LAPIS SEMICONDUCTOR

FJXTCCU8

CCU8
プログラミングガイド

プログラム開発支援ソフトウェア

2012 年 4 月発行

ご注意

本資料の一部または全部をラピスセミコンダクタの許可なく、転載・複写することを堅くお断りします。

本資料の記載内容は改良などのため予告なく変更することがあります。

本資料に記載されている内容は製品のご紹介資料です。ご使用にあたりましては、別途仕様書を必ずご請求のうえ、ご確認ください。

本資料に記載されております応用回路例やその定数などの情報につきましては、本製品の標準的な動作や使い方を説明するものです。したがいまして、量産設計をされる場合には、外部諸条件を考慮していただきますようお願いいたします。

本資料に記載されております情報は、正確を期すため慎重に作成したものですが、万が一、当該情報の誤り・誤植に起因する損害がお客様に生じた場合においても、ラピスセミコンダクタはその責任を負うものではありません。

本資料に記載されております技術情報は、製品の代表的動作および応用回路例などを示したものであり、ラピスセミコンダクタまたは他社の知的財産権その他のあらゆる権利について明示的にも黙示的にも、その実施または利用を許諾するものではありません。上記技術情報の使用に起因して紛争が発生した場合、ラピスセミコンダクタはその責任を負うものではありません。

本資料に掲載されております製品は、一般的な電子機器（AV 機器、OA 機器、通信機器、家電製品、アミューズメント機器など）への使用を意図しています。

本資料に掲載されております製品は、「耐放射線設計」はなされておりません。

ラピスセミコンダクタは常に品質・信頼性の向上に取り組んでおりますが、種々の要因で故障することもあり得ます。

ラピスセミコンダクタ製品が故障した際、その影響により人身事故、火災損害等が起こらないようご使用機器でのディレーティング、冗長設計、延焼防止、フェイルセーフ等の安全確保をお願いします。定格を超えたご使用や使用上の注意書が守られていない場合、いかなる責任もラピスセミコンダクタは負うものではありません。

極めて高度な信頼性が要求され、その製品の故障や誤動作が直接人命を脅かしあるいは人体に危害を及ぼすおそれのある機器・装置・システム（医療機器、輸送機器、航空宇宙機、原子力制御、燃料制御、各種安全装置など）へのご使用を意図して設計・製造されたものではありません。上記特定用途に使用された場合、いかなる責任もラピスセミコンダクタは負うものではありません。上記特定用途への使用を検討される際は、事前にローム営業窓口までご相談願います。

本資料に記載されております製品および技術のうち「外国為替及び外国貿易法」に該当する製品または技術を輸出する場合、または国外に提供する場合には、同法に基づく許可が必要です。

Windows は、米国 Microsoft Corporation の米国およびその他の国における登録商標です。また、その他の製品名や社名などは、一般に商標または登録商標です。

ラピスセミコンダクタ株式会社
〒193-8550 東京都八王子市東浅川町 550 番地 1
http://www.lapis-semi.com/jp/

目次

1 プログラミング

1.1 SFR の参照方法

CCU8 コンパイラパッケージでは，SFR 名を定義したターゲットヘッダファイルを提供しています。

#include 前処理指令によってターゲットヘッダファイルを読み込むことで，対象のマイクロコントローラが持っているすべての SFR 名を，絶対アドレスを持つ変数として使用できます。

1.1.1 ターゲットヘッダファイル名

ターゲットヘッダファイルの名前は，対象となるマイクロコントローラ名の"ML"の部分を"M"にした文字列と".H"を結合したものです。

例えば，マイクロコントローラ名が"ML610001"であれば，ターゲットヘッダファイル名は"M610001.H"となります。

1.1.2 ターゲットヘッダファイルの内容

ターゲットヘッダファイルには，SFR 名が次のようにマクロで定義されています。

```
/***************************************************
 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 SFR の参照例

ターゲットヘッダファイルをインクルードして，SFR をアクセスするプログラムの例を以下に示します。

```
#include <m610001.h> /* ターゲットヘッダファイルのインクルード */
void initial_timer(void)
{
    TM0CON0 = 0x08; /* タイマコントロールレジスタの設定 */
    TM0D = 0x7f;    /* タイマデータレジスタの設定 */
    ETM0 = 1;       /* タイマ割り込み許可 */
    T0RUN = 1;      /* タイマカウント開始 */
}
```

1.1.4 volatile 修飾子の効果

ターゲットヘッダファイルでは，SFR 名のマクロ定義においてそれぞれ volatile 修飾子を指定しています。ここでは，volatile 修飾子がどのような効果を持つのかを説明します。

volatile を指定しなかった場合

例えば，次のようなプログラムを記述したとします。

```
unsigned char status; /* 割り込み処理で値が設定される */
void proc1(unsigned char);
void fn(void)
{
    status = 0;
    while(status == 0)
        ;       /* 割り込み処理によって status が設定されるまで待つ */
```

```
    proc1(status);
}
```

上記のプログラムでは，割り込み処理において変数 status の値が設定されるものとし，変数 status の値が 0 以外になるまで待つようにしています。このプログラムを CCU8 でコンパイルすると，次のようなアセンブリコードを生成します。

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

変数 status の値は while 文の直前で 0 に設定されているため，CCU8 は status の値は常に 0 であると解釈して最適化を行い，無限ループとなるアセンブリコードを生成してしまいます。このため，このプログラムはユーザの意図したとおりには動作しません。

/Od オプションを指定すれば，最適化が抑止され，意図したとおりのアセンブリコードが生成されますが，コンパイル対象のすべてのプログラムに対して最適化が抑止されてしまうため，プログラムサイズが大きくなってしまいます。

volatile で修飾した場合

このような場合には，次のように変数 status を volatile で修飾します。

```
volatile unsigned char status; /* 割り込み処理で値が設定される */
void proc1(unsigned char);
void fn(void)
{
    status = 0;
    while(status == 0)
        ;       /* 割り込み処理によって status が設定されるまで待つ */
    proc1(status);
}
```

volatile で修飾された変数 status が使用されている式に対しては，最適化が抑止されます。したがって，上記のプログラムにおいては，無限ループにはなりません。このプログラムを CCU8 でコンパイルした場合，出力アセンブリコードは次のようになります。

```
_fn     :
;;    status = 0; /* 割り込み処理で値が変更される */
        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 割り込み処理の記述方法

CCU8 では，ハードウェア割り込み処理およびソフトウェア割り込み処理を，C 言語の関数として定義するためのプラグマを用意しています。

ハードウェア割り込み処理を記述する場合には，INTERRUPT プラグマを使用します。

ソフトウェア割り込み処理を記述する場合には，SWI プラグマを使用します。

割り込み関数に戻り値および引数を記述することはできません。INTERRUPT プラグマおよび SWI プラグマの詳細については，『CCU8 ユーザーズマニュアル』を参照してください。

INTERRUPT プラグマ，および SWI プラグマの構文は以下のとおりです。

プラグマ	構文	*address* の範囲
INTERRUPT	#pragma INTERRUPT *function_name address* [*category*]	0x08～0x7E
SWI	#pragma SWI *function_name address* [*category*]	0x80～0xFE

INTERRUPT プラグマと SWI プラグマの構文は，指定できるアドレスの範囲を除いて同じです。

1.2.1 多重割り込み禁止の割り込み関数の記述

多重割り込み禁止の割り込み関数を記述する場合，INTERRUPT プラグマおよび SWI プラグマの *category* フィールドで 1 を指定します。多重割り込みを禁止する割り込み関数内で組み込み関数__EI を呼び出すと，CCU8 はエラーを表示します。

記述例

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

例のように記述すると，intr_fn_0A は多重割り込みを禁止する割り込み処理関数として扱われます。CCU8 は次のようなアセンブリコードを出力します。

出力例

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

```
        rti
```

割り込み関数では，割り込み処理内で使用される可能性のあるレジスタ（ここでは ER0 のみ）をスタックに保存します。多重割り込み禁止の割り込み関数から復帰する場合には”RTI”が使用されます。

次に，割り込み関数から他の関数を呼び出す場合の例を以下に示します。

記述例

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

出力例

```
_intr_fn_10     :
        push    lr, ea
        push    xr0
        l       r0,     DSR
        push    r0

;;    func();
        bl      _func

;;}
        pop     r0
        st      r0,     DSR
        pop     xr0
        pop     ea, lr
        rti
```

割り込み関数から他の関数を呼び出す場合，割り込み関数から他の関数を呼び出さない場合に比べて出力コードは冗長になり，その結果割り込みの処理時間も長くなります。これは，CCU8 には関数 func がどのようなレジスタを使用するかが分からないため，func を呼び出すことによって変更される可能性のあるレジスタをすべてスタックに退避してしまうためです。

注意

多重割り込みを禁止にした関数から他の関数を呼び出し、呼び出した関数内で割り込みを許可しないようにしてください。

許可した場合、多重割り込みが発生した際に、プログラムが暴走する可能性があります。

1.2.2 多重割り込み許可の割り込み関数の記述

多重割り込み許可の割り込み関数を記述する場合，INTERRUPT プラグマおよび SWI プラグマの *category* フィールドで 2 を指定します。*category* フィールドでの指定を省略しても多重割り込み許可となります。多重割り込みを許可する割り込み関数内では，組み込み関数__EI を呼び出すことができます。

記述例

```
static void intr_fn_20(void);
volatile unsigned short TM2msec;
#pragma interrupt intr_fn_20 0x20 2
static void intr_fn_20(void)
{
    __EI();  /* 多重割り込み許可 */
    TM2msec++;
    __DI();  /* 多重割り込み禁止 */
}
```

例のように記述すると，intr_fn_20()は多重割り込みを許可する割り込み処理関数として扱われます。CCU8 は次のようなアセンブリコードを出力します。

出力例

```
_intr_fn_20     :
        push    elr, epsw
        push    er0
;;    __EI();  /* 多重割り込み許可 */
        ei
;;    TM1msec++;
        l       er0,    NEAR _TM2msec
        add     er0,    #1
        st      er0,    NEAR _TM2msec
;;    __DI();  /* 多重割り込み禁止 */
        di
;;}
        pop     er0
        pop     psw, pc
```

多重割り込み許可の割り込み関数では，多重割り込みによって ELR と EPSW が破壊されないよう ELR と EPSW をスタックに退避します。この部分が多重割り込み禁止の割り込み関数と異なります。そして，割り込み関数から復帰する場合には”RTI”ではなく，”POP PSW, PC”が使用されます。

1.3 変数をアブソリュートなアドレスに割り付ける方法

変数を特定の領域に割り付けるには，通常は ABSOLUTE プラグマを使用します。ただし，

NVDATA プラグマで指定された変数を ABSOLUTE プラグマで指定することはできません。

ここでは，NVDATA プラグマで指定されていない変数の場合と，NVDATA プラグマで指定された変数の場合に分けて，変数に対するアブソリュートなアドレスの割り付け方法を説明します。

ABSOLUTE プラグマおよび NVDATA プラグマの詳細については，『CCU8 ユーザーズマニュアル』を参照してください。

1.3.1 NVDATA プラグマで指定されていない変数の場合

const で修飾されない変数をアブソリュートなアドレスに割り付ける場合の例を以下に示します。

記述例

```
#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;
```

物理セグメント#1 以上のアブソリュートなアドレスを変数に割り当てる場合，例のように変数を__far で修飾する必要があります。

上記に対し，CCU8 は次のアセンブリコードを出力します。

出力例

```
          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
```

const で修飾されない，ABSOLUTE プラグマで指定された変数は，アブソリュート DATA セグ

メントとして出力されます。さらに，その変数に対して初期化が行われている場合には，初期化コードがリロケータブル CODE セグメント$$content_of_init に出力されます。

$$content_of_init はスタートアップルーチンから呼ばれ，$$content_of_init に含まれる初期化プログラムが実行されます。

次に const で修飾された変数をアブソリュートなアドレスに割り付ける場合の例を以下に示します。

記述例

```
#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;
```

出力例

```
          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
```

const で修飾された，ABSOLUTE プラグマで指定された変数は，アブソリュート TABLE セグメントとして出力されます。

1.3.2 NVDATA プラグマで指定された変数

NVDATA プラグマで指定された変数は，ABSOLUTE プラグマを用いてアブソリュートアドレスを指定することができません。ここでは，NVDATA プラグマで指定された変数をアブソリュートなアドレスに割り付ける方法を説明します。

記述例

```
#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
    };
```

まず，上記のようにそれぞれの変数を NVDATA プラグマで指定します。アセンブリファイルへの出力において，nvdata 変数を宣言した順番に並べたい場合には，初期値を指定してください。変数に対して初期値を指定しなかった場合，変数の順番は保証されません。

上記の記述に対し，CCU8 は以下のアセンブリコードを出力します。

出力例

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

C ソースファイル名が sample.c であった場合，__near 修飾子で指定された nvdata 変数はリロケータブル NVDATA セグメント$$NNVDATAsample に出力され，__far 修飾子で指定された nvdata 変数はリロケータブル NVDATA セグメント$$FNVDATAsample に出力されます。

NVDATA プラグマで指定された変数をアブソリュートなアドレスに割り付ける場合には，リンカ RLU8 のコマンドラインオプション/NVDATA を使用します。このときに，セグメント名が必要となります。

例えば，$$NNVDATAsample を 0:0A000H 番地に，$$FNVDATAsample を 3:8000H 番地にそれぞれ割り付けたい場合には，次のようにリンカオプションを指定します。

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

IDEU8 をご使用になる場合は，[プロジェクト]メニューから[オプション][ターゲット]を選択して[ターゲットオプション]ダイアログを表示し，[セグメント]タブを選択します。そして，優先的に割り付ける NVDATA セグメントをチェックし，セグメント指定ボックスに次のように指定します。

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

1.4 NEAR/FAR

データのアクセスにおいて，物理セグメント#0 のメモリ空間をアクセスする場合と，物理セグメント#1 以上をアクセスする場合とでは，アクセスの方法が異なります。

物理セグメント#0 のデータメモリ空間をアクセスする場合，16 ビットのアドレス指定でアクセスが可能です。しかし，物理セグメント#1 以上のデータメモリ空間をアクセスする場合，はじめに，アクセスするデータメモリ空間の物理セグメントアドレスの設定を行い（この物理セグメントアドレスの設定を行う命令のことを DSR プリフィックス命令と呼びます），その後に物理セグメント内のオフセットアドレス（16 ビット）を指定してアクセスします。したがって，物理セグメント#1 以上のデータメモリ空間をアクセスする場合，物理セグメント#0 のデータメ

モリ空間をアクセスする場合よりも効率は悪くなります。

CCU8 では，物理セグメント#0 のみのデータメモリ空間をアクセスすることを near アクセス，物理セグメントを限定せずにデータメモリ空間をアクセスすることを far アクセスと呼んでいます。また，物理セグメント#0 のみに置かれるデータのことを near データ，配置先の物理セグメントアドレスを限定しないデータのことを far データと呼びます。さらに，near データをアクセスするポインタを near ポインタ（ポインタのサイズは 2 バイト），far データをアクセスするポインタを far ポインタ（ポインタのサイズは 3 バイト）と呼びます。

1.4.1 __near, __far 修飾子

__near 修飾子および__far 修飾子を用いると，個々の変数に対して near/far の設定を行なうことができます。これらの修飾子をデータアクセス指定子と呼ぶ場合もあります。

__near 修飾子および__far 修飾子の詳しい説明については『CCU8 ランゲージリファレンス』の「4.4.1 メモリモデル修飾子」を参照してください。

例

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

near_var は near データとして扱われます。near_var は物理セグメント#0 に配置されます。

far_var は far データとして扱われます。far_var はどの物理セグメントに配置されてもかまいません。

例

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

far_pointer は int 型のオブジェクトへの far ポインタとして扱われます。far_pointer 自身は物理セグメント#0 に配置されます。

near_pointer は int 型のオブジェクトへの near ポインタとして扱われます。near_pointer 自身はどの物理セグメントに配置されてもかまいません。

1.4.2 /near, /far オプション

/near オプションを使用すると，データアクセス指定子（__near，__far）が指定されていないすべてのデータを near データとして扱うよう CCU8 に指示することができます。

/far オプションを使用すると，データアクセス指定子が指定されていないすべてのデータを far データとして扱うよう CCU8 に指示することができます。

以下の C 言語プログラムを例に挙げて，説明します。

C 言語の記述

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

上記の C 言語プログラムに対し，/near オプションを指定してコンパイルした結果と，/far オプションを指定してコンパイルした結果を以下に示します。

/near オプション指定時のアセンブリ出力	/far オプション指定時のアセンブリ出力

`_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`

/near オプションを指定した場合，CCU8 は変数 a, b に対して次のように宣言されたものと解釈します。

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

変数 a は near データ，変数 b は near ポインタとして扱われます。a，b ともに物理セグメント #0 に配置されます。

/far オプションを指定した場合，CCU8 は変数 a, b に対して次のように宣言されたものと解釈します。

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

変数 a は far データ，変数 b は far ポインタとして扱われます。a，b ともに配置先の物理セグメントアドレスは限定されません。

/far オプションを指定した場合，/near オプションを指定した場合よりもプログラムサイズは大きくなり，その結果プログラムの実行速度も遅くなります。

したがって，プログラムを作成する場合には，変数をできるだけ near データと扱うようにコンパイル時には/near オプションを指定すべきです。そして，アプリケーションプログラムの仕

様上どうしても物理セグメント#0 に置くことができない変数や，リンク時に物理セグメント#0 に入りきらなかった変数を far データとして扱うようにするとよいでしょう。一部のデータを far データとして扱う場合には，それらの変数に対して__far 修飾子を指定するか，後述の FAR プラグマを使用してください。

1.4.3 NEAR, FAR プラグマ

NEAR プラグマおよび FAR プラグマを使用すると，ソースファイルの一部に対してデフォルトのデータアクセス指定子を指定することができます。

以下に例を示します。

例

```
int a, b;       /* /near, /far オプションに依存する */
#pragma near
char nbuf[16];  /* /near, /far オプションに関わらず,
                   near データとして扱われる */
#pragma far
char fbuf[16];  /* /near, /far オプションに関わらず,
                   far データとして扱われる */

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”);
                /* /near, /far オプションに関わらず,
                   文字列”near string”は near データとして扱われる */
#pragma far
    strcpy_ff(fbuf, ”far string”);
                /* /near, /far オプションに関わらず,
                   文字列”far string”は far データとして扱われる */
}
```

上記のプログラムに対し，/far オプションを指定してコンパイルすると，アセンブリ出力は次のようになります。

```
        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 データのサイズおよびメモリ配置

ここでは，各データ型のオブジェクトのサイズ，および各オブジェクトがメモリ上に配置される場合，どのように配置されるのかを説明します。

1.5.1 スカラ型

データ型	サイズ	境界調整
char	1 バイト	1 バイト
unsigned char	1 バイト	1 バイト
short	2 バイト	2 バイト
unsigned short	2 バイト	2 バイト
int	2 バイト	2 バイト
unsigned int	2 バイト	2 バイト
long	4 バイト	2 バイト
unsigned long	4 バイト	2 バイト
enum	2 バイト	2 バイト
float	4 バイト	2 バイト
double	8 バイト	2 バイト
near ポインタ	2 バイト	2 バイト
far ポインタ	3 バイト	2 バイト
関数へのポインタ (スモールモデル時)	2 バイト	2 バイト
関数へのポインタ (ラージモデル時)	3 バイト	2 バイト

1.5.2 配列

1.5.2.1 char 型の配列

char 型の配列の場合，指定したバイト数の領域が確保されます。

例

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

出力アセンブリソースの例を以下に示します。

```
        _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
```

配列 odd_arr1 のサイズは 7 バイト，配列 odd_arr2 のサイズは 9 バイトになります。

1.5.2.2 char 型以外の配列

char 型以外の配列において配列要素のサイズが奇数サイズの場合，各要素の間に 1 バイトのパディングが挿入されます。

far ポインタの配列の場合を例に挙げて説明します。

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

```
        rseg $$NINITTAB
        dw      OFFSET ($$S3) ;; fparr[0]の初期値
        db      SEG ($$S3)    ;;     同上
        align
        dw      OFFSET ($$S4) ;; fparr[1]の初期値
        db      SEG ($$S4)    ;;     同上
        align
        dw      OFFSET ($$S5) ;; fparr[2]の初期値
        db      SEG ($$S5)    ;;     同上
        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
```

far ポインタは 3 バイトですが，far ポインタの配列になると各要素の間に 1 バイトのパディングが挿入されます。これは，far ポインタは偶数アドレスに配置される必要があるためです。したがって，配列全体のサイズ（sizeof(fparr)）は（（far ポインタのサイズ（3 バイト））＋（パディング分（1 バイト））＊（要素数（5））＝ 20 バイトになります。ただし，各配列要素のサイズ（sizeof(fparr[n])）は 3 バイトとなりますので注意してください。

1.5.3 構造体

構造体の各メンバは，宣言した順に並べて格納されます。最初のメンバがもっとも小さいアドレスに，最後のメンバが最も大きいアドレスに割り付けられます。各メンバは，その型にあったメモリ境界から始まります。このため，メモリ上は構造体のメンバ間にパディングが挿入される場合があります。

パディングは次の条件の場合に挿入されます。

(1) 2 バイト以上のメンバのオフセットが奇数になる場合，そのメンバの直前に 1 バイトのパディングが挿入されます。

(2) 構造体のサイズが 3 バイト以上の奇数バイトになる場合，一番最後のメンバの直後に 1 バイトのパディングが挿入されます。すなわち，1 バイト以外の構造体は偶数サイズに調整されます。

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

上記のプログラムをコンパイルすると，出力アセンブリソースは次のようになります。

```
        rseg $$NINITTAB
        dw      0ah   ;; var.i の初期値
        dw      014h  ;; var.l の初期値
        dw      00h   ;;     同上
        db      01eh  ;; var.c の初期値
        align

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

この場合，構造体 var のサイズは 8 バイトになります。char 型のメンバ c のあとにサイズを調整するための 1 バイトが確保されます。

次に，メンバを並べる順序により構造体のサイズが異なる例を示します。

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

この場合，メンバ b とメンバ c との間に 1 バイトのパディングが挿入されます。そして，メンバ d の後ろに 1 バイトのパディングが挿入されます。この結果，stvar1 のサイズは 8 バイトになります。

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

この例では，char 型のメンバを連続した領域に割り付けるように宣言しています。この場合，パディングは挿入されません。したがって，strvar2 のサイズは 6 バイトになります。

1.5.4 ビットフィールド

CCU8 では，型指定子に unsigned char 型と unsigned int 型を指定することができます。

1.5.4.1 unsinged char 型のビットフィールド

ビットフィールドのメンバを unsigned char 型で宣言すると，全体のサイズは unsigned char のサイズ（1 バイト）で確保されます。

unsigned char 型のビットフィールド中の連続したメンバは，合計サイズが 1 バイト（8 ビット）の範囲内であれば同一バイト位置に格納されます。

```
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;
```

unsigned char 型のビットフィールドでは，連続したメンバの合計サイズが 8 ビットを超える場合は，入りきらないビットフィールドメンバを格納するための新しい 1 バイト領域が確保され

ます。

```
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;
```

ビットフィールドは，unsigned char 型で確保するほうがコード効率がよくなります。

1.5.4.2 unsigned int 型のビットフィールド

ビットフィールドを unsigned int 型で宣言すると 2 バイト単位で領域が確保されます。各ビットメンバはできる限り詰めて確保されます。

```
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;  /* 2 バイト単位で領域が確保されます */
    unsigned int     b7 : 1;
} bit_field;
```

unsigned int 型のビットフィールド中の連続したメンバは，合計サイズが 2 バイト（16 ビット）の範囲内であれば同一ワード（16 ビット）位置に格納されます。

unsigned int 型のビットフィールドでは，連続したメンバの合計サイズが 16 ビットを超える場合は，入りきらないビットフィールドメンバを格納するための新しい 2 バイト領域が確保されます。

1.5.5 共用体

共用体のサイズは，共用体の中で最も大きなサイズを持つメンバを格納するのに必要なサイズになります。共用体のサイズが 3 バイト以上の奇数バイトになる場合，その共用体のサイズは偶数バイトに調整されます。

例 1

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

この例では，共用体の中で最も大きいサイズを持つメンバは配列 x で，そのサイズは 3 バイトです。この場合，共用体のサイズは偶数バイトに調整され，4 バイトになります。

例 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;
```

この例では，共用体のメンバ uc と bf はともに 1 バイトです。この場合，共用体のサイズは 1 バイトとなります。

1.6 アセンブリ言語との結合

1.6.1 外部名(関数名および変数名)の参照方法

static 宣言されていない関数や，static 記憶クラス指定子で指定されていないグローバル変数は，外部名として他のファイルからも参照することができます。これらの外部名は，アセンブリプログラムからも参照が可能です。

```
const char array[] = ”string”;
int gvar;
void func(void)
{
    /* 関数本体の処理 */
}
```

このプログラムを CCU8 でコンパイルしたときの出力アセンブリコードを示します。

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

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

    rseg $$NTABgsym
_array :
    DB  "string", 00H
```

出力アセンブリコードを見て分かるように，C プログラムで記述された名前 array，gvar，func の先頭にそれぞれ下線（_）が付加されています。逆に言うと C プログラムから参照可能な名前をアセンブリ言語で記述する場合には，それぞれの名前の先頭に下線（_）を付加しなければなりません。

そして，関数_func および配列_array は public 擬似命令によりパブリック宣言されています。また，変数_gvar は comm 擬似命令を用いて宣言されています。これらの宣言により，他のファイルからも参照することができるようになります。public 擬似命令および comm 擬似命令の詳細については，『MACU8 アセンブラパッケージ ユーザーズマニュアル』の「5.9 リンケージ制御擬似命令」を参照してください。

それぞれの外部名にはユーセージタイプと呼ばれる属性があります。具体的には，_func のユーセージタイプは CODE，_array のユーセージタイプは TABLE，_gvar のユーセージタイプは DATA です。ユーセージタイプについては『MACU8 アセンブラパッケージ ユーザーズマニュアル』の「2.4.5 ユーセージタイプとセグメントタイプ」を参照してください。

関数や変数がどのようなユーセージタイプを持つのかを以下に示します。

種類	ユーセージタイプ
関数名	CODE
const で修飾された変数名	TABLE
const で修飾されない変数名	DATA
nvdata プラグマで指定された変数名	NVDATA

アセンブリプログラムから外部名を参照する場合，EXTRN 擬似命令を使用します。このときにユーセージタイプが必要となります。

上の例に示される_func，_array，_gvar をアセンブリプログラムから参照する場合には，次のように宣言します。

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

1.6.2 レジスタ規約

CCU8 では，一定の規約に従ってレジスタを使用します。C 言語のプログラムとアセンブリ言語のプログラムを結合する場合には，アセンブリ言語でプログラムを記述する場合でもこの規約に従う必要があります。

1.6.2.1 CCU8 が使用するレジスタ

CCU8 はアセンブリコードを生成するときに，いくつかの汎用レジスタ，およびコントロールレジスタを使用します。CCU8 がコード生成に使用するレジスタは，次のとおりです。

CCU8 がコード生成に使用するレジスタ	用途
R0～R3	関数の引数，関数の戻り値，ワークレジスタ
R4～R13	ローカル変数，ワークレジスタ
FP (ER14)	スタックに割り当てられる関数引数およびローカル変数のベースポインタ
SP	スタックの操作（スタックの補正やローカル変数の領域確保）
DSR	FAR データアクセス
EA	メモリへのデータアクセス

1.6.2.2 関数内部で保存するレジスタ

CCU8 では，関数内部で保存するレジスタを次のように定めています。

関数の種類	関数内部での条件	条件を満たす場合に必要な操作
一般の関数	他の関数を呼び出す場合	LR をスタックに保存する
	R4～R15 のいずれかを使用する場合	左記のうち，使用するレジスタをスタックに保存する
割り込み関数およびソフトウェア割り込み関数	多重割り込みを許可する場合	ELR, EPSW をスタックに保存する
	他の関数を呼び出す場合	LR, EA, DSR, XR0 をスタックに保存する
	R0～R15，DSR，EA のいずれかを使用する場合	左記のうち，使用するレジスタをスタックに保存する

上記の仕様により，一般の関数呼び出しを行った場合，R0～R3，DSR，および EA の内容は保証されません（破壊されます）が，R4～R15 の内容は保証されます。ハードウェア割り込み

およびソフトウェア割り込みの前後においては，すべての汎用レジスタおよびコントロールレジスタ（退避用レジスタを除く）の内容は保証されます。

なお，コントロールレジスタのうち，SP（スタックポインタ）については，CCU8 は関数の呼び出し前と関数実行後との間において内容が変化しないことを前提としているため，呼び出される関数の内部では SP の内容を保持しておく必要があります。PSW（プログラムステータスワード）については，CCU8 は関数の呼び出し前と関数実行後との間において内容が変化することを前提としているため，呼び出される関数の内部では PSW の内容を保持しておく必要はありません。

1.6.3 関数の呼び出し規約

CCU8 では一定の規約にしたがって，C 言語の関数に引数を渡したり，C 言語の関数呼び出しから値を受け取ります。アセンブリ言語で C 言語の関数を呼び出したり，関数本体を記述する場合にも，この規約に従わなくてはなりません。

ここでは，はじめに関数の呼び出しがどのようなアセンブリコードに展開されるのかを例に示し，その後で関数呼び出しに関する規約について詳しく述べていきます。

C 言語の記述例

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

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

void main(void)
{
    res = func(nbuf, fbuf);  /* 関数呼び出し */
}

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

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

上記のような C 言語のプログラムが記述されている場合について，関数 func を呼び出す場合のアセンブリコードの出力，および関数 func 本体のアセンブリコードの出力を以下に示します。

出力アセンブリコードの例（関数の呼び出し側）

```
  _main   :

  ;;      res = func(nbuf, fbuf);
          mov     r0,     #SEG _fbuf           ;; 第２引数の設定
          push    r0                           ;;
          mov     r0,     #BYTE1 OFFSET _fbuf  ;;
          mov     r1,     #BYTE2 OFFSET _fbuf  ;;
          push    er0                          ;; 引数をスタックに格納

          mov     r0,     #BYTE1 OFFSET _nbuf  ;; 第１引数の設定
          mov     r1,     #BYTE2 OFFSET _nbuf  ;; 引数をレジスタに格納

          bl      _func                        ;; 関数の呼び出し

          add     sp,     #4                   ;; スタックポインタの補正

          st      er0,    NEAR _res            ;; 戻り値の参照
  ;;}
  _$$end_of_main :
          bal     $
```

関数を呼び出す場合は，次の順序にしたがって関数を呼び出します。

(1) 呼び出す関数に引数がある場合，「1.6.4 関数引数の割り当て規則」にしたがって引数を設定します。

(2) 関数を呼び出します。

(3) 関数引数をスタックで渡した場合，スタックポインタの補正を行います。

(4) 戻り値を参照します。戻り値は「1.6.5 関数の戻り値規約」にしたがって，レジスタあるいは特定の領域に格納されます。

出力アセンブリコードの例（関数本体）

```
_func   :
        ;; この関数内で別の関数を呼び出す場合には，ここで"push lr"が必要
        push    fp              ;; 変更前の fp をスタックに退避

        mov     fp,     sp      ;; この時点での SP を FP にコピー

        add     sp,     #-02    ;; ローカル変数をスタックに確保

        push    bp              ;; この関数で使用するレジスタをスタック
        push    er8             ;; に退避
```

```
        mov     er8,    er0     ;; 第１引数を er8 にコピー

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

;;      return  cnt;
        l       er0,    -2[fp]  ;; 戻り値の設定
;;}
        pop     er8             ;; スタックに退避したレジスタを復帰する
        pop     bp              ;;

        mov     sp,     fp      ;; スタックポインタを関数呼び出し直後の
                                ;; 状態に戻す

        pop     fp              ;; fp を元に戻す

        rt  ;; 関数の入り口で”push lr”を行った場合には”pop pc”で戻る
```

関数の本体側では，次の順序にしたがって関数内の処理が行われます。

(1) 関数内で他の関数を呼び出す（ノード関数と呼びます）場合，リンクレジスタ LR をスタックに退避します。関数内で他の関数を呼び出さない（リーフ関数と呼びます）場合は，LR をスタックに退避する必要はありません。

(2) スタックで渡された引数を参照する場合，またはローカル変数をスタックに割り当てる場合，それらをアクセスするための準備を行います。スタックで渡された引数やスタックに割り当てられたローカル変数をアクセスする場合には，フレームポインタ FP を利用します。この場合，FP を更新する前にスタックに退避します。なお，ローカル変数をスタックに割り当てる場合，SP から定数を減算（ここでは“add sp, #-02”）することでローカル変数の領域を確保します。

(3) 関数内で保存する必要のあるレジスタを使用する場合，それらをスタックに退避します。関数内で保存する必要のあるレジスタについては，「1.6.2 レジスタ規約」を参照してください。上記プログラム例のスタックの様子（これをスタックフレームと呼びます）は，次のようになります。

関数 func 内でのスタックフレームの構造

(4) 関数本体内部の処理が行われます。

(5) 関数が戻り値を持つ場合，戻り値の設定を行います。

(6) スタックに退避したレジスタがあれば，それらを復帰します。

(7) スタックポインタ（SP），フレームポインタ（FP）の値を，関数呼び出し直後の状態に戻します。

(8) 関数入り口で“PUSH LR”を行っていれば（ノード関数であれば），“POP PC”を使用して呼び出し元に戻ります。“PUSH LR”を行っていなければ（リーフ関数であれば），“RT”を使用して呼び出し元に戻ります。

1.6.4 関数引数の割り当て規則

関数の引数はレジスタに割り当てられる場合と，スタックに割り当てられる場合があります。

(1) __noreg で修飾された関数の場合

__noregで修飾された関数の引数は，すべてスタックに割り当てられます。各引数は，「1.6.4.2 引数のスタックへの割り当て規則」にしたがってスタックに割り当てられます。

(2) 可変引数を扱う関数の場合

__noregで修飾されていない関数でも，可変引数を扱う関数の引数は，すべてスタックに割り当てられます。各引数は，「1.6.4.2 引数のスタックへの割り当て規則」にしたがってスタックに割り当てられます。

(3) 上記以外の関数の場合

レジスタに割り当てることが可能な引数は，「1.6.4.1 引数のレジスタへの割り当て規則」にしたがってレジスタに割り当てます。レジスタに割り当てることができない引数は，「1.6.4.2 引数のスタックへの割り当て規則」にしたがってスタックに割り当てられます。

1.6.4.1 引数のレジスタへの割り当て規則

引数をレジスタに割り当てる場合，レジスタは R0, R1, R2, R3 が使用されます。

レジスタに割り当てられる引数の型は，char 型，unsigned char 型，short 型，unsigned short 型，int 型，unsigned int 型，long 型，unsigned long 型，float 型，およびポインタ型です。それ以外の型のものや，レジスタが足りないためにレジスタに割り当てることができない場合には，引数はスタックに割り当てられます。

引数をレジスタに割り当てる場合の規則は次のとおりです。

(1) 引数の左から右の順で，引数をレジスタ R0 から R3 に割り当てていきます。

(2) char 型および unsigned char 型の引数は，R*n*(*n* は 0, 1, 2 または 3)に割り当てます。

(3) short 型，unsigned short 型，int 型および unsigned int 型の引数は，ER*n*(*n* は 0 または 2)に割り当てます。レジスタへの格納順序は，引数の下位バイトが R*n*，引数の上位バイトが R*n*+1 となります。

(4) long 型，unsigned long 型および float 型の引数は，XR0 に割り当てます。レジスタへの格納順序は，引数の最下位バイトが R0，2 バイト目が R1，3 バイト目が R2，最上位バイトが R3 となります。

(5) near ポインタの引数は，ER*n*(*n* は 0 または 2)に割り当てます。レジスタへの格納順序は，引数の下位バイトが R*n*，引数の上位バイトが R*n*+1 となります。

(6) far ポインタまたは huge ポインタの引数は，R2:ER0 に割り当てます。レジスタへの格納順序は，ポインタのオフセット部の下位バイトが R0，オフセット部の上位バイトが R1，物理セグメントアドレス部が R2 となります。

(7) 引数として割り当てられた R0，R1，R2 および R3 は，必要に応じて関数の入り口で R8，R9，R10 および R11 にコピーして保存します。この場合，R8 から R11 を事前にスタックに退避する必要があります。

例 1

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

この場合，引数 a は R0，引数 b は R1，引数 c は ER2 に割り当てられます。

例 2

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

この場合，引数 a は R0，引数 b は ER2 に割り当てられます。ただし，引数 c はレジスタには割り当てられずスタックに割り当てられます。

例 3

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

この場合，引数 fp は R2:ER0，引数 a は R3 に割り当てられます。

例 4

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

この場合，引数 a は R0 に割り当てられます。引数 fp はレジスタに割り当てられず，スタックに割り当てられます。

例 5

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

この場合，引数 a は R0 に割り当てられます。引数 fp はレジスタに割り当てられず，スタックに割り当てられますが，引数 b は R1 に，引数 c は ER2 に割り当てられます。

1.6.4.2 引数のスタックへの割り当て規則

(1) 引数をスタックに割り当てる場合，引数は右から左へプッシュしていきます。

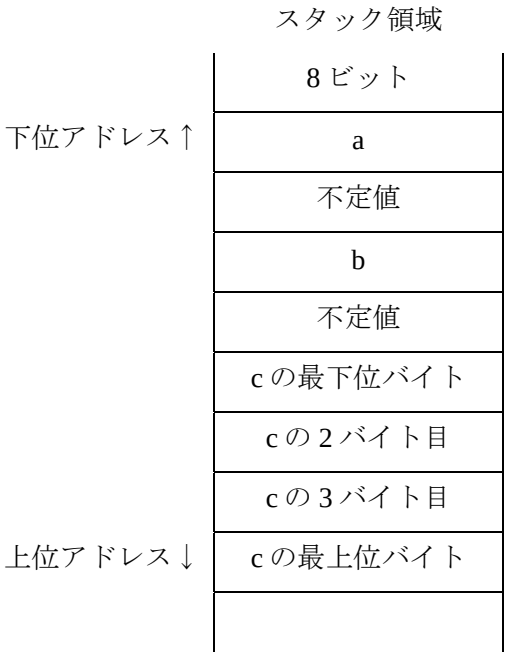
(2) 引数をスタックにプッシュする場合，上位アドレス側からプッシュします。

(3) 奇数バイトの引数をスタックにプッシュする場合，1 バイト余計に消費します。このとき，上位バイトの内容は不定値となります。

例

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

この例では，__noreg 修飾子により，関数の引数をすべてスタックに割り当てるように指示しています。引数は c，b，a の順にスタックにプッシュされます。この場合，スタックへの割り当てイメージは次のようになります。



1.6.5 関数の戻り値規約

関数の戻り値が 4 バイト以下の場合には CCU8 は関数の戻り値を R0，R1，R2，R3 レジスタ（使用するレジスタ数は戻り値のサイズによります）に割り当てます。戻り値が double 型，構造体型または共用体の場合には，CCU8 は戻り値が代入される変数のアドレスを最初の引数に割り当てます。

1.6.5.1 戻り値をレジスタに割り当てる場合

戻り値がレジスタに割り当てられる場合の規則を以下に示します。

戻り値の型	サイズ	レジスタ
char, unsigned char	1 バイト	R0
short, unsigned short, int, unsigned int	2 バイト	R1:R0（ER0）
long, unsinged long, float	4 バイト	R3:R2:R1:R0（XR0）
near ポインタ	2 バイト	R1:R0（ER0）
far ポインタ，huge ポインタ	3 バイト	R2:R1:R0（R2:ER0）

1.6.5.2 戻り値が double 型，構造体型，共用体型の場合

double 型，構造体型，共用体型を戻り値にもつ関数の場合，CCU8 は戻り値が代入される変数のアドレスを最初の引数に割り当てます。

例を以下に示します。

例

```
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;
}
```

関数呼び出し側では，戻り値を格納する変数が near データか far データによって関数に引き渡すポインタの内容が異なります。

関数呼び出し側は，次のように展開されます。

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

;;    ndbl = dbl_func();
        ;; 戻り値の格納先が near 変数の場合，変数のアドレスをそのまま渡す
        mov     r0,     #BYTE1 OFFSET _ndbl
        mov     r1,     #BYTE2 OFFSET _ndbl
        bl      _dbl_func ;; _dbl_func の内部で_ndbl に戻り値を書きこむ

;;    fdbl = dbl_func();
        ;; 戻り値の格納先が far 変数の場合，この関数内で確保したスタック上の
        ;; 領域を示すアドレスを渡す
        mov     er0,    fp
        add     er0,    #-8
        bl      _dbl_func ;; _dbl_func の内部で-8[fp]に戻り値を書きこむ
        lea     -8[fp]
        l       qr0,    [ea]    ;; スタック(-8[fp])から戻り値を取り出す
        mov     r8,     #SEG _fdbl
        lea     OFFSET _fdbl
        st      qr0,    r8:[ea] ;; スタックから取り出した戻り値を far 変数
                                ;; の領域にコピーする
;;}
```

```
        pop     r8
        mov     sp,     fp
        pop     fp
        pop     pc
```

戻り値を格納する変数が near 変数の場合には，変数のアドレスがそのまま関数に渡されます。この場合，呼び出された関数側の中では戻り値を near 変数（ここでは ndbl）に直接書き込むことになります。

戻り値を格納する変数が far 変数の場合には，呼び出す側の関数（ここでは fn()）で戻り値を一時的に格納する領域をスタック上に確保し，そのアドレスを引数として渡します。この場合，呼び出された関数の中では戻り値をスタックに書きこむことになります。したがって，関数の呼び出し側（ここでは fn()）に復帰したあと，さらにスタックから戻り値を取り出して，それを far 変数（ここでは fdbl）にコピーするという処理が行われるため，かなり冗長な処理になります。

例では double 型を返す関数について説明しましたが，構造体型，共用体型を返す関数も同様の展開になります。

関数本体部分は，次のように展開されます。

```
_dbl_func       :
        push    er8
        mov     er8,    er0
;;    return dbl_var;
        lea     OFFSET _$ST0
        l       qr0,    [ea]
        lea     [er8]           ;; 引数で指定された格納先のアドレスを設定
        st      qr0,    [ea]    ;; 戻り値を格納先のアドレスにコピー
;;}
        pop     er8
        rt
```

1.6.6 関数の間接呼び出し

関数の間接呼び出しは，スモールモデルとラージモデルの場合とで異なります。

スモールモデルの場合，関数へのポインタは 2 バイトですので，16 ビットのレジスタを使用した間接呼び出しが行われます。

ラージモデルの場合，関数へのポインタは 3 バイトになるため 16 ビットレジスタを使用した間接呼び出しはできません。このため，CCU8 ではエミュレーションライブラリルーチンを用いて関数の間接呼び出しを実現しています。

例

C 言語での記述

```
void (*fp)(void);
```

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

この例では，関数へのポインタ fp の内容を参照して，関数の間接呼び出しを行っています。

上記のプログラムをスモールモデルでコンパイルした場合と，ラージモデルでコンパイルした場合のアセンブリ出力を以下に示します。

アセンブリ出力（スモールモデルの場合）

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

スモールモデルの場合，関数へのポインタは 2 バイトになります。fp の内容をワードレジスタ ER0 にロードし，bl 命令を使用して関数を呼び出します。

アセンブリ出力（ラージモデルの場合）

```
;;    fp();
    l       r0,    NEAR _fp+02h
    push    r0
    l       er0,   NEAR _fp
    push    er0
    bl      __indru8lw ;;エミュレーションライブラリルーチン
                       ;;(実際には POP PC を行うのみ)
    ;; <= (A)
```

ラージモデルの場合，関数へのポインタは 3 バイトになります。fp の内容をスタックにプッシュし，bl 命令を使用してエミュレーションライブラリルーチン__indru8lw を呼び出していますが，ここが重要なポイントとなります。実際には，__indru8lw の内部では単に“POP PC”を行っているだけですが，“bl __indru8lw”のかわりにここで“POP PC”を行ってしまうと，リンクレジスタ（LR）の内容が更新されないために (A)の位置に戻ることができなくなってしまいます。“bl __indru8lw”を行うことによりリンクレジスタが更新され，(A)の位置に戻ることができるようになります。

1.6.7 C プログラムから呼び出す関数をアセンブリ言語で作成する方法

C プログラムから呼び出す関数をアセンブリ言語で作成する場合，はじめに C 言語でプログラムを作成してからそれをコンパイルし，その出力アセンブリコードに対して変更を加えていくようにすると比較的簡単に作成できます。

C プログラム記述例

```
int gi1, gi2;
unsigned char gc;
int retval;
```

```
int function(int arg1, unsigned char arg2)
{
    volatile int local = gi1;   /* ローカル変数へのアクセス */
    gi2 = arg1;                 /* 引数 arg1 へのアクセス */
    gc = arg2;                  /* 引数 arg2 へのアクセス */
    return retval;              /* 戻り値の設定 */
}
```

上記のプログラムでは，ローカル変数および引数へのアクセス，そして戻り値の設定を行うプログラムが記述されています。このプログラムを CCU8 でコンパイルすると，次のようなアセンブリコードを出力します。なお，コンパイルオプションは，他のプログラムと同じオプションを指定するようにしてください。

アセンブリコード出力例

```
_function       :               ;; 関数名の先頭に下線(_)が付加される
        push    fp              ;; スタックに割り付けられる変数をアクセス
        mov     fp,     sp      ;; するための準備
        add     sp,     #-02    ;; ローカル変数の領域確保
        push    er8             ;; er8 をこの関数で使用するため，スタック
                                ;; に退避する
        mov     er8,    er0     ;; 引数 arg1 を er8 にコピー

;; ********** ここからが関数本体の処理になります **********
;;    volatile int local = gi1;   /* ローカル変数へのアクセス */
        l       er0,    NEAR _gi1
        st      er0,    -2[fp]

;;    gi2 = arg1;                 /* 引数 arg1 へのアクセス */
        st      er8,    NEAR _gi2

;;    gc = arg2;                  /* 引数 arg2 へのアクセス */
        st      r2,     NEAR _gc

;;    return retval;              /* 戻り値の設定 */
        l       er0,    NEAR _retval

;; ********** ここまでが関数本体の処理になります **********
;;}
        pop     er8             ;; 関数から呼び出し元に戻るための処理
        mov     sp,     fp      ;; 同上
        pop     fp              ;; 同上
        rt                      ;; 呼び出し元に戻る
```

```
        public _function        ;; static 指定子がない場合，public 宣言
                                ;; される
```

CCU8 によって出力されたアセンブリコードを参照することで，引数やローカル変数がどこに割り付けられているのか，戻り値がどのように設定されるのかを知ることができます。上記の出力アセンブリコードに対し，関数本体部分について変更を加えていきます。

1.7 プログラム記述における注意事項

1.7.1 プロトタイプ宣言の必要性

関数のプロトタイプ宣言は非常に重要です。関数のプロトタイプ宣言を記述しないと，プログラムが誤動作をする恐れがあります。

次の例を見てください。

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

上記のプログラムを CCU8 でコンパイルすると，出力は次のようになります。

```
_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
```

関数 subfunc については関数のプロトタイプ宣言がありません。このため，CCU8 は関数 subfunc に対して引数の型，および戻り値の型チェックを行いません。

関数 subfunc の呼び出しにおいて，(1)では引数として near ポインタおよび int 型の値を渡しているのに対し，(2)では引数として far ポインタおよび long 型の値を渡しています。関数呼び出しにおけるアセンブリコードを見てみると，関数引数の設定の仕方が(1)と(2)とで明らかに異なっています。この結果，上記のようなプログラムを実行した場合に，正常に動作しなくなってしまいます。

このような関数引数の引き渡しにおける不整合を防ぐためにも，関数のプロトタイプ宣言を記述することを強く推奨いたします。

次のようにプロトタイプ宣言を記述した場合，

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

(2)の記述に対しては引数の型が異なりますので，CCU8 はその旨を伝えるメッセージを表示します。

なお，CCU8 ではコマンドラインオプション/Zg オプションを指定することにより，ファイル中に定義された関数に対するプロトタイプ宣言のリストを生成することができます。

例えば，次のような C プログラムの記述に対し，

```
/* ファイル名 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;
}
```

コンパイル時に/Zg オプションを指定すると，次のようなプロトタイプ宣言ファイルが生成されます。

プロトタイプ宣言ファイルの出力例 (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);
```

このプロトタイプ宣言ファイルを，#include 前処理指令を使用してインクルードすることで，CCU8 は関数の引数の型，および戻り値の型をチェックすることができます。

注意

/Zg オプションを使用する場合，構造体型，共用体型，および列挙型（またはこれらの型のオブジェクトへのポインタ）を引数，戻り値に持つ関数については，それらの構造体型，共用体型，および列挙型の型宣言においてタグ名を指定するようにしてください。

例えば，次のように記述した場合，

```
/* ファイル名 test2.c */
typedef struct { /* タグ名なしの構造体 */
    int     memb1;
    int     memb2;
} ST1;

typedef struct st2 { /* タグ名付きの構造体 */
    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;
}
```

プロトタイプ宣言ファイルは次のようになります。

```
extern  void fn1(struct  *pST1); /* タグ名がないため，正しく出力されない */
extern  void fn2(struct st2 *pST2);
```

この例に示すように，構造体型，共用体型，および列挙型にタグ名が存在しない場合，プロトタイプ宣言が正しく出力されませんので，型宣言のときにタグ名を指定するようにしてください。

また，プログラムによっては，/Zg オプションを指定して作成したプロトタイプ宣言ファイルを，そのままインクルードすることができない場合もあります。例えば，次のような場合です。

```
/* ファイル名 test3.c */
static void static_fn(void)
{
    /* 処理 */
}

void global_fn(void)
{
    /* 処理 */
}
```

この場合，プロトタイプ宣言ファイルは次のようになります。

```
static  void static_fn(void);
extern  void global_fn(void);
```

上記のプロトタイプ宣言ファイルには，static 宣言された関数も含まれていますので，他のファイルからそのままインクルードすることはできません。この場合，プロトタイプ宣言ファイルをもとに static 宣言された関数 static_fn のプロトタイプ宣言を削除したプロトタイプ宣言のファイルを作成し，それをインクルードするようにしてください。

2 コンパイル，リンク

2.1 コンパイラが生成するセグメント名

CCU8 は C 言語プログラムの記述に応じて，以下のリロケータブルセグメントを生成し，それらのリロケータブルセグメントにプログラムコードやデータを配置します。

Ver.3.30 から/Zc オプションの有無で、生成されるセグメントタイプ CODE のセグメント名が異なります。詳細については、『CCU8 ユーザーズマニュアル』を参照してください。

セグメント名	セグメントタイプ	物理セグメント属性	リロケータブルセグメントに格納される内容
$$*funcname*$*filename*	CODE	#0	スモールモデルの関数または割り込み関数（/Zc オプションを指定しない時のみ出力）
$$*funcname*$*filename*	CODE	ANY	ラージモデルの関数（/Zc オプションを指定しない時のみ出力）
$$NCOD*filename*	CODE	#0	スモールモデルの関数（/Zc オプション指定時のみ出力）
$$FCOD*filename*	CODE	ANY	ラージモデルの関数（Zc オプション指定時のみ出力）
$$INTERRUPTCODE	CODE	#0	割り込み関数（Zc オプション指定時のみ出力）
$$TAB*constname*$*filename*	TABLE	#0	const で修飾された near 変数（スタティックグローバル変数，スタティックローカル変数）（/Zc オプションを指定しない時のみ出力）
$$TAB*constname*$*filename*	TABLE	ANY	const で修飾された far 変数（スタティックグローバル変数，スタティックローカル変数）（/Zc オプションを指定しない時のみ出力）
$$NTAB*filename*	TABLE	#0	const で修飾された near 変数（スタティックグローバル変数，スタティックローカル変数）（Zc オプション指定時のみ出力）
$$FTAB*filename*	TABLE	ANY	const で修飾された far 変数（スタティックグローバル変数，スタティックローカル変数）（Zc オプション指定時のみ出力）

セグメント名	セグメントタイプ	物理セグメント属性	リロケータブルセグメントに格納される内容
$$NVAR*filename*	DATA	#0	const で修飾されていない初期化なしの near 変数（スタティックグローバル変数，スタティックローカル変数）
$$FVAR*filename*	DATA	ANY	const で修飾されていない初期化なしの far 変数（スタティックグローバル変数，スタティックローカル変数）
$$NINITVAR	DATA	#0	const で修飾されていない初期化付きの near 変数（グローバル変数，スタティックグローバル変数，スタティックローカル変数）
$$NINITTAB	TABLE	ANY	const で修飾されていない初期化付きの near 変数（グローバル変数，スタティックグローバル変数，スタティックローカル変数）の初期値
$$FINITVAR*filename*	DATA	ANY	const で修飾されていない初期化付きの far 変数（グローバル変数，スタティックグローバル変数，スタティックローカル変数）
$$FINITTAB*filename*	TABLE	ANY	const で修飾されていない初期化付きの far 変数（グローバル変数，スタティックグローバル変数，スタティックローカル変数）の初期値
$$init_info	TABLE	ANY	const で修飾されていない初期化付きの変数の初期化情報テーブル
$$NNVDATA*filename*	NVDATA	#0	NVDATA プラグマで指定された near 変数
$$FNVDATA*filename*	NVDATA	ANY	NVDATA プラグマで指定された far 変数
$$content_of_init	CODE	コンパイル時のメモリモデルに依存	ABSOLUTE プラグマで指定された，const で修飾されていない変数の初期化コード

funcname は C ソースファイル中の関数名，*constname* は C ソースファイル中の const 変数名、*filename* はコンパイル対象の C ソースファイル名のベース名です。

リンク時には/CODE，/DATA，/TABLE または/NVDATA オプションを使用して，上記のリロケータブルセグメントを特定の領域に配置することが可能です。

例えば，file1.c と file2.c をラージモデルでコンパイルし，それらのファイルに含まれるプログラムコードを 1:8000H 番地に配置したい場合には，次のように指定します。

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

/COMB オプションは CODE タイプおよび TABLE タイプのセグメントのみに対して有効です。

const で修飾されていない未初期化のグローバル変数に対しては，CCU8 は共有シンボルとして生成します。共有シンボルに対しては，リンカのオプションによって特定の領域に配置することはできません。

実際の C 言語プログラムの記述とコンパイラが生成するリロケータブルセグメントとの対応例を以下に示します。

C 言語の記述例	出力例および解説
`int __near gi_ram1;`	const なしの未初期化のグローバル変数は，共有シンボルとして生成され，RAM へ配置されます。 `_gi_ram1 comm data 02h #00h`

`int __near gi_ram2 = 10;`	const なしの初期化付きのグローバル変数は，初期値データと変数領域とに分けられて生成され，初期値データは ROM へ，変数領域は RAM へ配置されます。そして，プログラム開始時にスタートアップルーチンにより初期値が変数領域にコピーされます。
	
	`rseg $$NINITTAB`
	`dw  0ah`
	
	`rseg $$NINITVAR`
	`_gi_ram2 :`
	`ds  02h`

`const int __far gi_rom;`	const で修飾された変数は，ROM に配置されます。
	
	`rseg $$FTABsample`
	`_gi_rom :`
	`dw  00h`

C 言語の記述例	出力例および解説

int f(int, int);
void main(void)
{
　f(10, 20);
}
int f(int a, int b)
{
　return a + b;
}
	プログラム部分は，ROM に配置されます。
	
	　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 プログラム実行の流れ

CCU8 を用いて作成したプログラムは次の手順で実行されます。

(1) リセット端子入力によるパワーオンリセット

(2) スタートアップルーチン（$$start_up）の実行

(3) main 関数の呼び出し

(4) ユーザプログラムの実行

リセット後，ユーザが作成したプログラムを実行する前に，使用されるレジスタや RAM の初期化を適切に行わなければなりません。この初期化を行うのがスタートアップルーチンです。

CCU8 では，基本的な動作を記述したスタートアップルーチンを，スタートアップファイルという形で提供しています。

スタートアップファイルのソースファイルはCCU8 に添付されていますので，ユーザは，これをもとにカスタマイズすることができます。カスタマイズの方法については，「2.5 スタートアップファイルの説明」で説明しています。

2.3 main 関数について

CCU8 は，main 関数のあるファイルをコンパイルすると，リセットベクタ 2H にスタートアップルーチンを登録するコードを出力します。これによって，リセット端子入力によるリセットが起きた場合，スタートアップファイルのコードが実行されます。

スタートアップルーチンが実行されると，main 関数へ処理が移ります。CCU8 の出力するコードは，main 関数の処理を終了すると無限ループになります。

2.4 リンク時に必要となるファイル

ターゲット上で実行可能なモジュールを構築するためには，ユーザが作成するプログラムのほかに，スタートアップファイル，エミュレーションライブラリ，C ランタイムライブラリをリンク時に指定する必要があります。

2.4.1 スタートアップファイル

スタートアップファイルの名前は，マイクロコントローラの種類やメモリモデル，ROM ウィンドウの有無が分かるように命名されています。

スタートアップファイルの命名則は次のようになっています。

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

先頭の‘s’は固定です。

target はマイクロコントローラの種類を示すもので，マイクロコントローラ名の先頭の“ML”を取り除いたものです。

{s|l}はメモリモデルを示し，s か l のどちらかになります。s はスモールモデル用，l はラージモデル用を示します。

[w]は ROM ウィンドウの有無を示します。w が付加されているものは，ROM ウィンドウあり，w が付加されていないときは ROM ウィンドウなしであることを示します。

例えば，s610001sw.obj はターゲットのマイクロコントローラが ML610001 で ROM ウィンドウを使用するスモールモデル用のスタートアップファイルとなります。

アセンブリソースのファイル名は，スタートアップファイル名の拡張子を“.asm”に置き換えたものになります。

2.4.2 エミュレーションライブラリ

エミュレーションライブラリには，次の種類があります。

ライブラリファイル名	内容
longu8.lib	整数演算用エミュレーションライブラリ
doubleu8.lib	倍精度浮動小数点演算用エミュレーションライブラリ
floatu8.lib	単精度浮動小数点演算用エミュレーションライブラリ

エミュレーションライブラリは，リンク時のコマンドラインにおいて明示的に指定する必要はありません。リンク時に/CC オプションが指定されると，RLU8 は未解決なシンボルが残った場合に，上記のエミュレーションライブラリを自動的にサーチし，必要なモジュールをライブラリファイルから取り出してリンクします。

詳しい内容については，『MACU8 アセンブラパッケージ　ユーザーズマニュアル』の「7.2.1.4 *libraries* フィールド」を参照してください。

2.4.3 C ランタイムライブラリ

C ランタイムライブラリには，次の種類があります。

ライブラリファイル名	内容
LU8100SW.LIB	スモールモデル用 C ランタイムライブラリ
LU8100LW.LIB	ラージモデル用 C ランタイムライブラリ

C ランタイムライブラリは，ANSI/ISO9899 C が提唱するライブラリのサブセット版です。このライブラリに含まれる strcpy や memcpy などのライブラリルーチンを使用している場合は，リンク時のコマンドラインにおいて明示的に指定する必要があります。メモリモデルに応じて適切なライブラリファイルを指定してください。

C ランタイムライブラリの詳しい内容については，『RTLU8 ランタイムライブラリリファレンスマニュアル』を参照してください。

IDEU8 から指定する場合は、[ターゲットオプション][一般]タブの[ライブラリフィールドへ追加]フィールドに LU8100SW.LIB または LU8100LW.LIB を指定してください。

2.5 スタートアップファイルの説明

CCU8 で提供されるスタートアップファイルには，

- メモリモデルの設定
- ROM WINDOW の設定
- 割り込みベクタの設定
- 内部 RAM の初期化
- 初期値のある C 変数の初期化
- セグメントレジスタ（DSR）の設定
- main 関数の呼出し

などの処理が記述されています。

以下，スタートアップファイルの内容，および，カスタマイズの方法について順番に説明していきます。

2.5.1 コメントの読み方

```
/*********************************************************
    ML610001 start up assembly source file ←対象機種
    for CCU8 version 3.XX
    LARGE CODE MODEL                       ←メモリモデル
    ROMWINDOW  0-7FFFh                     ←ROM WINDOW 領域の範囲
```

```
    Version 1.00                                    ←本ファイルのバージョン
    Copyright (C) 2008 LAPIS Semiconductor CO., LTD.
*********************************************************/
```

スタートアップファイル先頭のコメントには，

- 対象機種
- メモリモデル
- ROM WINDOW 領域の範囲

が記述されています。 この情報には，コメントの後に記述してある擬似命令を使ったアセンブラへの初期設定が反映されています。

スタートアップファイルをアセンブルする場合は「2.5.15 スタートアップファイルの再アセンブル」を読み，オプションを適切に設定してください。

2.5.2 擬似命令による初期設定

スタートアップファイルの先頭では，擬似命令を使ってプログラムを実行する環境を指定しています。

```
type(M610001)           ←機種の指定
model   large, far      ←メモリモデルおよびデータモデルの指定
romwindow   0, 7fffh    ←ROM WINDOW 領域の範囲指定
```

以下，擬似命令を順に説明していきます。

TYPE 擬似命令

TYPE 擬似命令によって，対象となる機種の指定を行います。カッコ()の中には DCL ファイル名を記述します。DCL ファイルとは，機種依存の情報が書かれているテキストファイルで，RASU8 が使用します。

DCL ファイルは，提供したインストーラが作成した DCL という名前のディレクトリに格納されています。拡張子は“.DCL”です。

MODEL 擬似命令

メモリモデルおよびデータモデルを設定します。“model”の後にカンマ（,）で区切って記述します。以下の文字列を使って指定します。

メモリモデル

```
small : Small コードモデル
large : Large コードモデル
```

データモデル

```
near : near データモデル
far  : far データモデル
```

ROMWINDOW 擬似命令

ROMWINDOW 擬似命令によって，ROM ウィンドウの領域を設定しています。これはアセンブラのチェックやリンカのチェックに使用されます。

2.5.3 シンボルの宣言

ここではスタートアップファイル中で使用するシンボルの定義を行っています。

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

EXTRN 擬似命令

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

別のファイルで PUBLIC 宣言されたシンボルを参照するには，EXTRN 擬似命令を使います。“extrn”の後には，属性を，そのあとにシンボルを記述します。

“extrn code:_main”は，_main というコードに割り付けられたシンボルを宣言しています。_main は，main 関数を表し main へのジャンプに利用します。

“extrn data:_$$SP”では，_$$SP というデータに割り付けられたシンボルを宣言しています。_$$SP は，スタックシンボルという特殊なシンボルで，スタックセグメントの“終了アドレス+1”の値が格納されています。スタックポインタの初期化に使います。

PUBLIC 擬似命令

```
public  $$start_up
```

PUBLIC 擬似命令によってスタートアップファイルのラベルを宣言しています。別のファイルから参照されるシンボルは，必ず PUBLIC 擬似命令で宣言します。

main 関数のある C ソースファイルをコンパイルすると，リセットベクタ（プログラ ムメモリ空間のアドレス 2h）に$$start_up ルーチンの開始アドレスが設定されます。 これによって，リセット端子入力によるリセットがあると，$$start_up からプログラムの実行が始まります。

2.5.4 リセットベクタの設定

スタックポインタの初期化

ここではスタックポインタの初期化を行なっています。

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

CSEG 擬似命令でコードセグメントつまりプログラムメモリ空間がはじまります。CSEG 擬似命令は，アセンブル時にアドレスが決定するアブソリュートセグメントを定義します。

前述の EXTRN 擬似命令で宣言されたスタックポインタをあらわすシンボル_$$SP を，物理セ

グメント#0 の 0H 番地に設定します。このシンボルは，リンク時にスタックセグメントの“終了アドレス+1”の値に設定されます。スタックポインタの初期化は，リセット時に一番最初に行なわれます。

BRK リセットベクタの設定

ここでは，ELEVEL が 0 または 1 の状態において BRK 命令が実行されたときに参照されるリセットベクタを初期化しています。

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

DW 擬似命令のオペランドに記述しているのは，BRK リセット処理用のルーチンのラベルです。このルーチンは，スタートアップファイル内で定義されています。

リセット端子入力時，または ELEVEL が 2 または 3 の状態において BRK 命令が実行されたときに参照されるリセットベクタアドレス 2H 番地は，CCU8 によって main 関数が記述された C ソースファイルで定義されます。

2.5.5 スタートアップルーチンの開始アドレス

スタートアップルーチンのコードは，リロケータブルなコードセグメントの中に記述されています。したがって，スタートアップルーチンの開始アドレスはリンカによって決定されます。

もし，$$start_up のアドレスを固定にさせたい場合は，以下のように変更すれば可能です。ただし，変更する場合には使用するベクタテーブル領域や SWI テーブル領域と重複しないように注意してください。また，物理セグメントは必ず#0 でなければなりません。

下の例では，$$start_up のアドレスを物理セグメント#0 の 100H 番地に設定しています。

標準の設定

```
$$NCODs610001lw  segment code    #0
        rseg    $$NCODs610001lw
$$start_up:
        ↓ 変更
        cseg    at 0:100h
$$start_up:
```

2.5.6 リセット処理ルーチンの記述

ここでは，BRK リセットベクタ 4H に対応するルーチンを記述しています。この処理はユーザのプログラムによって異なるため，最低限の記述しかしていません。

この処理はユーザプログラムの用途にしたがって自由に変更することができます。

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

2.5.7 メモリモデルの設定

メモリモデルが選択可能な機種で，その選択方法が SFR の設定による機種の場合は，メモリモデルの設定をスタートアップファイルで行ないます。

メモリモデルの設定について，くわしくは対象機種のユーザーズマニュアルをご覧下さい。

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

2.5.8 ROM ウィンドウ領域の範囲の設定

ROM ウィンドウ領域の範囲が選択可能な機種で，その選択方法が SFR の設定による機種の場合は，ROM ウィンドウ領域の範囲の設定をスタートアップファイルで行ないます。

ROM ウィンドウ領域の範囲の設定について，くわしくは対象機種のユーザーズマニュアルをご覧下さい。

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

2.5.9 SFR の初期化

ユーザが使用する SFR の初期化は，以下の部分に記述できます。もちろんこれ以外の場所に記述してもかまいません。

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

2.5.10 暗黙のゼロクリア

C 言語では，初期化されていないグローバル変数は暗黙的にゼロクリアされます。 この初期化ルーチンにより，初期化されていないグローバル変数が 0 に初期化されます。ゼロクリアが必要なければ，この部分は削除してもかまいません。

2.5.10.1 物理セグメント#0 の RAM 領域のゼロクリア

```
;-----------------------------------------
;       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 物理セグメント#1 以上の RAM 領域のゼロクリア

こちらのゼロクリアは，物理セグメント#1 以上に RAM が実装されている時にのみ行ないます。以下にサンプルプログラムを示しますので，これを参考にして実装に合わせてカスタマイズしてください。

```
;-----------------------------------------
;       Far Data memory zero clear
;        (1:0000h - 1:FFFFh)
;        (2:8000h - 3:7FFFh)
;-----------------------------------------
;       MOV     ER0,    #0      ;すでに値 0 であれば省略可能
;       MOV     ER2,    #0      ;
;       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 変数の初期化

ここでは，初期値のある変数の初期化が行なわれます。ここに記述されているのは near データの初期化だけで，far データの初期化に必要な情報は，CCU8 がアセンブリソース中に生成します。

2.5.11.1 初期化を行うときの手順

(1) はじめに初期化情報テーブル$$init_info より，コピー元のオフセットアドレス，コピー先のオフセットアドレス，コピーを行うサイズ，コピー元の物理セグメントアドレス，コピー先の物理セグメントアドレスを得ます。

(2) １つの初期化情報テーブルについてサイズがゼロになるまでワード単位でコピーを行ないます。

(3) 初期化情報テーブルの内容をすべて読み込むまでこの処理を繰り返します。

(4) 初期化情報テーブルの最後にはターミネータとして 0xffff が置かれます。このターミネータを読み込んだときに処理が終了します。

$$init_infoは，スタートアップファイルの最後の方で定義されています。「2.5.14.2 グローバル変数初期化用のセグメント定義」を参照してください。

```
;---------------------------------------------------------------
;       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 ABSOLUTE プラグマで定義されたデータの初期化

ABSOLUTE プラグマで定義されたデータで初期値を持つデータの初期化は，$$content_of_init というコードセグメントの中にプログラムで初期化されています。スタートアップファイルでこれをコールすることにより，これらの初期化を行ないます。

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

2.5.12 セグメントレジスタの初期化

DSR の設定を行います。物理セグメント#0 を示すようにします。

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

2.5.13 main へのジャンプ

スタートアップの処理がすべて終了すると，main 関数へジャンプします。main 関数は物理セグメント#1 以上のプログラム空間にあってもかまいません。

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

2.5.14 セグメントの定義

2.5.14.1 データ初期化用のセグメント定義

$$content_of_init は，プログラムによるデータの初期化ルーチンが納められたセグメントです。$$end_of_init はそのセグメントのターミネータを示します。

```
;-----------------------------------------------------
;       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 グローバル変数初期化用のセグメント定義

near変数用の初期化用のセグメント定義です。「2.5.11 変数の初期化」で使用されます。

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

$$init_info_end は，初期化ルーチンの終了を表します。0xFFFF が読み込まれると初期化ルーチンは終了します。$$init_info_end セグメントは，$$init_info の最後にリンクされなければなりません。これは/CC オプションを指定すると，リンカによって自動的に行われます。

2.5.15 スタートアップファイルの再アセンブル

スタートアップファイルを変更した場合は，再アセンブルを行いオブジェクトファイルを作成しなおします。

例

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

C 言語では大文字と小文字を区別しているため，/CD オプションは必ず指定します。

2.6 注意事項

ここでは，コンパイルおよびリンク時において注意すべきことについて説明します。

2.6.1 ターゲット CPU の設定

コンパイル時には必ず/T オプションを指定してください。

マイクロコントローラの名称が ML610001 の場合には，コンパイル時のオプションとして /TM610001 を指定します。/T オプションで指定された名前が DCL ファイルの名前として認識されます。

IDEU8 で設定する場合には，[プロジェクト]メニューから[オプション][コンパイル/アセンブル…]を選択し，[コンパイル/アセンブル]ダイアログの[一般]タブを選択します。

[一般]タブの[ターゲット MCU]で DCL ファイルの名前（ベース名のみ）を指定します。正しい名前を指定しないと，正常にコンパイル/アセンブル/リンクが行われませんので注意してください。

2.6.2 メモリモデル

メモリモデルは，デフォルトでスモールモデルが選択されます。

コンパイル時のメモリモデルのオプションは/MS と/ML です。メモリモデルをスモールモデルに設定する場合は/MS を，メモリモデルをラージモデルに設定する場合には/ML を設定してください。メモリモデルの異なるモジュールをリンクすることはできません。

IDEU8 で設定する場合には，[プロジェクト]メニューから[オプション][コンパイル/アセンブル…]を選択し，[コンパイル/アセンブル]ダイアログの[一般]タブを選択します。

[一般]タブの[メモリモデル]でスモールかラージを選択します。

メモリモデルは，それぞれのマイクロコントローラでサポートされるメモリモデルを設定しないと正常に動作しませんので，設定を確実に行ってください。例えば，メモリモデルがラージモデル固定のマイクロコントローラの場合，メモリモデルは必ずラージモデルに設定してください。

2.6.3 ROM ウィンドウ領域

CCU8 では，デフォルトで ROM ウィンドウ領域を使用する設定になっていますが，C プログラム上で ROMWIN プラグマを使用して明示的に ROM ウィンドウ領域を指定しない限り，ROM ウィンドウ領域の範囲は未定義の状態となっています。このようなモジュールをリンクする場合，最終的に ROM ウィンドウ領域の範囲が決定していないと，RLU8 は次のエラーメッセージ

を表示して処理を中断してしまいます。

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

この問題を解決するには，リンカ RLU8 のコマンドラインオプション/ROMWIN を使用して ROM ウィンドウ領域を指定します。ROM ウィンドウ領域の範囲が 0-7FFFH の場合は，次のように指定します。

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

IDEU8 で設定する場合には，[プロジェクト]メニューから[オプション][ターゲット…]を選択し，[ターゲットオプション]ダイアログの[メモリ設定]タブを選択します。

[メモリ設定]タブの[ROMWINDOW の設定][ROMWINDOW 領域を設定する]をチェックし，ROMWINDOW 領域の範囲を指定します。

ただし，通常はスタートアップファイルに ROM ウィンドウ領域の範囲が設定されていますので，スタートアップファイルをリンクする場合には上記のエラーは発生しません。

2.7 リロケータブルセグメントを特定の領域に割り付けるには

リロケータブルセグメントを特定の領域に割り付けるには，リンカのコマンドラインオプション（/CODE, /TABLE, /DATA, /NVDATA）を使用します。これらのオプションの詳細については，『MACU8 アセンブラパッケージ ユーザーズマニュアル』の「7.5.3 各オプションの機能」を参照してください。

IDEU8 で設定する場合には，[プロジェクト]メニューから[オプション][ターゲット…]を選択します。

[ターゲットオプション]ダイアログの[セグメント]タブで，特定の領域に割り付けたいリロケータブルセグメントのセグメント名を対応するセグメントタイプのテキストボックスに指定していきます。

例えば，file1.c と file2.c をラージモデルでコンパイルし，それらのファイルに含まれるプログラムコードを 1:8000H 番地に連続させて配置したい場合，

[優先的に割り付ける CODE セグメント]をチェックし，[セグメント指定]ボックスには次のように指定します。

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

さらに，file1 と file2 に含まれるプログラムコードを結合するために，[CODE/TABLE タイプのセグメント結合を指定する]をチェックし，[結合順序]ボックスに対し次のように指定します。

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

両側の丸カッコは必ず必要です。

2.8 HEX ファイルの作成

HEX ファイルを作成する場合，基本的にはオブジェクトモジュール中のすべてのデータを HEX ファイルに変換しますが，オブジェクトモジュール中の一部のデータのみを取り出して

HEX ファイルに変換することもできます。

2.8.1 モジュール中のすべてのデータを変換する

オブジェクトモジュール中のすべてのデータを HEX ファイルに変換する場合，コマンドラインでも IDEU8 でも指定できます。OHU8 の詳しい使い方については，『MACU8 アセンブラパッケージ　ユーザーズマニュアル』の「9 OHU8」を参照してください。

IDEU8 で設定する場合，[プロジェクト]メニューから[オプション]→[ターゲット...]を選択します。

[ターゲットオプション]ダイアログの[一般]タブで，[オブジェクトコンバータ]の[HEX 形式ファイルを出力する]をチェックします。そして，HEX ファイルの出力形式を[インテル HEX]にするか[S フォーマット]にするかを選択します。

2.8.2 モジュール中の一部のデータを変換する

オブジェクトモジュール中の一部のデータを HEX ファイルに変換する場合，コマンドラインで指定します。IDEU8 を使用してオブジェクトモジュール中の一部のデータを HEX ファイルに変換することはできません。

オブジェクトモジュール中の一部のデータを HEX ファイルに変換する場合，/R オプションを使用して次のように指定します。

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

この例では，アブソリュートオブジェクトモジュール SAMPLE.ABS に含まれるデータのうち，3:0 番地から 3:FFFFH 番地に含まれるデータのみを取り出して，それを HEX ファイルに変換しています。OHU8 の詳しい使い方については，『MACU8 アセンブラパッケージ　ユーザーズマニュアル』の「9 OHU8」を参照してください。

3 付録

3.1 マップファイルについて

RLU8 が出力するマップファイルには，ユーザが作成したプログラムに対するさまざまな情報が含まれています。マップファイルの読み方の詳細については『MACU8 アセンブラパッケージユーザーズマニュアル』の「7.7 マップファイル」を参照してください。

3.1.1 リンクされたモジュールの情報

どのようなモジュールがリンクされたかを知りたい場合には，マップファイルの以下の部分を参照することで分かります。

```
                        ------------------------
                         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
```

ここには，リンクされたすべてのモジュールが一覧として出力されます。RLU8 によって自動的にリンクされたライブラリ中のモジュールも出力されます。

3.1.2 メモリのマッピング情報

ROM や RAM などのメモリがどのようにマッピングされているかを知りたい場合には，以下の部分を参照することで分かります。

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

上の例は，物理セグメント#0 のプログラムメモリ空間に実装されたメモリのマッピング情報

を示しています。

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

上の例は，物理セグメント#0 のデータメモリ空間に実装されたメモリのマッピング情報を示しています。

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

上の例は，物理セグメント#1 以上のメモリ空間に実装されたメモリのマッピング情報を示しています。

3.1.3 各メモリへの割り付け情報

各セグメントがどのメモリに割り付けられたかを知りたい場合には，以下の部分を参照することで分かります。

```
                          ------------------
                           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
```

上の例は，物理セグメント#0 のプログラムメモリ空間に割り付けられたセグメントの情報を示しています。各 Type フィールドの先頭に S という文字が表示されていますが，これは対象のシンボルがセグメントシンボルであることを示します。

CODE タイプのセグメントには，プログラムが格納されています。TABLE タイプのセグメントには，C プログラムの変数の初期値や const で修飾された参照専用のテーブルデータが格納されています。CCU8 によって生成されるセグメントの名前は「2.1 コンパイラが生成するセグメント名」の命名則に従っていますので，セグメント名からそれがどのモジュールに属しているのかを知ることができます。例えば，セグメント$$FCODkeytask はモジュール keytask に属しているプログラムコードを含んでいることが分かります。

```
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)
```

上の例は，物理セグメント#0 のデータメモリ空間に割り付けられたセグメントおよび共有シンボルの情報を示しています。各 Type フィールドの先頭に S または C という文字が表示されていますが，S は対象のシンボルがセグメントシンボルであることを示し，C は対象のシンボルが共有シンボルであることを示しています。

明示的に初期化されていないグローバル変数に対して，CCU8 は共有シンボルとして扱います。例えば，_Fifo_signalToKey や_Fifo_signalToMode などは先頭にアンダスコアが付加されている共有シンボルであることから，C プログラムのグローバル変数であることが推測できます。上記の

マップ情報を参照することでグローバル変数がどこに割り付けられているかを確認することができます。ただし，明示的に初期化されたグローバル変数に対しては，CCU8 は共有シンボルとしては扱わず，パブリックシンボルとして扱います。したがって，明示的に初期化されたグローバル変数を上記のセグメント割り付け情報から確認することはできません。この場合は，後述のモジュールごとのシンボル情報を参照するか，パブリックシンボルリストを参照してください。

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

上記は，サイズ 0 のセグメントが検出されたときに出力されるものです。$$NINITVAR，$$NINITTAB に対して上記のように出力されるのは，明示的に初期化されたグローバル変数がまったくなかった場合に限られます。この情報については無視してもかまいません。

3.1.4 プログラムおよびデータのサイズ

プログラムおよびデータのサイズは，以下の部分を参照することで分かります。

```
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)
```

それぞれのセグメントタイプ別にセグメントの合計サイズが出力されます。

3.1.5 各シンボルのアドレス

各シンボルのアドレスは，モジュールごとのシンボル情報またはパブリックシンボルリストを参照することで分かります。

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

Module         Value     Type         Symbol
------         -----     ----------   ------
s610001lw
              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
```

上の例は，モジュールごとのシンボル情報を示しています。この情報は，アセンブル時とリンク時に/D オプションが指定されたときにのみ出力されます。アセンブル時に/D オプションを指定していなかったモジュールに対しては，リンク時に/D オプションを指定してもシンボル情報は出力されません。

```
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
```

上の例は，プログラム中で使用されるすべてのパブリックシンボルリストを示しています。この情報は，リンク時に/S オプションが指定されたときに出力されます。グローバル関数や明示的に初期化されたグローバル変数などのアドレスが，このリストに表示されます。

3.2 スタック消費量の算出方法

ユーザが作成したプログラムのスタック消費量を算出するためには，CCU8 の/LE オプションと/CT オプションを使用します。

例

```
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;
}
```

CCU8 の起動時に/LE オプションを指定すると，各関数で使用しているスタックの消費量がエラーリストファイルに出力されます。

エラーリストファイルに出力されるスタック情報の例

```
                        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
```

スタック情報の各フィールドの意味は次のとおりです。

タイトル	意味
FUNCTION	関数名。関数の先頭にアンダスコア(_)が付加されて表示されます。
LOCALS	関数内で使用される自動変数，および一時退避用として使用される領域の合計サイズ。
CONTEXT	関数の入り口で保存されるレジスタの合計サイズ。
OTHERS	関数の呼び出しにおいて，スタックに積まれる引数の合計サイズ。関数呼び出しが複数存在する場合は，それらのうちの最大値。
TOTAL	LOCALS，CONTEXT，OTHERS フィールドの合計値。

ここで示されるスタック消費量は，関数単体のみのスタック消費量となります。他のサブ関数を呼び出している場合でも，サブ関数のスタック消費量は含まれません。したがって，この場合は関数の呼び出し関係を調べて，呼び出し元の関数のスタック消費量にサブ関数のスタック消費量を加算する必要があります。

関数の呼び出し関係を調べるには，CCU8 の/CT オプションを使用します。

コールツリーリストファイルの出力例

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

上記のコールツリーリストは，前ページの C プログラムに対し/CT オプションを指定してコンパイルしたときに出力されたものです。

このコールツリーリストとエラーリストファイルに出力されたスタック情報から，各関数のスタック消費量は次のように算出されます。

関数名	スタック消費量	解説
main	14 バイト	main からは fn1，fn2，fn3 を呼び出しており，このうち fn2 のスタック消費量（10 バイト）が最大値となるので，その値と main のスタック消費量（4 バイト）の合計（14 バイト）がスタック消費量となる。
fn1	8 バイト	他の関数を呼び出していないので，そのまま fn1 のスタック消費量となる
fn2	10 バイト	fn2 から fn3 を呼び出しているので，fn2 のスタック消費量（6 バイト）と fn3 のスタック消費量（4 バイト）の合計（10 バイト）が fn2 のスタック消費量となる
fn3	4 バイト	他の関数を呼び出していないので，そのまま fn3 のスタック消費量となる

CCU8
プログラミングガイド

SQ003023E004

2012 年 4 月　　第 3 版発行