

TOSHIBA

TLCS-900 Compiler System User's Guide

1st Edition

TOSHIBA Corporation Semiconductor Company

RESTRICTIONS ON PRODUCT USE

- The information contained herein is subject to change without notice. (W11AE-01)
- TOSHIBA is continually working to improve the quality and reliability of its products. Nevertheless, the hardware and/or software incorporated in the TOSHIBA products listed in this document ("TOSHIBA Products") in general can malfunction or fail due to their inherent electrical sensitivity and vulnerability to physical stress. It is the responsibility of the customer, when utilizing TOSHIBA Products, to fully comply with the standards of safety in making safety design for the entire system, and to avoid the situations in which a malfunction or failure of such TOSHIBA Products could cause loss of human life, bodily injury or damage to property.
In developing your designs, please ensure that TOSHIBA Products are used within specified operating ranges as set forth in the specifications for this product, the specifications for the semiconductor devices under evaluation, and any other related information. Also, please keep in mind the precautions and conditions set forth in the "TOSHIBA Semiconductor Reliability Handbook" and "Instruction Manual" or "Operation Manual" that accompany this product and any devices connected to this product.
Please always confirm the latest information of the TOSHIBA Products released on the web page of microcomputer in the web site of TOSHIBA Semiconductor Company.
(<http://www.semicon.toshiba.co.jp/eng/>) (W01AE-01)
- The TOSHIBA Products are intended for usage in the functional evaluation of semiconductor devices. TOSHIBA Products shall not be used for purposes other than functional evaluation, such as for verification of device reliability. The TOSHIBA Products shall not be incorporated this product into customer products. The TOSHIBA Products shall not be converted, disassembled, modified, or used outside its specified operating range of the TOSHIBA Products listed in this document.
- The TOSHIBA Products are intended for the functional evaluation of semiconductor devices that are designed for use in general electronics applications (e.g., computer, personal equipment, office equipment, measuring equipment, industrial robotics, and domestic appliances). These TOSHIBA Products are neither intended nor warranted for usage in equipment that requires extraordinarily high quality and/or reliability or a malfunction or failure of which may cause loss of human life or bodily injury ("Unintended Usage").
Without limiting the generality of the foregoing, unintended Usage include atomic energy control instruments, airplane or spaceship instruments, transportation instruments, traffic signal instruments, combustion control instruments, medical instruments, and all types of safety devices. The TOSHIBA Products shall not be used for Unintended Usage. (W02BE-01)
- The products described in this document shall not be used or embedded to any downstream products of which manufacture, use and/or sale are prohibited under any applicable laws and regulations. (W03AE-01)
- TOSHIBA does not take any responsibility for incidental damage (including loss of business profit, business interruption, loss of business information, and other pecuniary damage) arising out of the use or disability to use the product. (W04AE-01)
- The information contained herein is presented only as a guide for the applications of our products. No responsibility is assumed by TOSHIBA for any infringements of patents or other rights of the third parties which may result from its use. No license is granted by implication or otherwise under any patents or other rights of TOSHIBA or the third parties. (W06AE-02)
- Product names mentioned herein may be trademarks of their respective companies. (W07AE-02)

Preface

Thank you for using Toshiba microcomputer products.

This manual describes how to use the microcomputer development system product you have purchased. Please keep this manual to hand when you use the product.

Toshiba will continue to make every effort to improve our products to better meet the needs of our customers. We will highly appreciate your continued patronage of Toshiba microcomputer products also in future.

- Microsoft®, Windows®, Windows® 2000, Windows® XP, and Windows Vista® are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

prefaceE-03

Technical support

The "readme.txt" file is included with the product package to help you use this product. If you have any further questions regarding the content of this manual, please do not hesitate to contact your local Toshiba sales representative.

Our technical support service is available if you encounter any phenomenon that seems to be faulty while using this product. At your request we will investigate the cause of the phenomenon and report back to you. To use this service, you need to provide us with the data that enables us to reproduce the phenomenon, such as the operation procedure, etc. Please note that we may not be able to deal with a phenomenon that cannot be reproduced.

INDEX

Part 1	Compiler System Overview	1
Chapter 1	Before Using the Compiler System	3
1.1	Explanation of Each Manual	3
1.2	How to Read the Manuals	3
Chapter 2	Outline of System Development	4
2.1	Developing Microcomputer Application Systems	4
Chapter 3	Compiler Mechanism	5
3.1	Compiler Overview	5
Part 2	How to Use the Compiler System	7
Chapter 4	Mechanism for C Program Operation.....	9
4.1	Startup File	9
4.1.1	Preparation for Using Stack	10
4.1.2	Memory Initialization	11
4.1.3	Hardware Initialization	12
4.1.4	Calling the Main Function and Processing after End	12
4.2	SFR Header File	12
4.3	SFR, Vector Definition File	13
4.3.1	SFR Definition.....	14
4.3.2	Interrupt Function Definition	15
4.3.3	Program for Interrupt Processing	15
4.4	Link Command File	15
4.4.1	Memory Definition Part	16
4.4.2	Section Definition Part	17
4.4.3	Symbol Definition Part.....	17
4.5	Standard Library Definition File	17
4.5.1	Preparation for Using Arithmetic Functions and Floating Point Numbers	19
4.5.2	Preparation for Using Abort Function.....	19
4.5.3	Preparation for Using Exit Function	19
4.5.4	Preparation for Using Heap Areas	19
Chapter 5	Variable Type and Function Type.....	21
5.1	Variable Type.....	21
5.2	Function Type	21
Chapter 6	Structures and Bit Fields	23
6.1	Memory Allocation of Structure Members	23
6.2	Changing Alignment of Structure Members.....	23
6.3	Bit Fields Type.....	24
6.4	Memory Allocation of Bit Fields.....	24
6.5	Changing Memory Allocation of Bit Fields	25
Chapter 7	Interrupt Processing.....	26
7.1	Definition Interrupt Function	26
7.2	Definition Interrupt Vector	26
7.3	Enable Interrupt and Disable Interrupt.....	26
7.3.1	__DI(), __EI() and __EI900()	26
7.3.2	#pragma disinterrupt.....	27
Chapter 8	How to use Assembly Language in C Program	28
8.1	Inline Assembly	28
8.1.1	Inline Assembly Format	28
8.1.2	Caution When Using Inline Assembly	28
8.2	Register Pseudo Variables.....	29

8.3	Variable Name and Function Name.....	29
Chapter 9	How to use PIC/PID.....	30
9.1	PIC/PID Outline	30
9.2	PIC/PID Format	30
9.3	How to use PIC/PID	30
Part 3	Checking and Improving Programs	33
Chapter 10	Checking Programs	35
10.1	Confirming Method of Compiling, Assembling, and Linking Results	35
10.1.1	Confirming Compiling Result.....	35
10.1.2	Confirming Assembling Result	35
10.1.3	Confirming Linking Result	35
Chapter 11	Efficient Program Writing Methods	36
Part 4	Caution Items	39
Chapter 12	MCU Specification.....	41
12.1	Select CPU Type.....	41
Chapter 13	Compiler Specification	42
13.1	Source Files Rules	42
13.2	Caution about Compiler	42
13.3	Relation between Debugging Function of IDE and Compiler Optimization	42
13.4	Caution about Assembler	42
13.5	Caution about Library	42
Chapter 14	Error Meanings and Handling Methods.....	44
14.1	Link Errors and How to Handle Them.....	44
Part 5	Appendix	47
Chapter 15	Using by Command Line	49
15.1	Setting Environmental Variables.....	49
15.2	Command and Explanation Thereof.....	49
Chapter 16	Transition to Macro Preprocessor	51
16.1	Case of using Preprocessor	51
16.1.1	Comment.....	51
16.2	Case of using Macroprocessor	51
16.2.1	?include, ?ic	51
16.2.2	?if, ?while, ?repeat	51
16.2.3	Escape Function, Bracket Function.....	52
16.2.4	?reject, ?genonly, ?gen, ?in, ?list, ?maclib, ?nolist, ?out, ?title.....	52
16.2.5	?define	52
16.2.6	?set	53
16.2.7	?eqs, ?nes, ?lts, ?les, ?gts, ?ges.....	53
16.2.8	?substr.....	54
Chapter 17	Specification Change of PIC/PID Function	55
17.1	Specification Change of PIC	55
17.1.1	Method for Specifying PIC Function	55
17.1.2	Section Name which Allocates PIC	55
17.2	Specification Change of PID.....	56
17.2.1	Method for Specifying PID Variable	56
17.2.2	Section Name which Allocates PID	56

Part 1 Compiler System Overview

Chapter 1 Before Using the Compiler System

1.1 Explanation of Each Manual

Three manuals come with this product. We will give an overview of these three manuals.

Compiler System User's Guide

This manual includes specific methods of use of items such as the compiler, assembler, and linker when doing actual application development. It consists of things such as how to allocate variables and functions, how to view information output by an assembler or linker, customization of a source program, how to deal with errors, etc. You can know the specifications in even more detail from this manual by referring to the other two manuals.

C Compiler Reference

This manual includes specifications relating to C language, compiler option contents, and the like.

Assembler Reference

This manual includes specifications relating to assembly language, the linker and link command files, the macropreprocessor, the librarian, and the object converter. They are collectively indicated for every tool.

For TOSHIBA Integrated Development Environment, please refer to IDE help.

1.2 How to Read the Manuals

Here, we will explain the format description rules.

Format Description Rules

[Format Description Example]

```
#pragma section <Section Type> [<Section Name>]
                [<Displacement>|<Start Address>]
```

#pragma section For commands and options, etc., parts that do not have the enclosure symbols or delimiting symbols described hereafter are noted as is in the actual program.

<Section Type> Specifiers enclosed in < > describe character strings or numerical values specified within < > in the actual program.

[<Section Name>] Specifiers enclosed in [] can be omitted in the actual program.

[<Displacement> | <Start Address>]

For specifiers delimited by " | ", specify one of those items in the actual program.

Chapter 2 Outline of System Development

2.1 Developing Microcomputer Application Systems

With system development using microcomputers, the developing software that controls the subject microcomputer occupies an important position. Development tools such as a compiler, assembler, and debugger support this software development. Figure 2-1 below shows development tools suited for development processes.

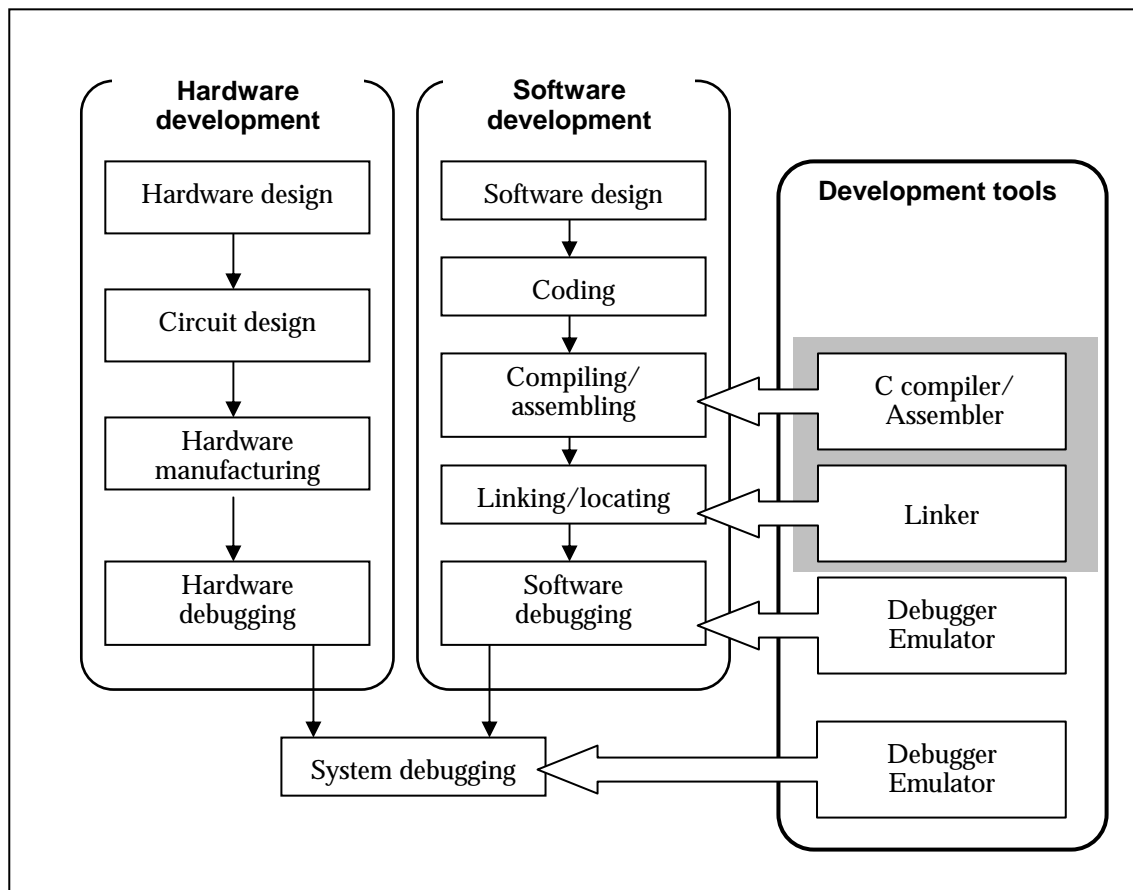


Figure 2-1 Development Tools for Handling System Development Processes

With TLCS-900 Family C Compiler, in the software development part shown in the figure above, the program after coding is compiled, assembled, and linked, and processing is done until an object is generated. The generated object is in either a format that the debugger can use or in a format that can be written to ROM.

Chapter 3 Compiler Mechanism

3.1 Compiler Overview

The compiler process flow is as follows.

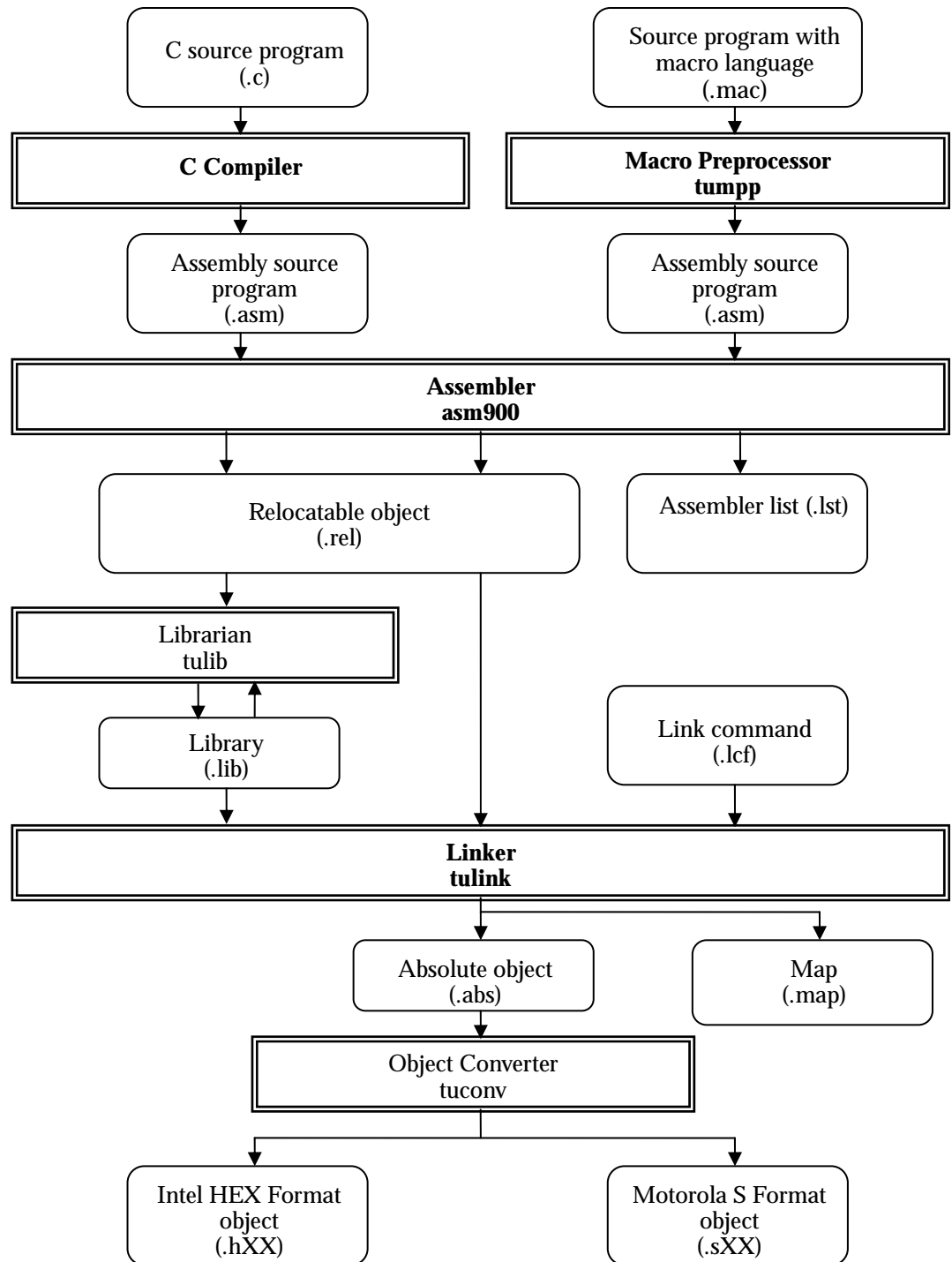


Figure 3-1 Compiler Process Flow

C Compiler

The C compiler handles files that have the suffix `.c` as input files. The C compiler is a tool that converts a C language source file to an assembly source file that uses TLCS-900 Family instruction set.

Assembler

The assembler handles files with the suffix `.asm` as input files. The assembler creates relocatable object files from assembly source files.

Linker

The linker handles files with the suffix `.rel/.lcf/.lib` as input files.

The linker links multiple relocatable object files according to the specifications of the link command file, and is a tool that creates one executable absolute object file. A link command file is a file that defines the allocation on memory and the link sequence of elements such as code and data in a program.

Macro Preprocessor

The macro preprocessor handles files with the suffix `.mac` as input files. It processes macro preprocessor source files that contain macros preprocessor language. As a result, an assembly source file that does not contain macros or conditional assemblies is created.

Librarian

The librarian handles files with the suffix `.rel` as input files. The librarian is a tool for consolidating multiple relocatable object files into one library file.

Object Converter

The object converter handles files with the suffix `.abs` as input files. The object converter is a tool that converts absolute object files to object file that is usable by EPROM writer or the like. The object converter can output Intel HEX and Motorola S Format as object file.

Part 2 How to Use the Compiler System

Chapter 4 Mechanism for C Program Operation

In this section, we will explain operation immediately after reset the CPU using the example of the following files.

- startup file
- SFR header file
- definision file for SFR and vector
- link command file
- definision files for standard library

If TLCS-900 Family C compiler is installed, the sample of these files can be referred to. We do NOT guarantee these sample files. When you use this sample, please confirm the operation.

Explanation of each file is using the sample file of TLCS-900/L1 Series.

4.1 Startup File

A startup file is a file that contains the startup routine.

When CPU is reset, the program is executed from the address written to reset vector. At this time, the program executed first is called startup routine and the preparations for performing main function are made. It is necessary to describe according as the specification of using CPU. Generally items required for startup file are as follows.

- Preparation for using stack
- Memory initialization
- Hardware initialization
- Calling the main function and processing after end
- Program for interrupt processing
- Preparation for using standard libraries

<stc91ml.asm>

```

;*****
;*          Sample Start up Program for TLCS-900/L1 Series          *
;*                      MCU : TMP91FW64FG                          *
;*-----*
;* (C)Copyright TOSHIBA CORPORATION 2009 All rights reserved *
;*****

$MAXIMUM
    module    stc91ml_asm

;=====
; [ External symbol declaration ]
;=====
    extern    medium    _WDMOD
    extern    medium    _WDCR
    extern    large     __FAreaOrg
    extern    medium    __FAreaSize
    extern    large     __FDataAddr
    extern    large     __FDataOrg
    extern    medium    __FDataSize
    extern    large     _main
    extern    medium    __BaseXSP

;=====
; [ Dummy section ]

```

```

; This part is for initialize of f_area and f_data section.
; Do not rewrite this part.
;=====
f_area section data large align=2,2
f_data section data large align=2,2

;=====
; [ Start Up Routine ]
;=====
;
;==[ Caution !! ]=====
; Don't use following instructions before setting XSP.
; ( CALL condition, dst ), ( CALR dst )
; ( LINK dst, num ) , ( POP dst )
; ( POP SR ) , ( PUSH SR )
; ( PUSH src ) , ( RET condition )
; ( RETD num ) , ( RETI )
; ( SWI num ) , ( UNLK dst )
;=====
f_code section code large
public __startup
__startup:

;---[ Disable interrupt ]-----
di

;---[ Disable Watchdog Timer(WDT) ]-----
ldb (_WDMOD),0x00; [for RTE] WDT disable
ldb (_WDCR),0xb1 ;

;---[ Setting Stack Pointer ]-----
ld XSP, __BaseXSP ; For Setting XSP

;---[ RAM clear for f_area section ]-----
ld xde, __FAreaOrg
ld bc, __FAreaSize
or bc, bc
j z, FAR_AREA_2
FAR_AREA_1:
ldb (xde+), 0
sub bc, 1
j nz, FAR_AREA_1
FAR_AREA_2:

;---[ Initialize of f_data section : Using as the need arises. ]--
ld xde, __FDataAddr
ld xhl, __FDataOrg
ld bc, __FDataSize
or bc, bc
j z, FAR_DATA_1
ldirb (xde+), (xhl+)
FAR_DATA_1:

;---[ Jump to main program ]-----
ei
j _main

end

```

4.1.1 Preparation for Using Stack

Before interruption and function call occur, it is necessary to set the value to the stack pointer (XSP). When not having set, unexpected operation of MCU may occur. Generally the final address of RAM is set to stack pointer. The stack area is extended in the direction of low address (numerical value is smaller). " __BaseXSP " of the following example is set up by link command file mentioned later.

<Exerpt from str91ml.asm>

```

;=====
; [ Start Up Routine ]

```

```

;=====
;
;==[ Caution !! ]=====
; Don't use following instructions before setting XSP.
; ( CALL condition, dst ), ( CALR dst )
; ( LINK dst, num ) , ( POP dst )
; ( POP SR ) , ( PUSH SR )
; ( PUSH src ) , ( RET condition )
; ( RETD num ) , ( RETI )
; ( SWI num ) , ( UNLK dst )
;=====
f_code section code large
public __startup
__startup:

; :

;---[ Setting Stack Pointer ]-----
ld XSP, __BaseXSP ; For Setting XSP

```

4.1.2 Memory Initialization

When power is turned on for the hardware or the like, the value of the object allocated in the RAM area is undefined. This is why global variable initialization is required. There are two types of global variables initialization, one is the initialization of global variables without initial value, and the other is the initialization of global variables with initial value.

Initialization of global variables without initial value (area section)

The variable areas without initial value are initialized by setting the RAM area to 0.

With C language specifications, the variable area without initial value must have initial settings of "0", so in the startup rutin, initialization processing is done before the main function is called. `__FAreaOrg` is the top address of RAM area, `__FAreaSize` is the size of RAM area. These variables are setup by link command file mentioned later.

<Exerpt from str91ml.asm>

```

;---[ RAM clear for f_area section ]-----
ld xde, __FAreaOrg
ld bc, __FAreaSize
or bc, bc
j z, FAR_AREA_2
FAR_AREA_1:
ldb (xde+), 0
sub bc, 1
j nz, FAR_AREA_1
FAR_AREA_2:

```

Initialization of global variables with initial values (data section)

The variable areas with initial values (`t_data`, `n_data`, `f_data`) are initialized by transferring the initial values from the ROM area to the RAM area.

The variable areas with initial values must be set the initial values, it specifies by link command file so that initial values is allocated in the ROM area, and variable is allocated in the RAM area. Then, the initial values are set by transferring the initial values that are allocated in the ROM area to the variables that allocated in the RAM area before calling the main function.

The variable areas with initial values are areas defined by section type "data", and normally the section name are declared using `t_data`, `n_data` and `f_data`. Here, we will give an example of `f_data` initialization.

__FDataOrg is the start address of transfer source, __FDataAddr is the start address of transfer destination, and __FDataSize is the number of transmission bytes, these variables are setup by link command file mentioned later.

<Exerpt from str91ml.asm>

```

;---[ Initialize of f_data section : Using as the need arises. ]--
    ld        xde,__FDataAddr
    ld        xhl,__FDataOrg
    ld        bc,__FDataSize
    or        bc,bc
    j         z,FAR_DATA_1
    ldirb     (xde+),(xhl+)
FAR_DATA_1:

```

4.1.3 Hardware Initialization

Hardware initialization is done by setting a suitable value in the Special Function Register (SFR). Specific examples for TLCS-900 Family include watchdog timer settings, system clock settings, and I/O port settings, etc. The required register types and settings differ depending on the application that is created. For the detail of the Special Function Register, see the "data sheet" of each microcomputer.

4.1.4 Calling the Main Function and Processing after End

The main function is called. Here, we will give an example of calling main function after interrupt enabled.

<Exerpt from str91ml.asm>

```

;=====
; [ External symbol declaration ]
;=====
extern    medium    _WDMOD
extern    medium    _WDCR
extern    large     __FAreaOrg
extern    medium    __FAreaSize
extern    large     __FDataAddr
extern    large     __FDataOrg
extern    medium    __FDataSize
extern    large     _main
extern    medium    __BaseXSP

;    :
;    :

;---[ Jump to main program ]-----
    ei
    j         _main

```

4.2 SFR Header File

In SFR header file, each SFR (special function register) define as I/O variables.

This example has described that it can use also for SFR external declaration. When this file is included after "#define IO_MEM 0" in a source file using SFR, it can be used as external declaration.

< io900l1.h>

```

/*****
;*          Sample SFR Header File for TLCS-900/L1 Series          *
;*                               MCU : TMP91FW64FG                  *
;*-----*
;* (C)Copyright TOSHIBA CORPORATION 2009 All rights reserved *

```

```

;*****/
#ifdef IO_MEM
/* This definition is for I/O variable and extern definition */
/* So please do not use the name "IO_MEM".*/
#define EXTERN
#else
#define EXTERN extern
#endif

/**[0x00]*****/
EXTERN unsigned char __io(0x00) P0; /* 0x00: Port0 */
EXTERN unsigned char __io(0x01) P1; /* 0x01: Port1 */
EXTERN unsigned char __io(0x02) P0CR; /* 0x02: Port0 control */
/*--- (0x03) Reserved ---*/
EXTERN unsigned char __io(0x04) P1CR; /* 0x04: Port1 control */
EXTERN unsigned char __io(0x05) P1FC; /* 0x05: Port1 function */
EXTERN unsigned char __io(0x06) P2; /* 0x06: Port2 */
/*--- (0x07) Reserved ---*/
EXTERN unsigned char __io(0x08) P2CR; /* 0x08: Port2 control */
:
:

```

4.3 SFR, Vector Definition File

Generally items required for SFR and vector definition file are as follows.

- SFR definition
- Interrupt function definition
- Program for interrupt processing

< io90011.c>

```

/*****
;*          Sample SFR/Vector File for TLCS-900/L1 Series          *
;*          MCU : TMP91FW64FG                                     *
;*-----*
;* (C)Copyright TOSHIBA CORPORATION 2009 All rights reserved *
;*****/

#define IO_MEM 1
/* This definition is for I/O variable and extern definition */
/* So please do not use the name "IO_MEM".*/

#include "io90011.h"

void _startup(void);

/*=====
[ Dummy function for interrupt ]
=====*/
void __interrupt _Int_dummy(void)
{
}

/*=====
[ Define interrupt table ]
This part must be rewrite.
=====*/
#pragma section const INT_VECTOR
void * const _IntTbl[] = {
    _startup      /* 0xffff00: reset / SWI0 */
, _Int_dummy     /* 0xffff04: SWI1 */
, _Int_dummy     /* 0xffff08: INTUNDEF / SWI2 */
, _Int_dummy     /* 0xffff0c: SWI3 */
, _Int_dummy     /* 0xffff10: SWI4 */
, _Int_dummy     /* 0xffff14: SWI5 */
, _Int_dummy     /* 0xffff18: SWI6 */
, _Int_dummy     /* 0xffff1c: SWI7 */

```

```

,_Int_dummy      /* 0xffff20: NMI */
,_Int_dummy      /* 0xffff24: INTWD */
,_Int_dummy      /* 0xffff28: INT0 */
,_Int_dummy      /* 0xffff2c: INT1 */
,_Int_dummy      /* 0xffff30: INT2 */
,_Int_dummy      /* 0xffff34: INT3 */
,_Int_dummy      /* 0xffff38: INT4 */
,_Int_dummy      /* 0xffff3c: INT5 */
,_Int_dummy      /* 0xffff40: INT6 */
,_Int_dummy      /* 0xffff44: INT7 */
,_Int_dummy      /* 0xffff48: INT8 */
,_Int_dummy      /* 0xffff4c: INT9 */
,_Int_dummy      /* 0xffff50: INT10 */
,_Int_dummy      /* 0xffff54: INTTA0 */
,_Int_dummy      /* 0xffff58: INTTA1 */
,_Int_dummy      /* 0xffff5c: INTTA2 */
,_Int_dummy      /* 0xffff60: INTTA3 */
,_Int_dummy      /* 0xffff64: INTTA4 */
,_Int_dummy      /* 0xffff68: INTTA5 */
,_Int_dummy      /* 0xffff6c: INTTB00 */
,_Int_dummy      /* 0xffff70: INTTB01 */
,_Int_dummy      /* 0xffff74: INTTB10 */
,_Int_dummy      /* 0xffff78: INTTB11 */
,_Int_dummy      /* 0xffff7c: INTTB20 */
,_Int_dummy      /* 0xffff80: INTTB21 */
,_Int_dummy      /* 0xffff84: INTTB30 */
,_Int_dummy      /* 0xffff88: INTTB31 */
,_Int_dummy      /* 0xffff8c: INTTB40 */
,_Int_dummy      /* 0xffff90: INTTB41 */
,_Int_dummy      /* 0xffff94: INTTBOF0 */
,_Int_dummy      /* 0xffff98: INTTBOF1 */
,_Int_dummy      /* 0xffff9c: INTTBOF2 */
,_Int_dummy      /* 0xffffa0: INTTBOF3 */
,_Int_dummy      /* 0xffffa4: INTTBOF3 */
,_Int_dummy      /* 0xffffa8: INTRX0 */
,_Int_dummy      /* 0xffffac: INTTX0 */
,_Int_dummy      /* 0xffffb0: INTRX1 */
,_Int_dummy      /* 0xffffb4: INTTX1 */
,_Int_dummy      /* 0xffffb8: INTRX2 */
,_Int_dummy      /* 0xffffbc: INTTX2 */
,_Int_dummy      /* 0xffffc0: INTSBI0 */
,_Int_dummy      /* 0xffffc4: INTSBI1 */
,_Int_dummy      /* 0xffffc8: INTRTC */
,_Int_dummy      /* 0xffffcc: INTAD */
,_Int_dummy      /* 0xffffd0: INTTC0 */
,_Int_dummy      /* 0xffffd4: INTTC1 */
,_Int_dummy      /* 0xffffd8: INTTC2 */
,_Int_dummy      /* 0xffffdc: INTTC3 */
};
#pragma section const /* return to default */

/*-eof-*/

```

4.3.1 SFR Definition

Each SFR is defined as I/O variables. The following example is defining SFR using SFR header file with "#define IO_MEM 1".

<Exerpt from io900l1.c>

```

#define IO_MEM      1
/* This definition is for I/O variable and extern definition */
/* So please do not use the name "IO_MEM".*/

#include "io900l1.h"

```

4.3.2 Interrupt Function Definition

The interrupt functions are defined. The following example is dummy routine which carries out only a return without processing anything.

<Exerpt from io90011.c>

```
/*=====
[ Dummy function for interrupt ]
=====*/
void __interrupt _Int_dummy(void)
{
}
```

4.3.3 Program for Interrupt Processing

The table which stores interrupt/reset vector is created. The array of function pointer which qualified const define interrupt vector. Reset vector is a start address when MCU has reset, and interrupt vector is a start address of interrupt processing routine. These vector are set to the vector table by function name. Please assign the dummy routine which carries out only a return without processing anything to the interrupts which is not used. The following example, interrupt vector is defined as the section name "INT_VECTOR".

<Exerpt from io90011.c>

```
/*=====
[ Define interrupt table ]
This part must be rewrite.
=====*/
#pragma section const INT_VECTOR
void * const _IntTbl[] = {
    _startup          /* 0xffff00: reset / SWI0 */
    ,_Int_dummy        /* 0xffff04: SWI1 */
    ,_Int_dummy        /* 0xffff08: INTUNDEF / SWI2 */
    ,_Int_dummy        /* 0xffff0c: SWI3 */
    ,_Int_dummy        /* 0xffff10: SWI4 */
    ,_Int_dummy        /* 0xffff14: SWI5 */
    :
    :
    ,_Int_dummy        /* 0xffffc8: INTRTC */
    ,_Int_dummy        /* 0xffffcc: INTAD */
    ,_Int_dummy        /* 0xffffd0: INTTC0 */
    ,_Int_dummy        /* 0xffffd4: INTTC1 */
    ,_Int_dummy        /* 0xffffd8: INTTC2 */
    ,_Int_dummy        /* 0xffffdc: INTTC3 */
};
#pragma section const /* return to default */
```

4.4 Link Command File

A link command file is a file which describes the definition of memory space to use, and the memory allocation of the programs and variables, and the definisiton of gloval symbol. Generally items required for link command file are as follows.

- Memory definition part : Memory space is defined.
- Section definition part : Memory allocation of the programs and variables is defined
- External definition symbol part : External definition symbol is defined.

<c91.lcf>

```

/******
;*      Sample Link Command File for TLCS-900/L1 Series      *
;*      MCU : TMP91FW64FG                                  *
;*      *
;*-----*
;* (C)Copyright TOSHIBA CORPORATION 2009 All rights reserved *
;*****/

memory
{
    IO      : org=0x000000, len=0x001000
    RAM     : org=0x001000, len=0x002000
    EMEM    : org=0x003000, len=0xfdd000
    code.l  : org=0xfe0000, len=0x01fff0
    INTTBL  : org=0xffff00, len=0x0000f0
    RESV2   : org=0xfffff0, len=0x000010
}

sections
{
    far_code    org=0xfe0000 : {*(f_code)}
    far_const   org=org(far_code)+sizeof(far_code) : {*(f_const)}
    far_area    org=0x001000 : {*(f_area)}
    far_data    org=org(far_const)+sizeof(far_const)
                addr=org(far_area)+sizeof(far_area) : {*(f_data)}
    int_table   org=0xffff00 : {*(INT_VECTOR)}
}

__BaseXSP      = 0x002fff;

__FAreaOrg     = org(far_area);
__FAreaSize    = sizeof(far_area);

__FDataAddr    = addr(far_data);
__FDataOrg     = org(far_data);
__FDataSize    = sizeof(far_data);

/*-eof-*/

```

[Caution] If a link is performed without specifying a link command file, it will be allocated sequentially from the address 0 irrespective of a section attribute. Because of this, usually, a link is performed with specifying a link command file.

4.4.1 Memory Definition Part

Memory definition part is described memory composition of the target MCU. For example, "IO : org=0x000000, len=0x001000" becomes the meaning that the memory area "IO" is between 0x000000 and 0x000fff (the size is 0x001000).

<Exerpt from c91.lcf>

```

memory
{
    IO      : org=0x000000, len=0x001000
    RAM     : org=0x001000, len=0x002000
    EMEM    : org=0x003000, len=0xfdd000
    code.l  : org=0xfe0000, len=0x01fff0
    INTTBL  : org=0xffff00, len=0x0000f0
    RESV2   : org=0xfffff0, len=0x000010
}

```

code.l is called Pre-defined memory, and this have meaning of the following.

Pre-defined memory	Displacement	Section type
data.l	far	area / data
data.m	near	
data.s	tiny	
code.l	far	code / const
code.m	near	

4.4.2 Section Definition Part

The section definition part specifies combining the input section, and collecting into an output section, and arranging to memory. The addressing of section is united specification of the memory definition part.

<Exerpt from c91.lcf>

```
sections
{
    far_code      org=0xfe0000 : {*(f_code)}
    far_const     org=org(far_code)+sizeof(far_code) : {*(f_const)}
    far_area      org=0x001000 : {*(f_area)}
    far_data      org=org(far_const)+sizeof(far_const)
                  addr=org(far_area)+sizeof(far_area) : {*(f_data)}
    int_table     org=0xfffff00 : {*(INT_VECTOR)}
}
```

Below, f_area section (far variable without initial value) was summarized to far_area, and it allocates to 0x001000.

```
far_area      org=0x001000 : {*(f_area)}
```

Below, f_data section (far variables with initial value) was summarized to far_data, and the initial value to transmit is allocated to immediatry after far_const, the using variable is allocated to immediatry far_area. When allocating the variables with initial value, the allocating address of the initial value to transmit to org= and the allocating address for the using variables to addr= are specified.

```
far_data      org=org(far_const)+sizeof(far_const)
                  addr=org(far_area)+sizeof(far_area) : {*(f_data)}
```

4.4.3 Symbol Definition Part

A link command file can define external definition symbol. The identifier that are referenced at ROM/RAM transfer in a startup file define in this part.

<Exerpt from c91.lcf>

```
__BaseXSP     = 0x002fff;

__FAreaOrg    = org(far_area);
__FAreaSize   = sizeof(far_area);

__FDataAddr   = addr(far_data);
__FDataOrg    = org(far_data);
__FDataSize   = sizeof(far_data);
```

4.5 Standard Library Definition File

When using standard library files, a required setup is included in the standard library definition file. Do the appropriate preparations for using standard library files.

- Using arithmetic functions and floating point numbers
- Using Heap Areas

<ini91ml.c>

```

/*****
/*      Sample Initial Program for TLCS-900/L1 Series      *
/*      MCU : TMP91FW64FG                                  *
/*-----*
/* (C)Copyright TOSHIBA CORPORATION 2009 All rights reserved *
/*-----*/

#include <stdlib.h>
#include <errno.h>

/*=====*/
/* [ Configuration part ]                                */
/* Following macro value means that:                      */
/* 0 : The function is not used.                          */
/* 1 : The function is used.                              */
/* When you use 'abort' or 'exit' or 'heap area',         */
/* Rewrite each program to fit your use.                 */
/*=====*/
#define __USE_FLOAT 0 /* Change 0 to 1, if use floating point.*/
#define __USE_ABORT 0 /* Change 0 to 1, if use abort function.*/
#define __USE_EXIT 0 /* Change 0 to 1, if use exit function.*/
#define __USE_HEAP 0 /* Change 0 to 1, if use heap area. */

#if __USE_FLOAT
/*=====*/
/* [ Define public variable ]                            */
/* When use floating point, this part is necessary.      */
/* Do not change this part.                             */
/* 'errno' is reserved word. Do not access to this variable. */
/*=====*/
volatile int errno; /* Refer from standard library */

#endif /* __USE_FLOAT */

#if __USE_ABORT
/*=====*/
/* [ Standard library function: abort() ]                */
/* When use abort function, this part is necessary.      */
/* Rewrite this part to fit your program.                */
/*=====*/
void abort(void)
{
    /*=====*/
    /* Write the process before reset, then jump to your start up */
    /* routine.                                                    */
    /*=====*/
    /* __asm(" j __startup");                                     SAMPLE */
    /*=====*/
}
#endif /* __USE_ABORT */

#if __USE_EXIT
/*=====*/
/* [ Standard library function: exit() ]                  */
/* When use exit function, this part is necessary.      */
/* Rewrite this part to fit your program.                */
/*=====*/
void exit(int status)
{
    /*=====*/
    /* Write the process as same as normal program ending.      */
    /*=====*/
    /* __asm(" halt");                                           SAMPLE */
    /*=====*/
}

```

```

#endif    /* __USE_EXIT */

#if __USE_HEAP
/*=====*/
/* [ Define Heap area ]                               */
/*   When use malloc, calloc or realloc, this part is necessary.  */
/*   Rewrite the address and size of Heap area to fit your program.*/
/*=====*/
#define HeapTop      (void*)0x2000 /* Heap area top address */
#define HeapSize 0x800           /* Heap area size */
unsigned long SBRK_break = HeapTop; /* Set Heap area top address */
unsigned long SBRK_size = HeapSize; /* Set Heap area size */
void* _alloca= ((void *)0);        /* Memory management pointer */

#endif    /* __USE_HEAP */

/*-eof-*/

```

4.5.1 Preparation for Using Arithmetic Functions and Floating Point Numbers

When using arithmetic functions and floating point numbers, change "__USE_FLOAT" into 1 and perform symbol "errno" definition.

```

#define __USE_FLOAT 0    /* Change 0 to 1, if use floating point.*/

```

4.5.2 Preparation for Using Abort Function

When using abort function, change variable "__USE_ABORT" into 1, and please rewrite the abort function according to your program.

```

#define __USE_ABORT 0    /* Change 0 to 1, if use abort function. */

```

4.5.3 Preparation for Using Exit Function

When using exit function, change variable "__USE_EXIT" into 1, and please rewrite the exit function according to your program.

```

#define __USE_EXIT 0    /* Change 0 to 1, if use exit function. */

```

4.5.4 Preparation for Using Heap Areas

When using items such as malloc function, calloc function, or realloc function, change variable "__USE_HEAP" into 1.

```

#define __USE_HEAP 0    /* Change 0 to 1, if use heap area.    */

```

(1) Define a heap area in memory

Using the memory definition part in the link command file, an addition needs to define the memory as a heap area.

(2) Define the start address and size of the heap area

Define the start address and size of the heap area. In the example, define symbols "HeapStart" and "HeapSize".

(3) Initialize the parameter to use the heap area

To use a heap area, variables need to be prepared for an address for accessing the heap area and for the area size. The variables that are prepared are as follows.

SBRK_break : Heap area start pointer
SBRK_size : Heap area size
_alloca : Heap area control pointer

For initialization of the heap area start pointer, set the start address of the memory space defined as the heap area. The heap area control pointer is initial set using 0.

<Exerpt from ini91ml.c>

```
#if __USE_HEAP
/*=====*/
/* [ Define Heap area ] */
/* When use malloc, calloc or realloc, this part is necessary. */
/* Rewrite the address and size of Heap area to fit your program.*/
/*=====*/
#define HeapTop (void*)0x2000 /* Heap area top address */
#define HeapSize 0x800 /* Heap area size */
unsigned long SBRK_break = HeapTop; /* Set Heap area top address */
unsigned long SBRK_size = HeapSize; /* Set Heap area size */
void* _alloca=((void *)0); /* Memory management pointer */
#endif /* __USE_HEAP */
```

Chapter 5 Variable Type and Function Type

5.1 Variable Type

far variables	These variables can be allocated to memory area for which displacement is "far". When <code>__far</code> qualifier is specified or the memory qualifier is omitted, it can become far variable, and it can allocate to the area from 0x0 to 0xfffff.
near variables	These variables can be allocated to memory area for which displacement is "near". When <code>__near</code> qualifier is specified, it can become near variable, and it can allocate to the area from 0x0 to 0xffff. When accessing a near variable, it may use the command which the object size becomes small. Therefore, it is possible to make object size small by carrying out the variable of high access frequency to near variable.
tiny variables	These variables can be allocated to memory area for which displacement is "tiny". When <code>__tiny</code> qualifier is specified, it can become tiny variable, and it can allocate to the area from 0x0 to 0xff. When accessing a tiny variable, it may use the command which the object size becomes small. Therefore, it is possible to make object size small by carrying out the variable of high access frequency to tiny variable.
io variables	These are variables using definition SFR. io variables are out of optimization.

5.2 Function Type

`__interrupt` function qualifier

"__interrupt" specifies maskable interrupt function. In interrupt functions, using registers are saved at function entry, and restored at exit, so that register value may be protected. Compiler outputs "reti" to return from interrupt function.

`__regbank` function qualifier

"__regbank" specifies maskable interrupt function. The difference from "__interrupt" function specification is being able to use register bank, and the register which are saved and restored are XIX, XIY and XIZ.

`__inline` function qualifier

When the inline function called, the inline function is unrolled directly to that location.

`__cdecl` function qualifier

The "__cdecl" function is a function type. In this function, arguments are recognized from right side, and all arguments are passed by stack. The default function type is "__cdecl" function. Compiler outputs "ret" to return from "__cdecl" function.

`__cdecl` function qualifier

The "__cdecl" function is a function type. In this function, arguments are recognized from left side, 1st, 2nd and 3rd arguments are passed by register XWA,

XBC and XDE. It passes by stack after the 4th argument. A return value is stored and returned to XHL register. Also larger type than 4 bytes cannot be used for a return value. Compiler outputs "ret" to return from "__cdecl" function, but if arguments are stored to stack, compiler outputs "retl" and stack is released.

__pic qualifier

"__pic" specifies PIC function.

Chapter 6 Structures and Bit Fields

6.1 Memory Allocation of Structure Members

The structure members are allocated in memory in the sequence as they are described in the source file. At this time, required padding is inserted according to alignment of data type, and alignment adjustments are made.

Table 6-1 Structure member type and alignment

Member Type	Alignment (byte)
char / signed char / unsigned char	1
short / signed short / unsigned short	2
int / signed int / unsigned int	2
long / signed long / unsigned long	2
float / double / long double	2
pointer	2
array	Determined by the alignment of element type
structure / union	Determined by the alignment of the maximum size member

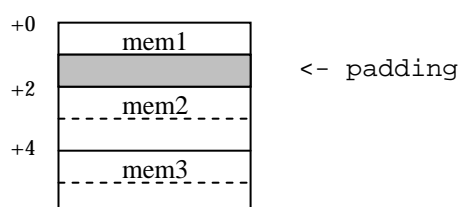
6.2 Changing Alignment of Structure Members

When changing alignment of structure members, there are 2 kinds of methods, add -Xp option and specify #pragma pack directive.

When changing alignment by file unit, compile with -Xp<alignment byte> option. The other methods by block unit, specify #pragma pack(<alignment byte>) directive in a file. 1 or 2 can be specified as <alignment byte>. When compile with -Xp2 option or specify #pragma pack(2) in a file, it becomes the same meaning as default alignment 2 byte.

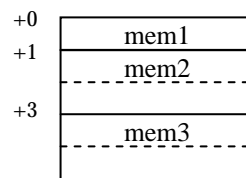
[Example1] default memory allocation

```
struct pack_sample {
    unsigned char    mem1;
    unsigned int     mem2;
    unsigned short   mem3;
};
```



[Example2] memory allocation specified #pragma pack(1)

```
#pragma pack(1)
struct pack_sample {
    unsigned char    mem1;
    unsigned int     mem2;
    unsigned short   mem3;
};
#pragma pack()
```



6.3 Bit Fields Type

The following types can be used as a bit fields type. When it declares without specifying signed and unsigned, it is recognized as signed. And if the signed type specifies, the highest-order bit is recognized as sign bit.

Table 6-2 Bit fields type

Type	Bit count
char / signed char / unsigned char	8
short / signed short / unsigned short	16
int / signed int / unsigned int	16
long / signed long / unsigned long	32

6.4 Memory Allocation of Bit Fields

Bit fields are allocated so that boundaries according to each member type are not extended across.

[Example1] unsigned short int type bit fields

```
struct field1 {
    unsigned short  a:1;
    unsigned short  b:2;
    unsigned short  c:3;
    unsigned short  d:1;
    unsigned short  e:8;
};
```

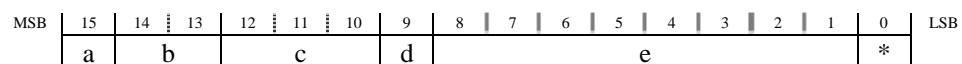


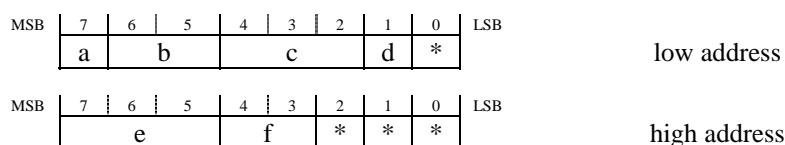
Figure 6-1 Default Bit Image of field1 (* denotes a blank bit)

[Example2] unsigned char type bit fields

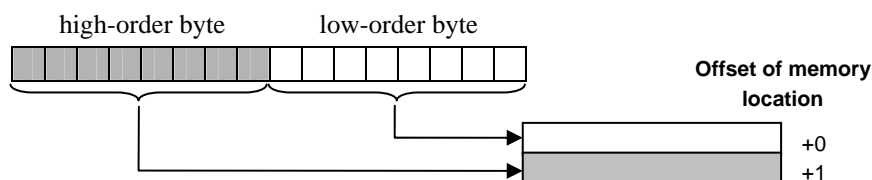
```

struct field2 {
    unsigned char    a:1;
    unsigned char    b:2;
    unsigned char    c:3;
    unsigned char    d:1;
    unsigned char    e:3;
    unsigned char    f:2;
};

```

**Figure 6-2 Default Bit Image of field2** (* denotes a blank bit)

Byte inversion is performed to the data type which size is 2 byte or more, and it is allocated to memory. By performing byte inversion, the low-order byte is allocated to smaller address, and the high-order byte is allocated to bigger address.



6.5 Changing Memory Allocation of Bit Fields

When changing the bit field memory allocation sequence, add the `-Xw` option and compile. With the default, the bit field is allocated from the most significant bit (MSB). When the `-Xw` option is specified, each element of the bit field is allocated from the least significant bit (LSB).

Chapter 7 Interrupt Processing

7.1 Definition Interrupt Function

In order to define maskable interrupt function, "__interrupt" or "__regbank" is specify. Compiler outputs "reti" to return from maskable interrupt function.

```
void __interrupt intr1(void)          /* maskable interrupt */
{
    intVar++;
}

void __regbank(1) regintr1(void)     /* maskable interrupt */
{
    intVar++;
}
```

7.2 Definition Interrupt Vector

In TLCS-900 Family CPU, interrupt vector table is divided into several function pointer (4 byte). When you create a interrupt vector by C program, please define by array of function pointer. The example of description is in 4.3.3 "Program for Interrupt Processing".

7.3 Enable Interrupt and Disable Interrupt

7.3.1 __DI(), __EI() and __EI900()

__DI(), __EI() and __EI900(<level>) are specified in order to perform interrupt-disable/interrupt-enable. __DI(), __EI() and __EI900(<level>) are defined by "stdlib.h" as macro. When you use them, please include "stdlib.h".

```
#include <stdlib.h>                  /* necessary */

void func(void)
{
    int  n;
    int  intVar;
    __DI();                          /* interrupt-disable */
    for(n=0; n<100; n++) {           /* interrupt-disable area */
        intVar += n;                /* : */
    }                               /* : */
    /* skip */                      /* : */
    __EI();                          /* interrupt-enable */
}
```

The command "DI" that means interrupt disable to the place which described "__DI();" is inserted. And, the command "EI" that means interrupt enable to the place which described "__EI();" or "__EI900(<level>);" is inserted. "__EI900(<level>);" can specify the interrupt level.

[Caution] When "__DI();", "__EI();" or "__EI900(<level>);" is described at top of function, "DI"/"EI" is outputted after entrance processing of a function (for example, allocating of

auto variable, or using register saving). Similarly, when "__DI();", "__EI();" or "__EI900(<level>);" is described at end of function, "DI"/"EI" is outputted front exit processing of a function (for example, releasing of auto variable, or using register restoring).

7.3.2 #pragma disinterrupt

"#pragma disinterrupt" is specified in order to perform interrupt-disable whole function. The format of "#pragma disinterrupt" is as follows.

<pre>#pragma disinterrupt([<level>]) <Function Name>[, <Function Name>, ...]</pre>

[Sample]

<pre>#pragma disinterrupt(0) func int func(void) { /* skip */ }</pre>

Chapter 8 How to use Assembly Language in C Program

8.1 Inline Assembly

8.1.1 Inline Assembly Format

There are 2 kinds of inline assembly format, they are `__ASM()` and `__asm()`. For inline assembly `__ASM()` and `__asm()`, the register allocation around these differs.

`__ASM()`

The compiler interprets it as the register to which the value was assigned not being updated, and performs register allocation. Therefore, when the register value is updated in `__ASM()`, a program may not operate correctly.

In the case of the following example, register allocation is not suspended at point (a), and is performed up through point (b).

:		
(a)	/* During this time, register allocation is not divided */	
__ASM("ld WA, 5");	/*	*/
(b)	/*	*/
:		

`__asm()`

After the `__asm()` statement, the compiler interprets that all registers other than the register to be saved by a function call are updated, and it performs register allocation of a variable anew.

In the case of the following sample, register allocation is temporarily divided at point (a), and new register allocation is performed from point (b).

:		
(a)	/* Divide the register allocation here */	
__asm("ld WA, 5");		
(b)	/* Start new register allocation from here */	
:		

8.1.2 Caution When Using Inline Assembly

An advantage when using `__ASM()` is not affecting code efficiency. An advantage when using `__asm()` is being able to use all register freely. Please use `__ASM()` and `__asm()` properly according to the purpose.

■ Detection of error locations

Assembly language source lines described using inline assembly are not checked by the compiler, and are transferred directly to the assembler. When an error is detected in the source line described by inline assembly, the error line is output using the line number within the assembly source file.

The error line can be confirmed when the assembler source file is created with compile -XF option.

- "end" instruction

In inline assembly, the "end" instruction must not be used.

- Restrictions on label use

In inline assembly, when the same label as the label which a compiler generates is defined, a duplicate definition error occurs. The compiler generates a label name such as one where a decimal number continues after "L" or "S", so when using a label, avoid this kind of label name.

8.2 Register Pseudo Variables

TLCS-900 family Compiler is provided with register pseudo variables for doing direct operation of CPU registers in C language. The purpose of doing direct operation of registers is mainly to deliver data with the inline assembly.

When using register pseudo variables, be careful of the following.

- The address operator "&" cannot be used for register pseudo variables.
- It cannot be used as an argument of a function.
- It cannot be used for reference and assignment of register pseudo variables in global.
- It cannot be used the register pseudo variables to some registers.
- If register pseudo variables is used, you have to make it register pseudo variables not have to interfere in the register using evaluation of expression. When you use register pseudo variables, avoid using it by the complicated operation expression. In addition, please check to see assembly source file that is outputted by compiler with -XF option, if the register value become intended result.

8.3 Variable Name and Function Name

The variable which are specified out of function in C source program, their name become the name as an underbar added head of name in assembly source program. And the variable which is specified in function, in order to use stack or register allocation, their name is not used in C source program.

If the function is normaly function without function qualifier (cdecl function) or interrupt function or interrupt_n function or regbank function or regbank_n function, C compiler outputs a function name as an underscore(_) added head of name. If the function is adecl function, C compiler outputs a function name as an period(.) added head of name.

Chapter 9 How to use PIC/PID

9.1 PIC/PID Outline

PIC is the abbreviation for Position Independent Code, it is the code which is able to execute in any location on a memory. PIC can be specified for each function, and the function which was specified as PIC, it is called PIC function.

Meanwhile, PID is the abbreviation for Position Independent Data, it is the data which can be accessed in same code without being dependent in any location on a memory. PID can be specified for each variable, and the variable which was specified as PID, it is called PID variable.

PIC functions and PID variables can be moved on memory while the program is executing, and they can also execute at move destination. Executing PIC functions and accessing to PID variables are performed by using relative address from the initial address of each section. That is, by moving the whole PIC/PID sections, they can perform completely like movement before.

9.2 PIC/PID Format

PIC function Format

The specification method of PIC function specifies `__pic` function qualifier.

It is also necessary to specify `__pic` qualifier at the prototype declaration, the external reference declaration and using a pointer of function.

```
<type specifier> __pic <function name>([<argument>]);  
                                /* Prototype declaration */  
<type specifier> __pic <function name>([<argument>])  
{  
    :  
}
```

PID variable Format

The specification method of PID variable has the method of specifying `__pid` qualifier and setting up individually, and a method of specifying `#pragma pid_on` and `#pragma pid_off`, and specifying two or more variables collectively.

It is also necessary to specify `__pid` qualifier at the external reference declaration.

```
<type specifier> __pid <variable name>;
```

```
#pragma pid_on  
<type specifier> <variable name>;  
<type specifier> <variable name>;  
    :  
#pragma pid_off
```

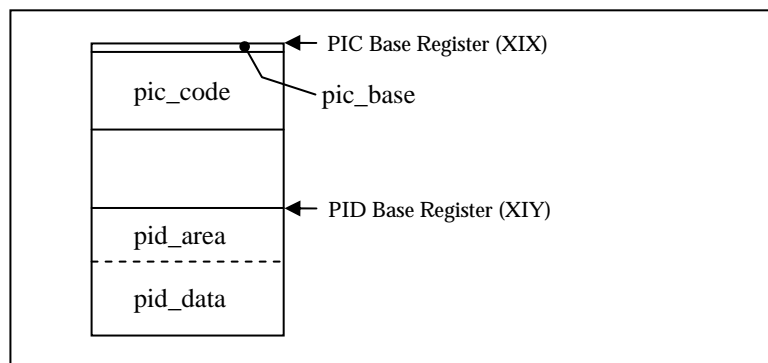
9.3 How to use PIC/PID

In order to use this function, specification of PIC function and PID variable, a setup of the base section and the base register, and a program transmission routine are required.

A setup of the base register is performed with the transmission routine of a program. The base register is certainly set up after transmission of PIC/PID section. The section name into which PIC function and PID variable are summarized, and the base register are as follows.

Table 9-1 Section Name and Base Register used by PIC/PID

Item	Section Name	Base Register
NULL pointe for PIC	pic_base	XIX
PIC function	pic_code	
Variables without initial value which had PID specified	pid_area	XIY
Variables with initial value which had PID specified	pid_data	
Constant (const object) which had PID specified		



[Caution] XIX and XIY register cannot be used for any uses other than the base register in a program using PIC function or PID variable.

The followings are examples for a setup using PIC and transmission routine. The same processing is required also at the case of using PID.

[Example] Flow of examples for a set up using PIC

- Startup File
 - (1) The base register setting.
- Link Command File
 - (2) The link command file setting.
- Transmission Routine
 - (3) Reference of symbol required for a transmission.
 - (4) Disabling interrupt.
 - (5) Code transmission.
 - (6) Enabling interrupt.
 - (7) The base register update.

< Startup File example (st_sample.asm) >

```
; Add the following setting to a startup file
pic_base section code large align=1,1      ;(1)
dw 0                                       ;(1)
```

< Link Command File example (lcf_sample.lcf) >

```
/* Add the following setting (2) to a link command file */
memory
{
    /* skip */
    pic_l : org=0x010000, len=0x010000 /* (2) */
}
```

```

EMEM2      : org=0x020000, len=0xfc0000
code.1     : org=0xfe0000, len=0x01ff00
}

sections
{
    /* skip */
    far_code org=0xfe0000 : {*(f_code)}
    pic_code org=0x020000 : {*(pic_base) *(pic_code)}      /* (2) */
}

/* skip */
__PCodeOrg = org(pic_code);      /* (2) */
__PCodeSize = sizeof(pic_code); /* (2) */

/* PIC destination address */
__dst_PIC = 0x100000;           /* (2) */

```

< Transmission Routine example (trans_sample.c) >

```

void movePIC()
{
    __ASM("extern __dst_PIC, __PCodeOrg, __PCodeSize"); /* (3) */

    __ASM("di"); /* (4) */

    __ASM(" ld XDE, __dst_PIC"); /* (5) */
    __ASM(" ld XHL, __PCodeOrg"); /* (5) */
    __ASM(" ld BC, __PCodeSize"); /* (5) */
    __ASM(" or BC, BC"); /* (5) */
    __ASM(" j z, PIC_CODE_0"); /* (5) */
    __ASM(" ldirb (XDE+), (XHL+)"); /* (5) */
    __ASM("PIC_CODE_0:"); /* (5) */

    __ASM("ei"); /* (6) */
}

int __pic pfunc()
{
    return 1;
}

int main()
{
    int val;

    movePIC();
    __ASM(" ld XIX, __dst_PIC"); /* (7) */

    val = pfunc();
    return val + 1;
}

```


Part 3 Checking and Improving Programs

Chapter 10 Checking Programs

10.1 Confirming Method of Compiling, Assembling, and Linking Results

10.1.1 Confirming Compiling Result

The compiling result can be confirmed with an assembler source file generated by specifying `-XF` option. The following information is output in an assembler source file.

- Compiler version number
- Assembly language corresponding to a C source file

10.1.2 Confirming Assembling Result

The assembling result can be confirmed with an assembler list file generated by specifying `-l` option(suffix `.lst`). The following information is output in an assembler list file.

- Assembler version number and assembler option
- Machine language corresponding to an assembly source file
- List of defined symbols
- Error or Warning message

10.1.3 Confirming Linking Result

The linking results can be confirmed with a map file generated by specifying `-la` option (suffix `.map`). The following information is output in an map file.

- Linker version and link options
- Link command file contents
- Error or Warning message
- List of link target files
- Memory image after linking sections
- Address information allocated to symbols

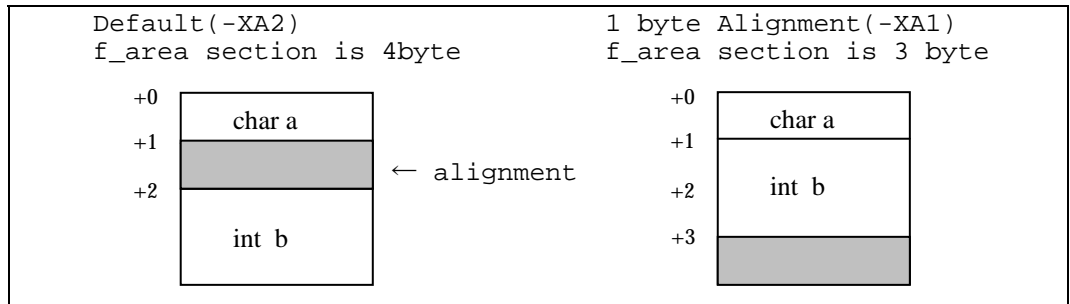
Chapter 11 Efficient Program Writing Methods

In order to improve object efficiency, please take each following item into consideration enough, and design and code it.

- The member of bit-field is set to unsigned char, and it is set as 1 bit.

```
struct bitpattern {  
    unsigned char b7:1;  
    unsigned char b6:1;  
    unsigned char b5:1;  
    unsigned char b4:1;  
    unsigned char b3:1;  
    unsigned char b2:1;  
    unsigned char b1:1;  
    unsigned char b0:1;  
};
```

- Avoid use of signed variables as much as possible.
- Avoid use of long type and floating type variables as much as possible.
- The integer type variables used frequently is allocated to the tiny area or near area.
By allocating integer type variables to tiny area or near area,, speed becomes quick. This setting uses extended qualifier `__tiny` or `__near`. Change of allocation area will also change the section name, therefore, the correction in a section definition part of a link command file or a memory initialization of a startup file may be required.
- It is made for operation of variables with a different type not to occur.
The type conversion is performed by the operation of variables with different type. Therefore, the operation of variables with same type becomes small in object size.
- The shift counter of a shift operation specifies the constant number, without using variables.
The object size will become small if a shift counter is made into a constant number. For the detail of the shift operation, see the "TLCS-900 C Compiler Reference".
- When making structure and array into the argument of function, pointer is passed as argument.
- The recursive function containing many arguments and auto variables is not used.
- -XS option is specified.
The code that the object size becomes small is outputted. Although the memory utilization improves, execution speed may deteriorate.
- -Xec option is specified.
The size of enumerator depends on the range of enumerators, the size of enumerator will become small.
- The alignment of section is set as 1 byte.
The data is allocated by 1 byte alignment, the datas allocate close, and the section size become small. This setting is used -ZA1, -ZD1, and -ZC1 options. However, since data may be allocated at the odd address, execute speed become slow.



Part 4 Caution Items

Chapter 12 MCU Specification

12.1 Select CPU Type

CPU type of TLCS-900 family is specified as this option.

```
-Nb[ <CPU-type> ]
```

<CPU-type> can be specify the value from 0 to 3, a corresponding type is as follows.

Table 12-1 CPU Type

CPU Type	Meaning
0	TLCS-900 series
1	TLCS-900/L series, TLCS-900/L1 series
2	TLCS-900/H series
3	TLCS-900/H1 series

If this option is omitted, it means -Nb1 is specified.

Unless it specifies CPU type, the control register cannot be used correctly, and malfunction may be carried out. Since the control register which can be used and the control register map differ from each CPU type, please be sure to specify it.

For example, in the processing using `__DMAD0` which is a control register pseudo variable, the object codes differ in TLCS-900/L, 900/L1 series and TLCS-900/H1 series. The detail of the control register, see the "TLCS-900 C Compiler Reference", or . "data sheet" of each microcomputer.

[Example1] the example of using `__DMAD0`

```
unsigned long    val1;
void sample()
{
    val1 = __DMAD0;
}
```

[Example1-a] the object code for TLCS-900/L, 900/L1 series (-Nb1)

Object	Source Statement
E82F10	ldc XWA,DMAD0
F200000060	ld (_a1),XWA

[Example1-b] the object code for TLCS-900/H1 series (-Nb3)

Object	Source Statement
E82F20	ldc XWA,DMAD0
	// when DMAS4 is used, E82F10 is outputted
F200000060	ld (_a1),XWA

Chapter 13 Compiler Specification

13.1 Source Files Rules

The end of file is a line feed code.

The end of the file that C source file, assembler source file, or header file included in by each, should finish with a line return. In the case of C language, this is determined in ANSI standard. When the end of a file is NOT a line return, unexpected error may occur, so be sure to insert a line return.

13.2 Caution about Compiler

CPU register mode

The CPU register mode of each CPU of TLCS-900 family is supporting only the maximum mode. The minimum mode is NOT supporting.

13.3 Relation between Debugging Function of IDE and Compiler Optimization

The debugger (IDE) uses as an information source that is the information for the debugger (debugging information) output by the compiler, making source level debugging possible. Debugging information is attached to objects output by the compiler. Because of this, with compiler optimization, when processes and variables are deleted and the execution sequence is changed, the following kind of behavior may occur.

- Can not see the variables with the debugger
- Not executed according to the source program sequence
- Not stopping at the breakpoint that is supposed to be executed
- Stopping at the breakpoint that is not to be executed

13.4 Caution about Assembler

Call C language function from assembly language program

When calling the function described by C language in the assembly language program, it calls using cal instruction or j instruction. The register which was being used in assembly program is not saved with the function of C language. Therefore, save the registers before and behind a function calling.

13.5 Caution about Library

TLCS-900 family C Compiler links the following libraries.

- c900ml.lib** TLCS-900 family library
- c900pic.lib** TLCS-900 family library for PIC

These library files are stored in the lib directory that is in TLCS-900 family C Compiler install directory. For detail of each libraries, see the "TLCS-900 C Compiler Reference".

Reentrant

Reentrancy means being able to guarantee operation when, before a certain function execution is completed, that function is called again. For example, this would be a case of when an interrupt occurs during execution of function `f()`, the same function `f()` is called during that interrupt processing.

A simple example of a function `f()` when reentrance is impossible would be a function that is using a global variable.

Reentrancy is not guaranteed for standard libraries and run time libraries that come with the product.

Chapter 14 Error Meanings and Handling Methods

14.1 Link Errors and How to Handle Them

```
TULINK-Error-231: Section "xxx" at "0xyyyy" load value overflow.
                Truncated
```

Since the value of the external reference symbol currently used within the applicable section won't fit in the object that is trying to embed, this error is occurred. The location which the error has occurred can be identified by the following methods, so please check "the size which can be specified as an operand" or "the size of symbol which can be specified as an operand" of the instruction which the error has occurred.

- (1) Generate a assemble list file(*.lst) and a mapping list file(*.map) with option -l option and -la option at build.
- (2) The address "0xyyyy" of the cause of an error identifies from "Link map" in a mapping list file, which section of which file it is.
- (3) The offset value from the head address of a section to an error occurring address is calculated.
- (4) With reference to the assemble list file of the file leading to an error, an error location is identified with the section specified by (2) based on the offset value calculated by (3).

```
TULINK-Warning-511: Unresolved external symbol "_errno"
TULINK-Error-209: Reference made to unresolved external symbol
                  "_errno"
```

Since "errno" used by error handling when using mathematic function or floating point number is not defined, these errors are occurred.

When using mathematic function or floating point number, please define the int type variable "errno" in a startup routine etc. and include "errno.h" in a file which using function point type. Please refer to "4.5.1 Preparation for Using Arithmetic Functions and Floating Point Numbers" for the example of use in a sample file.

```
TULINK-Error-210: "xxx" at "0xyyyy" won't fit into configured memory
```

Since the area which allocates output section "xxx" is insufficient, this error is occurred.

In section definition part of a link command file, it is specified that it allocates "xxx" section at address "0xyyyy" by allocation address specification (org) or start address specification (addr). When "xxx" section is not contained in the memory area to which address "0xyyyy" corresponds, this error occurs.

Please adjust by specification of area which can be allocated "xxx" section, and change of "xxx" section size.

```
TULINK-Error-211: No space for "xxx" in "yyy"
```

Since the area which allocates output section "xxx" is insufficient, this error is occurred.

In section definition part of a link command file, it is specified that it allocates "xxx" section at "yyy" memory. In this specification, there are a case of output memory is specified and other is allocated to

predefined memory. When "xxx" section is not contained in specified "yyy" memory area, this error occurs.

Please adjust by specification of area which can be allocated "xxx" section, and change of "xxx" section size.

Part 5 Appendix

Chapter 15 Using by Command Line

15.1 Setting Environmental Variables

When using TLCS-900 family C compiler from command line, environmental variables must be set.

Table 15-1 Required Environmental Setting

Environmental Variables	Setting Contents
THOME900	Directory path name in which TLCS-900 family C Compiler is installed.
TMP	Compiler working directory path name.
PATH	The path which attached \bin is added to the path specified by environmental variable THOME900.

We will explain an example in which the installation destination and the working directory are the following directory.

Installation desitnation	C:\Program Files\TOSHIBA\T900
Working directory	C:\TEMP\TOSHIBA\T900

Please set the environmental variables of Windows System as followin.

Decide whether to specify system environmental variables or user environmental variables according to your use format.

path	"C:\Program Files\TOSHIBA\T900\bin";%path%
THOME900	C:\Program Files\TOSHIBA\T900
TMP	C:\TEMP\TOSHIBA\T900

[Caution] When the environmental variable(THOME900) settings are enclosed in double quotation marks, the compiler may not start up correctly. Do not enclose the environmental variable (THOME900) settings in double quotation marks.

Language tools cannot handle path names that include full-width characters. Because of this, multi-byte characters cannot be used in the path name of environmental variables.

15.2 Command and Explanation Thereof

(1) Create an absolute object file

```
cc900 -Nb1 -O3 -g -o sample.abs file1.asm stc96.c linc96.lcf
```

- 1) Assemble file1.asm at TLCS-900/L1 series(-Nb1) and with debugging information(-g), and generate file1.rel.
- 2) Compile stc96.c at TLCS-900/L1 series(-Nb1) and optimization level 3(-O3), with output debugging information(-g), and generate stc96.rel.
- 3) Link file1.asm and stc96.rel according to linc96.lcf, and generate sample.abs(-o sample.abs.)

(2) Create a Intel HEX Format file

```
tuconv sample.abs
```

- 1) Convert sample.abs to Intel HEX file, generate sample.h16.

(3) Create a relocatable object file

```
cc900 -Nb1 -O3 -c -XF file2.c
```

- 1) Compile file2.c at TLCS-900/L1 series(-Nb1) and optimization level 3(-O3), and generate file2.rel(-c). When doing this, leave file2.asm without deleting it(-XF).

(4) Create a library

```
tulib -r x96.lib file2.rel
```

- 1) Create x96.lib, and register file2.rel in the library. When x96.lib exists, update file2.rel to that.

Chapter 16 Transition to Macro Preprocessor

TLCS-900 family C Compiler provided Macro Preprocessor.

This tool have the function both preprocessor (TUAPP) and macroprocessor (TUMPL) in previous version product, however the syntax is different. Therefore, when you use preprocessor or macroprocessor in the development, some modification needs in your source program to transfer for Macro Preprocessor.

For the detail of Macro Preprocessor, see the "TLCS-900 Assembler Reference".

16.1 Case of using Preprocessor

16.1.1 Comment

The comment starting with # after a preprocessing directive is abolished in Macro Preprocessor.

Macro Preprocessor regards the comment starting with # as a part of replace parameters, therefore it will become unexpected result, and no error or no warning is output. When using comment starting with #, modify comment to block comment using "/*" and "*/", or two consecutive slashes (//), or line comment starting with a semicolon (;).

```

; Before replacing :
  #define AAA BBB # comment
                      ;^^^^^^^^^^^^^^^^
                      ;This is regarded as replace parameter

; After replacing :
  #define AAA BBB /* comment */

```

16.2 Case of using Macroprocessor

16.2.1 ?include, ?ic

?include and ?ic function are abolished in Macro Preprocessor.

When using this function, modify this function to #include directive. Macro Preprocessor output a error when ?include or ?ic exist.

```

; Before replacing :
  ?include    <sample1.h>
  ?ic         "sample2.h"

; After replacing :
  #include    <sample1.h>
  #include    "sample2.h"

```

16.2.2 ?if, ?while, ?repeat

?if, ?while and ?repeat syntax are modified in Macro Preprocessor.

When using this function, modify old syntax to new syntax. Macro Preprocessor output a error when ?if, ?while or ?repeat exist.

```
; Before replacing :
?if (<condition>) then (
    AAA
)
else (
    BBB
) fi

; After replacing :
?if (<condition>)
    AAA
?else
    BBB
?endif
```

16.2.3 Escape Function, Bracket Function

The escape function and bracket function are abolished in Macro Preprocessor.

When using this function, modify source program using back slash or string control macro. Macro Preprocessor output a error when these function use.

```
; Before replacing :
?define(DDATA(datalist, name))(
    ?name: dw ?datalist
)

?DDATA(? (0x245, 0x1da, 0x333), tel)

; After replacing :
?macro DDATA datalist, name
    ?name: dw ?substr(?datalist, 0, ?len(?datalist))
?endm

DDATA "0x245, 0x1da, 0x333", tel
```

16.2.4 ?eject, ?genonly, ?gen, ?in, ?list, ?maclib, ?nolist, ?out, ?title

?eject, ?genonly, ?gen, ?in, ?list, ?maclib, ?nolist, ?out and ?title are abolished in Macro Preprocessor.

Macro Preprocessor output a error when these function exist.

16.2.5 ?define

?define function is abolished in Macro Preprocessor.

Currently, it enables to use ?define to consider about compatibility with Macroprocessor and Macro Preprocessor. However, use #define or ?macro in new program, because of this compatibility deprecated in future.

?define have two format as follows.

```
; Format 1

; Befor replacing :
?define(AAA)(BBB)

; After replacing :
#define AAA BBB
```

```

; Format 2

; Before replacing :
?define(AAA(par1, par2))
local lab1 lab2 (
    ?par1
    ?par2
    ?lab1
    ?lab2
    ...
)

?AAA(1, 2) ; call

; After replacing :
?macro AAA par1, par2
labels lab1, lab2
    ?par1
    ?par2
    ?lab1
    ?lab2
    ...
?endm

AAA 1, 2 ; call

```

16.2.6 ?set

?set function is abolished in Macro Preprocessor.

Currently, it enables to use ?set function to consider about compatibility with Macroprocessor and Macro Preprocessor. However, use ?variable in new program, because of this compatibility deprecated in future.

When ?set replace to ?variable, modify only declaration part.

```

; Before replacing :
?set(AAA, 1)
?AAA

; After replacing :
?variable AAA, 1
?AAA

```

16.2.7 ?eqs, ?nes, ?lts, ?les, ?gts, ?ges

?eqs, ?nes, ?lts, ?les, ?gts and ?ges specification are modified in Macro Preprocessor.

In Macroprocessor, if the result of these macro is true, the return value is 0xfffff, and if it is false, the return value is 0x00. However, in Macro Preprocessor, if the result is true, the return value is 1, and if it is false, the return value is 0. When you use these functions, please check about these macro result.

```

; Macroprocessor :
    ?eqs(abc, abc)          // 0x00
    ?nes(ABC,AbC)           // 0xffffffff
    ?lts(ABC,abc)           // 0xffffffff
    ?les(ABCD,ABC)          // 0x00
    ?gts(0x11,0x13)         // 0x00
    ?ges(abcde,abcde)       // 0xffffffff

; Macro Preprocessor :
    ?eqs(abc, abc)          // 0
    ?nes(ABC,AbC)           // 1
    ?lts(ABC,abc)           // 1
    ?les(ABCD,ABC)          // 0
    ?gts(0x11,0x13)         // 0
    ?ges(abcde,abcde)       // 1
```

16.2.8 ?substr

?substr specification are modified in Macro Preprocessor.

In Macroprocessor, the first character of character string is assumed as 1 column. However, in Macro Preprocessor, the first character of character string is assumed as 0 column. When you use this function, please check about this macro result.

```

; Macroprocessor :
    ?substr(abcdefg,8,1)     // NULL
    ?substr(abcdefg,3,0)     // NULL
    ?substr(abcdefg,5,1)     // e
    ?substr(abcdefg,5,100)   // efg

; Macro Preprocessor :
    ?substr(abcdefg,8,1)     // NULL
    ?substr(abcdefg,3,0)     // NULL
    ?substr(abcdefg,5,1)     // f
    ?substr(abcdefg,5,100)   // fg
```

Chapter 17 Specification Change of PIC/PID Function

TLCS-900 family C Compiler can be used PIC/PID function.

A specification change of this function is made from the PIC/PID function of previous version product. Therefore, when you use this function in your development by previous version product, some modification needs in your source program to upgrade our product.

For the detail of this function, see the "TLCS-900 C Compiler Reference".

17.1 Specification Change of PIC

17.1.1 Method for Specifying PIC Function

The method for specifying PIC function was modified.

The following functions and description methods that have been specified by the previous version products was abolished by this specification change.

```
#pragma near_pic_on
#pragma near_pic_off

#pragma far_pic_on
#pragma far_pic_off

__near_pic

__far_pic
```

When specify PIC function, use extended qualifier __pic. In addition, the displacement of PIC functions is only "far" by this specification change, please note.

```
; Before replacing :
#pragma far_pic_on
int fpic_func1(void);          /* prototype declaration(far) */
#pragma far_pic_off

char __far_pic fpic_func2(void) /* function definition(far) */
{
    return 1;
}

; After replacing :
int __pic fpic_func1(void);     /* prototype declaration(far) */

char __pic fpic_func2(void)     /* function definition(far) */
{
    return 1;
}
```

17.1.2 Section Name which Allocates PIC

The section name which allocates PIC functions was changed to "pic_code".

Please correct the files (link command file, etc.) which specifies the section name ("near_pic", "far_pic") outputted with previous version products.

17.2 Specification Change of PID

17.2.1 Method for Specifying PID Variable

The method for specifying PID variable was modified.

The following functions and description methods that have been specified by the previous version products was abolished by this specification change.

```
#pragma near_pid_on
#pragma near_pid_off

#pragma far_pid_on
#pragma far_pid_off

__near_pid

__far_pid
```

When specify PID variable, use extended qualifier __pid or #pragma pid_on / pid_off. In addition, the displacement of PID variables is only "far" by this specification change, please note.

```
; Before replacing :

#pragma far_pid_on
int fpid_vall;      /* far */
#pragma far_pid_off

char __far_pid fpid_val2; /* far */

; After replacing :
#pragma pid_on
int fpid_vall;      /* far */
#pragma pid_off

char __pid fpid_val2; /* far */
```

17.2.2 Section Name which Allocates PID

The section name which allocates PID variables was changed to following.

PID variables without initial value	: pid_area
PID variables with initial value	: pid_data
PID variables added const attribute	: pid_data

Please correct the files (link command file, etc.) which specifies the section name ("near_pid_area", "far_pid_area", "near_pid_data", "far_pid_data") outputted with previous version products.

History

Issue	Date	Update
1st Edition	7 Jan, 2009	1st Edition

TLCS-900 Compiler System User's Guide [1st Edition]

The Date of Issue: 7 Jan, 2009

TDE120-01