



HMS800C C Compiler Programming Guide

MCU Application Team,
ABOV Semiconductor, Ltd

CONTENTS

INTRODUCTION.....	1
INSTALLATION.....	1
Running the compiler	2
Command Line Versions	2
GUI Versions.....	2
C Compiler	2
Language Facilities	2
Performance.....	2
TUTORIAL	3
Creating a new project.....	3
Build a project.....	9
C PROGRAMMING GUIDELINE	10
Introduction to startup files	10
Using header files.....	11
Device-specific options and option files	18
Variable and label naming conventions	18
Attributes of functions and variables.....	19
Optimizing features of the HMS800 C Compiler	19
Access to control registers	20
Using <i>struct</i> and <i>union</i>	21
Using constant pointers.....	22

Using <i>sfr</i> and <i>sbit</i> types	22
Using <i>BIT</i> and <i>BYTE</i> types	23
Interrupt function definition.....	23
Efficient function call	24
Allocating variables user-specified memory.....	24
Using pointers.....	25
Restriction on use of RAM PAGE0	26
Tips to Reduce Code Size	26
HMS800C COMPILER COMMAND OPTIONS.....	27
Option Summary	27
Options Controlling the Kind of Output.....	31
Options Controlling C Dialect	33
Options to Request or Suppress Warnings.....	39
Options for Debugging Your Program	55
Options That Control Optimization	55
Options Controlling the Preprocessor.....	63
Passing Options to the Assembler	68
Options for Linking.....	68
Options for Directory Search.....	70
Options for Code Generation Conventions	72
EXTENSIONS TO C LANGUAGE FAMILY	75
Statements and Declarations in Expressions.....	75

Locally Declared Labels	76
Labels as Values.....	77
Naming an Expression's Type	78
Referring to a Type with `typeof'	79
Macros with a Variable Number of Arguments	80
Case Ranges.....	81
Cast to a Union Type.....	82
Declaring Attributes of Functions.....	82
Attribute Syntax	88
Specifying Attributes of Variables	91
Specifying Attributes of Types.....	94

INTRODUCTION

This chapter explains how to install and run the command line and GUI versions of our compiler, and gives an overview of the user guides supplied with them.

INSTALLATION

This section describes how to install and uninstall HMS800C C compiler.

What you need

- Windows NT 3.51 or later, Windows XP, or Windows Server 2003
- Up to 100 Mbytes of free disk space.
- A minimum of 10Mbytes of RAM for the applications.

Installation sequence

HMS800C_v.3.1.003.exe from <http://abov.co.kr>.

2. Double-click the file.
3. After all process is completed, all files for compiler are copied in **C:\HMS800C** folder and all environment variables are defined automatically.

Uninstallation sequence

1. Click the **Start** button in the taskbar, and then click **Control Panel**.
2. Double-click the **Add/Remove Programs** icon in the **Control Panel** folder
3. There is **HMS800C v3.1 Compiler**, and select it.
4. Then, **Remove** button is enabled and click the button.

Running the compiler

Command Line Versions

Type the appropriate command at the MS-DOS prompt.

Ex) C:\hms800-cc -Os -option-file=hms87C1902.opt -o test.out test.c

For more information refer to the batch file of sample programs in C:\HMS800C\Samples folder.

GUI Versions

Click the **Start** button in the taskbar, and then click the **ABOV DevStudio** icon in **HMS800C V3.1 Compiler** folder.

C Compiler

HMS800C C compiler offers the standard features of the C language, plus a few extensions designed to take advantage of the HMS800 core specific features.

Language Facilities

- Conformance to the ANSI specification.
- Linkage of user code with assembly routines.
- External extension for hardware control.
- GCC-based compiler.

Performance

- Fast compilation
- Simple type checking at compile time.
- Code size checking at link time.

TUTORIAL

This chapter illustrates how you might use the HMS800C C compiler to develop a series of typical programs, and illustrates some of the C compiler's most important features.

Before reading this chapter you should:

- Already have installed HMS800C C compiler.
- Be familiar with the architecture and instruction set of the HMS800 processor.

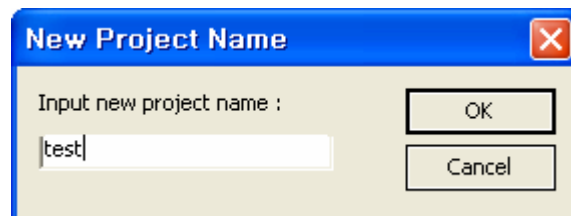
For more information see our data book which can be accessed from our website <http://abov.co.kr>.

Creating a new project

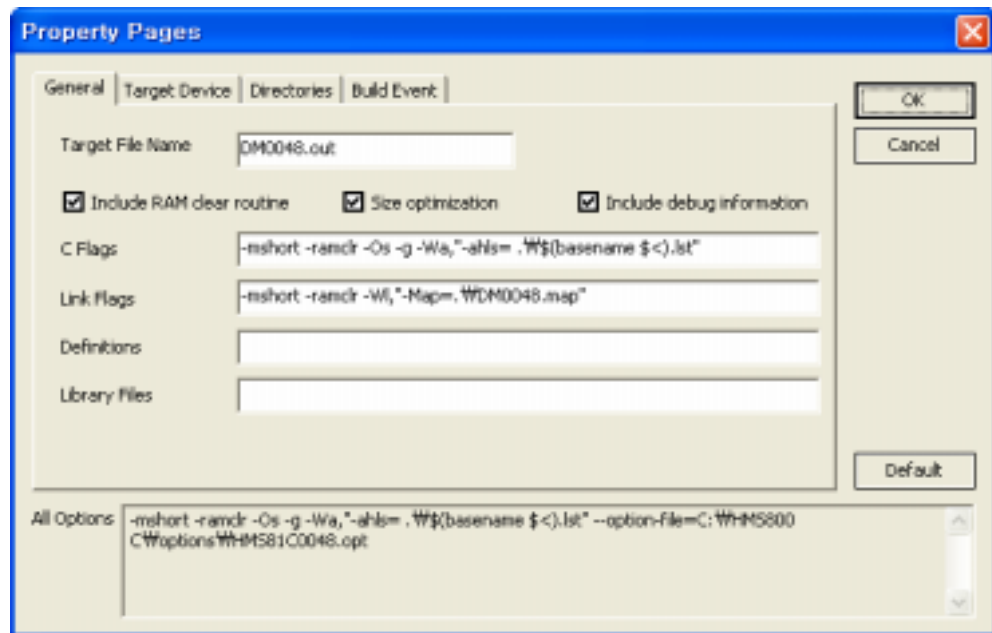
The first step is to create a new project for the sample programs.

First, run the MagnaChip DevStudio, and create a project for the tutorial as follows.

Choose **New** from the **Project** menu to display the following dialog box:



Type a project name and choose **OK** to display the **Property Pages** dialog box.

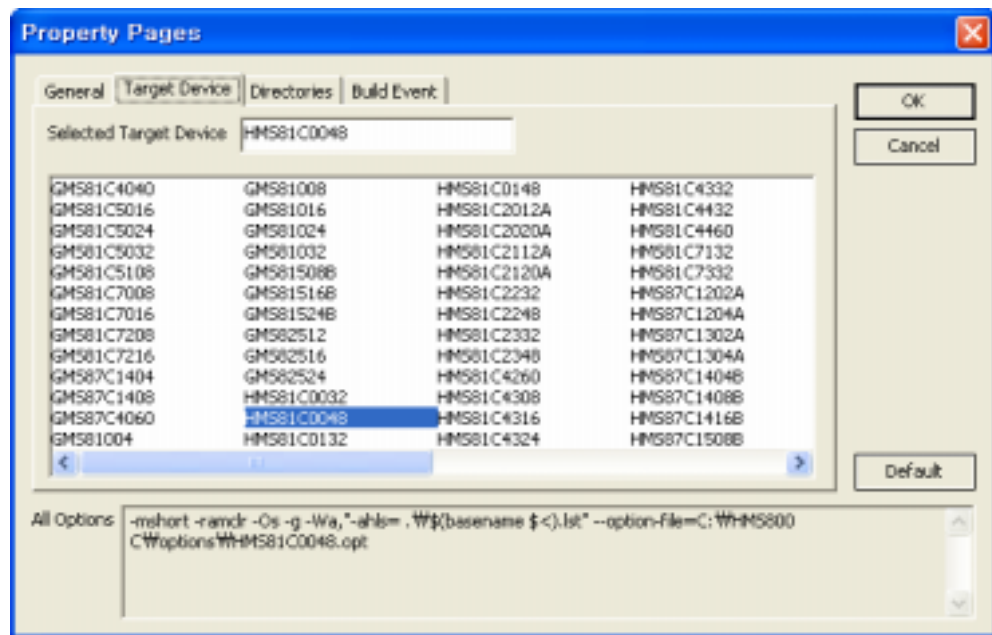


There is three tabs; **General**, **Target Device**, and **Directories**. In **General** tab, **Target File Name** means output file name which will be generated after compilation process. Default name is same with the project name which is specified in previous step. If you would like to use another name, you have to write a new name down.

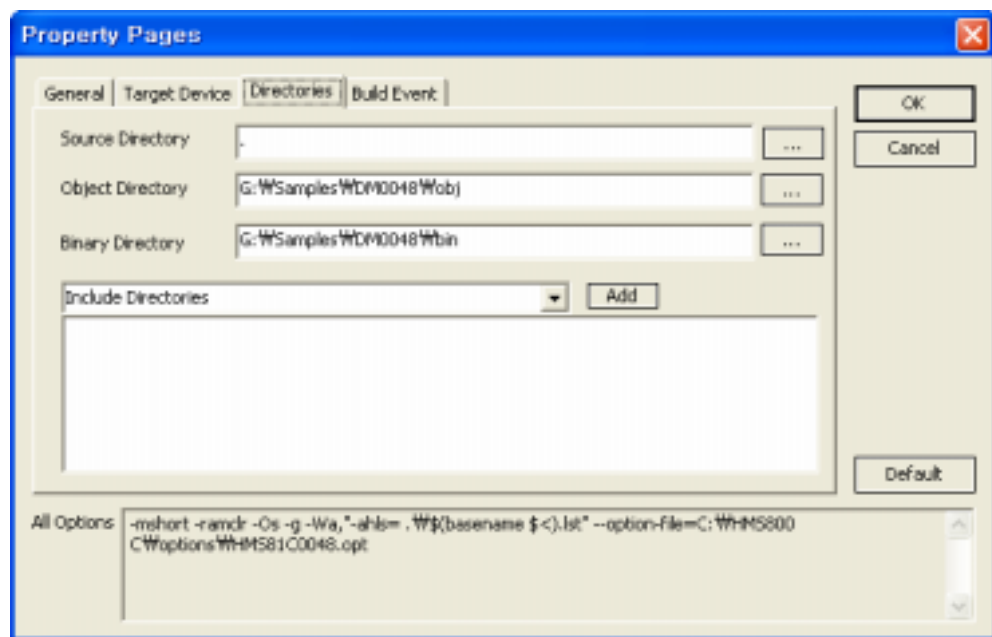
And there are three check boxes; **Include RAM clear routine**, **Size optimization**, and **Include debug information**. According to your need, they can be unchecked, but we strongly recommend all options are checked. The first option, **RAM clear routine**, means that whole RAM area is cleared as an initialization process before main routine is started. If this option is unchecked, RAM clear is responsible for programmer. If you don't have to clear RAM or want to clear some part of RAM, you uncheck this option and insert your routine in appropriate position.

Next option is **Size optimization**. In microcontroller program, one of the most important factors is code size because code size is closely related with production cost, and most microcontrollers have a constraint in memory size. If this option is check, various size optimization techniques are applied, and code size is decreased about 30%. Because all our benchmark programs are tested with this option, we recommend use this option. The last one is debug information. This has to be checked to debug appropriately.

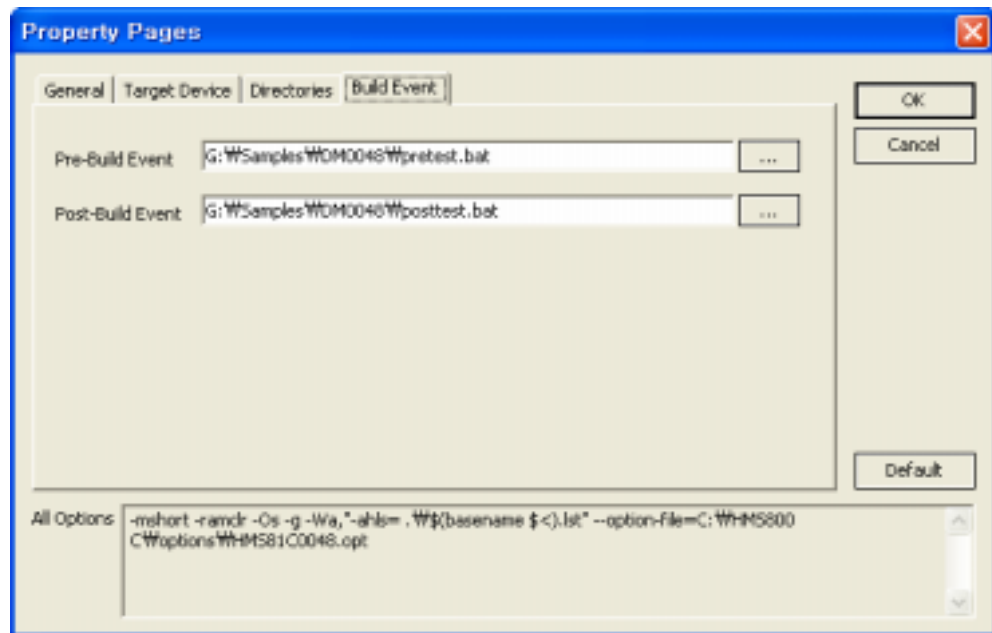
Others are options which can be transferred to compiler directly. If you are familiar to *gcc* (GNU Compiler Collection), you can modify the options according to your taste; otherwise, you had better be unchanged



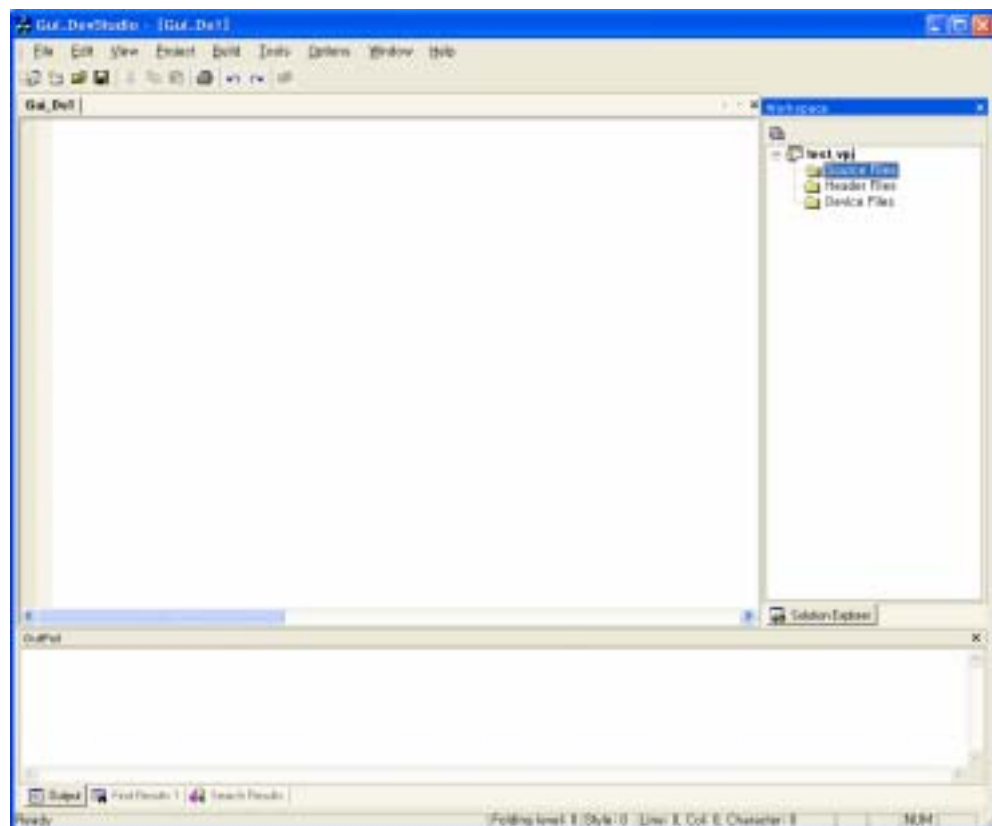
In the second tab of property pages, you can choose a target device.



The third tab provides directory settings. If you would like to use a specific directory for source, object, or binary, you can change the directory instead of current directory. And in the drop-down list box you can add new directories for header file and library. Then choose **OK** to create the new project.

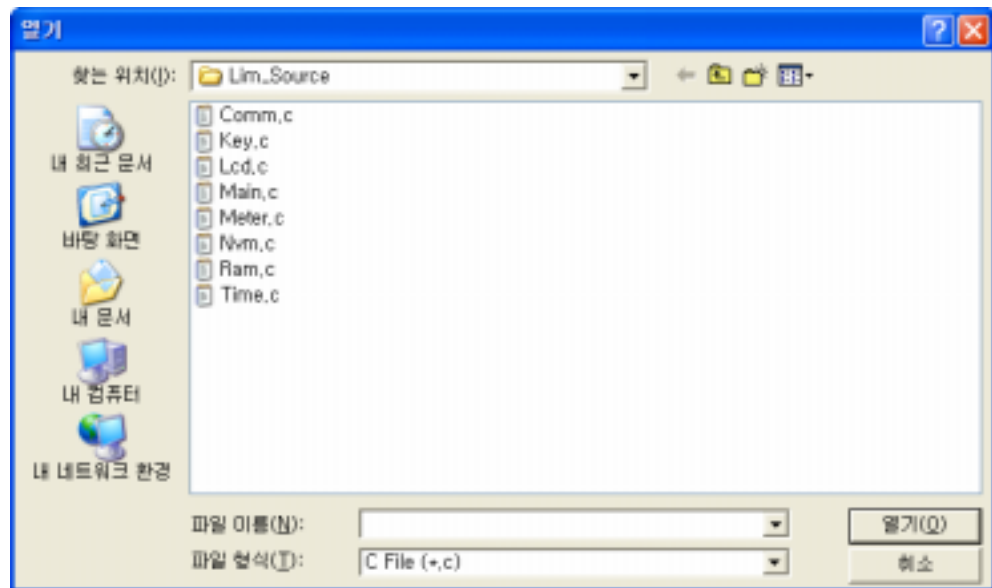


The fourth tab provides build event. When you want to execute an operation before/after compilation, the pre-build event/post-build event can be used.

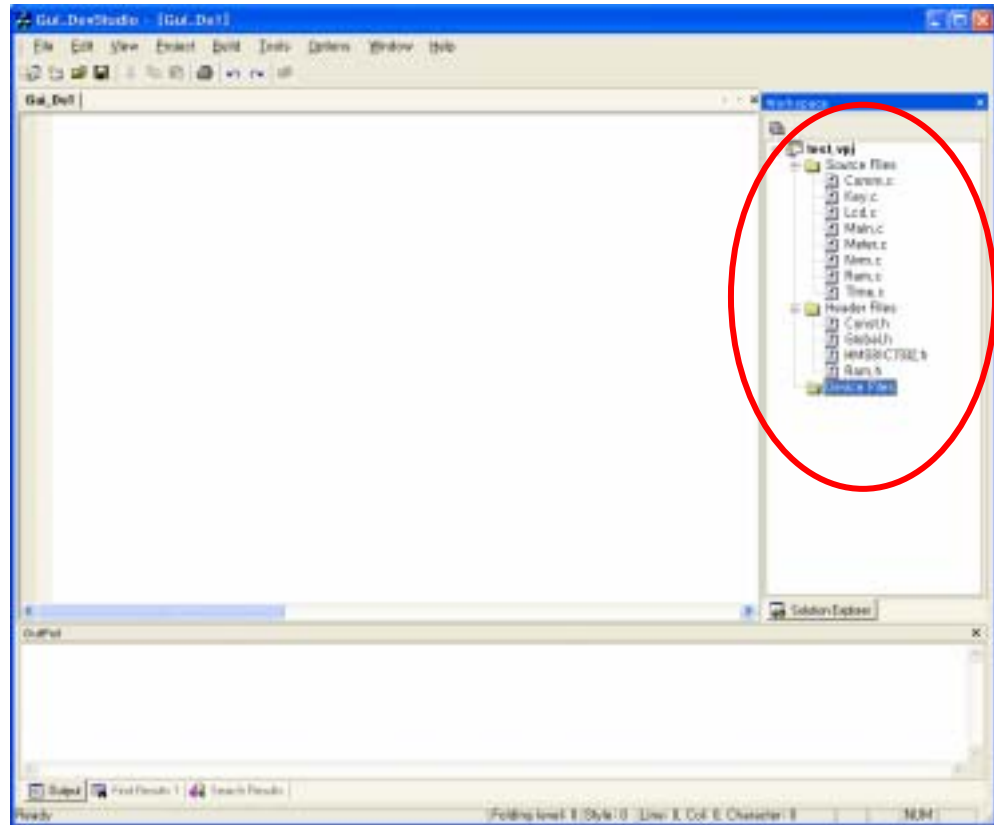


This figure shows initial display after making a project file. To compile your source files, you have to add C source files or header files.

1. Click the **Source Files** sub tree in the **Solution Explorer**.
2. Click the right button of the mouse, then the context menu is enabled, and you can see **Add Existing Item...** submenu.
3. Select the submenu, and then the file open dialog is activated.

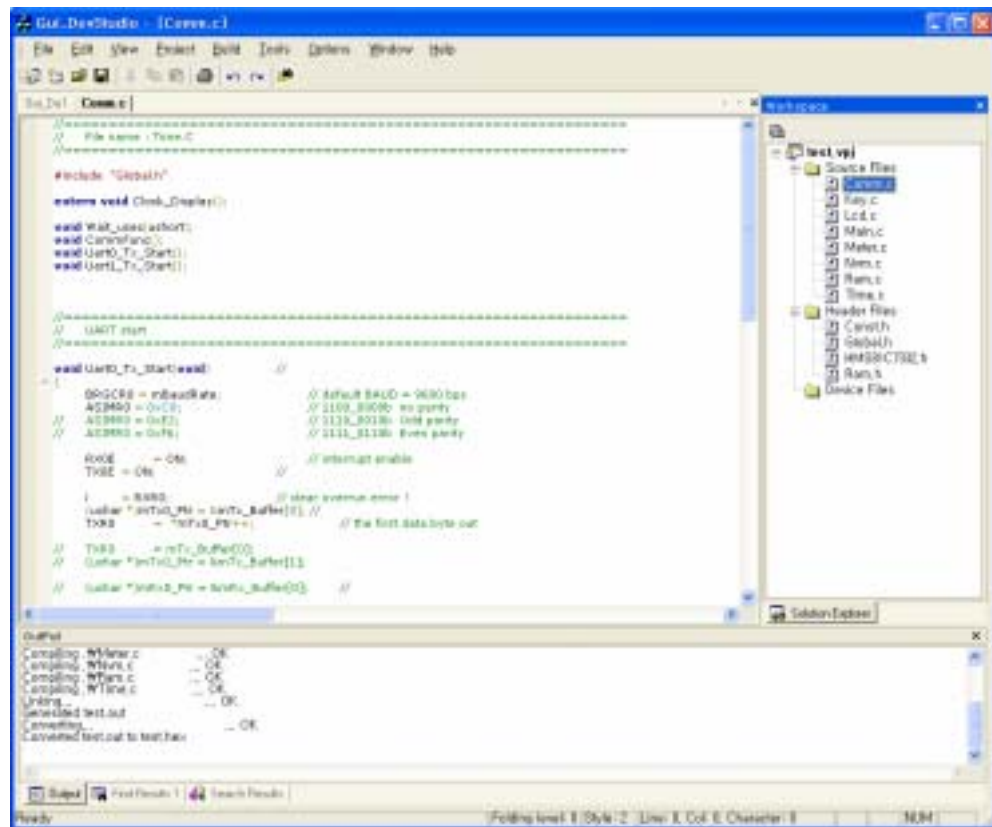


Select a file which you would like to add, Click **Open**.



You can see the added files correctly in the solution explorer.

Build a project



The Build menu will be described in this section.

1. **Build Project** or F7 key – C source files are compiled, linked, and translated to a hex file. To decrease compilation time, only modified files are compiled. Object files, binary file and hex file are generated in the directories which are specified in the directory tab of **Project Properties**. Default values are current directory which is same location with a directory having source files.
2. **Rebuild Project** or F8 key – This has a same function with selecting **Build Project** after choosing **Clean Project**.
3. **Clean Project** or F9 key – All intermediate files are removed; object files, map file, list files, out file, and hex file.
4. **Compile** or Ctrl+F7 key – Only selected file is compiled.

C Programming Guideline

This section describes some guidelines on HMS800 C programming. Some of the topics in this section are essential to write C programs on HMS800 series. The other topics provide insight into the operation of the HMS800 C Compiler and how more efficient programs can be written. This section discusses the following topics:

- Introduction to startup files
- Using header files
- Device-specific options and option files
- Variable and label naming conventions
- Attributes of functions and variables
- Optimizing features of the HMS800 C Compiler
- Access to control registers
- Using BIT and BYTE types
- Interrupt function definition
- Efficient function call
- Allocating variables user-specified memory
- Using pointers
- Restriction on use of RAM PAGE0

Introduction to startup files

Startup files are linked and executed before main function. These files are `crtmn.o`, `crtmmr.o` and `ram_clm.o`. An appropriate startup file is selected by the linker, `hms800-elf-lid.exe`. They are located in `C:\hms800C\libngcc-lib\hms800-elf\2.0.0`. In the startup files,

- Variables used by the compiler are declared.
- Interrupt vector table is defined.
- RAM clearing routines are defined.
- Some initialization operations are performed. Variables and registers used in the user program are initialized.

There are some variables used in the HMS800 C Compiler. These variables are not visible to users. Without these variables, the compiler fails to compile source codes. Because the number of compiler variables can be changed according to compilation, you can not estimate address of global variable. For example, if address of global variable `gVar` was allocated in `0x0012` in the first compilation, its value is not always

0x0012 in the next compilation. If you use address 0x0012 to reference the variable gVar, we cannot guarantee correct execution. Interrupt vector table is filled with the addresses of the user-provided interrupt-service routine which is designated by `__attribute__((interrupt))` keyword. When more than two functions are declared as interrupt routines, an error occurs and compilation is terminated. For some cases, a linker error will be reported with `_.INTn`. Elements of interrupt vector table not defined by users are filled with `_.NOT_USED` function. The body of the function just has `RETI`.

Using header files

The various machine specific features are defined in header files. The header files can be used by specifying a file in `#include <header_file_name>` or `#include "header_file_name"`. The bracket `<>` means that header file is located in system include directory (`C:\HMS800C\include`), and quotation mark `"` means that the file is located in the current working directory. For example, `hms800.h` can be included by using a preprocessor directive like `#include <hms800.h>`, if the file is located in system directory for header files. Also subdirectory name can be specified as in `#include <GMS810XX/GMS81032/GMS81032.h>` for device header files. The following source is contents of `hms800.h` which describes data structures for BIT and WORD, macros for memory pages and interrupts, and type definitions.

```
#ifndef _HMS800_H
#define _HMS800_H_

typedef unsigned char uchar;
typedef unsigned long ushort;

union byte_type_t
{
    uchar byte;
    struct p8_bit
    {
        char bit0:1    ; // 1
        char bit1:1    ; // 2
        char bit2:1    ; // 3
        char bit3:1    ; // 4
        char bit4:1    ; // 5
    }
};
```

```

        char bit5:1      ;// 6
        char bit6:1      ;// 7
        char bit7:1      ;// 8
    } bit;
};

```

```
typedef volatile union byte_type_t BYTE;
```

```

union word_type_t
{
    ushort word;
    struct p16_bit
    {
        char bit0:1      ;// 1
        char bit1:1      ;// 2
        char bit2:1      ;// 3
        char bit3:1      ;// 4
        char bit4:1      ;// 5
        char bit5:1      ;// 6
        char bit6:1      ;// 7
        char bit7:1      ;// 8
        char bit8:1      ;// 9
        char bit9:1      ;// 10
        char bit10:1     ;// 11
        char bit11:1     ;// 12
        char bit12:1     ;// 13
        char bit13:1     ;// 14
        char bit14:1     ;// 15
        char bit15:1     ;// 16
    } bit;
};

```

```
typedef volatile union word_type_t WORD;
```

```

#define      CODE __attribute__((section(".text")))
#define      PAGE0 __attribute__((section("PAGE0")))

```

```

#define PAGE1 __attribute__((section("PAGE1")))
#define PAGE2 __attribute__((section("PAGE2")))
#define PAGE3 __attribute__((section("PAGE3")))
#define PAGE4 __attribute__((section("PAGE4")))
#define PAGE5 __attribute__((section("PAGE5")))
#define PAGE6 __attribute__((section("PAGE6")))
#define PAGE7 __attribute__((section("PAGE7")))
#define PAGE8 __attribute__((section("PAGE8")))
#define PAGE9 __attribute__((section("PAGE9")))
#define PAGE10 __attribute__((section("PAGE10")))
#define PAGE11 __attribute__((section("PAGE11")))
#define PAGE12 __attribute__((section("PAGE12")))
#define PAGE13 __attribute__((section("PAGE13")))
#define PAGE14 __attribute__((section("PAGE14")))
#define PAGE15 __attribute__((section("PAGE15")))

#define INT0 __attribute__((interrupt ("0")))
#define INT1 __attribute__((interrupt ("1")))
#define INT2 __attribute__((interrupt ("2")))
#define INT3 __attribute__((interrupt ("3")))
#define INT4 __attribute__((interrupt ("4")))
#define INT5 __attribute__((interrupt ("5")))
#define INT6 __attribute__((interrupt ("6")))
#define INT7 __attribute__((interrupt ("7")))
#define INT8 __attribute__((interrupt ("8")))
#define INT9 __attribute__((interrupt ("9")))
#define INT10 __attribute__((interrupt ("10")))
#define INT11 __attribute__((interrupt ("11")))
#define INT12 __attribute__((interrupt ("12")))
#define INT13 __attribute__((interrupt ("13")))
#define INT14 __attribute__((interrupt ("14")))
#define INT15 __attribute__((interrupt ("15")))

#define HMS800_ENABLE_INTERRUPT asm("EI");
#define HMS800_DISABLE_INTERRUPT asm("DI");
#define HMS800_NOP asm("NOP");

```

```

#define HMS800_STOP                                asm("STOP");

/* These macros just can be used in interrupt service routines */
#define HMS800_PUSH_HI_REG                        asm("_PUSH_HI_REG"); /*
Push 16-bit registers */
#define HMS800_POP_HI_REG                        asm("_POP_HI_REG");
/* Pop 16-bit registers */

#endif

```

Addresses from 0xC0 to 0xFF of memory area are reserved for control registers. According to target device, addresses of control registers can be changed. Therefore all addresses of control registers are defined in header file of each device for convenient. The following source, GMS81032.h, describes target specific macros as an example of GMS81032 device; definitions of control register and bits. The **sfr** and **sbit** are described in more detail in the next section.

```

#ifndef _GMS81032_H_
#define _GMS81032_H_ 1

sfr    R0      = 0xC0 ; // [R/W] PORT R0 DATA REG.
sbit   R07     = R0^7 ;
sbit   R06     = R0^6 ;
sbit   R05     = R0^5 ;
sbit   R04     = R0^4 ;
sbit   R03     = R0^3 ;
sbit   R02     = R0^2 ;
sbit   R01     = R0^1 ;
sbit   R00     = R0^0 ;

sfr    R0DD    = 0xC1 ; // [W] PORT R0 DATA DIRECTION REG.

sfr    R1      = 0xC2 ; // [R/W] PORT R1 DATA REG.
sbit   R17     = R1^7 ;
sbit   R16     = R1^6 ;
sbit   R15     = R1^5 ;

```

```

sbit   R14    = R1^4  ;
sbit   R13    = R1^3  ;
sbit   R12    = R1^2  ;
sbit   R11    = R1^1  ;
sbit   R10    = R1^0  ;

sfr    R1DD   = 0xC3  ;    // [W] PORT R1 DATA DIRECTION REG.

sfr    R2     = 0xC4  ;    // [R/W] PORT R2 DATA REG.
sbit   R27    = R2^7  ;
sbit   R26    = R2^6  ;
sbit   R25    = R2^5  ;
sbit   R24    = R2^4  ;
sbit   R23    = R2^3  ;
sbit   R22    = R2^2  ;
sbit   R21    = R2^1  ;
sbit   R20    = R2^0  ;

sfr    R2DD   = 0xC5  ;    // [W] PORT R2 DATA DIRECTION REG.

sfr    CKCTRL = 0xC7  ;    // [W] CLOCK CONTROL REG.
sfr    BITR   = 0xC7  ;    // [R] BASIC INTERVAL TIMER REG.
sfr    WDTR   = 0xC8  ;    // [W] WATCH DOG TIMER REG.
sfr    PMR1   = 0xC9  ;    // [W] PORT R1 MODE REG.
sfr    IMOD   = 0xCA  ;    // [R/W] INT. MODE REG.
sfr    IEDS   = 0xCB  ;    // [W] EXT.INT.EDGE SELECTION

sfr    IENL   = 0xCC  ;    // [R/W] INT.ENABLE REG. LOW
sbit   WDTE   = IENL^6 ;    // WDT INT. ENABLE
sbit   BITE   = IENL^5 ;    // BIT INT. ENABLE

sfr    IRQL   = 0xCD  ;    // [R/W] INT.REQUEST FLAG REG. LOW
sbit   WDTIF  = IRQL^6 ;    // WDT INT. REQUEST
sbit   BITIF  = IRQL^5 ;    // BIT INT. REQUEST

sfr    IENH   = 0xCE  ;    // [R/W] INT.ENABLE REG.

```

```

sbit   KSCNE   = IENH^7 ; // KEY SCAN INT. ENABLE
sbit   INT1E   = IENH^6 ; // EXT.INT.1 ENABLE
sbit   INT2E   = IENH^5 ; // EXT.INT.2 ENABLE
sbit   T0E     = IENH^3 ; // TIMER0 INT. ENABLE
sbit   T1E     = IENH^2 ; // TIMER1 INT. ENABLE
sbit   T2E     = IENH^1 ; // TIMER2 INT. ENABLE

sfr    IRQH    = 0xCF   ; // [R/W] INT.REQUEST FLAG REG. HIGH
sbit   KSCNIF  = IRQH^7 ; // KEY SCAN INT. REQUEST
sbit   INT1IF  = IRQH^6 ; // EXT.INT.1 REQUEST
sbit   INT2IF  = IRQH^5 ; // EXT.INT.2 REQUEST
sbit   T0IF    = IRQH^3 ; // TIMER0 INT. REQUEST
sbit   T1IF    = IRQH^2 ; // TIMER1 INT. REQUEST
sbit   T2IF    = IRQH^1 ; // TIMER2 INT. REQUEST

sfr    TM0     = 0xD0   ; // [R/W] TIMER0 (16BIT) MODE REG.
sbit   CAP0    = TM0^7  ; // TIMER0/COUNTER OR INPUT CAPTURE
sbit   T0ST    = TM0^6  ; // TIMER0 START/STOP CONTROL
sbit   T0CN    = TM0^5  ; // TIMER0 COUNTER CONTINUATION/PAUSE
CONTROL
sbit   T0MOD   = TM0^4  ; // TIMER0 SINGLE/MODULO-N SELECTION
sbit   T0IFS   = TM0^3  ; // EVERY COUNTER OR EVERY 2ND COUNTER
sbit   T0SL2   = TM0^2  ; // TIMER0 INPUT CLOCK SELECTION2
sbit   T0SL1   = TM0^1  ; // TIMER0 INPUT CLOCK SELECTION1
sbit   T0SL0   = TM0^0  ; // TIMER0 INPUT CLOCK SELECTION0

sfr    TM1     = 0xD1   ; // [R/W] TIMER1 (8BIT) MODE REG.
sbit   T1ST    = TM1^7  ; // TIMER1 START/STOP CONTROL
sbit   T1CN    = TM1^6  ; // TIMER1 COUNTER CONTINUATION/PAUSE
CONTROL
sbit   T1MOD   = TM1^5  ; // TIMER1 SINGLE/MODULO-N SELECTION
sbit   T1IFS   = TM1^4  ; // EVERY COUNTER OR EVERY 2ND COUNTER
sbit   T1SL2   = TM1^2  ; // TIMER1 INPUT CLOCK SELECTION2
sbit   T1SL1   = TM1^1  ; // TIMER1 INPUT CLOCK SELECTION1
sbit   T1SL0   = TM1^0  ; // TIMER1 INPUT CLOCK SELECTION0

```

```

sfr    TM2      = 0xD2    ;    // [R/W] TIMER2 (8BIT) MODE REG.
sbit   T2ST     = TM2^4   ;    // TIMER2 START/STOP CONTROL
sbit   T2CN     = TM2^3   ;    // TIMER2 COUNTER CONTINUATION/PAUSE
CONTROL
sbit   T2SL2    = TM2^2   ;    // TIMER2 INPUT CLOCK SELECTION2
sbit   T2SL1    = TM2^1   ;    // TIMER2 INPUT CLOCK SELECTION1
sbit   T2SL0    = TM2^0   ;    // TIMER2 INPUT CLOCK SELECTION0

sfr    T0HMD    = 0xD3    ;    // [W] TIMER0 HIGH-MSB DATA REG.
sfr    T0HLD    = 0xD4    ;    // [W] TIMER0 HIGH-LSB DATA REG.
sfr    T0LMD    = 0xD5    ;    // [W] TIMER0 LOW-MSB DATA REG.
                                // [R] TIMER0 LOW-MSB COUNT REG.
sfr    T0LLD    = 0xD6    ;    // [W] TIMER0 LOW-LSB DATA REG.
                                // [R] TIMER0 LOW-LSB COUNT REG.
sfr    T1HD     = 0xD7    ;    // [W] TIMER1 HIGH DATA REG.
sfr    T1LD     = 0xD8    ;    // [W] TIMER1 LOW DATA REG.
                                // [R] TIMER1 LOW COUNT REG.
sfr    T2DR     = 0xD9    ;    // [W] TIMER2 DATA REG. [R] TIMER2 COUNT REG.

sfr    TM01     = 0xDA    ;    // [R/W] TIMER0/TIMER1 MODE REG.
sbit   TOUTS    = TM01^7   ;    // REMOUT PORT OUTPUT SELECTION
sbit   TOUTB    = TM01^6   ;    // REMOUT PORT BIT CONTROL
sbit   T0OUTP   = TM01^4   ;    // T0OUT POLARITY SELECTION
sbit   T0INIT   = TM01^3   ;    // TIMER0 OUTPUT INITIAL VALUE
sbit   T1INIT   = TM01^2   ;    // TIMER1 OUTPUT INITIAL VALUE
sbit   TOUT1    = TM01^1   ;    // TOUT LOGIC1
sbit   TOUT0    = TM01^0   ;    // TOUT LOGIC2

sfr    SMRR0    = 0xDC    ;    // [W] STANDBY MODE RELEASE REG0.
sfr    SMRR1    = 0xDD    ;    // [W] STANDBY MODE RELEASE REG1.
sfr    R1ODC    = 0xDE    ;    // [W] PORT R1 OPEN DRAIN ASSIGN REG.

```

```
#endif // _GMS81032_H_
```

Device-specific options and option files

In **C:\HMS800C\options** directory option files for devices are provided. To use an option file, you have two choices. The one is to specify full path name in ``--option-file=<name>'' as a parameter of **hms800-cc**, such as **--option-file=C:\HMS800C\options\GMS81032.opt** for GMS81032 device. The other is that you copy the option file from **C:\HMS800C\options** directory to current working directory and specify only option file name as a parameter, such as **--option-file=GMS81032.opt** for GMS81032 device.

```
section memory {
    /* code start address */
    STACK_LIMIT = 0x1FF;
    CODE_ROM_START = 0x8000;
    CODE_ROM_LENGTH = 0x7FDF;
    RAM_start = 0x00; /* data RAM start */
    pages = 2; /* number of pages */
    /*
     * start and end addresses of data pages
     * The order is not important.
     */
    page_0_start = 0x00;
    page_0_end = 0xBF;

    page_1_start = 0x100;
    page_1_end = 0x1FE;

    USE_GFLAG;
}
```

Variable and label naming conventions

The names of internal variables and labels start with ``_.'. Therefore, user source code must not use names starting with ``_.'. For example, if you define a global variable **ulCount** in C source, compiler uses **__ulCount** internally. If you use inline assembly and access global variables defined in C program, use **__ulCount** as a symbolic reference.

```

int ulCount;

int foo()
{
    ulCount = 1;

    asm("LDA #1");
    asm("STA _ulCount");
}

```

Attributes of functions and variables

By specifying the attributes of functions and variables, users can direct the compiler to generate smaller and faster code. For example, the attribute, ```noreturn```, can be used to inform the compiler that a function never returns. When a function ends up with **exit**, for example, it will never return. The compiler may produce better code when it knows that a function never returns. *An important use of function attributes is to declare a function as an interrupt routine.* When a function is declared as an interrupt function, the address of the function is registered in the interrupt vector table and **RETI** is used to return the caller.

A variable can be located in a specific memory location by using ```section(``section-name``)``` attribute. For example when there is more than one direct-paged memory, this attribute can designate the page in which a variable can be placed. In this case, the designated page must be declared as a separate section.

Optimizing features of the HMS800 C Compiler

The HMS800 C Compiler is an optimizing compiler. This means that the compiler takes certain steps to ensure the code that is generated and output to the object file is the most efficient (smaller and/or faster) code possible. The compiler analyzes the generated code to produce more efficient instruction sequences. This ensures that the user HMS800 program runs as quickly as possible.

- General optimizations
 - **Constant folding** Several constant values occurring in an expression or address calculation are combined as a constant.

- **Jump optimizing** Jumps are inverted or extended to the final target address when the program efficiency is thereby increased.
 - **Dead code elimination** Code which cannot be reached (dead code) is removed from the program.
 - **Global common sub-expression elimination** Identical sub-expressions or address calculations that occur multiple times in a function are recognized and calculated only once when possible.
- HMS800-specific optimizations
 - **Parameters passing via direct-paged memory** Function arguments are passed in direct-paged memory locations instead of stack when possible.
 - **Peephole optimization** Complex operations are replaced by simplified operations when memory space, execution time can be saved as a result, or bit manipulation operations.
 - **Extended access optimizing** Constants and variables are included directly in operations.
 - **Case/switch optimizing** Any switch and case statements are optimized by using a jump table of string of jumps.

Various optimizations may cause conflicting results. For example, an optimization for smaller code sizes may increase execution time or vice versa. Therefore, users must select suitable optimizations for their purpose.

Access to control registers

The control registers are used in C source program to control timers, counters, port I/O's and peripherals. The control registers reside from address 0xC0 to 0xFF and can be accessed as bits or bytes. Within HMS800 series, the number and type of the control registers vary. *Note that no control register name is predefined by the HMS800 C Compiler.* However, declaration for control registers are provided in header files in **C:\HMS800C\include**. HMS800 C Compiler provides the user with a number of header files for various HMS800 derivatives. Each header file contains the declarations for the control registers available on that derivative. There are three different methods to use control registers. In the following the methods are described.

Using *struct* and *union*

In this method to access a control register, the address of the control register must be specified. For the access to the bit-addressable control register, bit position must be specified. Instead of special data types, users can use bit-field declaration to represent the bits of the control registers. A variable is declared as a pointer to character (**char ***) and initialized to the address of a control register. The control register can be accessed through pointer dereferencing. The following code segment shows how control register can be accessed.

```
/*
 * Bit-field data type to represent bit position of a control register
 */
typedef struct bit_t {
    char bit_0:1;
    char bit_1:1;
    char bit_2:1;
    char bit_3:1;
    char bit_4:1;
    char bit_5:1;
    char bit_6:1;
    char bit_7:1;
} BIT;

typedef union cr_t {
    BIT A;
    char B;
} CR;          /* data structure for control register */

int main()
{
    CR R0;          /* R0 Port declaration */
    char *pR0;     /* pointer to R0 port */

    pR0 = (char *) 0xC0; /* initialize the pointer to the address of R0 */

    /* Store 0xAA in R0 */
```

```

R0.A.bit_0 = 0;          /* bit 0 of R0 is set to 0 */
R0.A.bit_1 = 1;          /* bit 1 of R0 is set to 1 */
R0.A.bit_2 = 0;
R0.A.bit_3 = 1;
R0.A.bit_4 = 0;
R0.A.bit_5 = 1;
R0.A.bit_6 = 0;
R0.A.bit_7 = 1;

*pR0 = R0.B;             /* put 0xAA in R0 to Port R0 */
}

```

Using constant pointers

There is another way to access control registers. As shown in the following example code, the address of a control register is used explicitly. This is possible because control registers are treated in the same way as other memory locations.

```

#define RA      ((unsigned char *) 0xC0) /* [R/W] RA Port Data Reg.      */
#define RAIO   ((unsigned char *) 0xC1) /* [W]   RA Port Direction Reg.*/
#define RB      ((unsigned char *) 0xC2) /* [R/W] RB Port Data Reg.      */
#define RBIO   ((unsigned char *) 0xC3) /* [W]   RB Port Direction Reg.*/
#define RC      ((unsigned char *) 0xC4) /* [R/W] RC Port Data Reg.      */
#define RCIO   ((unsigned char *) 0xC5) /* [W]   RC Port Direction Reg.*/
#define WDTR   ((unsigned char *) 0xED) /* [R/W] Watchdog Timer Register.*/
#define P1 *RA

*WDTR = 0xFF;           /* Watch dog timer refresh */
temporary_RC0 = *RC     /* Get the value of RC */
P1 = strobe_data[seg0]; /* Put the value of strobe_data[seg0] into RA */
temporary_RC0 = *RC & 0x01; /* get the bit 0 of RC */

```

Using *sfr* and *sbit* types

sfr and **sbit** types can be used to access control registers as shown in the following code. In **sfr** type declaration, = and **address** are used. **sbit** type variable can only be declared using previously declared **sfr** variables.

```

sfr    RA = 0xC0;    /* [R/W] RA Port Data Reg.    */
sbit   RA0 = RA^0;  /* bit 0 of RA                  */

sfr    RC = 0xC4;    /* [R/W] RC Port Data Reg.    */
sbit   RC0 = RC^0;  /* bit 0 of RC                  */

sfr    WDTR = 0xED;  /* [R/W] Watchdog Timer Register. */

WDTR = 0xFF;        /* Watch dog timer refresh */
temporary_RC0 = RC0; /* get the bit 0 of RC */

```

Using *BIT* and *BYTE* types

BIT and **BYTE** types defined in *hms800.h*. **BIT** type allows byte and bit accesses to 8-bit data. This type can be used to access byte access and bit access.

```

BYTE flag;
flag.byte = 10; /* byte access */
flag.bit.bit1 = 8; /* bit access. Flag = 8 */

```

To enable bit-access to 16-bit data **WORD** type can be used.

Interrupt function definition

Interrupt function can be defined as follows by using **__attribute__** keyword. **Interrupt** keyword has one string argument which value is between ``0" and ``15".

```

void __attribute__((interrupt ("3")))
int_timer(int i)
{
    /* body of function */
    ...
}

```

In the example above, **__attribute__** is used after type of a function at the function definition site. Interrupt function also can be declared within the prototype of a function as shown in the example below:

```
void int_timer(unsigned char) __attribute__((interrupt ("1")));
```

In *hms800.h*, macros, **INT0 ~ INT15** are defined. By using these macros the above examples can be modified as follows.

```
void INT3 int_timer(int i)
{
    /* body of function */
    ...
}

void int_timer(unsigned char) INT1;
```

Efficient function call

The HMS800 C Compiler can efficiently manipulate a function whose arguments are less than four. The excessive arguments are passed on the stack, which may cause larger code size. Therefore, it is desirable to reduce the number of arguments as small as possible.

Allocating variables user-specified memory

By using **__attribute__** keyword, variables can be declared in user-specified memory. The following code shows an example:

```
unsigned char cArray_src[] __attribute__((section (".text"))) =
    {1, 2, 3, 4, 5, 6, 7, 8, 9, 0xA};
```

The section name in the above example, **".text"**, can be replaced with appropriate section name where the variable is located. To place a variable in code memory, **".text"** should be used. The variable which is not modified during execution must be declared in code memory in order to reduce the amount of data memory usage.

For variables in **.text** section and RAM pages other than PAGE0, macros defined in **hms800.h** can be used. Variables in code memory (in **.text** section) can be declared using **CODE** macro. *unsigned table[] CODE = {0, 1, 2, 3};* will direct the compiler allocates *table* in code memory.

Without **PAGE1 ~ PAGE15** macros, a variable is allocated in PAGE0. With this macro a variable is allocated in specified page. *table* will be allocated in PAGE1 when *table* is defined in *unsigned table[] PAGE1 = {0, 1, 2, 3};*

Using pointers

In a general compiler, pointers are used to write efficient code in both code size and execution time. However, in HMS800 Compiler this is not always true since the size of address registers is not equal to that of pointers. Because the size of address register is 8 bits in HMS800 series and that of pointer is 16 bits, a memory access using pointers may cause indirect addressing using memory.

Consider the following example,

```
unsigned char cArray_src[] __attribute__((section(".text"))) =
{1, 2, 3, 4, 5, 6, 7, 8, 9, 0xA};
unsigned char cArray_dst[10];

int main()
{
    unsigned char *cPtr2, *cPtr1;
    unsigned char count;

    cPtr2 = cArray_dst;
    cPtr1 = cArray_src;

    for (count = 0; count < 10; count++)
    {
        *cPtr2++ = *cPtr1++;
    }
}
```

The above example moves 10 numbers from **cArray_src** to **cArray_dst**. Two pointers (**cPtr1** and **cPtr2**) are used to access source and destination arrays, respectively. Since the size of a pointer is 16 bits, 2-byte memory location is allocated for the pointer and address registers can not be used to store the pointer. This may cause the larger code size than the cases where address registers are used to contain address.

To avoid the overhead with pointers, the above code can be re-written as follows:

```
unsigned char cArray_src[] __attribute__((section(".text"))) =
{1, 2, 3, 4, 5, 6, 7, 8, 9, 0xA};
unsigned char cArray_dst[10];

int main()
{
    unsigned char count;

    for (count = 0; count < 10; count++)
    {
        cArray_dst[count] = cArray_src[count];
    }
}
```

In this code array name and index are used instead of pointers. The name of global array is treated as a label and can be used directly in the generated code. Therefore, the overhead of pointer is eliminated. For pointers to the variables allocated in PAGE0, can be accessed using pointers since the address of the variables in PAGE0 is 8 bits.

Restriction on use of RAM PAGE0

The first 16 bytes of PAGE0 are reserved for the use of the compiler and user codes must not access these reserved locations. For example, when RAM clearing routine is written in assembly code using *asm* keyword, the routine should not access the reserved bytes.

Tips to Reduce Code Size

- Use global variables whenever possible.

- Use no or one argument in functions.
- Compile with size optimization option (Default).
- Use **unsigned char** data type if applicable.
- Locate variables which are often used in PAGE 0.

HMS800C Compiler Command Options

When you invoke HMS800 C Compiler, it normally does preprocessing, compilation, assembly and linking. The "overall options" allow you to stop this process at an intermediate stage. For example, the `-c` option says not to run the linker. Then the output consists of object files output by the assembler.

Other options are passed on to one stage of processing. Some options control the preprocessor and others the compiler itself. Yet other options control the assembler and linker; most of these are not documented here, since you rarely need to use any of them.

The `hms800-cc` program accepts options and file names as operands. Many options have multi-letter names; therefore multiple single-letter options may not be grouped: `-dr` is very different from `-d -r`.

You can mix options and other arguments. For the most part, the order you use doesn't matter. Order does matter when you use several options of the same kind; for example, if you specify `-L`, which specifies a path for libraries more than once, the directories are searched in the order specified.

Many options have long names starting with `-f` or with `-W`, for example, `-fforce-mem`, `-fstrength-reduce`, `-Wformat` and so on. Most of these have both positive and negative forms; the negative form of `-ffoo` would be `-fno-foo`. This manual documents only one of these two forms, whichever one is not the default.

Option Summary

Here is a summary of all the options, grouped by type. Explanations are in the following sections.

- **Overall Options**

`-c -S -E -o FILE -pass-exit-codes -v --help`

- **C Language Options**

`-ansi -std=STANDARD -aux-info FILENAME`

-fno-asm -fno-builtin
-fhosted -ffreestanding
-trigraphs -traditional -traditional-cpp
-fallow-single-precision -fcond-mismatch
-fsigned-bitfields -fsigned-char
-funsigned-bitfields -funsigned-char
-fwritable-strings -fshort-wchar

- **Language Independent Options**

-fmessage-length=N
-fdiagnostics-show-location=[once|every-line]

- **Warning Options**

-fsyntax-only -pedantic -pedantic-errors
-w -W -Wall -Waggregate-return
-Wcast-align -Wcast-qual -Wchar-subscripts -Wcomment
-Wconversion -Wdisabled-optimization -Werror
-Wfloat-equal -Wformat -Wformat=2
-Wformat-nonliteral -Wformat-security
-Wid-clash-LEN -Wimplicit -Wimplicit-int
-Wimplicit-function-declaration
-Werror-implicit-function-declaration
-Wimport -Winline
-Wlarger-than-LEN -Wlong-long
-Wmain -Wmissing-braces -Wmissing-declarations
-Wmissing-format-attribute -Wmissing-noreturn
-Wmultichar -Wno-format-extra-args -Wno-format-y2k
-Wno-import -Wpacked -Wpadded
-Wparentheses -Wpointer-arith -Wredundant-decls
-Wreturn-type -Wsequence-point -Wshadow
-Wsign-compare -Wswitch -Wsystem-headers
-Wtrigraphs -Wundef -Wuninitialized
-Wunknown-pragmas -Wunreachable-code
-Wunused -Wunused-function -Wunused-label -Wunused-parameter
-Wunused-value -Wunused-variable -Wwrite-strings

- **C-only Warning Options**

-Wbad-function-cast -Wmissing-prototypes -Wnested-externs
-Wstrict-prototypes -Wtraditional

- **Debugging Options**

-a -ax -dLETTERS -dumpspecs -dumpmachine -dumpversion
-fdump-unnumbered -fdump-translation-unit[-N]
-fdump-class-hierarchy[-N]
-fdump-ast-original[-N] -fdump-ast-optimized[-N]
-fmem-report -fpretend-float
-fprofile-arcs -ftest-coverage -ftime-report
-g
-p -pg -print-file-name=LIBRARY -print-libgcc-file-name
-print-multi-directory -print-multi-lib
-print-prog-name=PROGRAM -print-search-dirs -Q
-save-temps -time

- **Optimization Options**

-falign-functions=N -falign-jumps=N
-falign-labels=N -falign-loops=N
-fbranch-probabilities -fcaller-saves
-fcse-follow-jumps -fcse-skip-blocks -fdata-sections -fdce
-fdelayed-branch -fdelete-null-pointer-checks
-fexpensive-optimizations -ffast-math -ffloat-store
-fforce-addr -fforce-mem -ffunction-sections -fgcse
-finline-functions -finline-limit=N -fkeep-inline-functions
-fkeep-static-consts -fmove-all-movables
-fno-default-inline -fno-defer-pop
-fno-function-cse -fno-guess-branch-probability
-fno-inline -fno-math-errno -fno-peephole -fno-peephole2
-fomit-frame-pointer -foptimize-register-move
-foptimize-sibling-calls -freduce-all-givs
-fregmove -frename-registers
-frerun-cse-after-loop -frerun-loop-opt
-fschedule-insns -fschedule-insns2
-fsingle-precision-constant -fssa

-fstrength-reduce
-funroll-all-loops -funroll-loops
--param NAME=VALUE
-O -O0 -O1 -O2 -O3 -Os

- **Preprocessor Options**

-\$ -AQUESTION=ANSWER -A-QUESTION[=ANSWER]
-C -dD -dl -dM -dN
-DMACRO[=DEFN] -E -H
-idirafter DIR
-include FILE -imacros FILE
-iprefix FILE -iwithprefix DIR
-iwithprefixbefore DIR -isystem DIR
-M -MM -MF -MG -MP -MQ -MT -nostdinc -P -remap
-trigraphs -undef -UMACRO -Wp,OPTION

- **Assembler Option**

-Wa,OPTION

- **Linker Options**

OBJECT-FILE-NAME -LIBRARY
-nostartfiles -nodefaultlibs -nostdlib
-s -static -static-libgcc -shared -shared-libgcc -symbolic
-Wl,OPTION -Xlinker OPTION
-u SYMBOL

- **Directory Options**

-BPREFIX -IDIR -I- -LDIR -specs=FILE

- **Code Generation Options**

-fcall-saved-REG -fcall-used-REG
-ffixed-REG -fexceptions
-fnon-call-exceptions -funwind-tables
-finhibit-size-directive -finstrument-functions
-fcheck-memory-usage -fprefix-function-name
-fno-common -fno-ident -fno-gnu-linker

-fpcc-struct-return -fpic -fPIC
-freg-struct-return -fshared-data -fshort-enums
-fshort-double -fvolatile
-fvolatile-global -fvolatile-static
-fverbose-asm -fpack-struct -fstack-check
-fstack-limit-register=REG -fstack-limit-symbol=SYM
-fargument-alias -fargument-noalias
-fargument-noalias-global -fleading-underscore

Options Controlling the Kind of Output

Compilation can involve up to four stages: preprocessing, compilation, assembly and linking, always in that order. The first three stages apply to an individual source file, and end by producing an object file; linking combines all the object files (those newly compiled, and those specified as input) into an executable file.

For any given input file, the file name suffix determines what kind of compilation is done:

- **FILE.c**

C source code which must be preprocessed.

- **FILE.i**

C source code which should not be preprocessed.

- **FILE.h**

C header file (not to be compiled or linked).

- **FILE.s**

Assembler code.

- **FILE.S**

Assembler code which must be preprocessed.

- **OTHER**

An object file to be fed straight into linking. Any file name with no recognized suffix is treated this way.

- **-pass-exit-codes**

Normally the ``hms800-cc'` program will exit with the code of 1 if any phase of the compiler returns a non-success return code. If you specify ``-pass-exit-codes'`, the ``hms800-cc'` program will instead return with numerically highest error produced by any phase that returned an error indication.

If you only want some of the stages of compilation, you can use ``-x'` (or filename suffixes) to tell ``hms800-cc'` where to start, and one of the options ``-c'`, ``-S'`, or ``-E'` to say where ``hms800-cc'` is to stop.

Note that some combinations (for example, ``-x cpp-output -E'`) instruct ``hms800-cc'` to do nothing at all.

- **-c**

Compile or assemble the source files, but do not link. The linking stage simply is not done. The ultimate output is in the form of an object file for each source file. By default, the object file name for a source file is made by replacing the suffix ``.c'`, ``.i'`, ``.s'`, etc., with ``.o'`. Unrecognized input files, not requiring compilation or assembly, are ignored.

- **-S**

Stop after the stage of compilation proper; do not assemble. The output is in the form of an assembler code file for each non-assembler input file specified. By default, the assembler file name for a source file is made by replacing the suffix ``.c'`, ``.i'`, etc., with ``.s'`.

Input files that don't require compilation are ignored.

- **-E**

Stop after the preprocessing stage; do not run the compiler proper. The output is in the form of preprocessed source code, which is sent to the standard output.

Input files which don't require preprocessing are ignored.

- **-o FILE**

Place output in file FILE. This applies regardless to whatever sort of output is being produced, whether it be an executable file, an object file, an assembler file or preprocessed C code.

Since only one output file can be specified, it does not make sense to use ``-o'` when compiling more than one input file, unless you are producing an executable file as output.

If `-o` is not specified, the default is to put an executable file in `a.out`, the object file for `SOURCE.SUFFIX` in `SOURCE.o`, its assembler file in `SOURCE.s`, and all preprocessed C source on standard output.

- **`-v`**

Print (on standard error output) the commands executed to run the stages of compilation. Also print the version number of the compiler driver program and of the preprocessor and the compiler proper.

- **`--help`**

Print (on the standard output) a description of the command line options understood by `hms800-cc`. If the `-v` option is also specified then `--help` will also be passed on to the various processes invoked by `hms800-cc`, so that they can display the command line options they accept. If the `-W` option is also specified then command line options which have no documentation associated with them will also be displayed.

Options Controlling C Dialect

The following options control the dialect of C that the compiler accepts:

- **`-ansi`**

In C mode, support all ISO C89 programs. This turns off certain features of HMS800 C Compiler that is incompatible with ISO C89 (when compiling C code) such as the `asm` and `typeof` keywords, and predefined macros such as `unix` and `vax` that identify the type of system you are using. It also enables the undesirable and rarely used ISO trigraph feature. For the C compiler, it disables recognition of C++ style `/**` comments as well as the `inline` keyword.

The alternate keywords `__asm__`, `__extension__`, `__inline__` and `__typeof__` continue to work despite `-ansi`. You would not want to use them in an ISO C program, of course, but it is useful to put them in header files that might be included in compilations done with `-ansi`. Alternate predefined macros, such as `__unix__` and `__vax__`, are also available with or without `-ansi`.

The `-ansi` option does not cause non-ISO programs to be rejected gratuitously. For that, `-pedantic` is required in addition to `-ansi`.

The macro `__STRICT_ANSI__` is predefined when the `-ansi` option is used. Some header files may notice this macro and refrain from declaring certain functions or

defining certain macros that the ISO standard doesn't call for; this is to avoid interfering with any programs that might use these names for other things.

Functions which would normally be built in but do not have semantics defined by ISO C (such as `'alloca'` and `'ffs'`) are not built-in functions with `'-ansi'` is used.

- **`-std=`**

Determine the language standard. This option is currently only supported when compiling C. A value for this option must be provided; possible values are

- `'c89'` `'iso9899:1990'`

ISO C89 (same as `'-ansi'`).

- `'iso9899:199409'`

ISO C89 as modified in amendment 1.

- `'c99'` `'c9x'` `'iso9899:1999'` `'iso9899:199x'`

ISO C99. Note that this standard is not yet fully supported. The names `'c9x'` and `'iso9899:199x'` are deprecated.

- `'gnu89'`

Default, ISO C89 plus GNU extensions (including some C99 features).

- `'gnu99'` `'gnu9x'`

ISO C99 plus GNU extensions. When ISO C99 is fully implemented in HMS800 C Compiler, this will become the default. The name `'gnu9x'` is deprecated.

Even when this option is not specified, you can still use some of the features of newer standards in so far as they do not conflict with previous C standards. For example, you may use `'__restrict__'` even when `'-std=c99'` is not specified.

The `'-std'` options specifying some version of ISO C have the same effects as `'-ansi'`, except that features that were not in ISO C89 but are in the specified version (for example, `'/'` comments and the `'inline'` keyword in ISO C99) are not disabled.

- **`'-aux-info FILENAME'`**

Output to the given filename prototyped declarations for all functions declared and/or defined in a translation unit, including those in header files. This option is silently ignored in any language other than C.

Besides declarations, the file indicates, in comments, the origin of each declaration (source file and line), whether the declaration was implicit, prototyped or unprototyped ('I', 'N' for new or 'O' for old, respectively, in the first character after the line number and the colon), and whether it came from a declaration or a definition ('C' or 'F', respectively, in the following character). In the case of function definitions, a K&R-style list of arguments followed by their declarations is also provided, inside comments, after the declaration.

- **'-fno-asm'**

Do not recognize 'asm', 'inline' or 'typeof' as a keyword, so that code can use these words as identifiers. You can use the keywords '__asm__', '__inline__' and '__typeof__' instead. '-ansi' implies '-fno-asm'.

- **'-fno-builtin'**

Don't recognize built-in functions that do not begin with '__builtin__' as prefix.

*Note Other built-in functions provided by HMS800 C Compiler: Other Builtins, for details of the functions affected, including those which are not built-in functions when '-ansi' or '-std' options for strict ISO C conformance are used because they do not have an ISO standard meaning.

HMS800 C Compiler normally generates special code to handle certain built-in functions more efficiently; for instance, calls to 'alloca' may become single instructions that adjust the stack directly, and calls to 'memcpy' may become inline copy loops. The resulting code is often both smaller and faster, but since the function calls no longer appear as such, you cannot set a breakpoint on those calls, nor can you change the behavior of the functions by linking with a different library.

- **'-fhosted'**

Assert that compilation takes place in a hosted environment. This implies '-fbuiltin'. A hosted environment is one in which the entire standard library is available, and in which 'main' has a return type of 'int'. Examples are nearly everything except a kernel. This is equivalent to '-fno-freestanding'.

- **'-ffreestanding'**

Assert that compilation takes place in a freestanding environment. This implies '-fno-builtin'. A freestanding environment is one in which the standard library may not exist, and program startup may not necessarily be at 'main'. The most obvious example is an OS kernel. This is equivalent to '-fno-hosted'.

- **`-trigraphs'**

Support ISO C trigraphs. The ``-ansi'` option (and ``-std'` options for strict ISO C conformance) implies ``-trigraphs'`.

- **`-traditional'**

Attempt to support some aspects of traditional C compilers. Specifically:

- All ``extern'` declarations take effect globally even if they are written inside of a function definition. This includes implicit declarations of functions.
- The newer keywords ``typeof'`, ``inline'`, ``signed'`, ``const'` and ``volatile'` are not recognized. (You can still use the alternative keywords such as ``__typeof__'`, ``__inline__'`, and so on.)
- Comparisons between pointers and integers are always allowed.
- Integer types ``unsigned short'` and ``unsigned char'` promote to ``unsigned int'`.
- Out-of-range floating point literals are not an error.
- Certain constructs which ISO regards as a single invalid preprocessing number, such as ``0xe-0xd'`, are treated as expressions instead.
- String "constants" are not necessarily constant; they are stored in writable space, and identical looking constants are allocated separately. (This is the same as the effect of ``-fwritable-strings'`.)
- All automatic variables not declared ``register'` are preserved by ``longjmp'`. Ordinarily, HMS800 C follows ISO C: automatic variables not declared ``volatile'` may be clobbered.
- The character escape sequences ``\x'` and ``\a'` evaluate as the literal characters ``x'` and ``a'` respectively. Without ``-traditional'`, ``\x'` is a prefix for the hexadecimal representation of a character, and ``\a'` produces a bell.

You may wish to use ``-fno-builtin'` as well as ``-traditional'` if your program uses names that are normally HMS800 C built-in functions for other purposes of its own.

You cannot use ``-traditional'` if you include any header files that rely on ISO C features. Some vendors are starting to ship systems with ISO C header files and you cannot use ``-traditional'` on such systems to compile files that include any system headers.

The ``-traditional'` option also enables ``-traditional-cpp'`, which is described next.

- **``-traditional-cpp'`**

Attempt to support some aspects of traditional C preprocessors. Specifically:

- Comments convert to nothing at all, rather than to a space. This allows traditional token concatenation.
- In a preprocessing directive, the ``#'` symbol must appear as the first character of a line.
- Macro arguments are recognized within string constants in a macro definition (and their values are stringified, though without additional quote marks, when they appear in such a context). The preprocessor always considers a string constant to end at a newline.
- The predefined macro ``__STDC__'` is not defined when you use ``-traditional'`, but ``__GNUC__'` is (since the GNU extensions which ``__GNUC__'` indicates are not affected by ``-traditional'`). If you need to write header files that work differently depending on whether ``-traditional'` is in use, by testing both of these predefined macros you can distinguish four situations: HMS800 C, traditional HMS800 C, other ISO C compilers, and other old C compilers. The predefined macro ``__STDC_VERSION__'` is also not defined when you use ``-traditional'`.
- The preprocessor considers a string constant to end at a newline (unless the newline is escaped with ``\'`). (Without ``-traditional'`, string constants can contain the newline character as typed.)

- **``-fcond-mismatch'`**

Allow conditional expressions with mismatched types in the second and third arguments. The value of such an expression is void. This option is not supported for C++.

- **``-funsigned-char'`**

Let the type ``char'` be unsigned, like ``unsigned char'`. Each kind of machine has a default for what ``char'` should be. It is either like ``unsigned char'` by default or like ``signed char'` by default.

Ideally, a portable program should always use ``signed char'` or ``unsigned char'` when it depends on the signedness of an object. But many programs have been written to use plain ``char'` and expect it to be signed, or expect it to be unsigned, depending on the machines they were written for. This option, and its inverse, let you make such a program work with the opposite default.

The type ``char'` is always a distinct type from each of ``signed char'` or ``unsigned char'`, even though its behavior is always just like one of those two.

- **``-fsigned-char'`**

Let the type ``char'` be signed, like ``signed char'`. Note that this is equivalent to ``-fno-unsigned-char'`, which is the negative form of ``-funsigned-char'`. Likewise, the option ``-fno-signed-char'` is equivalent to ``-funsigned-char'`.

- **``-fsigned-bitfields'`**

- **``-funsigned-bitfields'`**

- **``-fno-signed-bitfields'`**

- **``-fno-unsigned-bitfields'`**

These options control whether a bit-field is signed or unsigned, when the declaration does not use either ``signed'` or ``unsigned'`. By default, such a bit-field is signed, because this is consistent: the basic integer types such as ``int'` are signed types.

However, when ``-traditional'` is used, bit-fields are all unsigned no matter what.

- **``-fwritable-strings'`**

Store string constants in the writable data segment and don't uniquize them. This is for compatibility with old programs which assume they can write into string constants. The option ``-traditional'` also has this effect.

Writing into string constants is a very bad idea; "constants" should be constant.

- **`-fallow-single-precision`**

Do not promote single precision math operations to double precision, even when compiling with `-traditional`.

Traditional K&R C promotes all floating point operations to double precision, regardless of the sizes of the operands. On the architecture for which you are compiling, single precision may be faster than double precision. If you must use `-traditional`, but want to use single precision operations when the operands are single precision, use this option. This option has no effect when compiling with ISO or GNU C conventions (the default).

- **`-fshort-wchar`**

Override the underlying type for `wchar_t` to be `short unsigned int` instead of the default for the target. This option is useful for building programs to run under WINE.

Options to Request or Suppress Warnings

Warnings are diagnostic messages that report constructions which are not inherently erroneous but which are risky or suggest there may have been an error.

You can request many specific warnings with options beginning `-W`, for example `-Wimplicit` to request warnings on implicit declarations. Each of these specific warning options also has a negative form beginning `-Wno-` to turn off warnings; for example, `-Wno-implicit`. This manual lists only one of the two forms, whichever is not the default.

These options control the amount and kinds of warnings produced by HMS800 C Compiler:

- **`-fsyntax-only`**

Check the code for syntax errors, but don't do anything beyond that.

- **`-pedantic`**

Issue all the warnings demanded by strict ISO C; reject all programs that use forbidden extensions, and some other programs that do not follow ISO C and ISO C++. For ISO C, follows the version of the ISO C standard specified by any `-std` option used.

Valid ISO C programs should compile properly with or without this option (though a rare few will require `-ansi` or a `-std` option specifying the required version of ISO C). However, without this option, certain extensions and traditional C features are supported as well. With this option, they are rejected.

``-pedantic'` does not cause warning messages for use of the alternate keywords whose names begin and end with `__`. Pedantic warnings are also disabled in the expression that follows `__extension__`. However, only system header files should use these escape routes; application programs should avoid them.

Some users try to use ``-pedantic'` to check programs for strict ISO C conformance. They soon find that it does not do quite what they want: it finds some non-ISO practices, but not all--only those for which ISO C requires a diagnostic, and some others for which diagnostics have been added.

A feature to report any failure to conform to ISO C might be useful in some instances, but would require considerable additional work and would be quite different from ``-pedantic'`. We don't have plans to support such a feature in the near future.

Where the standard specified with ``-std'` represents a GNU extended dialect of C, such as ``gnu89'` or ``gnu99'`, there is a corresponding "base standard", the version of ISO C on which the GNU extended dialect is based. Warnings from ``-pedantic'` are given where they are required by the base standard. (It would not make sense for such warnings to be given only for features not in the specified GNU C dialect, since by definition the GNU dialects of C include all features the compiler supports with the given option, and there would be nothing to warn about.)

- **``-pedantic-errors'`**

Like ``-pedantic'`, except that errors are produced rather than warnings.

- **``-w'`**

Inhibit all warning messages.

- **``-Wno-import'`**

Inhibit warning messages about the use of ``#import'`.

- **``-Wchar-subscripts'`**

Warn if an array subscript has type ``char'`. This is a common cause of error, as programmers often forget that this type is signed on some machines.

- **``-Wcomment'`**

Warn whenever a comment-start sequence ``/*'` appears in a ``/*'` comment, or whenever a Backslash-Newline appears in a ``//'` comment.

- **`-Wformat`**

Check calls to `printf` and `scanf`, etc., to make sure that the arguments supplied have types appropriate to the format string specified, and that the conversions specified in the format string make sense.

`-Wformat` is included in `-Wall`. For more control over some aspects of format checking, the options `-Wno-format-y2k`, `-Wno-format-extra-args`, `-Wformat-nonliteral`, `-Wformat-security` and `-Wformat=2` are available, but are not included in `-Wall`.

- **`-Wno-format-y2k`**

If `-Wformat` is specified, do not warn about `strftime` formats which may yield only a two-digit year.

- **`-Wno-format-extra-args`**

If `-Wformat` is specified, do not warn about excess arguments to a `printf` or `scanf` format function. The C standard specifies that such arguments are ignored.

- **`-Wformat-nonliteral`**

If `-Wformat` is specified, also warn if the format string is not a string literal and so cannot be checked, unless the format function takes its format arguments as a `va_list`.

- **`-Wformat-security`**

If `-Wformat` is specified, also warn about uses of format functions that represent possible security problems. At present, this warns about calls to `printf` and `scanf` functions where the format string is not a string literal and there are no format arguments, as in `printf (foo);`. This may be a security hole if the format string came from untrusted input and contains `%n`. (This is currently a subset of what `-Wformat-nonliteral` warns about, but in future warnings may be added to `-Wformat-security` that is not included in `-Wformat-nonliteral`.)

- **`-Wformat=2`**

Enable `-Wformat` plus format checks not included in `-Wformat`. Currently equivalent to `-Wformat -Wformat-nonliteral -Wformat-security`.

- **`-Wimplicit-int`**

Warn when a declaration does not specify a type.

- **'-Wimplicit-function-declaration'**

- **'-Werror-implicit-function-declaration'**

Give a warning (or error) whenever a function is used before being declared.

- **'-Wimplicit'**

Same as '-Wimplicit-int' and '-Wimplicit-function-declaration'.

- **'-Wmain'**

Warn if the type of 'main' is suspicious. 'main' should be a function with external linkage, returning int, taking either zero arguments, two, or three arguments of appropriate types.

- **'-Wmissing-braces'**

Warn if an aggregate or union initializer is not fully bracketed. In the following example, the initializer for 'a' is not fully bracketed, but that for 'b' is fully bracketed.

```
int a[2][2] = { 0, 1, 2, 3 };
int b[2][2] = { { 0, 1 }, { 2, 3 } };
```

- **'-Wmultichar'**

Warn if a multicharacter constant ("FOOF") is used. Usually they indicate a typo in the user's code, as they have implementation-defined values, and should not be used in portable code.

- **'-Wparentheses'**

Warn if parentheses are omitted in certain contexts, such as when there is an assignment in a context where a truth value is expected, or when operators are nested whose precedence people often get confused about.

Also warn about constructions where there may be confusion to which 'if' statement an 'else' branch belongs. Here is an example of such a case:

```
{
    if (a)
        if (b)
            foo ();
    else
```

```

    bar ();
}

```

In C, every `else' branch belongs to the innermost possible `if' statement, which in this example is `if (b)'. This is often not what the programmer expected, as illustrated in the above example by indentation the programmer chose. When there is the potential for this confusion, HMS800 C Compiler will issue a warning when this flag is specified. To eliminate the warning, add explicit braces around the innermost `if' statement so there is no way the `else' could belong to the enclosing `if'. The resulting code would look like this:

```

{
  if (a)
  {
    if (b)
      foo ();
    else
      bar ();
  }
}

```

- **`-Wsequence-point'**

Warn about code that may have undefined semantics because of violations of sequence point rules in the C standard.

The C standard defines the order in which expressions in a C program are evaluated in terms of "sequence points", which represent a partial ordering between the execution of parts of the program: those executed before the sequence point, and those executed after it. These occur after the evaluation of a full expression (one which is not part of a larger expression), after the evaluation of the first operand of a `&&', `||', `?:' or `,' (comma) operator, before a function is called (but after the evaluation of its arguments and the expression denoting the called function), and in certain other places. Other than as expressed by the sequence point rules, the order of evaluation of subexpressions of an expression is not specified. All these rules describe only a partial order rather than a total order, since, for example, if two functions are called within one expression with no sequence point between them, the order in which the functions are called is not specified. However, the standards committee have ruled that function calls do not overlap.

It is not specified when between sequence points modifications to the values of objects take effect. Programs whose behavior depends on this have undefined behavior; the C standard specifies that "Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be read only to determine the value to be stored". If a program breaks these rules, the results on any particular implementation are entirely unpredictable.

Examples of code with undefined behavior are ``a = a++;``, ``a[n] = b[n++]`` and ``a[i++] = i;``. Some more complicated cases are not diagnosed by this option, and it may give an occasional false positive result, but in general it has been found fairly effective at detecting this sort of problem in programs.

The present implementation of this option only works for C programs. A future implementation may also work for C++ programs.

- **``-Wreturn-type``**

Warn whenever a function is defined with a return-type that defaults to ``int``. Also warn about any ``return`` statement with no return-value in a function whose return-type is not ``void``.

For C++, a function without return type always produces a diagnostic message, even when ``-Wno-return-type`` is specified. The only exceptions are ``main`` and functions defined in system headers.

- **``-Wswitch``**

Warn whenever a ``switch`` statement has an index of enumerable type and lacks a ``case`` for one or more of the named codes of that enumeration. (The presence of a ``default`` label prevents this warning.) ``case`` labels outside the enumeration range also provoke warnings when this option is used.

- **``-Wtrigraphs``**

Warn if any trigraphs are encountered that might change the meaning of the program (trigraphs within comments are not warned about).

- **``-Wunused-function``**

Warn whenever a static function is declared but not defined or a non-inline static function is unused.

- **``-Wunused-label'`**

Warn whenever a label is declared but not used. To suppress this warning use the ``unused'` attribute.

- **``-Wunused-parameter'`**

Warn whenever a function parameter is unused aside from its declaration. To suppress this warning use the ``unused'` attribute.

- **``-Wunused-variable'`**

Warn whenever a local variable or non-constant static variable is unused aside from its declaration. To suppress this warning use the ``unused'` attribute.

- **``-Wunused-value'`**

Warn whenever a statement computes a result that is explicitly not used. To suppress this warning cast the expression to ``void'`.

- **``-Wunused'`**

All all the above ``-Wunused'` options combined. In order to get a warning about an unused function parameter, you must either specify ``-W -Wunused'` or separately specify ``-Wunused-parameter'`.

- **``-Wuninitialized'`**

Warn if an automatic variable is used without first being initialized or if a variable may be clobbered by a ``setjmp'` call.

These warnings are possible only in optimizing compilation, because they require data flow information that is computed only when optimizing. If you don't specify ``-O'`, you simply won't get these warnings.

These warnings occur only for variables that are candidates for register allocation. Therefore, they do not occur for a variable that is declared ``volatile'`, or whose address is taken, or whose size is other than 1, 2, 4 or 8 bytes. Also, they do not occur for structures, unions or arrays, even when they are in registers.

Note that there may be no warning about a variable that is used only to compute a value that itself is never used, because such computations may be deleted by data flow analysis before the warnings are printed.

These warnings are made optional because HMS800 C Compiler is not smart enough to see all the reasons why the code might be correct despite appearing to have an error. Here is one example of how this can happen:

```
{
  int x;
  switch (y)
  {
    case 1: x = 1;
           break;
    case 2: x = 4;
           break;
    case 3: x = 5;
           }
  foo (x);
}
```

If the value of `y` is always 1, 2 or 3, then `x` is always initialized, but HMS800 C Compiler doesn't know this. Here is another common case:

```
{
  int save_y;
  if (change_y) save_y = y, y = new_y;
  ...
  if (change_y) y = save_y;
}
```

This has no bug because `save_y` is used only if it is set.

This option also warns when a non-volatile automatic variable might be changed by a call to `longjmp`. These warnings as well are possible only in optimizing compilation.

The compiler sees only the calls to `setjmp`. It cannot know where `longjmp` will be called; in fact, a signal handler could call it at any point in the code. As a result, you may get a warning even when there is in fact no problem because `longjmp` cannot in fact be called at the place which would cause a problem.

Some spurious warnings can be avoided if you declare all the functions you use that never return as `noreturn`.

- **`-Wunknown-pragmas'**

Warn when a #pragma directive is encountered which is not understood by HMS800 C Compiler. If this command line option is used, warnings will even be issued for unknown pragmas in system header files. This is not the case if the warnings were only enabled by the ``-Wall'` command line option.

- **`-Wall'**

All of the above ``-W'` options combined. This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros.

- **`-Wsystem-headers'**

Print warning messages for constructs found in system header files. Warnings from system headers are normally suppressed, on the assumption that they usually do not indicate real problems and would only make the compiler output harder to read. Using this command line option tells HMS800 C Compiler to emit warnings from system headers as if they occurred in user code. However, note that using ``-Wall'` in conjunction with this option will not warn about unknown pragmas in system headers--for that, ``-Wunknown-pragmas'` must also be used.

The following ``-W...'` options are not implied by ``-Wall'`. Some of them warn about constructions that users generally do not consider questionable, but which occasionally you might wish to check for; others warn about constructions that are necessary or hard to avoid in some cases, and there is no simple way to modify the code to suppress the warning.

- **`-W'**

Print extra warning messages for these events:

- A function can return either with or without a value. (Falling off the end of the function body is considered returning without a value.) For example, this function would evoke such a warning:

```
foo (a)
{
    i f (a > 0)
```

```
    return a;
}
```

- An expression-statement or the left-hand side of a comma expression contains no side effects. To suppress the warning, cast the unused expression to void. For example, an expression such as `x[i,j]` will cause a warning, but `x[(void)i,j]` will not.
- An unsigned value is compared against zero with `<` or `<=`.
- A comparison like `x<=y<=z` appears; this is equivalent to `(x<=y ? 1 : 0) <= z`, which is a different interpretation from that of ordinary mathematical notation.
- Storage-class specifiers like `static` are not the first things in a declaration. According to the C Standard, this usage is obsolescent.
- The return type of a function has a type qualifier such as `const`. Such a type qualifier has no effect, since the value returned by a function is not an lvalue. (But don't warn about the GNU extension of `volatile void` return types. That extension will be warned about if `-pedantic` is specified.)
- If `-Wall` or `-Wunused` is also specified, warn about unused arguments.
- A comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned. (But don't warn if `-Wno-sign-compare` is also specified.)
- An aggregate has a partly bracketed initializer. For example, the following code would evoke such a warning, because braces are missing around the initializer for `x.h`:

```
struct s { int f, g; };
struct t { struct s h; int i; };
struct t x = { 1, 2, 3 };
```

- An aggregate has an initializer which does not initialize all members. For example, the following code would cause such a warning, because `x.h` would be implicitly initialized to zero:

```
struct s { int f, g, h; };  
struct s x = { 3, 4 };
```

- **`-Wfloat-equal`**

Warn if floating point values are used in equality comparisons.

The idea behind this is that sometimes it is convenient (for the programmer) to consider floating-point values as approximations to infinitely precise real numbers. If you are doing this, then you need to compute (by analysing the code, or in some other way) the maximum or likely maximum error that the computation introduces, and allow for it when performing comparisons (and when producing output, but that's a different problem). In particular, instead of testing for equality, you would check to see whether the two values have ranges that overlap; and this is done with the relational operators, so equality comparisons are probably mistaken.

- **`-Wtraditional`**

Warn about certain constructs that behave differently in traditional and ISO C. Also warn about ISO C constructs that have no traditional C equivalent, and/or problematic constructs which should be avoided.

- Macro parameters that appear within string literals in the macro body. In traditional C macro replacement takes place within string literals, but does not in ISO C.
- In traditional C, some preprocessor directives did not exist. Traditional preprocessors would only consider a line to be a directive if the `\#` appeared in column 1 on the line. Therefore `-Wtraditional` warns about directives that traditional C understands but would ignore because the `\#` does not appear as the first character on the line. It also suggests you hide directives like `#pragma` not understood by traditional C by indenting them. Some traditional implementations would not recognize `#elif`, so it suggests avoiding it altogether.

- A function-like macro that appears without arguments.
- The unary plus operator.
- The `'U'` integer constant suffix, or the `'F'` or `'L'` floating point constant suffixes. (Traditional C does support the `'L'` suffix on integer constants.)
Note, these suffixes appear in macros defined in the system headers of most modern systems, e.g. the `'_MIN'/'_MAX'` macros in `'<limits.h>'`.
Use of these macros in user code might normally lead to spurious warnings; however gcc's integrated preprocessor has enough contexts to avoid warning in these cases.
- A function declared external in one block and then used after the end of the block.
- A `'switch'` statement has an operand of type `'long'`.
- A non-`'static'` function declaration follows a `'static'` one. This construct is not accepted by some traditional C compilers.
- The ISO type of an integer constant has a different width or signedness from its traditional type. This warning is only issued if the base of the constant is ten. I.e. hexadecimal or octal values, which typically represent bit patterns, are not warned about.
- Usage of ISO string concatenation is detected.
- Initialization of automatic aggregates.
- Identifier conflicts with labels. Traditional C lacks a separate namespace for labels.
- Initialization of unions. If the initializer is zero, the warning is omitted. This is done under the assumption that the zero initializer in user code appears conditioned on e.g. `'__STDC__'` to avoid missing initializer warnings and relies on default initialization to zero in the traditional C case.

- Conversions by prototypes between fixed/floating point values and vice versa. The absence of these prototypes when compiling with traditional C would cause serious problems. This is a subset of the possible conversion warnings, for the full set use ``-Wconversion'`.

- **``-Wundef'`**

Warn if an undefined identifier is evaluated in a ``#if'` directive.

- **``-Wshadow'`**

Warn whenever a local variable shadows another local variable, parameter or global variable or whenever a built-in function is shadowed.

- **``-Wid-clash-LEN'`**

Warn whenever two distinct identifiers match in the first LEN characters. This may help you prepare a program that will compile with certain obsolete, brain-damaged compilers.

- **``-Wlarger-than-LEN'`**

Warn whenever an object of larger than LEN bytes is defined.

- **``-Wpointer-arith'`**

Warn about anything that depends on the "size of" a function type or of ``void'`. GNU C assigns these types a size of 1, for convenience in calculations with ``void *'` pointers and pointers to functions.

- **``-Wbad-function-cast'`**

Warn whenever a function call is cast to a non-matching type. For example, warn if ``int malloc()'` is cast to ``anything *'`.

- **``-Wcast-qual'`**

Warn whenever a pointer is cast so as to remove a type qualifier from the target type. For example, warn if a ``const char *'` is cast to an ordinary ``char *'`.

- **``-Wcast-align'`**

Warn whenever a pointer is cast such that the required alignment of the target is increased. For example, warn if a `char *` is cast to an `int *` on machines where integers can only be accessed at two- or four-byte boundaries.

- **`-Wwrite-strings`**

When compiling C, give string constants the type `const char[LENGTH]` so that copying the address of one into a non-`const char *` pointer will get a warning; when compiling C++, warn about the deprecated conversion from string constants to `char *`. These warnings will help you find at compile time code that can try to write into a string constant, but only if you have been very careful about using `const` in declarations and prototypes. Otherwise, it will just be a nuisance; this is why we did not make `-Wall` request these warnings.

- **`-Wconversion`**

Warn if a prototype causes a type conversion that is different from what would happen to the same argument in the absence of a prototype. This includes conversions of fixed point to floating and vice versa, and conversions changing the width or signedness of a fixed point argument except when the same as the default promotion.

Also, warn if a negative integer constant expression is implicitly converted to an unsigned type. For example, warn about the assignment `x = -1` if `x` is unsigned. But do not warn about explicit casts like `(unsigned) -1`.

- **`-Wsign-compare`**

Warn when a comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned. This warning is also enabled by `-W`; to get the other warnings of `-W` without this warning, use `-W -Wno-sign-compare`.

- **`-Waggregate-return`**

Warn if any functions that return structures or unions are defined or called. (In languages where you can return an array, this also elicits a warning.)

- **`-Wstrict-prototypes`**

Warn if a function is declared or defined without specifying the argument types. (An old-style function definition is permitted without a warning if preceded by a declaration which specifies the argument types.)

- **'-Wmissing-prototypes'**

Warn if a global function is defined without a previous prototype declaration. This warning is issued even if the definition itself provides a prototype. The aim is to detect global functions that fail to be declared in header files.

- **'-Wmissing-declarations'**

Warn if a global function is defined without a previous declaration. Do so even if the definition itself provides a prototype. Use this option to detect global functions that are not declared in header files.

- **'-Wmissing-noreturn'**

Warn about functions which might be candidates for attribute `'noreturn'`. Note these are only possible candidates, not absolute ones. Care should be taken to manually verify functions actually do not ever return before adding the `'noreturn'` attribute, otherwise subtle code generation bugs could be introduced. You will not get a warning for `'main'` in hosted C environments.

- **'-Wmissing-format-attribute'**

If `'-Wformat'` is enabled, also warn about functions which might be candidates for `'format'` attributes. Note these are only possible candidates, not absolute ones. HMS800 C Compiler will guess that `'format'` attributes might be appropriate for any function that calls a function like `'vprintf'` or `'vscanf'`, but this might not always be the case, and some functions for which `'format'` attributes are appropriate may not be detected. This option has no effect unless `'-Wformat'` is enabled (possibly by `'-Wall'`).

- **'-Wpacked'**

Warn if a structure is given the packed attribute, but the packed attribute has no effect on the layout or size of the structure. Such structures may be mis-aligned for little benefit. For instance, in this code, the variable `'f.x'` in `'struct bar'` will be misaligned even though `'struct bar'` does not itself have the packed attribute:

```
struct foo {  
    int x;  
    char a, b, c, d;  
} __attribute__((packed));
```

```
struct bar {
    char z;
    struct foo f;
};
```

- **'-Wpadded'**

Warn if padding is included in a structure, either to align an element of the structure or to align the whole structure. Sometimes when this happens it is possible to rearrange the fields of the structure to reduce the padding and so make the structure smaller.

- **'-Wredundant-decls'**

Warn if anything is declared more than once in the same scope, even in cases where multiple declarations is valid and changes nothing.

- **'-Wnested-externs'**

Warn if an 'extern' declaration is encountered within a function.

- **'-Wunreachable-code'**

Warn if the compiler detects that code will never be executed.

This option is intended to warn when the compiler detects that at least a whole line of source code will never be executed, because some condition is never satisfied or because it is after a procedure that never returns.

It is possible for this option to produce a warning even though there are circumstances under which part of the affected line can be executed, so care should be taken when removing apparently-unreachable code.

For instance, when a function is inlined, a warning may mean that the line is unreachable in only one inlined copy of the function.

This option is not made part of '-Wall' because in a debugging version of a program there is often substantial code which checks correct functioning of the program and is, hopefully, unreachable because the program does work. Another common use of unreachable code is to provide behaviour which is selectable at compile-time.

- **'-Winline'**

Warn if a function can not be inlined and it was declared as inline.

- **'-Wlong-long'**

Warn if `'long long'` type is used. This is default. To inhibit the warning messages, use `'-Wno-long-long'`. Flags `'-Wlong-long'` and `'-Wno-long-long'` are taken into account only when `'-pedantic'` flag is used.

- **`'-Wdisabled-optimization'`**

Warn if a requested optimization pass is disabled. This warning does not generally indicate that there is anything wrong with your code; it merely indicates that HMS800 C Compiler's optimizers were unable to handle the code effectively. Often, the problem is that your code is too big or too complex; HMS800 C Compiler will refuse to optimize programs when the optimization itself is likely to take inordinate amounts of time.

- **`'-Werror'`**

Make all warnings into errors.

Options for Debugging Your Program

HMS800 C Compiler has various special options that are used for debugging your program:

- **`'-g'`**

Produce debugging information in the operating system's native format (DWARF). GDB can work with this debugging information.

Unlike most other C compilers, HMS800 C Compiler allows you to use `'-g'` with `'-O'`. The shortcuts taken by optimized code may occasionally produce surprising results: some variables you declared may not exist at all; flow of control may briefly move where you did not expect it; some statements may not be executed because they compute constant results or their values were already at hand; some statements may execute in different places because they were moved out of loops.

Nevertheless it proves possible to debug optimized output. This makes it reasonable to use the optimizer for programs that might have bugs.

The following options are useful when HMS800 C Compiler is generated with the capability for more than one debugging format.

Options That Control Optimization

These options control various sorts of optimizations:

- ``-O'`
- ``-O1'`

Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.

Without ``-O'`, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code.

Without ``-O'`, the compiler only allocates variables declared ``register'` in registers. The resulting compiled code is a little worse than produced by PCC without ``-O'`.

With ``-O'`, the compiler tries to reduce code size and execution time.

When you specify ``-O'`, the compiler turns on ``-fthread-jumps'` and ``-fdefer-pop'` on all machines. The compiler turns on ``-fdelayed-branch'` on machines that have delay slots, and ``-fomit-frame-pointer'` on machines that can support debugging even without a frame pointer. On some machines the compiler also turns on other flags.

- ``-O2'`

Optimize even more. HMS800 C Compiler performs nearly all supported optimizations that do not involve a space-speed tradeoff. The compiler does not perform loop unrolling or function inlining when you specify ``-O2'`. As compared to ``-O'`, this option increases both compilation time and the performance of the generated code.

``-O2'` turns on all optional optimizations except for loop unrolling, function inlining, and register renaming. It also turns on the ``-fforce-mem'` option on all machines and frame pointer elimination on machines where doing so does not interfere with debugging.

Please note the warning under ``-fgcse'` about invoking ``-O2'` on programs that use computed gotos.

- ``-O3'`

Optimize yet more. ``-O3'` turns on all optimizations specified by ``-O2'` and also turns on the ``-finline-functions'` and ``-frename-registers'` options.

- ``-O0'`

Do not optimize.

- ``-Os'`

Optimize for size. `-Os` enables all `-O2` optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size.

If you use multiple `-O` options, with or without level numbers, the last such option is the one that is effective.

Options of the form `-fFLAG` specify machine-independent flags. Most flags have both positive and negative forms; the negative form of `-ffoo` would be `-fno-foo`. In the table below, only one of the forms is listed--the one which is not the default. You can figure out the other form by either removing `no-` or adding it.

- **`-ffloat-store`**

Do not store floating point variables in registers, and inhibit other options that might change whether a floating point value is taken from a register or memory.

- **`-fno-defer-pop`**

Always pop the arguments to each function call as soon as that function returns. For machines which must pop arguments after a function call, the compiler normally lets arguments accumulate on the stack for several function calls and pops them all at once.

- **`-fforce-mem`**

Force memory operands to be copied into registers before doing arithmetic on them. This produces better code by making all memory references potential common subexpressions. When they are not common subexpressions, instruction combination should eliminate the separate register-load. The `-O2` option turns on this option.

- **`-fforce-addr`**

Force memory address constants to be copied into registers before doing arithmetic on them. This may produce better code just as `-fforce-mem` may.

- **`-fomit-frame-pointer`**

Don't keep the frame pointer in a register for functions that don't need one. This avoids the instructions to save, set up and restore frame pointers; it also makes an extra register available in many functions.

- **`-foptimize-sibling-calls`**

Optimize sibling and tail recursive calls.

- **``-fno-inline'`**

Don't pay attention to the ``inline'` keyword. Normally this option is used to keep the compiler from expanding any functions inline. Note that if you are not optimizing, no functions can be expanded inline.

- **``-finline-functions'`**

Integrate all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way.

If all calls to a given function are integrated, and the function is declared ``static'`, then the function is normally not output as assembler code in its own right.

- **``-finline-limit=N'`**

By default, HMS800 C Compiler limits the size of functions that can be inlined. This flag allows the control of this limit for functions that are explicitly marked as inline. N is the size of functions that can be inlined in number of pseudo instructions (not counting parameter handling). The default value of N is 600. Increasing this value can result in more inlined code at the cost of compilation time and memory consumption. Decreasing usually makes the compilation faster and less code will be inlined (which presumably means slower programs).

- **``-fkeep-inline-functions'`**

Even if all calls to a given function are integrated, and the function is declared ``static'`, nevertheless output a separate run-time callable version of the function. This switch does not affect ``extern inline'` functions.

- **``-fkeep-static-consts'`**

Emit variables declared ``static const'` when optimization isn't turned on, even if the variables aren't referenced.

HMS800 C Compiler enables this option by default. If you want to force the compiler to check if the variable was referenced, regardless of whether or not optimization is turned on, use the ``-fno-keep-static-consts'` option.

- **``-fno-function-cse'`**

Do not put function addresses in registers; make each instruction that calls a constant function contain the function's address explicitly.

This option results in less efficient code, but some strange hacks that alter the assembler output may be confused by the optimizations performed when this option is not used.

- **``-ffast-math'`**

This option allows HMS800 C Compiler to violate some ISO or IEEE rules and/or specifications in the interest of optimizing code for speed. For example, it allows the compiler to assume arguments to the ``sqrt'` function are non-negative numbers and that no floating-point values are NaNs.

This option causes the preprocessor macro ``__FAST_MATH__'` to be defined.

This option should never be turned on by any ``-O'` option since it can result in incorrect output for programs which depend on an exact implementation of IEEE or ISO rules/specifications for math functions.

- **``-fno-math-errno'`**

Do not set ERRNO after calling math functions that are executed with a single instruction, e.g., `sqrt`. A program that relies on IEEE exceptions for math error handling may want to use this flag for speed while maintaining IEEE arithmetic compatibility.

The default is ``-fmath-errno'`. The ``-ffast-math'` option sets ``-fno-math-errno'`.

The following options control specific optimizations. The ``-O2'` option turns on all of these optimizations except ``-funroll-loops'` and ``-funroll-all-loops'`. On most machines, the ``-O'` option turns on the ``-fthread-jumps'` and ``-fdelayed-branch'` options, but specific machines may handle it differently.

You can use the following flags in the rare cases when "fine-tuning" of optimizations to be performed is desired.

- **``-fstrength-reduce'`**

Perform the optimizations of loop strength reduction and elimination of iteration variables.

- **``-fthread-jumps'`**

Perform optimizations where we check to see if a jump branches to a location where another comparison subsumed by the first is found. If so, the first branch is redirected to either the destination of the second branch or a point immediately following it, depending on whether the condition is known to be true or false.

- **``-fcse-follow-jumps'`**

In common subexpression elimination, scan through jump instructions when the target of the jump is not reached by any other path. For example, when CSE encounters an ``if'` statement with an ``else'` clause, CSE will follow the jump when the condition tested is false.

- **``-fcse-skip-blocks'`**

This is similar to ``-fcse-follow-jumps'`, but causes CSE to follow jumps which conditionally skip over blocks. When CSE encounters a simple ``if'` statement with no else clause, ``-fcse-skip-blocks'` causes CSE to follow the jump around the body of the ``if'`.

- **``-frerun-cse-after-loop'`**

Re-run common subexpression elimination after loop optimizations has been performed.

- **``-frerun-loop-opt'`**

Run the loop optimizer twice.

- **``-fgcse'`**

Perform a global common subexpression elimination pass. This pass also performs global constant and copy propagation.

- **``-fdelete-null-pointer-checks'`**

Use global dataflow analysis to identify and eliminate useless null pointer checks. Programs which rely on NULL pointer dereferences not halting the program may not work properly with this option. Use ``-fno-delete-null-pointer-checks'` to disable this optimizing for programs which depend on that behavior.

- **``-fexpensive-optimizations'`**

Perform a number of minor optimizations that are relatively expensive.

- **``-foptimize-register-move'`**

- **``-fregmove'`**

Attempt to reassign register numbers in move instructions and as operands of other simple instructions in order to maximize the amount of register tying. This is especially

helpful on machines with two-operand instructions. HMS800 C Compiler enables this optimization by default with `-O2` or higher.

Note `-fregmove` and `-foptimize-register-move` are the same optimization.

- **`-fschedule-insns`**

If supported for the target machine, attempt to reorder instructions to eliminate execution stalls due to required data being unavailable. This helps machines that have slow floating point or memory load instructions by allowing other instructions to be issued until the result of the load or floating point instruction is required.

- **`-fschedule-insns2`**

Similar to `-fschedule-insns`, but requests an additional pass of instruction scheduling after register allocation has been done. This is especially useful on machines with a relatively small number of registers and where memory load instructions take more than one cycle.

- **`-fcaller-saves`**

Enable values to be allocated in registers that will be clobbered by function calls, by emitting extra instructions to save and restore the registers around such calls. Such allocation is done only when it seems to result in better code than would otherwise be produced. This option is always enabled by default on certain machines, usually those which have no call-preserved registers to use instead.

For all machines, optimization level 2 and higher enables this flag by default.

- **`-funroll-loops`**

Perform the optimization of loop unrolling. This is only done for loops whose number of iterations can be determined at compile time or run time. `-funroll-loops` implies both `-fstrength-reduce` and `-frerun-cse-after-loop`.

- **`-funroll-all-loops`**

Perform the optimization of loop unrolling. This is done for all loops and usually makes programs run more slowly. `-funroll-all-loops` implies `-fstrength-reduce` as well as `-frerun-cse-after-loop`.

- **`-fmove-all-movables`**

Forces all invariant computations in loops to be moved outside the loop.

- **``-freduce-all-givs'`**

Forces all general-induction variables in loops to be strength-reduced.

- **``-fno-peephole'`**

- **``-fno-peephole2'`**

Disable any machine-specific peephole optimizations. The difference between ``-fno-peephole'` and ``-fno-peephole2'` is in how they are implemented in the compiler; some targets use one, some use the other, a few use both.

- **``-falign-functions'`**

- **``-falign-functions=N'`**

Align the start of functions to the next power-of-two greater than N, skipping up to N bytes. For instance, ``-falign-functions=32'` aligns functions to the next 32-byte boundary, but ``-falign-functions=24'` would align to the next 32-byte boundary only if this can be done by skipping 23 bytes or less.

``-fno-align-functions'` and ``-falign-functions=1'` are equivalent and mean that functions will not be aligned.

Some assemblers only support this flag when N is a power of two; in that case, it is rounded up.

If N is not specified, use a machine-dependent default.

- **``-falign-labels'`**

- **``-falign-labels=N'`**

Align all branch targets to a power-of-two boundary, skipping up to N bytes like ``-falign-functions'`. This option can easily make code slower, because it must insert dummy operations for when the branch target is reached in the usual flow of the code.

If ``-falign-loops'` or ``-falign-jumps'` are applicable and are greater than this value, then their values are used instead.

If N is not specified, use a machine-dependent default which is very likely to be ``1'`, meaning no alignment.

- **``-falign-loops'`**

- **``-falign-loops=N'`**

Align loops to a power-of-two boundary, skipping up to N bytes like ``-falign-functions'`. The hope is that the loop will be executed many times, which will make up for any execution of the dummy operations.

If N is not specified, use a machine-dependent default.

- ``-falign-jumps'`
- ``-falign-jumps=N'`

Align branch targets to a power-of-two boundary, for branch targets where the targets can only be reached by jumping, skipping up to N bytes like ``-falign-functions'`. In this case, no dummy operations need be executed.

If N is not specified, use a machine-dependent default.

- ``-fssa'`

Perform optimizations in static single assignment form. Each function's flow graph is translated into SSA form, optimizations are performed, and the flow graph is translated back from SSA form. Users should not specify this option, since it is not yet ready for production use.

- ``-fdce'`

Perform dead-code elimination in SSA form. Requires ``-fssa'`. Like ``-fssa'`, this is an experimental feature.

- ``-fsingle-precision-constant'`

Treat floating point constant as single precision constant instead of implicitly converting it to double precision constant.

- ``-frename-registers'`

Attempt to avoid false dependencies in scheduled code by making use of registers left over after register allocation. This optimization will most benefit processors with lots of registers. It can, however, make debugging impossible, since variables will no longer stay in a "home register".

Options Controlling the Preprocessor

These options control the C preprocessor, which is run on each C source file before actual compilation. If you use the ``-E'` option, nothing is done except preprocessing.

Some of these options make sense only together with ``-E'` because they cause the preprocessor output to be unsuitable for actual compilation.

- **``-include FILE'`**

Process FILE as input before processing the regular input file. In effect, the contents of FILE are compiled first. Any ``-D'` and ``-U'` options on the command line are always processed before ``-include FILE'`, regardless of the order in which they are written. All the ``-include'` and ``-imacros'` options are processed in the order in which they are written.

- **``-imacros FILE'`**

Process FILE as input, discarding the resulting output, before processing the regular input file. Because the output generated from FILE is discarded, the only effect of ``-imacros FILE'` is to make the macros defined in FILE available for use in the main input. All the ``-include'` and ``-imacros'` options are processed in the order in which they are written.

- **``-idirafter DIR'`**

Add the directory DIR to the second include path. The directories on the second include path are searched when a header file is not found in any of the directories in the main include path (the one that ``-I'` adds to).

- **``-iprefix PREFIX'`**

Specify PREFIX as the prefix for subsequent ``-iwithprefix'` options.

- **``-iwithprefix DIR'`**

Add a directory to the second include path. The directory's name is made by concatenating PREFIX and DIR, where PREFIX was specified previously with ``-iprefix'`. If you have not specified a prefix yet, the directory containing the installed passes of the compiler is used as the default.

- **``-iwithprefixbefore DIR'`**

Add a directory to the main include path. The directory's name is made by concatenating PREFIX and DIR, as in the case of ``-iwithprefix'`.

- **``-isystem DIR'`**

Add a directory to the beginning of the second include path, marking it as a system directory, so that it gets the same special treatment as is applied to the standard system directories.

- **``-nostdinc'`**

Do not search the standard system directories for header files. Only the directories you have specified with ``-I'` options (and the current directory, if appropriate) are searched.

By using both ``-nostdinc'` and ``-I'`, you can limit the include-file search path to only those directories you specify explicitly.

- **``-remap'`**

When searching for a header file in a directory, remap file names if a file named ``header.gcc'` exists in that directory. This can be used to work around limitations of file systems with file name restrictions. The ``header.gcc'` file should contain a series of lines with two tokens on each line: the first token is the name to map, and the second token is the actual name to use.

- **``-undef'`**

Do not predefine any nonstandard macros. (Including architecture flags).

- **``-E'`**

Run only the C preprocessor. Preprocess all the C source files specified and output the results to standard output or to the specified output file.

- **``-C'`**

Tell the preprocessor not to discard comments. Used with the ``-E'` option.

- **``-P'`**

Tell the preprocessor not to generate ``#line'` directives. Used with the ``-E'` option.

- **``-M'`**

Instead of outputting the result of preprocessing, output a rule suitable for ``make'` describing the dependencies of the main source file. The preprocessor outputs one ``make'` rule containing the object file name for that source file, a colon, and the names of all the included files. Unless overridden explicitly, the object file name consists of the

basename of the source file with any suffix replaced with object file suffix. If there are many included files then the rule is split into several lines using ``'-newline`. ``-M'` implies ``-E'`.

- **``-MM'`**

Like ``-M'`, but mention only the files included with ``#include "FILE"`. System header files included with ``#include <FILE>` are omitted.

- **``-MD'`**

Like ``-M'` but the dependency information is written to a file rather than stdout. ``hms800-cc'` will use the same file name and directory as the object file, but with the suffix ``.d'` instead.

This is in addition to compiling the main file as specified--``-MD'` does not inhibit ordinary compilation the way ``-M'` does, unless you also specify ``-MG'`.

With Mach, you can use the utility ``md'` to merge multiple dependency files into a single dependency file suitable for using with the ``make'` command.

- **``-MMD'`**

Like ``-MD'` except mention only user header files, not system header files.

- **``-MF FILE'`**

When used with ``-M'` or ``-MM'`, specifies a file to write the dependencies to. This allows the preprocessor to write the preprocessed file to stdout normally. If no ``-MF'` switch is given, CPP sends the rules to stdout and suppresses normal preprocessed output.

- **``-MG'`**

When used with ``-M'` or ``-MM'`, ``-MG'` says to treat missing header files as generated files and assume they live in the same directory as the source file. It suppresses preprocessed output, as a missing header file is ordinarily an error. This feature is used in automatic updating of makefiles.

- **``-H'`**

Print the name of each header file used, in addition to other normal activities.

- **``-DMACRO'`**

Define macro MACRO with the string `1' as its definition.

- **`-DMACRO=DEFN'**

Define macro MACRO as DEFN. All instances of `-D' on the command line are processed before any `-U' options.

Any `-D' and `-U' options on the command line are processed in order, and always before `-imacros FILE'`, regardless of the order in which they are written.

- **`-UMACRO'**

Undefine macro MACRO. `-U' options are evaluated after all `-D' options, but before any `-include'` and `-imacros'` options.

Any `-D' and `-U' options on the command line are processed in order, and always before `-imacros FILE'`, regardless of the order in which they are written.

- **`-dM'**

Tell the preprocessor to output only a list of the macro definitions that are in effect at the end of preprocessing. Used with the `-E'` option.

- **`-dD'**

Tell the preprocessing to pass all macro definitions into the output, in their proper sequence in the rest of the output.

- **`-dN'**

Like `-dD'` except that the macro arguments and contents are omitted. Only `#define NAME'` is included in the output.

- **`-dl'**

Output `#include'` directives in addition to the result of preprocessing.

- **`-fpreprocessed'**

Indicate to the preprocessor that the input file has already been preprocessed. This suppresses things like macro expansion, trigraph conversion, escaped newline splicing, and processing of most directives. In this mode the integrated preprocessor is little more than a tokenizer for the front ends.

`-fpreprocessed'` is implicit if the input file has one of the extensions `.i'`, `.ii'` or `.mi'` indicating it has already been preprocessed.

- **`-Wp,OPTION'**

Pass OPTION as an option to the preprocessor. If OPTION contains commas, it is split into multiple options at the commas.

Passing Options to the Assembler

You can pass options to the assembler.

- **`-Wa,OPTION'**

Pass OPTION as an option to the assembler. If OPTION contains commas, it is split into multiple options at the commas.

Options for Linking

These options come into play when the compiler links object files into an executable output file. They are meaningless if the compiler is not doing a link step.

- **`OBJECT-FILE-NAME'**

A file name that does not end in a special recognized suffix is considered to name an object file or library. (Object files are distinguished from libraries by the linker according to the file contents.) If linking is done, these object files are used as input to the linker.

- **`-c'**

- **`-S'**

- **`-E'**

If any of these options is used, then the linker is not run, and object file names should not be used as arguments.

- **`-ILIBRARY'**

Search the library named LIBRARY when linking. It makes a difference where in the command you write this option; the linker searches processes libraries and object files in the order they are specified. Thus, ``foo.o -lz bar.o'` searches library ``z'` after file ``foo.o'` but before ``bar.o'`. If ``bar.o'` refers to functions in ``z'`, those functions may not be loaded.

The linker searches a standard list of directories for the library, which is actually a file named ``libLIBRARY.a'`. The linker then uses this file as if it had been specified precisely by name.

The directories searched include several standard system directories plus any that you specify with ``-L'`.

Normally the files found this way are library files--archive files whose members are object files. The linker handles an archive file by scanning through it for members which define symbols that have so far been referenced but not defined. But if the file that is found is an ordinary object file, it is linked in the usual fashion. The only difference between using an ``-l'` option and specifying a file name is that ``-l'` surrounds LIBRARY with ``lib'` and ``.a'` and searches several directories.

- **``-nostartfiles'`**

Do not use the standard system startup files when linking. The standard system libraries are used normally, unless ``-nostdlib'` or ``-nodefaultlibs'` is used.

- **``-nodefaultlibs'`**

Do not use the standard system libraries when linking. Only the libraries you specify will be passed to the linker. The standard startup files are used normally, unless ``-nostartfiles'` is used. The compiler may generate calls to `memcpy`, `memset`, and `memcpy` for System V (and ISO C) environments or to `bcopy` and `bzero` for BSD environments. These entries are usually resolved by entries in `libc`. These entry points should be supplied through some other mechanism when this option is specified.

- **``-nostdlib'`**

Do not use the standard system startup files or libraries when linking. No startup files and only the libraries you specify will be passed to the linker. The compiler may generate calls to `memcpy`, `memset`, and `memcpy` for System V (and ISO C) environments or to `bcopy` and `bzero` for BSD environments. These entries are usually resolved by entries in `libc`. These entry points should be supplied through some other mechanism when this option is specified.

One of the standard libraries bypassed by ``-nostdlib'` and ``-nodefaultlibs'` is ``libgcc.a'`, a library of internal subroutines that HMS800 C Compiler uses to overcome shortcomings of particular machines, or special needs for some languages. In most cases, you need ``libgcc.a'` even when you want to avoid other standard libraries. In other

words, when you specify ``-nostdlib'` or ``-nodefaultlibs'`, you should usually specify ``-lgcc'` as well.

This ensures that you have no unresolved references to internal HMS800 C Compiler library subroutines.

- **``-s'`**

Remove all symbol table and relocation information from the executable.

- **``-Xlinker OPTION'`**

Pass `OPTION` as an option to the linker. You can use this to supply system-specific linker options which HMS800 C Compiler does not know how to recognize.

If you want to pass an option that takes an argument, you must use ``-Xlinker'` twice, once for the option and once for the argument. For example, to pass ``-assert definitions'`, you must write ``-Xlinker -assert -Xlinker definitions'`. It does not work to write ``-Xlinker "-assert definitions"'`, because this passes the entire string as a single argument, which is not what the linker expects.

- **``-WI,OPTION'`**

Pass `OPTION` as an option to the linker. If `OPTION` contains commas, it is split into multiple options at the commas.

- **``-u SYMBOL'`**

Pretend the symbol `SYMBOL` is undefined, to force linking of library modules to define it. You can use ``-u'` multiple times with different symbols to force loading of additional library modules.

Options for Directory Search

These options specify directories to search for header files, for libraries and for parts of the compiler:

- **``-IDIR'`**

Add the directory `DIR` to the head of the list of directories to be searched for header files. This can be used to override a system header file, substituting your own version, since these directories are searched before the system header file directories. However, you should not use this option to add directories that contain vendor-supplied system

header files (use `-isystem` for that). If you use more than one `-I` option, the directories are scanned in left-to-right order; the standard system directories come after.

- **`-I`**

Any directories you specify with `-I` options before the `-I` option are searched only for the case of `#include "FILE"`; they are not searched for `#include <FILE>`.

If additional directories are specified with `-I` options after the `-I`, these directories are searched for all `#include` directives. (Ordinarily all `-I` directories are used this way.)

In addition, the `-I` option inhibits the use of the current directory (where the current input file came from) as the first search directory for `#include "FILE"`. There is no way to override this effect of `-I`. With `-I.` you can specify searching the directory which was current when the compiler was invoked. That is not exactly the same as what the preprocessor does by default, but it is often satisfactory.

`-I` does not inhibit the use of the standard system directories for header files. Thus, `-I` and `-nostdinc` are independent.

- **`-LDIR`**

Add directory DIR to the list of directories to be searched for `-I`.

- **`-BPREFIX`**

This option specifies where to find the executables, libraries, include files, and data files of the compiler itself. The compiler driver program runs one or more of the subprograms `cpp`, `cc1`, `as` and `ld`. For each subprogram to be run, the compiler driver first tries the `-B` prefix, if any. If that name is not found, or if `-B` was not specified, the driver tries two standard prefixes. If neither of those results in a file name that is found, the unmodified program name is searched for using the directories specified in your `PATH` environment variable.

`-B` prefixes that effectively specify directory names also apply to libraries in the linker, because the compiler translates these options into `-L` options for the linker. They also apply to include files in the preprocessor, because the compiler translates these options into `-isystem` options for the preprocessor. In this case, the compiler appends `include` to the prefix.

- **`-specs=FILE`**

Process FILE after the compiler reads in the standard `specs` file, in order to override the defaults that the `gcc` driver program uses when determining what switches

to pass to ``cc1'`, ``cc1plus'`, ``as'`, ``ld'`, etc. More than one ``-specs=FILE'` can be specified on the command line, and they are processed in order, from left to right.

Options for Code Generation Conventions

These machine-independent options control the interface conventions used in code generation. Most of them have both positive and negative forms; the negative form of ``-foo'` would be ``-fno-foo'`. In the table below, only one of the forms is listed--the one which is not the default. You can figure out the other form by either removing ``no-'` or adding it.

- **``-fpcc-struct-return'`**

Return **short** ``struct'` and ``union'` values in memory like longer ones, rather than in registers. This convention is less efficient, but it has the advantage of allowing intercallability between HMS800 C Compiler compiled files and files compiled with other compilers.

Short structures and unions are those whose size and alignment match that of some integer type.

- **``-freg-struct-return'`**

Use the convention that ``struct'` and ``union'` values are returned in registers when possible. This is more efficient for small structures than ``-fpcc-struct-return'`.

If you specify neither ``-fpcc-struct-return'` nor its contrary ``-freg-struct-return'`, HMS800 C Compiler defaults to whichever convention is standard for the target. If there is no standard convention, HMS800 C Compiler defaults to ``-fpcc-struct-return'`, except on targets where HMS800 C Compiler is the principal compiler. In those cases, we can choose the standard, and we chose the more efficient register return alternative.

- **``-fshort-enums'`**

Allocate to an ``enum'` type only as many bytes as it needs for the declared range of possible values. Specifically, the ``enum'` type will be equivalent to the smallest integer type which has enough room.

- **``-fshort-double'`**

Use the same size for ``double'` as for ``float'`.

- **``-fno-common'`**

In C, allocate even uninitialized global variables in the data section of the object file, rather than generating them as common blocks. This has the effect that if the same variable is declared (without ``extern'`) in two different compilations, you will get an error when you link them. The only reason this might be useful is if you wish to verify that the program will work on other systems which always work this way.

- **``-fno-ident'`**

Ignore the ``#ident'` directive.

- **``-fverbose-asm'`**

Put extra commentary information in the generated assembly code to make it more readable. This option is generally only of use to those who actually need to read the generated assembly code (perhaps while debugging the compiler itself).

``-fno-verbose-asm'`, the default, causes the extra information to be omitted and is useful when comparing two assembler files.

- **`'-fvolatile'`**

Consider all memory references through pointers to be volatile.

- **``-fvolatile-global'`**

Consider all memory references to extern and global data items to be volatile.

HMS800 C Compiler does not consider static data items to be volatile because of this switch.

- **``-fvolatile-static'`**

Consider all memory references to static data to be volatile.

- **``-fpic'`**

- **``-fPIC'`**

Generate position-independent code (PIC) suitable for use in a shared library, if supported for the target machine. Such code accesses all constant addresses through a global offset table (GOT). The dynamic loader resolves the GOT entries when the program starts (the dynamic loader is not part of HMS800 C Compiler; it is part of the operating system). If the GOT size for the linked executable exceeds a machine-specific maximum size, you get an error message from the linker indicating that ``-fpic'` does not work; in that case, recompile with ``-fPIC'` instead.

- **'-fpack-struct'**

Pack all structure members together without holes. Usually you would not want to use this option, since it makes the code suboptimal, and the offsets of structure members won't agree with system libraries.

- **'-fprefix-function-name'**

Request HMS800 C Compiler to add a prefix to the symbols generated for function names. HMS800 C Compiler adds a prefix to the names of functions defined as well as functions called. Code compiled with this option and code compiled without the option can't be linked together, unless stubs are used.

If you compile the following code with `'-fprefix-function-name'`

```
extern void bar (int);
void
foo (int a)
{
    return bar (a + 5);
}
```

HMS800 C Compiler will compile the code as if it was written:

```
extern void prefix_bar (int);
void
prefix_foo (int a)
{
    return prefix_bar (a + 5);
}
```

- **'-fleading-underscore'**

This option and its counterpart, `'-fno-leading-underscore'`, forcibly change the way C symbols are represented in the object file. One use is to help link with legacy assembly code. Be warned that you should know what you are doing when invoking this option, and that not all targets provide complete support for it.

Extensions to C Language Family

HMS800 C provides several language features not found in ISO standard C. (The `-pedantic` option directs HMS800 C Compiler to print a warning message if any of these features is used.)

Statements and Declarations in Expressions

A compound statement enclosed in parentheses may appear as an expression in HMS800 C. This allows you to use loops, switches, and local variables within an expression.

Recall that a compound statement is a sequence of statements surrounded by braces; in this construct, parentheses go around the braces. For example:

```
(( int y = foo (); int z;  
   if (y > 0) z = y;  
   else z = - y;  
   z;  })
```

is a valid (though slightly more complex than necessary) expression for the absolute value of `foo ()`.

The last thing in the compound statement should be an expression followed by a semicolon; the value of this subexpression serves as the value of the entire construct. (If you use some other kind of statement last within the braces, the construct has type `void`, and thus effectively no value.)

This feature is especially useful in making macro definitions "safe" (so that they evaluate each operand exactly once). For example, the "maximum" function is commonly defined as a macro in standard C as follows:

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

But this definition computes either A or B twice, with bad results if the operand has side effects. In HMS800 C, if you know the type of the operands (here let's assume `int`), you can define the macro safely as follows:

```
#define maxint(a, b) \  
  ((int _a = (a), _b = (b); _a > _b ? _a : _b;  })
```

Embedded statements are not allowed in constant expressions, such as the value of an enumeration constant, the width of a bit-field, or the initial value of a static variable. If you don't know the type of the operand, you can still do this, but you must use ``typeof'` or type naming.

Locally Declared Labels

Each statement expression is a scope in which `{\em local labels}` can be declared. A local label is simply an identifier; you can jump to it with an ordinary ``goto'` statement, but only from within the statement expression it belongs to.

A local label declaration looks like this:

```
__label__ LABEL;
```

or

```
__label__ LABEL1, LABEL2, ...;
```

Local label declarations must come at the beginning of the statement expression, right after the `{'`, before any ordinary declarations.

The label declaration defines the label name, but does not define the label itself. You must do this in the usual way, with ``LABEL:'`, within the statements of the statement expression.

The local label feature is useful because statement expressions are often used in macros. If the macro contains nested loops, a ``goto'` can be useful for breaking out of them. However, an ordinary label whose scope is the whole function cannot be used: if the macro can be expanded several times in one function, the label will be multiply defined in that function. A local label avoids this problem. For example:

```
#define SEARCH(array, target) \
({ \
    __label__ found; \
    typeof (target) _SEARCH_target = (target); \
    typeof (*(array)) *_SEARCH_array = (array); \
    int i, j; \
    int value; \
    for (i = 0; i < max; i++) \
        for (j = 0; j < max; j++) \
```

```

        if (_SEARCH_array[i][j] == _SEARCH_target) \
            { value = i; goto found; }          \
    value = -1;                                  \
found:                                          \
    value;                                      \
})

```

Labels as Values

You can get the address of a label defined in the current function (or a containing function) with the unary operator `&&`. The value has type `{tt void *}`. This value is a constant and can be used wherever a constant of that type is valid. For example:

```

void *ptr;
...
ptr = &&foo;

```

To use these values, you need to be able to jump to one. This is done with the computed goto statement(1), `goto *EXP;`. For example,

```

goto *ptr;

```

Any expression of type `{tt void *}` is allowed.

One way of using these constants is in initializing a static array that will serve as a jump table:

```

static void *array[] = { &&foo, &&bar, &&hack };

```

Then you can select a label with indexing, like this:

```

goto *array[i];

```

Note that this does not check whether the subscript is in bounds—array indexing in C never does that.

Such an array of label values serves a purpose much like that of the `switch` statement. The `switch` statement is cleaner, so use that rather than an array unless the

problem does not fit a `switch' statement very well. Another use of label values is in an interpreter for threaded code. The labels within the interpreter function can be stored in the threaded code for super-fast dispatching. You may not use this mechanism to jump to code in a different function. If you do that, totally unpredictable things will happen. The best way to avoid this is to store the label address only in automatic variables and never pass it as an argument. An alternate way to write the above example is

```
static const int array[] = { &&foo - &&foo, &&bar - &&foo,
                             &&hack - &&foo };
goto *(&&foo + array[i]);
```

This is friendlier to code living in shared libraries, as it reduces the number of dynamic relocations that are needed, and by consequence, allows the data to be read-only.

Naming an Expression's Type

You can give a name to the type of an expression using a `typedef' declaration with an initializer. Here is how to define NAME as a type name for the type of EXP:

```
typedef NAME = EXP;
```

This is useful in conjunction with the statements-within-expressions feature. Here is how the two together can be used to define a safe "maximum" macro that operates on any arithmetic type:

```
#define max(a, b) \
    ({typedef _ta = (a), _tb = (b); \
      _ta _a = (a); _tb _b = (b); \
      _a > _b ? _a : _b; })
```

The reason for using names that start with underscores for the local variables is to avoid conflicts with variable names that occur within the expressions that are substituted for `a' and `b'. Eventually we hope to design a new form of declaration syntax that allows you to declare variables whose scopes start only after their initializers; this will be a more reliable way to prevent such conflicts.

Referring to a Type with `typeof`

Another way to refer to the type of an expression is with `typeof`. The syntax of using of this keyword looks like `sizeof`, but the construct acts semantically like a type name defined with `typedef`. There are two ways of writing the argument to `typeof`: with an expression or with a type. Here is an example with an expression:

```
typeof (x[0](1))
```

This assumes that `x` is an array of pointers to functions; the type described is that of the values of the functions. Here is an example with a typename as the argument:

```
typeof (int *)
```

Here the type described is that of pointers to `int`.

If you are writing a header file that must work when included in ISO C programs, write `__typeof__` instead of `typeof`. A `typeof`-construct can be used anywhere a `typedef` name could be used. For example, you can use it in a declaration, in a cast, or inside of `sizeof` or `typeof`.

- This declares `y` with the type of what `x` points to.

```
typeof (*x) y;
```

- This declares `y` as an array of such values.

```
typeof (*x) y[4];
```

- This declares `y` as an array of pointers to characters:

```
typeof (typeof (char *)[4]) y;
```

It is equivalent to the following traditional C declaration:

```
char *y[4];
```

To see the meaning of the declaration using `typeof`, and why it might be a useful way to write, let's rewrite it with these macros:

```
#define pointer(T)  typeof(T *)
```

```
#define array(T, N) typedef(T [N])
```

Now the declaration can be rewritten this way:

```
array (pointer (char), 4) y;
```

Thus, 'array (pointer (char), 4)' is the type of arrays of 4 pointers to 'char'.

Macros with a Variable Number of Arguments

In the ISO C standard of 1999, a macro can be declared to accept a variable number of arguments much as a function can. The syntax for defining the macro is similar to that of a function. Here is an example:

```
#define debug(format, ...) fprintf (stderr, format, __VA_ARGS__)
```

Here '...' is a **variable argument**. In the invocation of such a macro, it represents the zero or more tokens until the closing parenthesis that ends the invocation, including any commas. This set of tokens replaces the identifier '__VA_ARGS__' in the macro body wherever it appears. See the CPP manual for more information.

HMS800 C Compiler has long supported variadic macros, and used a different syntax that allowed you to give a name to the variable arguments just like any other argument. Here is an example:

```
#define debug(format, args...) fprintf (stderr, format, args)
```

This is in all ways equivalent to the ISO C example above, but arguably more readable and descriptive.

HMS800 C Compiler CPP has two further variadic macro extensions, and permits them to be used with either of the above forms of macro definition.

In standard C, you are not allowed to leave the variable argument out entirely; but you are allowed to pass an empty argument. For example, this invocation is invalid in ISO C, because there is no comma after the string:

```
debug ("A message")
```

HMS800 C Compiler CPP permits you to completely omit the variable arguments in this way. In the above examples, the compiler would complain, though since the expansion of the macro still has the extra comma after the format string.

To help solve this problem, CPP behaves specially for variable arguments used with the token paste operator, `##`. If instead you write

```
#define debug(format, ...) fprintf(stderr, format, ## __VA_ARGS__)
```

and if the variable arguments are omitted or empty, the `##` operator causes the preprocessor to remove the comma before it. If you do provide some variable arguments in your macro invocation, GNU CPP does not complain about the paste operation and instead places the variable arguments after the comma. Just like any other pasted macro argument, these arguments are not macro expanded.

Case Ranges

You can specify a range of consecutive values in a single `case` label, like this:

```
case LOW ... HIGH:
```

This has the same effect as the proper number of individual `case` labels, one for each integer value from LOW to HIGH, inclusive.

This feature is especially useful for ranges of ASCII character codes:

```
case 'A' ... 'Z':
```

Write spaces around the `...`, for otherwise it may be parsed wrong when you use it with integer values. For example, write this:

```
case 1 ... 5:
```

rather than this:

```
case 1..5:
```

Cast to a Union Type

A cast to union type is similar to other casts, except that the type specified is a union type. You can specify the type either with ``union TAG'` or with a typedef name. A cast to union is actually a constructor though, not a cast, and hence does not yield an lvalue like normal casts.

The types that may be cast to the union type are those of the members of the union. Thus, given the following union and variables:

```
union foo { int i; double d; };
int x;
double y;
```

Both ``x'` and ``y'` can be cast to type ``union foo'`.

Using the cast as the right-hand side of an assignment to a variable of union type is equivalent to storing in a member of the union:

```
union foo u;
...
u = (union foo) x == u.i = x
u = (union foo) y == u.d = y
```

You can also use the union cast as a function argument:

```
void hack (union foo);
...
hack ((union foo) x);
```

Declaring Attributes of Functions

In HMS800 C, you declare certain things about functions called in your program which help the compiler optimize function calls and check your code more carefully.

The keyword ``__attribute__'` allows you to specify special attributes when making a declaration. This keyword is followed by an attribute specification inside double parentheses. Fourteen attributes, ``noreturn'`, ``pure'`, ``const'`, ``format'`, ``format_arg'`, ``no_instrument_function'`, ``section'`, ``constructor'`, ``destructor'`, ``unused'`, ``weak'`, ``malloc'`, ``alias'` and ``no_check_memory_usage'` are currently defined for functions. Several other

attributes are defined for functions on particular target systems. Other attributes, including ``section'` are supported for variables declarations and for types.

You may also specify attributes with ``__'` preceding and following each keyword. This allows you to use them in header files without being concerned about a possible macro of the same name. For example, you may use ``__noreturn__'` instead of ``noreturn'`.

- **``noreturn'`**

A few standard library functions, such as ``abort'` and ``exit'`, cannot return. HMS800 C Compiler knows this automatically. Some programs define their own functions that never return. You can declare them ``noreturn'` to tell the compiler this fact. For example,

```
void fatal () __attribute__((noreturn));

void
fatal (...)
{
    ... /* Print error message. */ ...
    exit (1);
}
```

The ``noreturn'` keyword tells the compiler to assume that ``fatal'` cannot return. It can then optimize without regard to what would happen if ``fatal'` ever did return. This makes slightly better code. More importantly, it helps avoid spurious warnings of uninitialized variables.

Do not assume that registers saved by the calling function are restored before calling the ``noreturn'` function.

It does not make sense for a ``noreturn'` function to have a return type other than ``void'`.

- **``pure'`**

Many functions have no effects except the return value and their return value depends only on the parameters and/or global variables. Such a function can be subject to common subexpression elimination and loop optimization just as an arithmetic operator would be. These functions should be declared with the attribute ``pure'`. For example,

```
int square (int) __attribute__ ((pure));
```

says that the hypothetical function `square` is safe to call fewer times than the program says.

Some of common examples of pure functions are `strlen` or `memcmp`. Interesting non-pure functions are functions with infinite loops or those depending on volatile memory or other system resource, that may change between two consecutive calls (such as `feof` in a multithreading environment).

- **`const`**

Many functions do not examine any values except their arguments, and have no effects except the return value. Basically this is just slightly more strict class than the `pure` attribute above, since function is not allowed to read global memory.

Note that a function that has pointer arguments and examines the data pointed to must not be declared `const`. Likewise, a function that calls a non-`const` function usually must not be `const`. It does not make sense for a `const` function to return `void`.

- **`format (ARCHETYPE, STRING-INDEX, FIRST-TO-CHECK)`**

The `format` attribute specifies that a function takes `printf`, `scanf`, `strptime` or `strfmon` style arguments which should be type-checked against a format string. For example, the declaration:

```
extern int  
my_printf (void *my_object, const char *my_format, ...)  
__attribute__ ((format (printf, 2, 3)));
```

causes the compiler to check the arguments in calls to `my_printf` for consistency with the `printf` style format string argument `my_format`.

The parameter ARCHETYPE determines how the format string is interpreted, and should be `printf`, `scanf`, `strptime` or `strfmon`. (You can also use `__printf__`, `__scanf__`, `__strptime__` or `__strfmon__`.) The parameter STRING-INDEX specifies which argument is the format string argument (starting from 1), while FIRST-TO-CHECK is the number of the first argument to check against the format string. For functions where the arguments are not available to be checked (such as `vprintf`), specify the third

parameter as zero. In this case the compiler only checks the format string for consistency. For ``strftime'` formats, the third parameter is required to be zero.

In the example above, the format string (``my_format'`) is the second argument of the function ``my_print'`, and the arguments to check start with the third argument, so the correct parameters for the format attribute are 2 and 3.

The ``format'` attribute allows you to identify your own functions which take format strings as arguments, so that HMS800 C Compiler can check the calls to these functions for errors. The compiler always (unless ``-ffreestanding'` is used) checks formats for the standard library functions ``printf'`, ``fprintf'`, ``sprintf'`, ``scanf'`, ``fscanf'`, ``sscanf'`, ``strftime'`, ``vprintf'`, ``vfprintf'` and ``vsprintf'` whenever such warnings are requested (using ``-Wformat'`), so there is no need to modify the header file ``stdio.h'`. In C99 mode, the functions ``snprintf'`, ``vsnprintf'`, ``vscanf'`, ``vfscanf'` and ``vsscanf'` are also checked. Except in strictly conforming C standard modes, the X/Open function ``strfmon'` is also checked. *Note Options Controlling C Dialect: C Dialect Options.

- **``format_arg (STRING-INDEX)'`**

The ``format_arg'` attribute specifies that a function takes a format string for a ``printf'`, ``scanf'`, ``strftime'` or ``strfmon'` style function and modifies it (for example, to translate it into another language), so the result can be passed to a ``printf'`, ``scanf'`, ``strftime'` or ``strfmon'` style function (with the remaining arguments to the format function the same as they would have been for the unmodified string). For example, the declaration:

```
extern char *
my_dgettext (char *my_domain, const char *my_format)
    __attribute__ ((format_arg (2)));
```

causes the compiler to check the arguments in calls to a ``printf'`, ``scanf'`, ``strftime'` or ``strfmon'` type function, whose format string argument is a call to the ``my_dgettext'` function, for consistency with the format string argument ``my_format'`. If the ``format_arg'` attribute had not been specified, all the compiler could tell in such calls to format functions would be that the format string argument is not constant; this would generate a warning when ``-Wformat-nonliteral'` is used, but the calls could not be checked without the attribute.

The parameter STRING-INDEX specifies which argument is the format string argument (starting from 1). The ``format-arg'` attribute allows you to identify your own functions which modify format strings, so that HMS800 C Compiler can check the calls to

``printf`, `scanf`, `strftime` or `strfmon`` type function whose operands are a call to one of your own function. The compiler always treats ``gettext`, `dgettext`, and `dcgettext`` in this manner except when strict ISO C support is requested by ``-ansi`` or an appropriate ``-std`` option, or ``-ffreestanding`` is used.

- **``section ("SECTION-NAME")``**

Normally, the compiler places the code it generates in the ``text`` section.

Sometimes, however, you need additional sections, or you need certain particular functions to appear in special sections. The ``section`` attribute specifies that a function lives in a particular section. For example, the declaration:

```
extern void foobar (void) __attribute__((section ("bar")));
```

puts the function ``foobar`` in the ``bar`` section.

Some file formats do not support arbitrary sections so the ``section`` attribute is not available on all platforms. If you need to map the entire contents of a module to a particular section, consider using the facilities of the linker instead.

- **``constructor``**

- **``destructor``**

The ``constructor`` attribute causes the function to be called automatically before execution enters ``main ()``. Similarly, the ``destructor`` attribute causes the function to be called automatically after ``main ()`` has completed or ``exit ()`` has been called. Functions with these attributes are useful for initializing data that will be used implicitly during the execution of the program.

- **``unused``**

This attribute, attached to a function, means that the function is meant to be possibly unused. HMS800 C Compiler will not produce a warning for this function.

- **``weak``**

The ``weak`` attribute causes the declaration to be emitted as a weak symbol rather than a global. This is primarily useful in defining library functions which can be overridden in user code, though it can also be used with non-function declarations. Weak symbols are supported for ELF targets, and also for a.out targets when using the GNU assembler and linker.

- **`malloc'**

The ``malloc'` attribute is used to tell the compiler that a function may be treated as if it were the `malloc` function. The compiler assumes that calls to `malloc` result in pointers that cannot alias anything. This will often improve optimization.

- **`no_check_memory_usage'**

The ``no_check_memory_usage'` attribute causes HMS800 C Compiler to omit checks of memory references when it generates code for that function. Normally if you specify ``-fcheck-memory-usage'` (see **note Code Gen Options::*), HMS800 C Compiler generates calls to support routines before most memory accesses to permit support code to record usage and detect uses of uninitialized or unallocated storage. Since HMS800 C Compiler cannot handle ``asm'` statements properly they are not allowed in such functions. If you declare a function with this attribute, HMS800 C Compiler will not generate memory checking code for that function, permitting the use of ``asm'` statements without having to compile that function with different options. This also allows you to write support routines of your own if you wish, without getting infinite recursion if they get compiled with ``-fcheck-memory-usage'`.

- **`function_vector'**

Use this option to indicate that the specified function should be called through the function vector. Calling a function through the function vector will reduce code size, however; the function vector has a limited size and shares space with the interrupt vector.

- **`interrupt'**

- **`interrupt_handler'**

Use this option to indicate that the specified function is an interrupt handler. The compiler will generate function entry and exit sequences suitable for use in an interrupt handler when this attribute is present. The compiler will generate function entry and exit sequences suitable for use in an interrupt handler when this attribute is present.

- **`eightbit_data'**

Use this option to indicate that the specified variable should be placed into the eight bit data section. The compiler will generate more efficient code for certain operations on data in the eight bit data area. Note the eight bit data area is limited to 256 bytes of data.

You can specify multiple attributes in a declaration by separating them by commas within the double parentheses or by immediately following an attribute declaration with another attribute declaration.

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. Because of infelicities in the grammar for attributes, some forms described here may not be successfully parsed in all cases.

An **attribute specifier** is of the form `__attribute__((ATTRIBUTE-LIST))`. An **attribute list** is a possibly empty comma-separated sequence of **attributes**, where each attribute is one of the following:

- Empty. Empty attributes are ignored.
- A word (which may be an identifier such as `unused`, or a reserved word such as `const`).
- A word followed by, in parentheses, parameters for the attribute. These parameters take one of the following forms:
 - An identifier. For example, `mode` attributes use this form.
 - An identifier followed by a comma and a non-empty comma-separated list of expressions. For example, `format` attributes use this form.
 - A possibly empty comma-separated list of expressions. For example, `format_arg` attributes use this form with the list being a single integer constant expression, and `alias` attributes use this form with the list being a single string constant.

An **attribute specifier list** is a sequence of one or more attribute specifiers, not separated by any other tokens.

An attribute specifier list may appear after the colon following a label, other than a `case` or `default` label. The only attribute it makes sense to use after a label is `unused`.

This feature is intended for code generated by programs which contains labels that may be unused but which is compiled with `-Wall`. It would not normally be appropriate to use in human-written code, though it could be useful in cases where the code that jumps to the label is contained within a `#ifdef` conditional.

An attribute specifier list may appear as part of a `struct`, `union` or `enum` specifier. It may go either immediately after the `struct`, `union` or `enum` keyword, or after the closing brace. It is ignored if the content of the structure, union or enumerated type is not defined in the specifier in which the attribute specifier list is used--that is, in usages such as `struct __attribute__((foo)) bar` with no following opening brace. Where attribute specifiers follow the closing brace, they are considered to relate to the structure, union or enumerated type defined, not to any enclosing declaration the type specifier appears in, and the type defined is not complete until after the attribute specifiers.

Otherwise, an attribute specifier appears as part of a declaration, counting declarations of unnamed parameters and type names, and relates to that declaration (which may be nested in another declaration, for example in the case of a parameter declaration). In future, attribute specifiers in some places may however apply to a particular declarator within a declaration instead; these cases are noted below. Where an attribute specifier is applied to a parameter declared as a function or an array, it should apply to the function or array rather than the pointer to which the parameter is implicitly converted, but this is not yet correctly implemented.

Any list of specifiers and qualifiers at the start of a declaration may contain attribute specifiers, whether or not such a list may in that context contain storage class specifiers. (Some attributes, however, are essentially in the nature of storage class specifiers, and only make sense where storage class specifiers may be used; for example, `section`.) There is one necessary limitation to this syntax: the first old-style parameter declaration in a function definition cannot begin with an attribute specifier, because such an attribute applies to the function instead by syntax described below (which, however, is not yet implemented in this case). In some other cases, attribute specifiers are permitted by this grammar but not yet supported by the compiler. All attribute specifiers in this place relate to the declaration as a whole. In the obsolescent usage where a type of `int` is implied by the absence of type specifiers, such a list of specifiers and qualifiers may be an attribute specifier list with no other specifiers or qualifiers.

An attribute specifier list may appear immediately before a declarator (other than the first) in a comma-separated list of declarators in a declaration of more than one identifier using a single list of specifiers and qualifiers. At present, such attribute specifiers apply not only to the identifier before whose declarator they appear, but to all subsequent

identifiers declared in that declaration, but in future they may apply only to that single identifier. For example, in ``__attribute__((noreturn)) void d0 (void), __attribute__((format(printf, 1, 2))) d1 (const char *, ...), d2 (void)``, the ``noreturn`` attribute applies to all the functions declared; the ``format`` attribute should only apply to ``d1``, but at present applies to ``d2`` as well (and so causes an error).

An attribute specifier list may appear immediately before the comma, ``='` or semicolon terminating the declaration of an identifier other than a function definition. At present, such attribute specifiers apply to the declared object or function, but in future they may attach to the outermost adjacent declarator. In simple cases there is no difference, but, for example, in ``void (****f) (void) __attribute__((noreturn));``, at present the ``noreturn`` attribute applies to ``f``, which causes a warning since ``f`` is not a function, but in future it may apply to the function ``****f``. The precise semantics of what attributes in such cases will apply to are not yet specified. Where an assembler name for an object or function is specified, at present the attribute must follow the ``asm`` specification; in future, attributes before the ``asm`` specification may apply to the adjacent declarator and those after it to the declared object or function.

An attribute specifier list may, in future, be permitted to appear after the declarator in a function definition (before any old-style parameter declarations or the function body).

An attribute specifier list may appear at the start of a nested declarator. At present, there are some limitations in this usage: the attributes apply to the identifier declared, and to all subsequent identifiers declared in that declaration (if it includes a comma-separated list of declarators), rather than to a specific declarator. When attribute specifiers follow the ``*`` of a pointer declarator, they must presently follow any type qualifiers present, and cannot be mixed with them. The following describes intended future semantics which make this syntax more useful only. It will make the most sense if you are familiar with the formal specification of declarators in the ISO C standard.

Consider (as in C99 subclause 6.7.5 paragraph 4) a declaration ``T D1``, where ``T`` contains declaration specifiers that specify a type TYPE (such as ``int``) and ``D1`` is a declarator that contains an identifier IDENT. The type specified for IDENT for derived declarators whose type does not include an attribute specifier is as in the ISO C standard.

If ``D1`` has the form ``(ATTRIBUTE-SPECIFIER-LIST D)``, and the declaration ``T D`` specifies the type "DERIVED-DECLARATOR-TYPE-LIST TYPE" for IDENT, then ``T D1`` specifies the type "DERIVED-DECLARATOR-TYPE-LIST ATTRIBUTE-SPECIFIER-LIST TYPE" for IDENT.

If ``D1'` has the form ``* TYPE-QUALIFIER-AND-ATTRIBUTE-SPECIFIER-LIST D'`, and the declaration ``T D'` specifies the type "DERIVED-DECLARATOR-TYPE-LIST TYPE" for IDENT, then ``T D1'` specifies the type "DERIVED-DECLARATOR-TYPE-LIST TYPE-QUALIFIER-AND-ATTRIBUTE-SPECIFIER-LIST TYPE" for IDENT.

For example, ``void (__attribute__((noreturn)) ****f)();'` specifies the type "pointer to pointer to pointer to pointer to non-returning function returning ``void`". As another example, ``char *__attribute__((aligned(8))) *f;'` specifies the type "pointer to 8-byte-aligned pointer to ``char`". Note again that this describes intended future semantics, not current implementation.

Specifying Attributes of Variables

The keyword ``__attribute__'` allows you to specify special attributes of variables or structure fields. This keyword is followed by an attribute specification inside double parentheses. Eight attributes are currently defined for variables: ``aligned'`, ``mode'`, ``nocommon'`, ``packed'`, ``section'`, ``transparent_union'`, ``unused'`, and ``weak'`. Some other attributes are defined for variables on particular target systems. Other attributes are available for functions and for types.

You may also specify attributes with ``__'` preceding and following each keyword. This allows you to use them in header files without being concerned about a possible macro of the same name. For example, you may use ``__aligned__'` instead of ``aligned'`.

- ``aligned (ALIGNMENT)'`

This attribute specifies a minimum alignment for the variable or structure field, measured in bytes. For example, the declaration:

```
int x __attribute__((aligned (16))) = 0;
```

causes the compiler to allocate the global variable ``x'` on a 16-byte boundary. On a 68040, this could be used in conjunction with an ``asm'` expression to access the ``move16'` instruction which requires 16-byte aligned operands.

You can also specify the alignment of structure fields. For example, to create a double-word aligned ``int'` pair, you could write:

```
struct foo { int x[2] __attribute__((aligned (8))); };
```

This is an alternative to creating a union with a `double` member that forces the union to be double-word aligned.

It is not possible to specify the alignment of functions; the alignment of functions is determined by the machine's requirements and cannot be changed. You cannot specify alignment for a typedef name because such a name is just an alias, not a distinct type.

As in the preceding examples, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given variable or structure field. Alternatively, you can leave out the alignment factor and just ask the compiler to align a variable or field to the maximum useful alignment for the target machine you are compiling for. For example, you could write:

```
short array[3] __attribute__((aligned));
```

Whenever you leave out the alignment factor in an `aligned` attribute specification, the compiler automatically sets the alignment for the declared variable or field to the largest alignment which is ever used for any data type on the target machine you are compiling for. Doing this can often make copy operations more efficient, because the compiler can use whatever instructions copy the biggest chunks of memory when performing copies to or from the variables or fields that you have aligned this way.

The `aligned` attribute can only increase the alignment; but you can decrease it by specifying `packed` as well. See below.

Note that the effectiveness of `aligned` attributes may be limited by inherent limitations in your linker. On many systems, the linker is only able to arrange for variables to be aligned up to a certain maximum alignment. (For some linkers, the maximum supported alignment may be very very small.) If your linker is only able to align variables up to a maximum of 8 byte alignment, then specifying `aligned(16)` in an `__attribute__` will still only provide you with 8 byte alignment. See your linker documentation for further information.

- `mode (MODE)'

This attribute specifies the data type for the declaration--whichever type corresponds to the mode MODE. This in effect lets you request an integer or floating point type according to its width.

You may also specify a mode of `'byte'` or `'__byte__'` to indicate the mode corresponding to a one-byte integer, `'word'` or `'__word__'` for the mode of a one-word integer, and `'pointer'` or `'__pointer__'` for the mode used to represent pointers.

- `'nocommon'`

This attribute specifies requests GCC not to place a variable "common" but instead to allocate space for it directly. If you specify the `'-fno-common'` flag, GCC will do this for all variables. Specifying the `'nocommon'` attribute for a variable provides an initialization of zeros. A variable may only be initialized in one source file.

- `'packed'`

The `'packed'` attribute specifies that a variable or structure field should have the smallest possible alignment--one byte for a variable, and one bit for a field, unless you specify a larger value with the `'aligned'` attribute. Here is a structure in which the field `'x'` is packed, so that it immediately follows `'a'`:

```
struct foo
{
    char a;
    int x[2] __attribute__((packed));
};
```

- `'section ("SECTION-NAME")'`

Normally, the compiler places the objects it generates in sections like `'data'` and `'bss'`. Sometimes, however, you need additional sections, or you need certain particular variables to appear in special sections, for example to map to special hardware. The `'section'` attribute specifies that a variable (or function) lives in a particular section. For example, this small program uses several specific section names:

```
struct duart a __attribute__((section ("DUART_A"))) = { 0 };
struct duart b __attribute__((section ("DUART_B"))) = { 0 };
char stack[10000] __attribute__((section ("STACK"))) = { 0 };
int init_data __attribute__((section ("INITDATA"))) = 0;

main()
{
```

```

/* Initialize stack pointer */
init_sp (stack + sizeof (stack));

/* Initialize initialized data */
memcpy (&init_data, &data, &edata - &data);

/* Turn on the serial ports */
init_duart (&a);
init_duart (&b);
}

```

Use the ``section'` attribute with an initialized definition of a global variable, as shown in the example. GCC issues a warning and otherwise ignores the ``section'` attribute in uninitialized variable declarations.

You may only use the ``section'` attribute with a fully initialized global definition because of the way linkers work. The linker requires each object be defined once, with the exception that uninitialized variables tentatively go in the ``common'` (or ``bss'`) section and can be multiply "defined". You can force a variable to be initialized with the ``-fno-common'` flag or the ``nocommon'` attribute.

Some file formats do not support arbitrary sections so the ``section'` attribute is not available on all platforms. If you need to map the entire contents of a module to a particular section, consider using the facilities of the linker instead.

- ``unused'`

This attribute, attached to a variable, means that the variable is meant to be possibly unused. GCC will not produce a warning for this variable.

- ``weak'`

The ``weak'` attribute is described in **Note Function Attributes::*

Specifying Attributes of Types

The keyword ``__attribute__'` allows you to specify special attributes of ``struct'` and ``union'` types when you define such types. This keyword is followed by an attribute specification inside double parentheses. Four attributes are currently defined for types:

`aligned', `packed', `transparent_union', and `unused'. Other attributes are defined for functions and for variables.

You may also specify any one of these attributes with `__' preceding and following its keyword. This allows you to use these attributes in header files without being concerned about a possible macro of the same name. For example, you may use `__aligned__' instead of `aligned'.

You may specify the `aligned' and `transparent_union' attributes either in a `typedef' declaration or just past the closing curly brace of a complete enum, struct or union type definition and the `packed' attribute only past the closing brace of a definition.

You may also specify attributes between the enum, struct or union tag and the name of the type rather than after the closing brace.

- `aligned (ALIGNMENT)'

This attribute specifies a minimum alignment (in bytes) for variables of the specified type. For example, the declarations:

```
struct S { short f[3]; } __attribute__((aligned (8)));  
typedef int more_aligned_int __attribute__((aligned (8)));
```

force the compiler to insure (as far as it can) that each variable whose type is `struct S' or `more_aligned_int' will be allocated and aligned at least on a 8-byte boundary. On a Sparc, having all variables of type `struct S' aligned to 8-byte boundaries allows the compiler to use the `ldd' and `std' (doubleword load and store) instructions when copying one variable of type `struct S' to another, thus improving run-time efficiency.

Note that the alignment of any given `struct' or `union' type is required by the ISO C standard to be at least a perfect multiple of the lowest common multiple of the alignments of all of the members of the `struct' or `union' in question. This means that you can effectively adjust the alignment of a `struct' or `union' type by attaching an `aligned' attribute to any one of the members of such a type, but the notation illustrated in the example above is a more obvious, intuitive, and readable way to request the compiler to adjust the alignment of an entire `struct' or `union' type.

As in the preceding example, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given `struct' or `union' type. Alternatively, you can leave out the alignment factor and just ask the compiler to align a type to the maximum

useful alignment for the target machine you are compiling for. For example, you could write:

```
struct S { short f[3]; } __attribute__((aligned));
```

Whenever you leave out the alignment factor in an ``aligned'` attribute specification, the compiler automatically sets the alignment for the type to the largest alignment which is ever used for any data type on the target machine you are compiling for. Doing this can often make copy operations more efficient, because the compiler can use whatever instructions copy the biggest chunks of memory when performing copies to or from the variables which have types that you have aligned this way.

In the example above, if the size of each ``short'` is 2 bytes, then the size of the entire ``struct S'` type is 6 bytes. The smallest power of two which is greater than or equal to that is 8, so the compiler sets the alignment for the entire ``struct S'` type to 8 bytes.

Note that although you can ask the compiler to select a time-efficient alignment for a given type and then declare only individual stand-alone objects of that type, the compiler's ability to select a time-efficient alignment is primarily useful only when you plan to create arrays of variables having the relevant (efficiently aligned) type. If you declare or use arrays of variables of an efficiently-aligned type, then it is likely that your program will also be doing pointer arithmetic (or subscripting, which amounts to the same thing) on pointers to the relevant type, and the code that the compiler generates for these pointer arithmetic operations will often be more efficient for efficiently-aligned types than for other types.

The ``aligned'` attribute can only increase the alignment; but you can decrease it by specifying ``packed'` as well. See below.

Note that the effectiveness of ``aligned'` attributes may be limited by inherent limitations in your linker. On many systems, the linker is only able to arrange for variables to be aligned up to a certain maximum alignment. (For some linkers, the maximum supported alignment may be very very small.) If your linker is only able to align variables up to a maximum of 8 byte alignment, then specifying ``aligned(16)'` in an ``__attribute__'` will still only provide you with 8 byte alignment. See your linker documentation for further information.

- ``packed'`

This attribute, attached to an ``enum'`, ``struct'`, or ``union'` type definition, specified that the minimum required memory be used to represent the type.

Specifying this attribute for `'struct'` and `'union'` types is equivalent to specifying the `'packed'` attribute on each of the structure or union members. Specifying the `'-short-enums'` flag on the line is equivalent to specify the `'packed'` attribute on all `'enum'` definitions.

You may only specify this attribute after a closing curly brace on an `'enum'` definition, not in a `'typedef'` declaration, unless that declaration also contains the definition of the `'enum'`.

- `'transparent_union'`

This attribute, attached to a `'union'` type definition, indicates that any function parameter having that union type causes calls to that function to be treated in a special way.

First, the argument corresponding to a transparent union type can be of any type in the union; no cast is required. Also, if the union contains a pointer type, the corresponding argument can be a null pointer constant or a void pointer expression; and if the union contains a void pointer type, the corresponding argument can be any pointer expression. If the union member type is a pointer, qualifiers like `'const'` on the referenced type must be respected, just as with normal pointer conversions.

Second, the argument is passed to the function using the calling conventions of first member of the transparent union, not the calling conventions of the union itself. All members of the union must have the same machine representation; this is necessary for this argument passing to work properly.

Transparent unions are designed for library functions that have multiple interfaces for compatibility reasons. For example, suppose the `'wait'` function must accept either a value of type `'int *'` to comply with Posix, or a value of type `'union wait *'` to comply with the 4.1BSD interface. If `'wait'`'s parameter were `'void *'`, `'wait'` would accept both kinds of arguments, but it would also accept any other pointer type and this would make argument type checking less useful. Instead, `'<sys/wait.h>'` might define the interface as follows:

```
typedef union
{
    int *__ip;
    union wait *__up;
} wait_status_ptr_t __attribute__((transparent_union));
```

```
pid_t wait (wait_status_ptr_t);
```

This interface allows either `int *` or `union wait *` arguments to be passed, using the `int *` calling convention. The program can call `wait` with arguments of either type:

```
int w1 () { int w; return wait (&w); }  
int w2 () { union wait w; return wait (&w); }
```

With this interface, `wait`'s implementation might look like this:

```
pid_t wait (wait_status_ptr_t p)  
{  
    return waitpid (-1, p, __i p, 0);  
}
```

- `unused`

When attached to a type (including a `union` or a `struct`), this attribute means that variables of that type are meant to appear possibly unused. GCC will not produce a warning for any variables of that type, even if the variable appears to do nothing. This is often the case with lock or thread classes, which are usually defined and then not referenced, but contain constructors and destructors that have nontrivial bookkeeping functions.

To specify multiple attributes, separate them by commas within the double parentheses: for example, `__attribute__ ((aligned (16), packed))`.