LAPIS SEMICONDUCTOR

FEXTCCU8

CCU8
Programming Guide

Program Development Support Software

ISSUE DATE: Apr. 2012

NOTICE

No copying or reproduction of this document, in part or in whole, is permitted without the consent of LAPIS Semiconductor Co., Ltd.

The content specified herein is subject to change for improvement without notice.

The content specified herein is for the purpose of introducing LAPIS Semiconductor's products (hereinafter "Products"). If you wish to use any such Product, please be sure to refer to the specifications, which can be obtained from LAPIS Semiconductor upon request.

Examples of application circuits, circuit constants and any other information contained herein illustrate the standard usage and operations of the Products. The peripheral conditions must be taken into account when designing circuits for mass production.

Great care was taken in ensuring the accuracy of the information specified in this document. However, should you incur any damage arising from any inaccuracy or misprint of such information, LAPIS Semiconductor shall bear no responsibility for such damage.

The technical information specified herein is intended only to show the typical functions of and examples of application circuits for the Products. LAPIS Semiconductor does not grant you, explicitly or implicitly, any license to use or exercise intellectual property or other rights held by LAPIS Semiconductor and other parties. LAPIS Semiconductor shall bear no responsibility whatsoever for any dispute arising from the use of such technical information.

The Products specified in this document are intended to be used with general-use electronic equipment or devices (such as audio visual equipment, office-automation equipment, communication devices, electronic appliances and amusement devices).

The Products specified in this document are not designed to be radiation tolerant.

While LAPIS Semiconductor always makes efforts to enhance the quality and reliability of its Products, a Product may fail or malfunction for a variety of reasons.

Please be sure to implement in your equipment using the Products safety measures to guard against the possibility of physical injury, fire or any other damage caused in the event of the failure of any Product, such as derating, redundancy, fire control and fail-safe designs. LAPIS Semiconductor shall bear no responsibility whatsoever for your use of any Product outside of the prescribed scope or not in accordance with the instruction manual.

The Products are not designed or manufactured to be used with any equipment, device or system which requires an extremely high level of reliability the failure or malfunction of which may result in a direct threat to human life or create a risk of human injury (such as a medical instrument, transportation equipment, aerospace machinery, nuclear-reactor controller, fuel-controller or other safety device). LAPIS Semiconductor shall bear no responsibility in any way for use of any of the Products for the above special purposes. If a Product is intended to be used for any such special purpose, please contact a ROHM sales representative before purchasing.

If you intend to export or ship overseas any Product or technology specified herein that may be controlled under the Foreign Exchange and the Foreign Trade Law, you will be required to obtain a license or permit under the Law.

Windows is a registered trademark of Microsoft Corporation (USA) in USA and other countries, and other product names and company names are trademarks or registered trademarks.

Contents

1 Programming

1.1 Referencing SFRs

The CCU8 compiler package provides header files defining SFR names for individual target microcontrollers.

Reading in the appropriate target header file with the #include preprocessing directive defines variables with absolute addresses for all special function registers (SFRs) available on the target microcontroller.

1.1.1 Target Header File Names

The header file name is derived from the target microcontroller name by replacing the ML prefix with M and adding the file extension .H. M610001.H is the one for the ML610001 microcontroller, for example.

1.1.2 Header File Contents

The following code fragment shows how the target header file defines SFR names using macros.

```
/************************************************
 BIT FIELD DEFINITION
************************************************/

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

/************************************************
        DATA ADDRESS SYMBOLS
************************************************/
#define DSR (*(volatile unsigned char __near *)0xF000)
#define _B_DSR (*(volatile _BYTE_FIELD __near *)0xF000)
#define STPACP (*(volatile unsigned char __near *)0xF008)
#define SBYCON (*(volatile unsigned char __near *)0xF009)
#define _B_SBYCON (*(volatile _BYTE_FIELD __near *)0xF009)
     :
     :
/************************************************
```

```
        END OF DATA ADDRESS SYMBOLS
*************************************************/
/*************************************************
        BIT ADDRESS SYMBOLS
*************************************************/

#define DSR0            (_B_DSR.b0)
#define HLT             (_B_SBYCON.b0)
#define STP             (_B_SBYCON.b1)
     :
     :
/*************************************************
        END OF BIT ADDRESS SYMBOLS
*************************************************/
```

1.1.3 Example Referencing SFRs

The following code fragment accesses SFRs using names from the #included header file.

```
#include <m610001.h> /* include target header file */
void initial_timer(void)
{
    TM0CON0 = 0x08; /* initialize timer control register */
    TM0D = 0x7f;    /* initialize timer data register */
    ETM0 = 1;       /* enable timer interrupt */
    T0RUN = 1;      /* start timer counter */
}
```

1.1.4 volatile Modifier

Target header files feature the ANSI C volatile modifier in each macro defining an SFR name. This Section describes why this modifier is necessary.

Without volatile Modifier

Consider the following sample source code.

```
unsigned char status; /* interrupt service routine changes value */
void proc1(unsigned char);
void fn(void)
{
    status = 0;
    while(status == 0)
        ;       /* wait until interrupt service routine changes status
*/
    proc1(status);
}
```

The intention is to wait until an interrupt service routine sets the variable status to a nonzero value. The

compiler, however, produces the following assembly language code.

```
_fn     :
;;    while(status == 0)
_$L3 :
        bal     _$L3
;;}
```

Because the function initializes the variable status to 0 immediately before the while statement, the compiler assumes that the value is always 0, so optimizes away the test, leaving just the assembly language code for an infinite loop--definitely not what was intended.

Specifying the /Od option suppresses this overenthusiastic optimization, producing the intended operation, but at the cost of increasing the size of every module being compiled.

With volatile Modifier

The solution is to add the volatile modifier.

```
volatile unsigned char status; /* interrupt service routine changes
value */
void proc1(unsigned char);
void fn(void)
{
    status = 0;
    while(status == 0)
        ;       /* wait until interrupt service routine changes status
*/
    proc1(status);
}
```

The volatile modifier before the variable status suppresses variable optimization, thus avoiding the infinite loop. The compiler therefore produces the following assembly language output.

```
_fn     :
;;    status = 0; /* interrupt service routine changes value */
        mov     r0,     #00h
        st      r0,     NEAR _status
;;    while(status == 0)
_$L3 :
        l       r0,     NEAR _status
        beq     _$L3
;;    proc1(status);
        l       r0,     NEAR _status
        b       _proc1
```

1.2 Writing Interrupt Service Routines

The compiler provides two pragmas for writing hardware and software interrupt service routines as C programming language functions: INTERRUPT and SWI, respectively.

Note that these C functions have neither arguments nor return values.

For further details on these two pragmas, refer to the CCU8 User's Manual.

These pragmas have the following syntax.

Pragma	Syntax	Address Range
INTERRUPT	#pragma INTERRUPT *function_name address* [*category*]	0x08 - 0x7E
SWI	#pragma SWI *function_name address* [*category*]	0x80 - 0xFE

Note that, apart from the address ranges available, the syntax is the same for both pragmas.

1.2.1 Interrupt Nesting Disabled

To disable interrupt nesting in an interrupt service routine, specify 1 in the INTERRUPT or SWI pragma category field.

Calling the embedded function __EI() inside such an interrupt service routine produces an error message from the compiler.

C Source Code

```
static void intr_fn_0A(void);
#pragma interrupt intr_fn_0A 0x0A 1
volatile unsigned short TM1msec;
static void intr_fn_0A(void)
{
    TM1msec++;
}
```

The sample source code disables interrupt nesting in the interrupt service routine intr_fn_0A().

The compiler converts the above to the following assembly language code.

Assembly Language Output

```
_intr_fn_0A     :
        push    er0
;;    TM1msec++;
        l       er0,    NEAR _TM1msec
        add     er0,    #1
        st      er0,    NEAR _TM1msec
;;}
        pop     er0
        rti
```

This type of interrupt service routine starts by saving to the stack any registers (here only ER0) that it may alter. It ends with an RTI instruction.

The next example is an interrupt service routine that calls another function.

C Source Code

```
static void intr_fn_10(void);
#pragma interrupt intr_fn_10 0x10 1
void func(void);
static void intr_fn_10(void)
{
    func();
}
```

Assembly Language Output

```
_intr_fn_10     :
        push    lr, ea
        push    r0
        l       r0,     DSR
        push    r0
        push    qr0

;;    func();
        bl      _func

;;}
        pop     qr0
        pop     r0
        st      r0,     DSR
        pop     r0
        pop     ea, lr
        rti
```

Calling another function from an interrupt service routine adds considerable overhead to the output code, increasing the processing time. This overhead is necessary because the compiler does not know which registers the function (func()) actually alters, so must bracket the call with potentially redundant PUSH and POP instructions for saving all such registers that might be altered.

Note

Please do not enable an interrupt in the calling another function from a disabled interrupt nesting function in the interrupt service routine.

If the interrupt is enabled and occurs, the running program may go out of control.

1.2.2 Interrupt Nesting Enabled

To enable interrupt nesting in an interrupt service routine, specify 2 in the INTERRUPT or SWI pragma category field. Omitting the category field also enables interrupt nesting.

If interrupt nesting is enabled, the interrupt service routine can call the embedded function __EI().

C Source Code

```
static void intr_fn_20(void);
volatile unsigned short TM2msec;
#pragma interrupt intr_fn_20 0x20 2
static void intr_fn_20(void)
{
    __EI();  /* enable interrupt nesting */
    TM2msec++;
    __DI();  /* disable interrupt nesting */
}
```

The sample source code enables interrupt nesting in the interrupt service routine intr_fn_20().

The compiler converts the above to the following assembly language code.

Assembly Language Output

```
_intr_fn_20     :
        push    elr, epsw
        push    er0
;;    __EI();  /* enable interrupt nesting */
        ei
;;    TM1msec++;
        l       er0,    NEAR _TM2msec
        add     er0,    #1
        st      er0,    NEAR _TM2msec
;;    __DI();  /* disable interrupt nesting */
        di
;;}
        pop     er0
        pop     psw, pc
```

This type of interrupt service routine differs from the non-nesting type in two places. It starts by saving the contents of ELR and EPSW to the stack. It ends with a POP PSW, PC instruction instead of RTI.

1.3 Assigning Variables to Absolute Addresses

The normal procedure for assigning a variable to a specific region is with an ABSOLUTE pragma. This procedure is, however, not available for a variable specified with an NVDATA pragma, so we must give the following separate procedures for the two cases.

For further details on the ABSOLUTE and NVDATA pragmas, refer to the CCU8 User's Manual.

1.3.1 Variable Not Specified with NVDATA Pragma

The following illustrates the procedure for assigning variables not modified with const to absolute addresses.

C Source Code

```
#pragma absolute near_data_0_8000h 0x8000
int __near near_data_0_8000h = 10;
#pragma absolute far_data_2_1000h 2:0x1000
int __far far_data_2_1000h = 100;
```

Note how assigning such variables to absolute addresses in physical segment #1 and higher requires the __far modifier.

The compiler converts the above to the following assembly language code.

Assembly Language Output

```
            rseg $$content_of_init
            mov     er0,    #10
            st      er0,    NEAR _near_data_0_8000h

            mov     r0,     #064h
            mov     r1,     #00h
            st      er0,    FAR _far_data_2_1000h

            public _far_data_2_1000h
            public _near_data_0_8000h

            dseg #00h at 08000h
    _near_data_0_8000h :
            ds      02h

            dseg #02h at 01000h
    _far_data_2_1000h :
            ds      02h
```

The compiler output for a non-const variable specified with an ABSOLUTE pragma is an absolute DATA segment.

If the declaration includes initialization, the compiler adds the code for initializing the variable to the relocatable CODE segment $$content_of_init. The start-up routine calls this initialization code.

The following illustrates the procedure for assigning variables modified with const to absolute addresses.

C Source Code

```
#pragma absolute near_table_0_4000h 0x4000
const int __near  near_table_0_4000h = 20;
```

```
#pragma absolute far_table_1_8000h 1:0x8000
const int __far far_table_1_8000h = 200;
```

Assembly Language Output

```
        public _far_table_1_8000h
        public _near_table_0_4000h

        tseg #00h at 04000h
_near_table_0_4000h :
        dw      014h

        tseg #01h at 08000h
_far_table_1_8000h :
        dw      0c8h
```

The compiler output for a const variable specified with an ABSOLUTE pragma is an absolute TABLE segment.

1.3.2 Variable Specified with NVDATA Pragma

The ABSOLUTE pragma is not available for variables specified with an NVDATA pragma. Assigning such variables to absolute addresses requires a different procedure.

C Source Code

```
#pragma nvdata near_nvdata1
unsigned char __near near_nvdata1[8] = {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07
    };
#pragma nvdata near_nvdata2
unsigned char __near near_nvdata2[8] = {
    0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17
    };
#pragma nvdata far_nvdata1
unsigned char __far far_nvdata1[8] = {
    0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27
    };
#pragma nvdata far_nvdata2
unsigned char __far far_nvdata2[8] = {
    0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37
    };
```

The first step is to specify NVDATA pragmas for all such variables as shown above. To ensure that these NVDATA variables appear in the assembly file in the declared order, specify an initial value for each one. The order is not guaranteed for variables without initial values. The compiler converts the above to the following assembly language code.

Assembly Language Output

```
$$NNVDATAsample segment nvdata 2h #0h
$$FNVDATAsample segment nvdata 2h any

        public _near_nvdata1
        public _near_nvdata2
        public _far_nvdata1
        public _far_nvdata2

        rseg $$NNVDATAsample
_near_nvdata1 :
        db      00h
        db      01h
        db      02h
        db      03h
        db      04h
        db      05h
        db      06h
        db      07h
_near_nvdata2 :
        db      010h
        db      011h
        db      012h
        db      013h
        db      014h
        db      015h
        db      016h
        db      017h

        rseg $$FNVDATAsample
_far_nvdata1 :
        db      020h
        db      021h
        db      022h
        db      023h
        db      024h
        db      025h
        db      026h
        db      027h
_far_nvdata2 :
        db      030h
        db      031h
        db      032h
        db      033h
        db      034h
```

```
        db      035h
        db      036h
        db      037h
```

If the C source code file is sample.c, the compiler assigns NVDATA variables with the __near modifier to the relocatable NVDATA segment $$NNVDATAsample and those with the __far modifier to the relocatable NVDATA segment $$FNVDATAsample. These segment names are necessary when assigning variables specified with NVDATA pragmas to absolute addresses with the RLU8 linker /NVDATA command line option.

To assign $$NNVDATAsample to the address 0:0A000H and $$FNVDATAsample to 3:8000H, for example, use the following command line option.

```
/NVDATA($$NNVDATAsample-0:0A000H $$FNVDATAsample-3:8000H)
```

To produce the same results with IDEU8, choose Project -> Options -> Target, select the Segments tab in the Target Options dialog box that appears, select the check box for Assign specified segment preferentially to NVDATA, and enter the following into the segment specification box.

```
$$NNVDATAsample-0:0A000H $$FNVDATAsample-3:8000H
```

1.4 Near and Far

Physical segment #0 differs from higher ones with regard to data access. Accessing data in physical segment #0 requires only a 16-bit address. Accessing that in physical segments #1 and higher is a two-step process specifying first the 8-bit physical segment number with a DSR prefix instruction and then the 16-bit offset within that segment. Data access is thus less efficient.

The compiler therefore distinguishes between near access limited to physical segment #0 and far access covering the entire data memory space, between near data in physical segment #0 and far data anywhere in the data memory space, and between 2-byte near pointers accessing near data and 3-byte far pointers accessing far data.

1.4.1 __near and __far Modifiers

Specifying whether variables are near or far is the job of the __near and __far modifiers. They are therefore also called data access specifiers.

For further details on these modifiers, refer to Section 4.4.1 "Memory Model Modifiers" in the CCU8 Language Reference.

Examples

```
int __near near_var;
int __far far_var;
```

near_var is treated as near data, so is assigned to physical segment #0. far_var, in contrast, is treated as far data, so the compiler can assign it to any physical segment.

Examples

```
int __far * __near far_pointer;
int __near * __far near_pointer;
```

far_pointer is treated as a far pointer to an object of type int. Because the pointer itself is near, however, it is assigned to physical segment #0.

near_pointer is treated as a near pointer to an object of type int. Because the pointer itself is far, however, the compiler can assign it to any physical segment.

1.4.2 /near and /far Command Line Options

The /near command line option tells the compiler to treat all data without a data access specifier (__near or __far) as near data; /far, as far.

Consider the following C language source code.

C Source Code

```
int a;
int *b;
void fn(void)
{
    a = *b;
}
```

The following table compares the results of compiling the above C language source code with the two command line options.

Assembly language output with /near command line option	Assembly language output with /far command line option

`_fn :`
`  push  bp`
`;;  a = *b;`
`  l  bp, NEAR _b`
`  l  bp, [bp]`
`  st  bp, NEAR _a`
`;;}`
`  pop bp`
`  rt  public _fn`
`  _a comm data 02h #00h`
`  _b comm data 02h #00h`
	`_fn :`
	`  push  bp`
	`;;  a = *b;`
	`  l  bp, FAR _b`
	`  l  r0, FAR _b+02h`
	`  l  er0,  r0:[bp]`
	`  st  er0,  FAR _a`
	`;;}`
	`  pop bp`
	`  rt`
	`  public _fn`
	`  _a comm data 02h ANY`
	`  _b comm data 03h ANY`

Using /near causes the compiler to interpret the variables a and b as having the following declarations.

```
int __near a;
int __near * __near b;
```

Variable a becomes near data; variable b, a near pointer. Both a and b are assigned to physical segment #0.

Using /far produces the following declarations.

```
int __far a;
int __far * __far b;
```

Variable a becomes far data; variable b, a far pointer. The compiler can assign a and b to any physical segment.

Note that /far produces larger code and thus slower program execution. We therefore advise using near data variables to the furthest extent possible by using the compiler's /near command line option and resorting to far data only when user application program specifications require too many variables to fit within physical segment #0 or the linker cannot fit all variables within physical segment #0. To force particular variables to far data, use the __far modifier or the NEAR and FAR pragmas described in the next section.

1.4.3 NEAR and FAR Pragmas

The NEAR and FAR pragmas are for switching the default data access specifier inside a source code file.

The following is an example of how they are used.

Example

```
int a, b;       /* Treatment depends on /near and /far command
                   line options */
#pragma near
char nbuf[16];  /* Treated as NEAR data regardless of any
                   /near or /far command line option */
#pragma far
char fbuf[16];  /* Treated as FAR data regardless of any
                   /near or /far command line option */

char __near * strcpy_nn(char __near *s1, char __near *s2);
char __far * strcpy_ff(char __far *s1, char __far *s2);
void fn(void)
{
    a = b;
#pragma near
    strcpy_nn(nbuf, ”near string”);
                /* String is treated as NEAR data regardless of any
                   /near or /far command line option */
#pragma far
    strcpy_ff(fbuf, ”far string”);
                /* String is treated as FAR data regardless of any
                   /near or /far command line option */
```

```
}
```

Using the /far command line option converts the above to the following assembly language code.

```
        type (u8)
        model small, far
        $$NTABsample segment table 2h #0h
        $$FTABsample segment table 2h any
        $$NCODsample segment code 2h #0h
        rseg $$NCODsample
_fn     :
        push    lr
;;    a = b;
        l       er0,    FAR _b
        st      er0,    FAR _a
;;    strcpy_nn(nbuf, "near string");
        mov     r2,     #BYTE1 OFFSET $$S1
        mov     r3,     #BYTE2 OFFSET $$S1
        mov     r0,     #BYTE1 OFFSET _nbuf
        mov     r1,     #BYTE2 OFFSET _nbuf
        bl      _strcpy_nn
;;    strcpy_ff(fbuf, "far string");
        mov     r0,     #SEG $$S2
        push    r0
        mov     r0,     #BYTE1 OFFSET $$S2
        mov     r1,     #BYTE2 OFFSET $$S2
        push    er0
        mov     r0,     #BYTE1 OFFSET _fbuf
        mov     r1,     #BYTE2 OFFSET _fbuf
        mov     r2,     #SEG _fbuf
        bl      _strcpy_ff
        add     sp,     #4
;;}
        pop     pc

        public _fn
        _a comm data 02h ANY
        _b comm data 02h ANY
        _fbuf comm data 010h ANY
        _nbuf comm data 010h #00h
        extrn code near : _strcpy_ff
        extrn code near : _strcpy_nn
        extrn code near : _main

        rseg $$NTABsample
$$S1 :
        DB      "near string", 00H
```

```
        rseg $$FTABsample
$$S2 :
        DB      "far string", 00H

        end
```

1.5 Data Sizes and Boundary Alignment

This Section gives, for each data type, the object size and the alignment in memory.

1.5.1 Scalars

Data Type	Size (bytes)	Boundary Alignment Unit (bytes)
char	1	1
unsigned char	1	1
short	2	2
unsigned short	2	2
int	2	2
unsigned int	2	2
long	4	2
unsigned long	4	2
enum	2	2
float	4	2
double	8	2
Near pointer	2	2
Far pointer	3	2
SMALL model pointer to function	2	2
LARGE model pointer to function	3	2

1.5.2 Arrays

1.5.2.1 Arrays of Type char

For an array of type char, the compiler simply reserves the specified number of bytes.

Example

```
char odd_arr1[7];
char odd_arr2[9] = {0, 1, 2};
```

The compiler converts the above to the following assembly language code.

```
        _odd_arr1 comm data 07h #00h
        rseg $$NINITTAB
        db      00h
        db      01h
        db      02h
        dw      00h
        dw      00h
        dw      00h
        rseg $$NINITVAR
_odd_arr2 :
        ds      09h
```

The array size for odd_arr1 is seven bytes; for odd_arr2, nine.

1.5.2.2 Arrays of Type Other than char

For arrays of type other than char, the compiler forces word alignment by padding elements with an odd number of bytes with an extra byte.

Consider the following array of far (3-byte) pointers.

```
char __far * fparr[5] = {”apple”, ”banana”, ”cherry”, };

        rseg $$NINITTAB
        dw      OFFSET ($$S3) ;; initial offset portion of fparr[0]
        db      SEG ($$S3)    ;; segment number
        align
        dw      OFFSET ($$S4) ;; initial offset portion of fparr[1]
        db      SEG ($$S4)    ;; segment number
        align
        dw      OFFSET ($$S5) ;; initial offset portion of fparr[2]
        db      SEG ($$S5)    ;; segment number
        align
        dw      00h
        dw      00h
        dw      00h
        dw      00h

        rseg $$FTABdbl
$$S3 :
        DB      "apple", 00H
$$S4 :
```

```
        DB      "banana", 00H
$$S5 :
        DB      "cherry", 00H

        rseg $$NINITVAR
_fparr :
        ds      014h
```

Because a far pointer occupies three bytes, the compiler pads each element in the array with an extra byte to make sure that each far pointer falls at an even-numbered address. sizeof(fparr), the total size of the array, therefore, is the (far pointer size + padding) times the number of elements or (3 + 1) * 5 = 20 bytes.

Note, however, that sizeof(fparr[n]), the element size is still three bytes.

1.5.3 Structures

The compiler stores structure members upwards from the starting address in the order that they are declared, inserting padding as necessary to align the next member with a memory boundary appropriate to its type.

The following are the rules for inserting extra bytes.

(1) The compiler inserts a byte to prevent an odd offset for a multibyte member.

(2) The compiler inserts a byte at the end of a multibyte structure with an odd number of bytes. In other words, apart from single-byte structures, the structure size is always even.

```
struct st {
    int  i;
    long l;
    char c;
} var = {10, 20, 30};
```

The compiler converts the above to the following assembly language code.

```
        rseg $$NINITTAB
        dw      0ah   ;; initial value for var.i
        dw      014h  ;; initial value for var.l
        dw      00h   ;; (continued)
        db      01eh  ;; initial value for var.c
        align

        rseg $$NINITVAR
_var :
        ds      08h
```

This structure has a size of eight bytes. There is an extra byte after member c, of type char, to adjust the size.

The next example shows how changing member order can change the structure size.

```
struct st1 {
    int     a;
    char    b;
    int     c;
    char    d;
} stvar1;
```

With this order, the compiler inserts one byte between members b and c and another after member d. As a result, stvar1 has a size of eight bytes.

```
struct st2 {
    char    b;
    char    d;
    int     a;
    int     c;
} stvar2;
```

This order assigns the two members of type char next to each other. There is no need for padding, so stvar2 is only six bytes long.

1.5.4 Bit Fields

The compiler offers a choice of type specifiers for bit fields: unsigned char and unsigned int. The type specifier determines the space allocation unit: byte (sizeof(char)) or word (sizeof(int)).

1.5.4.1 Type Specifier unsigned char

Using unsigned char for bit field members allocates space in byte increments. The compiler packs adjacent members into the same byte if they fit.

```
struct bit8{
    unsigned char    b0 : 1;  /*  7                    0  */
    unsigned char    b1 : 1;  /* +--+--+--+--+--+--+--+--+ */
    unsigned char    b2 : 1;  /* |b7|b6|b5|b4|b3|b2|b1|b0| */
    unsigned char    b3 : 1;  /* +--+--+--+--+--+--+--+--+ */
    unsigned char    b4 : 1;
    unsigned char    b5 : 1;
    unsigned char    b6 : 1;
    unsigned char    b7 : 1;
} bit_field;
```

If a member overflows the byte boundary, the compiler allocates a new byte for it.

```
struct bitA{
    unsigned char    b0 : 1;  /*  7                    0  */
    unsigned char    b1 : 1;  /* +--+--+--+--+--+--+--+--+ */
    unsigned char    b2 : 1;  /* |b7|b6|b5|b4|b3|b2|b1|b0| */
    unsigned char    b3 : 1;  /* +--+--+--+--+--+--+--+--+ */
    unsigned char    b4 : 1;  /* |  |  |  |  |  |  |b9|b8| */
    unsigned char    b5 : 1;  /* +--+--+--+--+--+--+--+--+ */
    unsigned char    b6 : 1;
    unsigned char    b7 : 1;
    unsigned char    b8 : 1;
    unsigned char    b9 : 1;
} bit_field;
```

Using unsigned char as the type specifier boosts code efficiency.

1.5.4.2 Type Specifier unsigned int

Using unsigned int for bit field members allocates space in word (2-byte) increments. The compiler packs adjacent members into the same word if they fit.

```
struct bit8{
    unsigned int     b0 : 1;  /*  7                    0  */
    unsigned int     b1 : 1;  /* +--+--+--+--+--+--+--+--+ */
    unsigned int     b2 : 1;  /* |b7|b6|b5|b4|b3|b2|b1|b0| */
    unsigned int     b3 : 1;  /* +--+--+--+--+--+--+--+--+ */
    unsigned int     b4 : 1;  /* |  |  |  |  |  |  |  |  | */
    unsigned int     b5 : 1;  /* +--+--+--+--+--+--+--+--+ */
    unsigned int     b6 : 1;  /* Compiler allocates two bytes */
    unsigned int     b7 : 1;  /* at a time */
} bit_field;
```

If a member overflows the word boundary, the compiler allocates a new word for it.

1.5.5 Unions

The size of a union is the number of bytes required to hold the largest member. Note, however, that the compiler pads the union so that multibyte unions always have even sizes.

Example 1

```
union union_tag {
    char x[3];
    int  y;
    char z;
} unvar;
```

In this example, the largest member is the array x, with three bytes. The compiler therefore reserves four bytes to keep the size even.

Example 2

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

union union_tag {
    unsigned char   uc;
    BIT_FLD         bf;
}un2;
```

In this example, union members uc and bf are both one byte long, so the union size is also one byte.

1.6 Linking with Assembly Language

1.6.1 Referencing External Functions and Variables

Unless declared static, all functions and global variables are automatically public names visible to other modules, including assembly language ones.

```
const char array[] = ”string”;
int gvar;
void func(void)
{
    /* function body */
}
```

The compiler converts the above to the following assembly language code.

```
    rseg $$NCODgsym
_func   :
;;}
    rt

    public _func
    public _array
    _gvar comm data 02h #00h
    extrn code near : _main

    rseg $$NTABgsym
_array :
```

```
DB  "string", 00H
```

Note that the compiler adds a leading underscore (_) to the names array, gvar, and func. It also makes the function _func and array _array public with PUBLIC directives and the variable _gvar communal with a COMM directive. These directives allow other files to reference these three names.

Each external name has an attribute called the usage type. The usage types here are CODE for _func, TABLE for _array, and DATA for _gvar.

For further details on usage types, refer to Section 2.4.5 "Usage and Segment Types" in the MACU8 Assembler Package User's Manual.

The following Table summarizes usage types for functions and variables.

Type	Usage Type
Function name	CODE
Variable name modified with const	TABLE
Variable name not modified with const	DATA
Variable name specified with NVDATA pragma	NVDATA

Assembly language source code referencing such external names must first declare them with EXTRN directives. These directives must also specify the usage type.

The following are the assembly language directives for referencing _func, _array, and _gvar from the above example.

```
extrn code :_func
extrn table : _array
extrn data : _gvar
```

1.6.2 Using Registers Inside Functions

Assembly language modules linked with C programming language modules must observe the compiler's conventions for register use.

1.6.2.1 Registers Used by Compiler

The compiler assigns particular roles to the following general-purpose and control registers when generating assembly language code.

Registers	Use
R0, R1, R2, R3	Function arguments, function return value, and work space
R4, R5, R6, R7, R8, R9, R10, R11, R12, R13	Local variables and work space

Registers	Use
FP (ER14)	Base pointer for function arguments assigned to stack and local variables
SP	Stack operations (adjusting stack size and allocating space for local variables)
DSR	FAR data access
EA	Data access to memory

1.6.2.2 Registers Preserved Across Functions

The compiler specifies that each function internally preserve the contents of the certain registers under the following conditions.

Type of Function	Condition	Operation(s) Necessary
General functions	Function calls other function	Save LR to stack
	Function uses R4 to R15	Save all such registers used to stack
Hardware and software interrupt service routines	Function enables nested interrupts	Save ELR and EPSW to stack
	Function calls other function	Save LR, EA, DSR, and XR0 to stack
	Function uses R0 to R15, DSR, or EA	Save all such registers used to stack

The above specifications allow general function calls to destroy the contents of registers R0 to R3, DSR, and EA, but require that they preserve the contents of R4 to R15. They also guarantee the contents of all general-purpose and control registers (except for backup registers) over hardware and software interrupts.

Note, however, that the called function must also internally preserve the contents of the stack pointer (SP) because the compiler assumes that the contents of this control register after executing the function are the same as when the function was called. There is no such requirement on the program status word (PSW), in contrast, because the compiler assumes that the called function modifies its contents.

1.6.3 Calling Functions

Any assembly language code calling C language functions or replacing the relevant portions of one must follow the conventions that the compiler adopts for passing parameters and return values between C language functions.

Let us start by examining the CCU8 assembly language output for such a function call before detailing these conventions.

C Source Code

```
char __near nbuf[64];
char __far  fbuf[64];

int func(char __near *npt, char __far *fpt);

void main(void)
{
    int res;
    res = func(nbuf, fbuf);  /* function call */
}

int func(char __near *npt, char __far *fpt)
{
    int cnt;

    for (cnt = 0; *fp != 0; cnt++)
    {
        *npt++ = *fpt++;
    }
    *npt = '¥0';
    return  cnt;
}
```

We focus on two sections from the CCU8 assembly language output for the above C language source code: the call to function func() and the function itself.

CCU8 Assembly Language Output for Call

```
_main   :
        mov     fp,     sp
        add     sp,     #-02

;;      res = func(nbuf, fbuf);
        mov     r0,     #SEG _fbuf          ;;
        push    r0                          ;;
        mov     r0,     #BYTE1 OFFSET _fbuf  ;;
        mov     r1,     #BYTE2 OFFSET _fbuf  ;; push second parameter
        push    er0                         ;; onto stack

        mov     r0,     #BYTE1 OFFSET _nbuf  ;; load first parameter
        mov     r1,     #BYTE2 OFFSET _nbuf  ;; into register

        bl      _func                       ;; call function

        add     sp,     #4                  ;; adjust stack pointer
```

```
        st      er0,    -2[fp]              ;; load return value
;;}
_$$end_of_main :
        bal     $
```

The call sequence consists of the following steps.

(1) Set up function parameters, if any, as prescribed in Section 1.6.4 " Passing Function Parameters".

(2) Call the function.

(3) Move the stack pointer over the function parameters, if any, on the stack.

(4) Access the return value in the register or specific location prescribed in Section 1.6.5 " Function Return Values."

CCU8 Assembly Language Output for Function Body

```
_func   :
        ;; Insert PUSH LR here if function calls others.
        push    fp              ;; copy FP to stack before modifying it

        mov     fp,     sp      ;; copy current SP to FP

        add     sp,     #-02    ;; point to local variable on stack

        push    bp              ;; save copies of registers used by
        push    er8             ;; this function to stack

        mov     er8,    er0     ;; copy first parameter into ER8

;;      for (cnt = 0; *fpt != 0; cnt++)
        mov     er0,    #0
        bal     _$L8
_$L4 :
        ;; (omitted)

;;      return  cnt;
        l       er0,    -2[fp]  ;; load return value
;;}
        pop     er8             ;; restore registers from stack
        pop     bp              ;;

        mov     sp,     fp      ;; restore stack pointer to contents
                                ;; just after entry point

        pop     fp              ;; restore FP
```

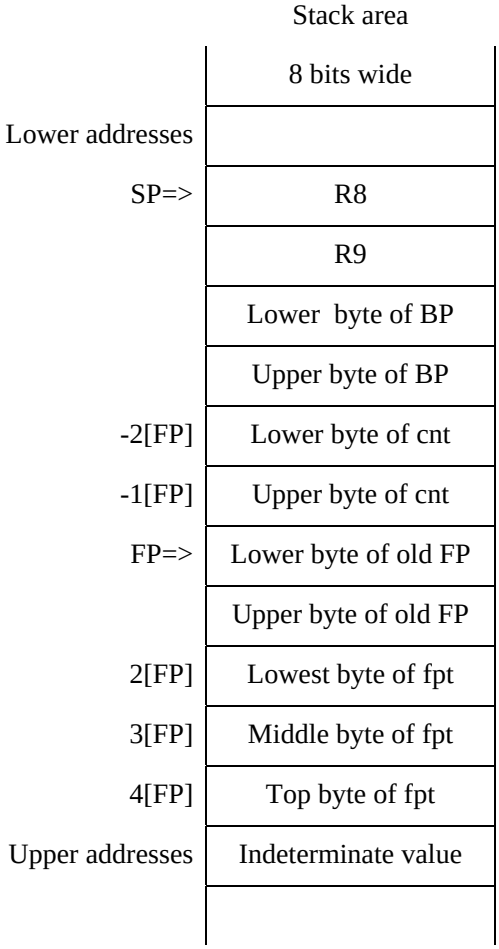
```
rt  ;; If function starts with PUSH LR,
    ;; return with POP PC instead.
```

The function body must incorporate the following steps in the order given.

(1) If the function is a node function, calling others, save a copy of the current link register (LR) contents to the stack. A leaf function, one which does not, can skip this step.

(2) Set up the frame pointer (FP) used to access parameters passed on the stack and local variables, if any. Note that the function allocates space for local variables by subtracting a constant from the stack pointer. (Here, the instruction is "add sp, #-02.")

(3) Save onto the stack any registers that the function must preserve as prescribed in Section 1.6.2 " Using Registers Inside Functions." The accompanying Figure shows the stack frame or stack state for the above source code.



Stack Frame Structure Inside Function func()

(4) Perform function's internal processing.

(5) If the function returns a value, set it up.

(6) Restore any registers saved to the stack.

(7) Restore the stack pointer (SP) and frame pointer (FP) to the contents that they had just after the entry point.

(8) If the function starts with PUSH LR (node function), return with POP PC. Otherwise (leaf function), return with RT instead.

1.6.4 Passing Function Parameters

There are two ways to pass function parameters: in registers and on the stack.

(1) Function with __noreg

A function modified with __noreg takes all its parameters from the stack.

The procedure for passing function parameters on the stack appears below in Section 1.6.4.2 "Passing Function Parameters on Stack."

(2) Function with variable number of parameters

Even without the __noreg modifier, a function with a variable number of parameters takes all its parameters from the stack.

The procedure for passing function parameters on the stack appears below in Section 1.6.4.2 "Passing Function Parameters on Stack."

(3) Other functions

Functions other than the above use the stack for parameters that cannot be passed in registers as described below in Section 1.6.4.1 " Passing Function Parameters in Registers."

The procedure for passing function parameters on the stack appears below in Section 1.6.4.2 "Passing Function Parameters on Stack."

1.6.4.1 Passing Function Parameters in Registers

The registers available for passing parameters are R0, R1, R2, and R3. They hold parameters of the following types: char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, float, and pointer. Other types and any additional parameters of the listed types left over after all registers have been used are passed on the stack.

Moving through the parameters from left to right, the compiler assigns as many parameters as possible to registers R0 through R3 using the following rules.

(1) The compiler assigns a parameter of type char or unsigned char to R*n* (*n* = 0, 1, 2 or 3).

(2) The compiler assigns a parameter of type short, unsigned short, int, or unsigned int to ER*n* (*n* = 0 or 2), loading the lower byte into register R*n* and the upper byte into R*n*+1.

(3) The compiler assigns a parameter of type long, unsigned long, or float to XR0, loading the lowest byte into register R0, the second byte into R1, the third byte into R2, and the top byte into R3.

(4) The compiler assigns a near pointer to ER*n* (*n* = 0 or 2), loading the lower byte into register R*n* and the upper byte into R*n*+1.

(5) The compiler assigns a far or huge pointer to R2:ER0, loading the lower byte of the offset into register

R0, the upper byte into R1, and the physical segment number into R2.

Example 1

```
void fn1(char a, char b, int c);
```

The compiler passes parameter a in R0, parameter b in R1, and parameter c in ER2.

Example 2

```
void fn2(char a, int b, char c);
```

The compiler passes parameters a and b in R0 and ER2, respectively. There is no room for parameter c in the registers, so it is passed on the stack.

Example 3

```
void fn3(char __far *fp, char a);
```

The compiler passes parameters fp and a in R2:ER0 and R3, respectively.

Example 4

```
void fn4(char a, char __far *fp);
```

The compiler passes parameter a in R0. There is no room for parameter fp in the registers, so it must go on the stack.

Example 5

```
void fn5(char a, char __far *fp, char b, int c);
```

The compiler passes parameters a, b, and c in R0, R1, and ER2, respectively. Parameter fp must go on the stack because there is no room for it in the registers.

1.6.4.2 Passing Function Parameters on Stack

(1) The compiler pushes these parameters onto the stack from right to left.

(2) The compiler pushes the individual bytes in decreasing address order.

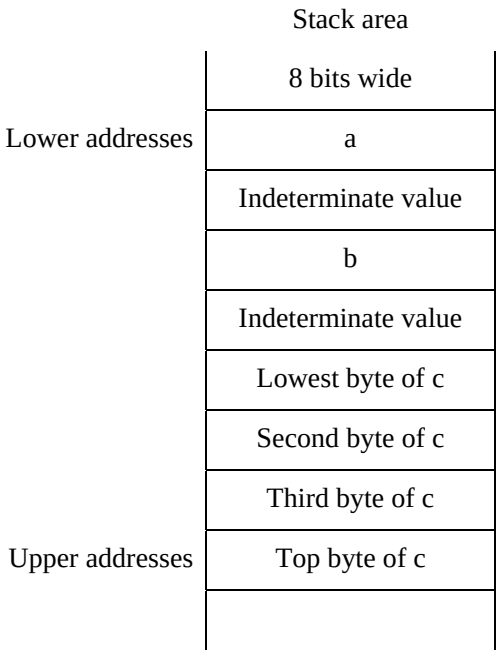
(3) If the parameter has an odd number of bytes, the compiler pushes an extra byte with indeterminate contents onto the stack.

Example

```
void __noreg fn(char a, char b, long c);
```

The __noreg modifier in this example tells the compiler to pass all function parameters on the stack. The compiler therefore pushes them onto the stack in the order c, b, a.

The following stack image results.



1.6.5 Function Return Values

If the return value fits within four bytes, the compiler uses registers R0 through R3. (See Table below.) Otherwise (double, structure, or union), the compiler passes, as the first function parameter, an address for storing the return value.

1.6.5.1 Returning Values in Registers

The following Table summarizes the rules for returning values in registers.

Type	Size	Registers
char, unsigned char	1 byte	R0
short, unsigned short, int, unsigned int	2 bytes	R1:R0 (ER0)
long, unsinged long, float	4 bytes	R3:R2:R1:R0 (XR0)
Near pointer	2 bytes	R1:R0 (ER0)
Far or huge pointer	3 bytes	R2:R1:R0 (R2:ER0)

1.6.5.2 Returning a Double, Structure, or Union

A double, structure, or union does not fit within four bytes, so the compiler passes, as the first function parameter, an address for storing the return value.

Consider the following code.

Example

```
double __near ndbl;
```

```
double __far fdbl;
double dbl_func(void);
void fn(void)
{
    ndbl = dbl_func();
    fdbl = dbl_func();
}
double dbl_func(void)
{
    static double dbl_var;
    return dbl_var;
}
```

The pointer passed to the function depends on whether the caller variable to receive the return value represents near data or far.

The C caller expands to the following assembly language code.

```
_fn     :
        push    lr
        push    fp
        mov     fp,     sp
        add     sp,     #-08
        push    r8

;;    ndbl = dbl_func();
        ;; if caller variable represents NEAR data, pass its address
        mov     r0,     #BYTE1 OFFSET _ndbl
        mov     r1,     #BYTE2 OFFSET _ndbl
        bl      _dbl_func       ;; _dbl_func writes return value
                                ;; directly to _ndbl

;;    fdbl = dbl_func();
        ;; if caller variable represents FAR data, pass address of
        ;; local variable for this function
        mov     er0,    fp
        add     er0,    #-8
        bl      _dbl_func       ;; _dbl_func writes return value
                                ;; to -8[FP]
        lea     -8[fp]
        l       qr0,    [ea]    ;; load return value from stack
                                ;; (-8[FP])
        mov     r8,     #SEG _fdbl
        lea     OFFSET _fdbl
        st      qr0,    r8:[ea] ;; store return value in far variable
;;}
        pop     r8
```

```
        mov     sp,     fp
        pop     fp
        pop     pc
```

If the caller variable represents near data, the compiler passes its address to the function, so the called function writes the return value directly to the specified variable--ndbl, in this example.

If the caller variable represents far data, however, the compiler takes a more convoluted approach. It allocates the necessary space on the stack for a caller local variable and passes the function that address instead. When the called function returns, the caller must then copy the return value from its temporary location in that local variable to the intended caller variable.

The above example features a return value of type double. The procedures for structures and unions are similar.

The body of the called function takes the following form.

```
_dbl_func       :
        push    er8
        mov     er8,    er0
;;    return dbl_var;
        lea     OFFSET _$ST0
        l       qr0,    [ea]
        lea     [er8]           ;; load destination address
                                ;; specified by first parameter
        st      qr0,    [ea]    ;; copy return value to specified
                                ;; destination
;;}
        pop     er8
        rt
```

1.6.6 Indirect Function Calls

The treatment of indirect function calls depends on the memory model. The SMALL model uses 2-byte function pointers, which fit into the 16-bit register used for indirect calls. The LARGE model, however, uses 3-byte ones which are too large, so the compiler uses a workaround involving an emulation library routine instead.

Example

C Source Code

```
void (*fp)(void);
void f(void)
{
    fp();
}
```

The sample source code calls a function indirectly by dereferencing fp, a pointer to a function. Compiling this code with the SMALL and LARGE models produces markedly different assembly language output.

Assembly Language Output (SMALL Model)

```
;;  fp();
    l       er0,    NEAR _fp
    bl      er0
```

The SMALL model uses 2-byte pointers, so calling the function is as easy as loading the contents of fp in the word register ER0 and branching with the bl instruction.

Assembly Language Output (Large model)

```
;;    fp();
    l       r0,    NEAR _fp+02h
    push    r0
    l       er0,   NEAR _fp
    push    er0
    bl      __indru8lw ;; emulation library routine
                       ;;(single line: POP PC)
    ;; <= (A)
```

The sample source code calls a function indirectly by dereferencing fp, a pointer to a function. In the LARGE model, this is a 3-byte pointer. The workaround pushes the contents of this pointer onto the stack and calls the emulation library routine __indru8lw. Substituting POP PC, the only instruction in the function, does not work because that does not modify the contents of the link register (LR), the key step in returning properly to the point marked (A) in the listing.

1.6.7 Writing Assembly Language Called from C

A relatively painless way to write an assembly language function to call from C programs is to first write the function in C, compile it, and modify the assembly language output.

Sample C Program

```
int gi1, gi2;
unsigned char gc;
int retval;
int function(int arg1, unsigned char arg2)
{
    volatile int local = gi1;   /* access local variable */
    gi2 = arg1;                 /* access argument arg1 */
    gc = arg2;                  /* access argument arg2 */
    return retval;              /* specify return value */
}
```

The sample source code covers the key issues: accessing local variables and arguments and specifying a return value for the function. Compiling it produces the following assembly language output. Note that the compile options must be the same as for other modules.

Sample Assembly Language Output

```
_function       :         ;; function name starts with underscore (_)
        push    fp              ;; set up stack frame for accessing
        mov     fp,     sp      ;; variables assigned to stack
        add     sp,     #-02    ;; allocate space for local variables
        push    er8             ;; save er8 to stack because it is
                                ;; used by this function
        mov     er8,    er0     ;; copy argument arg1 into er8

;; ********** function body starts here **********
;;    volatile int local = gi1;   /* access local variable */
        l       er0,    NEAR _gi1
        st      er0,    -2[fp]

;;    gi2 = arg1;                 /* access argument arg1 */
        st      er8,    NEAR _gi2

;;    gc = arg2;                  /* access argument arg2 */
        st      r2,     NEAR _gc

;;    return retval;              /* specify return value */
        l       er0,    NEAR _retval

;; ********** function body ends here **********
;;}
        pop     er8             ;; prepare to return to caller
        mov     sp,     fp      ;; ditto
        pop     fp              ;; ditto
        rt                      ;; return to caller

        public _function        ;; in the absence of static specifier,
                                ;; function is public
```

Examining the compiler output reveals the procedures for accessing local variables and arguments and specifying a return value. All that remains is to modify the assembly language code for the body of the function.

1.7 Important Programming Notes

1.7.1 Necessity for Prototype Declarations

Function prototypes play a key role. Omitting them risks faulty program operation.

Consider the following example.

```
char __near nbuf[32];
```

```
char __far  fbuf[32];
int  gi;
long gl;
void fn(void)
{
    subfunc(nbuf, gi);    /* [1] */
    subfunc(fbuf, gl);    /* [2] */
}
```

Compiling the above produces the following output.

```
_fn     :
        push    lr

;;    subfunc(nbuf, gi);    /* [1] */
        l       er2,    NEAR _gi
        mov     r0,     #BYTE1 OFFSET _nbuf
        mov     r1,     #BYTE2 OFFSET _nbuf
        bl      _subfunc

;;    subfunc(fbuf, gl);    /* [2] */
        l       er0,    NEAR _gl
        l       er2,    NEAR _gl+02h
        push    xr0
        mov     r0,     #BYTE1 OFFSET _fbuf
        mov     r1,     #BYTE2 OFFSET _fbuf
        mov     r2,     #SEG _fbuf
        bl      _subfunc
        add     sp,     #4

;;}
        pop     pc
```

The source code provides no function prototype for the function subfunc(), so the compiler skips type checking for its arguments and return value.

The first call to subfunc() has as its arguments a near pointer and a value of type int; the second, a far pointer and a value of type long. Examining the compiler output reveals that these two calls produce clearly different assembly language code. The resulting program therefore does not operate as intended.

We therefore strongly the use of function prototypes to prevent such problems with the passing of function arguments.

```
void subfunc(char __near *, int);
void fn(void)
{
    subfunc(nbuf, gi);    /* [1] */
    subfunc(fbuf, gl);    /* [2] */
}
```

Adding the prototype produces a warning message from the compiler for the second call because the argument types do not match those in the the prototype.

Note that specifying the compiler /Zg command line option generates a list of prototypes for all functions defined in the file.

Consider the following sample C source code.

```
/* file name test.c */
int int_fn(int a, int b)
{
    return a+b;
}

long long_fn(long a, long b)
{
    return a+b;
}

double double_fn(double a, double b)
{
    return a+b;
}

void *voidp_fn(void)
{
    return (void *)0x8000;
}
```

Compiling it with the /Zg command line option produces a prototype list file.

Sample Prototype List File Output (test.pro)

```
extern  int int_fn(int a,int b);
extern  long long_fn(long a,long b);
extern  double double_fn(double a,double b);
extern  void *voidp_fn(void);
```

Adding to the source code an #include preprocessing directive with the name of this prototype list file as its operand enables type checking for all arguments and return values.

Note

The /Zg option requires that, if an argument or return value is of type structure, union, enumeration, or pointer to same, the source code must declare that structure, union, or enumeration with a tag name.

Consider the following examples.

```
/* file name test2.c */
typedef struct { /* structure without tag name */
    int     memb1;
```

```
    int     memb2;
} ST1;

typedef struct st2 { /* structure with tag name */
    int     memb1;
    int     memb2;
    int     memb3;
} ST2;

void fn1(ST1 *pST1)
{
    pST1->memb1 = 10;
    pST1->memb2 = 20;
}

void fn2(ST2 *pST2)
{
    pST1->memb1 = 100;
    pST2->memb2 = 200;
}
```

The /Zg option produces the following prototype list file.

```
extern  void fn1(struct  *pST1); /* missing tag name means improper
output */
extern  void fn2(struct st2 *pST2);
```

As this example shows, omitting the tag name from a structure, union, or enumeration declaration prevents the /Zg option from producing correct function prototypes.

2 Compiling and Linking

2.1 Segment Names Generated by Compiler

The compiler distributes executable code and data from the C language source code among the following relocatable segments.

In CCU8 Ver.3.30, The generated segment name of the segment type ‘CODE’ in having /Zc command line option or not are different.

For further details, refer to the CCU8 User's Manual.

Segment Name	Segment Type	Physical Segment Attribute	Contents
$$*funcname*$*filename*	CODE	#0	Near functions (compiled with SMALL model) or Interrupt service routines [Only /Zc command line option not specified.]
$$*funcname*$*filename*	CODE	ANY	Far functions (compiled with LARGE model) [Only /Zc command line option not specified.]
$$NCOD*filename*	CODE	#0	Near functions (compiled with SMALL model) [Only /Zc command line option specified.]
$$FCOD*filename*	CODE	ANY	Far functions (compiled with LARGE model) [Only /Zc command line option specified.]
$$INTERRUPTCODE	CODE	#0	Interrupt service routines. [Only /Zc command line option specified.]
$$TAB*constname*$*filename*	TABLE	#0	near variables modified with const (static global variables and static local variables) [Only /Zc command line option not specified.]
$$TAB*constname*$*filename*	TABLE	ANY	far variables modified with const (static global variables and static local variables) [Only /Zc command line option not specified.]
$$NTAB*filename*	TABLE	#0	near variables modified with const (static global variables and static local variables) [Only /Zc command line option specified.]

Segment Name	Segment Type	Physical Segment Attribute	Contents
$$FTAB*filename*	TABLE	ANY	far variables modified with const (static global variables and static local variables) [Only /Zc command line option specified.]
$$NVAR*filename*	DATA	#0	Uninitialized near variables not modified with const (static global variables and static local variables)
$$FVAR*filename*	DATA	ANY	Uninitialized far variables not modified with const (static global variables and static local variables)
$$FTAB*filename*	TABLE	ANY	far variables modified with const (static global variables and static local variables)
$$NINITVAR	DATA	#0	Initialized near variables not modified with const (global variables, static global variables, and static local variables)
$$NINITTAB	TABLE	ANY	Initial values for above
$$FINITVAR*filename*	DATA	ANY	Initialized far variables not modified with const (global variables, static global variables, and static local variables)
$$FINITTAB*filename*	TABLE	ANY	Initial values for above
$$init_info	TABLE	ANY	Initialization parameter table for initialized variables not modified with const
$$NNVDATA*filename*	NVDATA	#0	near variables specified with NVDATA pragmas
$$FNVDATA*filename*	NVDATA	ANY	far variables specified with NVDATA pragmas
$$content_of_init	CODE	Depends on memory model specified to compiler	Code for initializing variables specified with ABSOLUTE pragmas and not modified with const

funcname is a function name, *constname* is a const variable name, *.filename* is the base name of the C

source code file.

The linker's /CODE, /DATA, /TABLE, and /NVDATA command line options allow the programmer to assign the above relocatable segments to specific address regions. Compiling file1.c and file2.c using the LARGE model and specifying the following command line options, for example, assigns the executable code from those files starting at the address 1:8000H.

```
/CODE($$FCODfile1-1:8000h) /COMB($$FCODfile1 $$FCODfile2)
```

The /COMB command line option applies only to segment types CODE and TABLE.

The compiler converts uninitialized global variables not modified with const to communal symbols. There is no linker command line option for assigning communal symbols to specific address regions.

The following Table shows the relocatable segments that the compiler generates for actual C language source code.

C Source Code	Assembly Language Output
int __near gi_ram1;	The compiler converts uninitialized global variables not modified with const to communal symbols and assigns them to RAM. _gi_ram1 comm data 02h #00h
int __near gi_ram2 = 10;	The compiler reserves space for initialized global variables in RAM and assigns the initial values to ROM. The start-up routine copies the latter to the former at the start of the user application program. rseg $$NINITTAB dw 0ah rseg $$NINITVAR _gi_ram2 : ds 02h
const int __far gi_rom;	The compiler assigns global variables modified with const to ROM. rseg $$FTABsample _gi_rom : dw 00h

C Source Code	Assembly Language Output

int f(int, int);
void main(void)
{
  f(10, 20);
}
int f(int a, int b)
{
  return a + b;
}
	The compiler assigns executable code to ROM.
	
	  rseg $$NCODsample
	_main :
	;; f(10, 20);
	  mov er2, #20
	  mov er0, #10
	  bl _f
	;;}
	_$$end_of_main :
	  bal $
	
	_f :
	;; return a + b;
	  add er0, er2
	;;}
	  rt

2.2 Program Execution Flow

Execution of a user application program compiled with the CCU8 compiler package goes through the following steps.

1. Power on reset triggered by RESET pin input
2. Execution of start-up routine ($$start_up)
3. Branch to main() function
4. Execution of user application program

After a reset, and before the execution of the actual user application program, the registers and RAM used must be properly initialized. This initialization is performed by the start-up routine.

The CCU8 compiler provides a start-up routine containing the basic operations in the form of a start-up file. The CCU8 compiler package also provides the source code so that the user can customize this routine. This document gives the procedures for customizing this code.

2.3 main() Function

When the CCU8 compiler compiles a file containing a main() function, it inserts code storing a pointer to the start-up routine in the reset vector at address 2H so that a reset with RESET pin input automatically executes the code in the start-up file.

The start-up routine branches to the main() function.

The CCU8 compiler inserts code that produces an endless loop when the main() function returns.

2.4 Files Required by Linker

Building a module to execute on the target microcontroller requires linking the user modules with the

appropriate start-up file and emulation libraries.

2.4.1 Start-Up Files

A start-up file is an object file containing the start-up routine, the code that the microcontroller first executes when it is reset with RESET pin input. The CCU8 compiler package also provides the assembly language source code.

The start-up files included with the CCU8 compiler package cover only the minimum processing necessary for executing a C program. Because initialization needs vary between user application programs, start-up files generally require customization. The safest approach is work with a copy of the desired file instead of directly modifying the original.

For further details on start-up file customization, refer to the document Start-Up File Description.

The start-up file name reflects microcontroller type, memory model, and the presence or absence of a ROM window using the following schema.

s+*target*+{s|l}[w].obj

The leading 's' is always present.

target, which indicates the microcontroller type, is the microcontroller name without the ML prefix.

The next letter, s or l, indicates the memory model: SMALL or LARGE.

The 'w', if present, indicates the presence of a ROM window.

s610001sw.obj, for example, is the start-up file for the ML610001 microcontroller used with the SMALL model and a ROM window.

This assembly language source file has the same name as the start-up file with the file extension .ASM instead of .OBJ.

2.4.2 Emulation Libraries

The compiler provides the following emulation libraries.

Library File Name	Contents
longu8.lib	Integer arithmetic
doubleu8.lib	Double-precision floating point arithmetic
floatu8.lib	Single-precision floating point arithmetic

Overtly specifying emulation libraries on the linker command line is not necessary. Simply specifying the /CC command line option causes RLU8 to search the above emulation libraries for any unresolved symbols and link in the necessary modules from those library files.

For further details, refer to Section 7.2.1.4 "*libraries* Field" in the MACU8 Assembler Package User's Manual.

2.4.3 C Runtime Libraries

The compiler provides the following C runtime libraries.

Library File Name	Contents
LU8100SW.LIB	C runtime library for small memory model
LU8100LW.LIB	C runtime library for large memory model

C runtime libraries are a subset of the library specifications stipulated by the ANSI/ISO 9899 C Standard. To use library routines such as strcpy or memcpy included in this libraries, it is a necessary to specify the library file name in the command-line at link time. At link time, always specify the same memory model used when the program was compiled with the CCU8 compiler.

For further details, refer to the RTLU8 Runtime Library Reference Manual.

To specify the library file name on IDEU8, please specify ‘LU8100SW.LIB’ or ‘LU8100LW.LIB’ in Additional library field of General tab in Target options dialog.

2.5 Start-Up File Description

The start-up file included with the CCU8 compiler package covers the following areas.

- Specifying the memory model
- Specifying the ROM window
- Initializing interrupt vectors
- Initializing the internal RAM region
- Initializing C variables
- Initializing the data segment register (DSR)
- Calling main() function

The following describes the contents of the start-up file and gives the procedures for customizing individual portions.

2.5.1 Initial Comment

```
/*************************************************************
    ML610001 start up assembly source file      (1)
    for CCU8 version 3.XX
    LARGE CODE MODEL                            (2)
    ROMWINDOW  0-7FFFh                          (3)
    Version 1.00                                (4)
    Copyright  2008 - 2011 LAPIS Semiconductor Co., Ltd.
*************************************************************/
```

(1) Target device

(2) Memory model

(3) ROM window region boundaries

(4) Version number for this file

The start-up file starts with a comment specifying the following.

- Target device
- Memory model
- ROM window region boundaries

This information reflects the settings appearing in the assembler initialization directives following this comment.

Before assembling the start-up file, read the section "Reassembling the Start-Up File" below and use the appropriate command line options.

2.5.2 Assembler Initialization Directives

The start-up file starts with directives specifying the program execution environment.

```
type(M610001)           <- Target device
model   large, far      <- Memory and data models
romwindow   0, 7fffh    <- ROM window region boundaries
```

The following describes these directives individually.

Specifying Target Device with TYPE Directive

The TYPE directive specifies the target device for running the user application program. The operand inside the parentheses is the base name of the corresponding DCL file.

DCL files are text files providing device-specific information to the RASU8 assembler. The installer stores these in the directory DCL. They have the file extension .DCL.

Specifying Memory and Data Models with MODEL Directive

This directive consists of the word "model" followed by the memory and data models separated with a comma.

Memory models

`small` : `Small` code model
`large` : `Large` code model

Data models

`near` : `Near` data model

`far` : `Far` data model

Specifying ROM Window Region with ROMWINDOW Directive

The ROMWINDOW directive specifies the ROM window boundaries. Both the assembler and the linker check these settings.

2.5.3 Symbol Declarations

This portion defines symbols used in the start-up file.

```
extrn   code: _main
extrn   data: _$$SP
public  $$start_up
```

Defining References to External Symbols with the EXTRN Directive

```
extrn   code:_main
extrn   data:_$$SP
```

EXTRN directives allow the start-up file to reference symbols declared with PUBLIC directives in other files. The word "extrn" is followed by the attribute and then the symbol name.

The line "extrn code:_main" declares the symbol _main assigned to a code address. This symbol is the entry point to the main() function, used to jump to the user application program's main routine.

The line "extrn data near: _$$SP" declares the symbol _$$SP assigned to a near data address. This special symbol is the stack symbol, one past the last address in the stack segment. It is used to initialize the stack pointer.

Making Symbols Public with PUBLIC Directive

```
public  $$start_up
```

This PUBLIC directive declares the label for the start-up file entry point. All symbols referenced from another file must be declared with PUBLIC directives.

Compiling a file containing a main() function produces code initializing the reset vector (address 2h in the code memory space) to the starting address ($$start_up) for the start-up routine. A reset with RESET pin input then starts program execution from that address.

2.5.4 Specifying Reset Vector

Initializing Stack Pointer

```
cseg    at 0:0h
dw      _$$SP
```

This portion initializes the stack pointer.

The CSEG directive starts a code segment--that is, one in the program memory space. This CSEG directive defines an absolute segment, one whose address is decided by the assembler.

This code saves _$$SP, the address symbol corresponding to the stack pointer declared with the EXTRN directive above at offset 0H in physical segment #0. The linker sets this symbol to one past the last address in the stack segment. Stack pointer initialization is the first step after a reset.

Initializing BRK Reset Vector

This portion initializes the reset vector referenced by a BRK instruction when ELEVEL is 0 or 1.

```
cseg    at 0:4h
dw      $$brk_reset
```

The operand to the DW directive is the label for the BRK reset routine. This routine is defined in the start-up file.

Reset pin input or a BRK instruction when ELEVEL is 2 or 3 references a different reset vector, at the address 2H. The compiler initializes that vector in the C source code file containing the function main().

2.5.5 Starting Address for Start-Up Routine

The code for the start-up routine is in a relocatable code segment, so the linker determines the starting address ($$start_up). If you want it to start at a specific address, modify the code as shown below.

Make sure, however, that the code does not overlap the used portion of the vector table region or the SWI table region. Note also that the code must be in physical segment #0.

The following example locates the code at offset 100H in physical segment #0.

Default code

```
$$NCODs610001lw  segment code    #0
        rseg    $$NCODs610001lw
$$start_up:
```

Modification

```
        cseg    at 0:100h
$$start_up:
```

2.5.6 Break Reset Routine

This portion represents the routine accessed by reset vector 4H. The default start-up file provides only a minimal stub because the contents depend on the user application program.

Feel free to expand this code to match the needs of the user application program.

```
        Sj      $$begin
$$brk_reset:
        bal     $       ;endless loop
$$begin:
```

2.5.7 Specifying Memory Model

This portion is for code specifying the memory model by setting special function registers (SFRs). (At the time of this writing, no target devices offer this option.)

For further details, refer to the User's Manual for the target device.

```
;---------------------------------------
;       setting Memory Model
;---------------------------------------
; nothing (fixed as Large model)
```

2.5.8 Specifying ROM Window Boundaries

This portion is for code specifying the ROM window boundaries by setting special function registers (SFRs). (At the time of this writing, no target devices offer this option.)

For further details, refer to the User's Manual for the target device.

```
;---------------------------------------
;       setting Rom Window range
;---------------------------------------
; nothing (fixed as range 0-7fffh)
```

2.5.9 Initializing SFRs

This portion is for code initializing special function registers (SFRs). Note that this code can be located elsewhere, if desired.

```
;---------------------------------------
;       user SFR definition
;---------------------------------------
; nothing
```

2.5.10 Zeroing Data Areas

The C programming language assumes that all uninitialized global variables are initialized to zero. If such initialization is not necessary, these portions may be deleted.

2.5.10.1 Zeroing RAM Region in Physical Segment #0

```
;---------------------------------------
;       Near Data memory zero clear
;---------------------------------------
NEAR_RAM_START  data    8000h
NEAR_RAM_END    data    8fffh

        mov     er0,    #0
        mov     er2,    #0
```

```
        mov     er4,    #0
        mov     er6,    #0

        mov     r8,     #BYTE1 NEAR_RAM_START
        mov     r9,     #BYTE2 NEAR_RAM_START
        lea     [er8]
__near_ram_loop:
        st      qr0,    [ea+]
        add     er8,    #8              ;er8 += 8
        cmp     r9,     #BYTE2 (NEAR_RAM_END+1)
        bne     __near_ram_loop
        cmp     r8,     #BYTE1 (NEAR_RAM_END+1)
        bne     __near_ram_loop
```

2.5.10.2 Zeroing RAM Regions in Other Physical Segments

The following sample code illustrates the process for zeroing RAM regions in physical segments #1 and higher. Customize it for the RAM regions physically present in the user application system.

```
;-----------------------------------------
;       Far Data memory zero clear
;        (1:0000h - 1:FFFFh)
;        (2:8000h - 3:7FFFh)
;-----------------------------------------
;       MOV     ER0,    #0       ;optional if these registers have
;       MOV     ER2,    #0       ; already been initialized
;       MOV     ER4,    #0       ;
;       MOV     ER6,    #0       ;

        LEA     0000h
        MOV     ER8,    #00h            ;ER8 <= 0000h
        MOV     R10,    #1              ;R10 <= 1
__clear_loop2:
        ST      QR0,    R10:[EA+]
        ADD     ER8,    #8
        ADDC    R10,    #0              ;R10,R9,R8 += 8
        CMP     R10,    #2
        BNE     __clear_loop2

        LEA     8000h
        MOV     R8,     #00h
        MOV     R9,     #80h            ;ER8 <= 8000h
        MOV     R10,    #2              ;R10 <= 2
__clear_loop3:
        ST      QR0,    R10:[EA+]
```

```
        ADD     ER8,    #8
        ADDC    R10,    #0              ;R10,R9,R8 += 8
        CMP     R10,    #3
        BNE     __clear_loop3
        CMP     R9,     #80h
        BNE     __clear_loop3
```

2.5.11 Initializing Variables

This portion is for code initializing variables to the specified initial values. Note that the procedure covers only near variables. The CCU8 compiler generates the assembly language code for initializing far data.

2.5.11.1 Initialization Procedure

1. Read the initialization parameters from the first entry in the initialization table $$init_info: copy source offset, copy destination offset, copy size in words, copy source physical segment number, and copy destination physical segment number.
2. Loop, copying one word at a time, until the word count drops to zero.
3. Repeat the above steps for the remaining entries in the initialization table. The terminator is the number 0xffff in the first field.

The initialization table $$init_info is located toward the end of the start-up file. (See "Defining Global Variable Initialization Segment" below.)

```
;-----------------------------------------------------------------
;       data variable initialization
;-----------------------------------------------------------------
        mov     r10,    #SEG $$init_info
        lea     OFFSET $$init_info
__init_loop:
        ;-----------------------------------------------
        ; get source offset address and set in ER0
        ;-----------------------------------------------
        l       er0,    r10:[ea+]
        cmp     r0,     #0ffh
        bne     __skip
        cmp     r1,     #0ffh
        beq     __init_end              ;if er0==0ffffh then exit
__skip:
        ;-----------------------------------------------
        ; get destination offset address and set in ER2
        ;-----------------------------------------------
        l       er2,    r10:[ea+]

        ;-----------------------------------------------
```

```
        ; get size of objects and set in ER4
        ;-------------------------------------------
        l       er4,    r10:[ea+]

        ;-------------------------------------------
        ; get source phy_seg address and set in R6
        ;-------------------------------------------
        l       r6,     r10:[ea+]

        ;-------------------------------------------
        ; get destination phy_seg address and set in R7
        ;-------------------------------------------
        l       r7,     r10:[ea+]

        ;-------------------------------------------
        ; copy
        ;-------------------------------------------
__init_loop2:
        cmp     r4,     #0
        bne     __skip2
        cmp     r5,     #0
        beq     __init_loop             ;if er4==0000 then next
__skip2:
        l       er8,    r6:[er0]
        add     er0,    #2              ;er0 += 2
        st      er8,    r7:[er2]
        add     er2,    #2              ;er2 += 2

        add     er4,    #-2             ;er4 -= 2
        bal     __init_loop2

__init_end:
```

2.5.11.2 Initializing Data Declared with ABSOLUTE Pragmas

Data defined with ABSOLUTE pragmas is initialized by the user application program in the code segment $$content_of_init. The start-up file therefore calls that code segment for that initialization.

```
;-----------------------------------------
;       call initializing routine
;-----------------------------------------
        bl      $$content_of_init
```

2.5.12 Initializing Segment Register

This portion is for code initializing the data segment register (DSR) to 0.

```
;----------------------------------------
;       initialize DSR zero
;----------------------------------------
;;*** mov    DSR, #0              ;not available
        l       r0,     0:0
```

2.5.13 Branch to main()

The start-up routine concludes by branching to the main() function. Note that the target address can be in any physical segment.

```
;----------------------------------------
;      jump to main routine
;----------------------------------------
        b       _main
```

2.5.14 Segment Definitions

2.5.14.1 Data Initialization Segment for ABSOLUTE Pragmas

The code segment $$content_of_init initializes data defined with ABSOLUTE pragmas. $$end_of_init is a terminator indicating the end of that segment.

```
;--------------------------------------------------
;       segment definition for initializing routine
;--------------------------------------------------
$$content_of_init segment code
        rseg    $$content_of_init

$$end_of_init segment code
        rseg    $$end_of_init
      rt
```

2.5.14.2 Defining Global Variable Initialization Segment

The table segment $$init_info contains the initialization table for near global variables. (See "Initializing Variables" above.)

```
;------------------------------------------------------------
;       segment definition for data variable initialization
;------------------------------------------------------------
$$init_info segment table 2
```

```
        rseg    $$init_info
        dw      $$NINITTAB
        dw      $$NINITVAR
        dw      size $$NINITTAB
        db      seg $$NINITTAB
        db      seg $$NINITVAR

$$init_info_end segment table
        rseg    $$init_info_end
        dw      0ffffh

$$NINITVAR segment data 2 #0
$$NINITTAB segment table 2
```

The table segment $$init_info_end contains the terminator for the table. The initialization routine stops when it reads 0xFFFF in the first field. Note that this segment must immediately follow $$init_info. Specifying the linker's /CC command line option automatically guarantees this.

2.5.15 Reassembling the Start-Up File

If you have modified the start-up file, create a new object file by reassembling the source code.

Example

```
RASU8 S610001LW.ASM /CD /NPR
```

The /CD command line option tells the assembler to distinguish case in symbols. It may be omitted, however, because that behavior is the RASU8 default.

2.6 Keeping In Mind

This Section lists points to keep in mind when compiling and linking program modules.

2.6.1 Specifying Target CPU

Always specify the target microcontroller to the compiler with the /T command line option--/TM610001 if the microcontroller name is ML610001, for example. The compiler interprets the operand as the base name for the corresponding DCL file.

To produce the same results with IDEU8, choose the Project menu's Options -> Compile/assemble... menu command, select the General tab in the dialog box that appears, and enter the DCL base name (M610001 in our example) in the Target microcontroller box.

Note that failing to provide the correct base name interferes with proper compiling, assembling, and linking.

2.6.2 Specifying Memory Model

The memory model defaults to SMALL, but can be overtly specified (SMALL or LARGE) with the /MS

and /ML command line options, respectively.

To produce the same results with IDEU8, choose the Project menu's Options -> Compile/assemble... menu command, select the General tab in the dialog box that appears, and select SMALL or LARGE under Memory model.

Make sure that the target microcontroller supports the specified memory model. If the microcontroller offers only LARGE, for example, this setting must always be LARGE.

2.6.3 Specifying ROM Window Region

The compiler defaults to using a ROM window region, but leaves the boundaries undefined until a C module specifies them with a ROMWIN pragma. If no module specifies them, the linker aborts with the following error message.

```
Fatal error F025: No ROM window specification
```

To eliminate this problem, specify the ROM window region with the linker's /ROMWIN command line option.

The following, for example, specifies the address range 0-7FFFH.

```
/ROMWIN(0, 7FFFH)
```

To produce the same results with IDEU8, choose the Project menu's Options -> Target... menu command, select the Memory settings tab in the dialog box that appears, select the ROM window region check box, and specify the address range.

Note, however, that the above procedures are normally unnecessary because the start-up file that is always linked in already contains this specification.

2.7 Assigning Relocatable Segments to Specific Regions

The linker's /CODE, /DATA, /TABLE, and /NVDATA command line options allow the programmer to assign the relocatable segments to specific address regions.

For further details on these command line options, refer to Section 7.5.3 "Command Line Option Functions" in the MACU8 Assembler Package User's Manual.

To produce the same results with IDEU8, choose the Project menu's Options -> Target... menu command, select the Segments tab in the dialog box that appears, select the segment type corresponding to the segment name, and specify the address range for that relocatable segment.

Compiling file1.c and file2.c using the LARGE model and specifying the following, for example, assigns the executable code from those files starting at the address 1:8000H.

Select the Assign specified segment preferentially to CODE check box and enter the following in the Segment specification text box.

```
$$FCODfile1-1:8000h
```

To merge the executable code from the two files, select the Specify merging for Combine segments check

box and enter the following in the Link order text box.

```
($$FCODfile1 $$FCODfile2)
```

Note that the parentheses are required.

2.8 Creating HEX Files

Although the customary procedure is to write all data in the object module to the HEX file output, limiting the data is also possible it.

2.8.1 Converting All Data in Module

It is possible to write all data in the object module to the HEX file output either from the command line or from within IDEU8.

For further details on using OHU8, refer to Chapter 9 "OHU8" in the MACU8 Assembler Package User's Manual.

When using IDEU8, choose the Project menu's Options -> Target... menu command, select the General tab in the dialog box that appears, select the Create HEX file check box under Object converter, and specify the format: Intel HEX or Motorola S format.

2.8.2 Converting Only a Portion

Writing a portion of the data in the object module to the HEX file output is only available from the command line. IDEU8 provides no means for limiting the address range.

Specify the address range with the OHU8 /R command line option.

```
OHU8 SAMPLE /R(3:0H, 3:0FFFFH);
```

This example restricts HEX file output to SAMPLE.ABS data between the addresses 3:0 and 3:FFFFH.

For further details on using OHU8, refer to Chapter 9 "OHU8" in the MACU8 Assembler Package User's Manual.

3 Appendix

3.1 Map Files

Map files created by the linker contain a wide variety of information about the user application program. For further details on reading the contents, refer to Section 7.7 "Map files" in the MACU8 assembler package User's Manual.

3.1.1 Object Module Synopsis

This portion of the map file tells you which object modules make up the user application program.

```
------------------------
 Object Module Synopsis
------------------------

Module Name           File Name                              Creator
-----------           ---------                              -------
fifo                  fifo.obj                               RASU8 Ver.1.03
keydebouncer          keydebouncer.obj                       RASU8 Ver.1.03
keyinterrupt          keyinterrupt.obj                       RASU8 Ver.1.03
keystate              keystate.obj                           RASU8 Ver.1.03
keystateerror         keystateerror.obj                      RASU8 Ver.1.03
keystateidle          keystateidle.obj                       RASU8 Ver.1.03
keystatepress         keystatepress.obj                      RASU8 Ver.1.03
keytask               keytask.obj                            RASU8 Ver.1.03
main                  main.obj                               RASU8 Ver.1.03
s610001lw             s610001lw.obj                          RASU8 Ver.1.03
INDRLW                C:¥Progra~1¥U8Dev¥Lib¥longu8.lib       RASU8 Ver.1.00


Number of Modules: 11
```

This portion lists all modules linked into the user application program--including those that the linker automatically linked from library files.

3.1.2 Memory Map

This portion of the map file gives you the mappings for ROM, RAM, and other types of memory.

```
Memory Map - Program memory space #0:
      Type     Start      Stop
     ----------------------------
      ROM      00:0000    00:FFBF
```

The above example gives memory mapping information for physical segment #0 in the program memory space.

```
Memory Map - Data memory space #0:
      Type     Start      Stop
     ----------------------------
      RAM      00:8000    00:8FFF
      RAM      00:F000    00:FFFF
```

The above example gives memory mapping information for physical segment #0 in the data memory space.

```
Memory Map - Memory space above #1:
      Type     Start      Stop
     ----------------------------
      ROM      01:0000    01:FFFF
```

The above example gives memory mapping information for physical segments #1 and higher.

3.1.3 Segment Synopsis

This portion of the map file tells you where segments have been assigned in memory.

```
                          ------------------
                           Segment Synopsis
                          ------------------


Link Map - Program memory space #0 ( ROMWINDOW: 0000 - 7FFF )

      Type     Start      Stop       Size              Name
     -----------------------------------------------------------
      S CODE   00:0000    00:0001    0002(2)           (absolute)
      S CODE   00:0002    00:0003    0002(2)           (absolute)
      S TABLE  00:0004    00:001B    0018(24)          $$NTABkeystate
      S CODE   00:001C    00:001C    0000(0)           $$content_of_init
      S CODE   00:001C    00:001D    0002(2)           $$end_of_init
      S CODE   00:001E    00:001F    0002(2)           $$indru8lw
      S TABLE  00:0020    00:0027    0008(8)           $$init_info
      S TABLE  00:0028    00:0029    0002(2)           $$init_info_end
>GAP<          00:002A    00:0031    0008(8)           (ROM)
      S CODE   00:0032    00:0033    0002(2)           (absolute)
      S CODE   00:0034    00:006D    003A(58)          $$INTERRUPTCODE
>GAP<          00:006E    00:007F    0012(18)          (ROM)
      S CODE   00:0080    00:0081    0002(2)           (absolute)
      S CODE   00:0082    00:00DD    005C(92)          $$NCODs610001lw
      S CODE   00:00DE    00:00F5    0018(24)          $$FCODkeystateerror
      S CODE   00:00F6    00:011F    002A(42)          $$FCODkeydebouncer
      S CODE   00:0120    00:0137    0018(24)          $$FCODkeystateidle
      S CODE   00:0138    00:0191    005A(90)          $$FCODkeystate
      S CODE   00:0192    00:026B    00DA(218)         $$FCODfifo
      S CODE   00:026C    00:02C7    005C(92)          $$FCODkeystatepress
      S CODE   00:02C8    00:0303    003C(60)          $$FCODkeytask
```

```
       S CODE   00:0304    00:032B    0028(40)          $$FCODmain
```

The above example gives segment mapping information for physical segment #0 in the program memory space. The letter S at the start of each entry in the Type column indicates that the corresponding symbol is a segment symbol.

CODE in the Type column indicates a segment containing program code; TABLE, one containing initial values for C program variables, variables modified with const, and other read-only table data.

The compiler generates segment names using the rules in Section 2.1 "Segment Names Generated by Compiler," so it is easy determine the source module from the segment name. The segment $$FCODkeytask, for example, contains program code from the module keytask.

```
Link Map - Data memory space #0

       Type     Start      Stop       Size              Name
      ------------------------------------------------------
       Q ROMWIN 00:0000    00:7FFF    8000(32768)       (ROMWIN)
>GAP<           00:8000.0  00:8BCF.7  0BD0.0(3024.0)    (RAM)
       S DATA   00:8BD0    00:8FCF    0400(1024)        $STACK
       C DATA   00:8FD0    00:8FD9    000A(10)          _Fifo_signalToKey
       C DATA   00:8FDA    00:8FE3    000A(10)          _Fifo_signalToMode
       S DATA   00:8FE4    00:8FE5    0002(2)           $$NVARkeystatepress
       C DATA   00:8FE6    00:8FEF    000A(10)          _Fifo_signalToLcd
       S DATA   00:8FF0    00:8FF0    0001(1)           $$NVARkeystate
>GAP<           00:8FF1.0  00:8FF1.7  0001.0(1.0)       (RAM)
       S DATA   00:8FF2    00:8FF3    0002(2)           $$NVARkeydebouncer
       C DATA   00:8FF4    00:8FFD    000A(10)          _Fifo_signalToTimer
       S DATA   00:8FFE    00:8FFE    0001(1)           $$NVARkeyinterrupt
>GAP<           00:8FFF.0  00:8FFF.7  0001.0(1.0)       (RAM)
       Q SFR    00:F000    00:FFFF    1000(4096)        (SFR)
```

The above example gives segment mapping and communal symbol table for physical segment #0 in the data memory space.

The letters S and C at the start of each entry in the Type column indicate segment and communal symbols, respectively.

The compiler treats uninitialized global variables as communal symbols. The underscore at the beginning of the communal symbols _Fifo_signalToKey and _Fifo_signalToMode, for example, strongly suggests that they are C program global variables. The above map information thus tells you where global variables have been assigned.

Note, however, that the compiler treats initialized global variables not as communal symbols, but as public symbols, so the above segment assignment information does not give their locations. Consult the symbol tables for the individual modules or the public symbol list instead. Both are described below.

```
Size 0 segments symbols:
      S DATA                                                $$NINITVAR
      S TABLE                                               $$NINITTAB
```

The above is the output for segments of size 0. This warning appears only when the segments $$NINITVAR and $$NINITTAB contain not a single overtly initialized global variable. It can be safely ignored.

3.1.4 Program and Data Sizes

This portion of the map file gives aggregate program and data sizes for each segment type.

```
Total size (CODE  ) = 002F0   (752)
Total size (DATA  ) = 0042E   (1070)
Total size (BIT   ) = 00000.0 (0.0)
Total size (NVDATA) = 00000   (0)
Total size (NVBIT ) = 00000.0 (0.0)
Total size (TABLE ) = 00022   (34)
```

3.1.5 Symbol Addresses

This portion of the map file gives symbol tables for each module and a public symbol list.

```
            ------------------------
             Symbol Table Synopsis
            ------------------------

Module          Value     Type          Symbol
------          -----     ----------    ------
s6100011w
               000000FF   Loc NUMBER    _$$WINVAL
                00:7FFF   Loc TABLE     _$$ROMWINEND
                00:0000   Loc TABLE     _$$ROMWINSTART
                00:009E   Loc CODE      __init_loop
                00:00D2   Loc CODE      __init_end
                00:0090   Loc CODE      __clear_loop
                00:00C2   Loc CODE      __skip2
                00:00AA   Loc CODE      __skip
                00:00BA   Loc CODE      __init_loop2
                00:0082   Pub CODE      $$start_up
```

The above example is a symbol table for a module. These tables only appear, however, if the assembler and linker command lines both specify the /D option. Specifying this option on the linker command line does not produce symbol table output for modules assembled without the /D option.

```
Public Symbols Reference

Symbol                              Value      Type    Module
------                              -----      ----    ------
$$start_up                          00:0082    CODE    s610001lw
_$$SP                               00:8FD0    DATA    fifo
_Fifo_deque                         00:0262    CODE    fifo
_Fifo_dequeInInterrupt              00:022E    CODE    fifo
_Fifo_enque                         00:0256    CODE    fifo
_Fifo_enqueInInterrupt              00:0204    CODE    fifo
_Fifo_init                          00:01F6    CODE    fifo
_Fifo_mainloop                      00:01CC    CODE    fifo
_KeyDebouncer_getDebouncedKey       00:0108    CODE    keydebouncer
_KeyDebouncer_getUndebouncedKey     00:0102    CODE    keydebouncer
_KeyDebouncer_init                  00:00F6    CODE    keydebouncer
        :
        :
_KeyStatePress_init                 00:026C    CODE    keystatepress
_KeyStatePress_process              00:027A    CODE    keystatepress
_KeyState_getSignal                 00:0174    CODE    keystate
_KeyState_getState                  00:0144    CODE    keystate
_KeyState_init                      00:0138    CODE    keystate
_KeyState_process                   00:014A    CODE    keystate
_KeyTask_init                       00:02C8    CODE    keytask
_KeyTask_schedule                   00:02D4    CODE    keytask
__indru8lw                          00:001E    CODE    INDRLW
_main                               00:0304    CODE    main
```

The above example lists all public symbols used in the program. Adding the /S option to the linker command line produces this list of addresses for global functions, overtly initialized global variables, and the like.

3.2 Calculating Stack Consumption

The compiler's /LE and /CT command line options are for calculating the user application program's stack consumption.

Example

```
int fn1(void);
int fn2(int a, int b);
int fn3(int a, int b, int c);
double dblfn(void);

void main(void)
{
    volatile int i;
    i = fn1();
    fn2(10, 20);
    fn3(10, 20, 30);
}

int fn1(void)
{
    volatile int    i, j, k;
    i = j = k = 0;
    return i + j + k;
}

int fn2(int a, int b)
{
    return fn3(a, b, a+b);
}

int fn3(int a, int b, int c)
{
    volatile i;
    i = a + b + c;
    return  i;
}
```

Specifying the compiler's /LE option sends the stack consumption for each function to the error list file.

Sample Stack Information from an Error List File

```
                        STACK INFORMATION
                        -----------------

  FUNCTION                  LOCALS    CONTEXT    OTHERS    TOTAL
  --------                  ------    -------    ------    -----
  _main                     2         0          2         4
  _fn1                      6         2          0         8
  _fn2                      0         4          2         6
  _fn3                      2         2          0         4
```

The columns contain the following information.

Title	Description
FUNCTION	Function name. The compiler prefixes this with an underscore (_).
LOCALS	Total size for all automatic variables used by the function plus regions used for preserving the contents of registers.
CONTEXT	Total size for registers saved upon entry.
OTHERS	Space for arguments that the function pushes onto the stack during function calls. If the function calls multiple functions, this figure is the maximum for all such calls.
TOTAL	Sum of the figures in the LOCALS, CONTEXT, and OTHERS columns.

Note that these figures represent the stack consumption for the function by itself. They do not include the stack consumptions for any subfunctions called. You must therefore investigate the call hierarchy and add up the figures for the function and any subfunctions. To display the call tree, use the /CT command line option.

Sample Call Tree List File Output

```
main
|   fn1
|   fn2
|   |   fn3
|   fn3...
```

The above call tree list is the result of compiling the C source code on the preceding page with the /CT command line option.

The following illustrates the procedure for calculating the total stack consumption from this call tree list and the stack information in the error list file.

Function Name	Stack Consumption	Discussion
main	14bytes	This function calls fn1(), fn2(), and fn3(). The fn2() call has the highest stack consumption (10 bytes), so the total stack consumption is the 4 bytes for main() alone plus those 10 bytes for a total of 14 bytes.
fn1	8bytes	This function does not call any others, so the stack consumption is just the 8 bytes for the function alone.
fn2	10bytes	This function calls fn3(), so the total stack consumption is the 6 bytes for fn2() alone plus the 4 bytes for fn3() for a total of 10 bytes.
fn3	4bytes	This function does not call any others, so the stack consumption is just the 4 bytes for the function alone.