 1;
      array[0] = 4;
    break;
      case 2:
      array[2] = 0xff;
      array[3] = 1;
      array[1] = 3;
      array[0] = 4;
      break;
    case 3:
      array[3] = 0xff;
      array[2] = 2;
      array[1] = 3;
      array[0] = 4;
      break;
    default : break;
  }
}
```

Code size:29

10.3.2 Replace repeated operations with a loop

When there are repeated and regular operations in the program, then try to replace them with a loop.

```
unsigned char show_data[6];
unsigned long hex;
void main()
{
  show_data[5]=hex%10;
  show_data[4]=hex/10%10;
  show_data[3]=hex/100%10;
  show_data[2]=hex/1000%10;
  show_data[1]=hex/10000%10;
  show_data[0]=hex/100000%10;
}
```

Code size:378

```
unsigned char show_data[6];
unsigned long hex;
void main()
{
  unsigned long temp=hex;
  unsigned char i;
  for(i=6;i>0;)
  {
    i--;
    show_data[i]=temp%10;
    temp/=10;
  }
}
```

Code size:156

10.4 Function Call

10.4.1 Avoid unnecessary function calls

If a function is called many times, but there is no difference in its return value, it is recommended to define a variable to receive the return value and use it instead.

int fun(int i)
{
  return i;
}
int i;
void main(void)
{
  if(fun(i)== 0)
  i = 0;
  else if (fun(i) == 1)
  i = 6;
  else if (fun(i) == 2)
  i = 4;
}
	int fun(int i)
	{
	  return i;
	}
	int i;
	void main(void)
	{
	  int temp = fun(i);
	  if(temp== 0)
	  i = 0;
	  else if (temp == 1)
	  i = 6;
	  else if (temp == 2)
	  i = 4;
	}

	reduce the run time

10.4.2 Encapsulate frequently used codes as a function

If a piece of code is frequently used in the program, encapsulate the code as a function to reduce the code size.

char array[10][10];
void func1()
{
  unsigned char i,j;
  for(i = 0;i <= 9; i++)
  for(j = 0;j<= 9; j++)
  array[i][j] = 0;
}
void func2()
{
  unsigned char i,j;
  for(i = 0;i <= 9; i++)
  for(j = 0;j<= 9; j++)
  array[i][j] = 0xff;
}
	char array[10][10];
	void init_array(char n)
	{
	  unsigned char i,j;
	  for(i = 0;i <= 9; i++)
	  for(j = 0;j<= 9; j++)
	  array[i][j] = n;
	}
	void func1()
	{
	  init_array(0);
	}
	void func2()
	{
	  init_array(0xff);
	}

Code size:108	Code size:52

10.4.3 If the function can only be called in the current file, it can be defined as static

float s;
unsigned char func(unsigned char *dx)
{
  unsigned char sum;
  sum = dx[0]+ dx[1]+ dx[2] ;
  return sum;
}
unsigned char sum;
unsigned char array[4];
void main()
{
  sum = func(array);
  s = sum * (float)3.14;
}
	float s;
	static unsigned char func(unsigned char *dx)
	{
	  unsigned char sum;
	  sum = dx[0]+ dx[1]+ dx[2] ;
	  return sum;
	}
	unsigned char sum;
	unsigned char array[4];
	void main()
	{
	  sum = func(array);
	  s = sum * (float)3.14;
	}

Code size:222	Code size:184

10.5 Global Variable Assignment

For those without extended instruction architecture MCUs, banks other than Bank0 in RAM must be addressed indirectly. As shown in the following example, the number of instructions for indirect addressing is 5 more than direct addressing. So when RAM Bank0 overflows, users can define the infrequently used variables to other banks and the more frequently used variables to bank0.

Direct addressing (Bank 0)	Indirect addressing (not Bank 0)
Rambank 0 ds ds .section 'data' _var0 db ?	Rambank 1 ds ds .section 'data' _var1 db ?

MOV A,40H
MOV _var0, A
	MOV A,BANK _var1
	OR A,ROM_BANK FUNC
	MOV BP,A
	MOV A,OFFSET _var1
	MOV MP1,A
	MOV A,40H
	MOV IAR1,A

Code size:2 words	Code size:7 words

10.6 Interrupt Service Routines

Generally, if two functions do not have a call relationship with each other, their local variables can share the same address but interrupt service routines can not share the local variable address with the main function. So in order to reduce the use of RAM, the interrupt program should be written as briefly as possible and not be too complicated.

10.7 Variable Initialization

If there is a Clear RAM function in the program, the initial value defined by the global/static variable can be removed, because the initial value is invalid at this time.

Appendix A: ASCII CODE TABLE

DEC	HEX	Symbol	DEC	HEX	Symbol	DEC	HEX	Symbol	DEC	HEX	Symbol
0	0	NUL	32	20	space	64	40	@	96	60	`
1	1	SOH	33	21	!	65	41	A	97	61	a
2	2	STX	34	22	“	66	42	B	98	62	b
3	3	ETX	35	23	#	67	43	C	99	63	c
4	4	EOT	36	24	$	68	44	D	100	64	d
5	5	ENQ	37	25	%	69	45	E	101	65	e
6	6	ACK	38	26	&	70	46	F	102	66	f
7	7	BEL	39	27	‘	71	47	G	103	67	g
8	8	BS	40	28	(	72	48	H	104	68	h
9	9	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	

Appendix B: Operator Priority

Priority	Operators	Description	operation type	Associativity
1	( )	parenthesis	unary	left to right
	[ ]	subscription		
	->	structure pointer		
	.	structure member		
2	!	Deny	unary	right to left
	~	Take one's complement		
	++ --	increment, decrement		
	(type key word)	type coercions		
	+ -	unary plus, unary minus		
	*	pointer		
	&	address		
	sizeof	size of type		
3	* / %	Multiplication, Division, Remainder	binary	left to right
4	+ -	unary plus, unary minus	binary	left to right
5	<<	shift left	binary	left to right
	>>	shift right		
6	< <= > >=	Smaller than, Smaller than or equal to, Greater than, Greater than or equal to	relational	left to right
7	= = !=	Equal, Unequal	relational	left to right
8	&	bitwise AND	bit operation	left to right
9	^	bitwise XOR	bit operation	left to right
10	|	bitwise OR	bit operation	left to right
11	&&	logical AND	bit operation	left to right
12	||	logical OR	bit operation	left to right
13	? :	conditional expression	ternary	right to left
14	= += -= *= /= %= << = >>= &= ^= |=	assignment	binary	right to left
15	,	comma	sequential	left to right

Appendix C: Command line mode command parameters and functions

1. Compiler parameter

Command: hgcc32 [options] cfile –o asmfile

Parameter	Description
-g	Generate an debug message
-O0/-O1/-O2/-O3/-Os	Optimization parameter
-D<macro>[=<val>]	Macro definition
-I<path>	Set the header file path
-msingle-ram-bank	Single RAM
-mmulti-ram-bank	Multiple RAM(default)
-msingle-rom-bank	Single ROM
-mmulti-rom-bank	Multiple ROM(default)
-fno-builtin	Without using gcc build in function
-mno-tbhp	Without TBHP
-mtbhp=addr	Specify TBHP address addr(1fH is default)
-mlong-instruction	Extended instruction MCU

2. Assembler parameter

Command: hasmgcc32 [options] source, object, listing

Parameter	Description
/chip=chip-name	Specify MCU type
/case	Case-sensitive
/d<macro>	Macro definition
/i<include-path>	Set the search path of the header file
/z	Generate a debug message
/h (/?)	Display help
source	Asm file that will be compiled
object	Specify the name of the generated obj files
listing	Specify the name of the generated list files

3. Linker parameter

Command: /HIDE=xxx /MCU=xxx [/NOLOGO] [/novectornest] [/OptimizeParam=x] [/OptimizeLInst=x] [/Startup0] [/EEPROM=xx] [/TBHP=x] [/ERRORLOG="xxx"] [/option] objectfile [,taskfile [,mapfile [,dbgfile [,libraryfiles]]]] [;]

Note: These parameters must be set in accordance with the above order, otherwise an error message will be generated by the linker. The parameters are case-sensitive.

The environment variable LIB is the searching path of the lib file.

Parameters	Description
/HIDE	IDE control code. 8 hexadecimal numbers
/MCU	MCU project name
/NOLOGO	Hidden LOGO
/novectornest	This parameter requests no ISR nesting which can save RAM space (for C Compiler V3 only)
/OptimizeParam=x	Default 0, The low half byte of x means the BP optimize option, 0 means disable, 2 means enable The high half byte of x means the dead section optimize option, 0 means disable, 1 means enable (for C Compiler V3 only)
/OptimizeLInst=x	Expresses the extended instruction optimization, 0 means no optimization, 1 means optimization. Without optimization by default (for C Compiler V3 only)
/Startup0	Initialize the global variables without initial values to 0 (for C Compiler V3 only)
/TBHP=x	x means the address of TBHP, default 9
/EEPROM=xx	Specify EEPROM_DATA_SIZE, default 0 (for C Compiler V3 only)
/ERRORLOG	Save the error log path
/MAP	Generate MAP file
/ADDR:section_name=addr [,section_name=addr]	Specify that some section assignment addresses are started from the specified addresses. Note: addr is hexadecimal
/HELP (/?)	Display the command line format help information

Note: To set the environment variable CFG before calling the assembler and linker CFG set HTCFG=IDE path\MCU.

Reference books

《HT-IDE3000 User's Guide》

The User's Guide introduces how to use the tools including the HT-IDE, assembler and linker. It can be obtained from the HT-IDE3000 installment file DOC .

《C standard library user's guide》

This user's guide introduces the standard function library and their use supported by Holtek C. This can be obtained from the HT-IDE3000 installment file DOC.

《Holtek C Compiler V3 FAQ》

C Compiler V3 FAQ common problems. These FAQs are continuously updated and can be obtained from the HT-IDE3000 installment file DOC.

《gcc manual》

GCC manual can be downloaded from http://gcc.gnu.org/onlinedocs/gcc-4.8.1/gcc.pdf