XLINK ガイド

IAR Embedded Workbench

XLINK-1-j

内容

1　はじめに

1.1　このガイドの対象者

このガイドは、XLINK を使用している、IAR Embedded Workbench（以降、EW と略す）の基本機能およびオペレーションを理解しており、リンカにて自由にコードやデータを配置する方法を習得されたい方を対象にしております。

1.2　対象製品

以下の製品、バージョンに基づいて書かれています。

- EWRL78 V1.30.7
- EW430 V5.60.2
- EW8051 V5.60.2

これ以前のバージョン、他のデバイスをご利用の場合は、それぞれのマニュアルをご参照ください。また、このガイドは XLINK を対象にしておりますので、ILINK を使用している EWARM V5.xx 以後のバージョン、EWRH850、EWRX、EWSH、EWSTM8 をご使用の方は別資料「ILINK ガイド」を御参照ください。

1.3　このガイドの見方について

このガイドには、リンカ設定ファイルとソースコードのサンプルが記載されていますが、この 2 つのサンプルを区別するため、前者は、白地に黒の文字、後者は、赤地に黒の文字で示します。

例：

- リンカ設定ファイルサンプル

```
-Z(CODE)RCODE,CODE=000D8-0FBFF
-Z(CONST)SWITCH=000D8-0FBFF
```

- ソースコードサンプル

```
#pragma location = "MYARRAY"
int myArray[100];
```

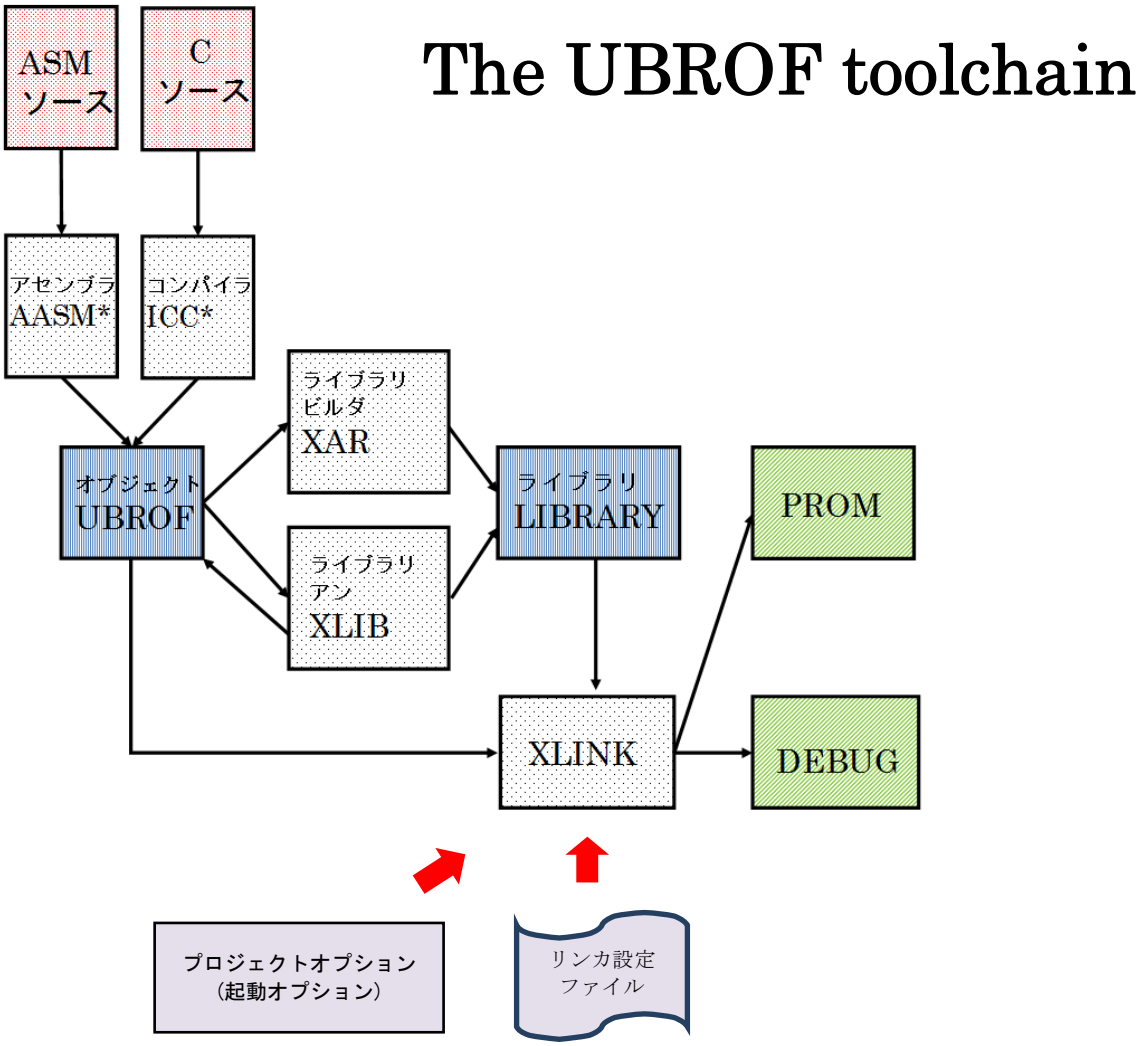
2　XLINK の概要

XLINK は、大きなサイズの再配置可能なマルチモジュールの C/C++プログラム、C/C++プログラムとアセンブラプログラムの混合リンク、サイズの小さい単一ファイルの絶対アドレスを持つアセンブラプログラムのリンクなど、幅広く組込みアプリケーションの開発に適した強力で柔軟性のあるリンカです。

EWRL78 では、選択したデバイス毎に、適切なリンカ設定ファイル（.xcl ファイル）が準備されています。また、EWRL78 のプロジェクトオプションから、

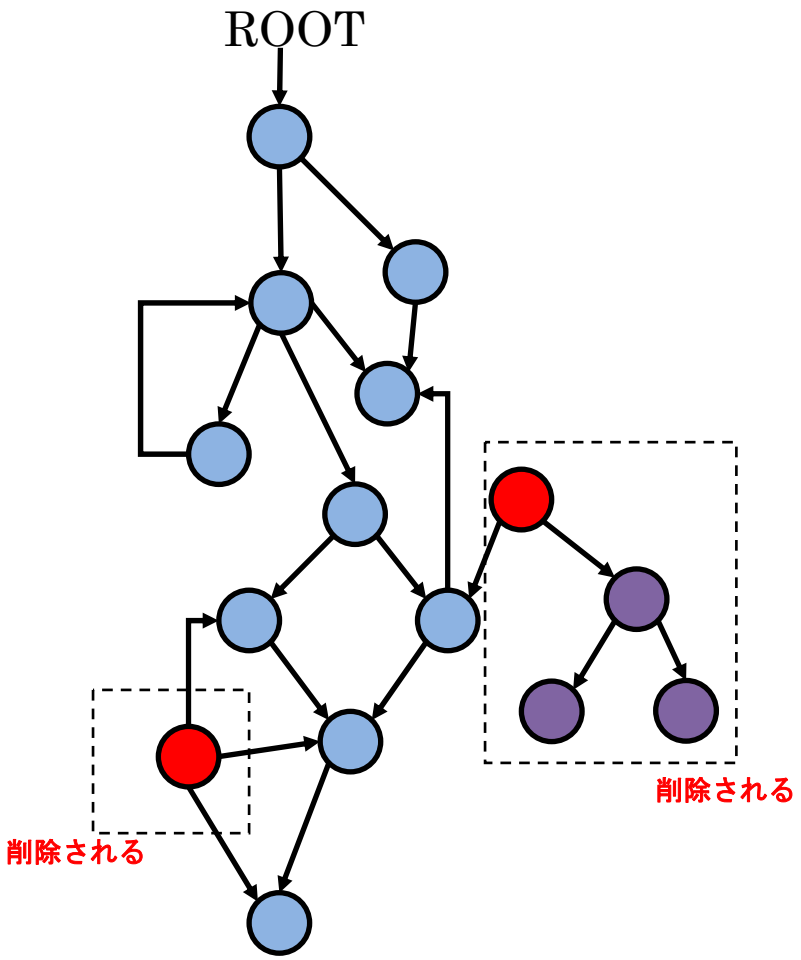
- スタック、ヒープ領域の大きさの指定
- コード、メモリモデルの選択
- ライブラリ選択

が可能です。



2.1　セグメントパート

- XLINK はモジュール単位ではなく、セグメントパート単位で、最終ロードモジュールに含めたり除外したりします。
- セグメントパートは、通常１つの関数か、１つのグローバル変数です。
- プログラムから参照されているセグメントパートは最終ロードモジュールに含まれ、参照されていないセグメントパートは含まれません。
- コンパイラ、アセンブラ、リンカのオプションで、プログラムから参照されていないかも知れないセグメントパートを最終ロードモジュールに強制的に含めることができます。(3.6 章参照)



2.2　XLINK の実行方法とオプションの指定方法

XLINK を使うには、次の2通りの方法があります。

- コマンドラインから実行する方法
- EW 内でビルド作業の一部として実行する方法

本ガイドでは、EW 内で XLINK を実行する場合を中心に説明します。コマンドラインから実行する場合の詳細な情報は、”IAR リンカおよびライブラリツール リファレンスガイド” を参照下さい。

3　リンカ設定ファイルの選択

EW 内で XLINK を実行する場合、次の2つの情報を XLINK 実行オプションに反映させることができます。

- デバイス毎の情報
- プロジェクト毎に設定する情報

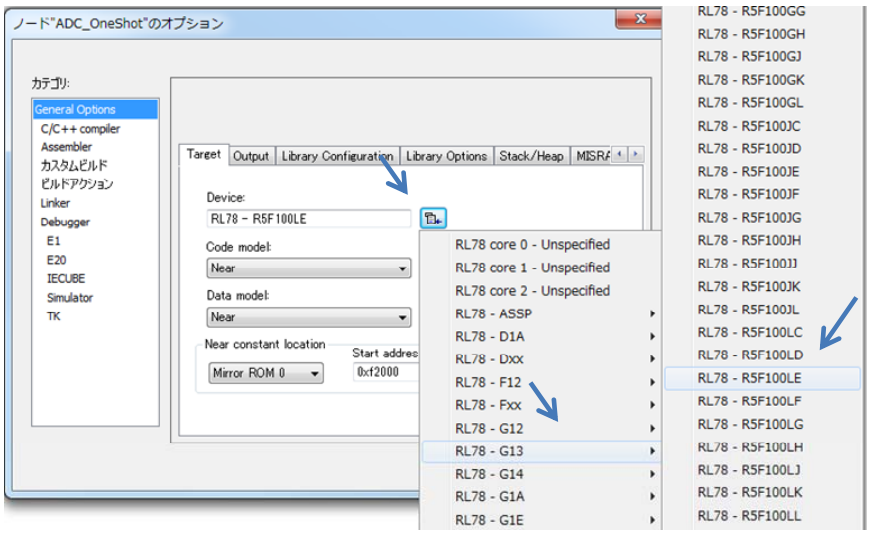
EW では、デバイス毎にリンカ設定ファイルが予め用意されており、プロジェクトオプション画面で指定したデバイスに合致するものがデフォルトで選択されます。これにより、目的のデバイスに合った情報が XLINK 実行オプションに反映されます。なお、リンカ設定ファイルは、予め用意されたファイルの代わりに、独自にカスタマイズしたものを指定することもできます。

リンカ設定ファイルは、スタックサイズやヒープサイズ、メモリのどの番地を使うかといった具体的な値を、別途指定できるように構成されています。これらの値は、EW のプロジェクトオプション画面に項目が用意されており、プロジェクト毎に変更できます。指定した値は、EW 内で XLINK を実行する際に自動的に、リンカ設定ファイルの情報とともに XLINK に渡されます。

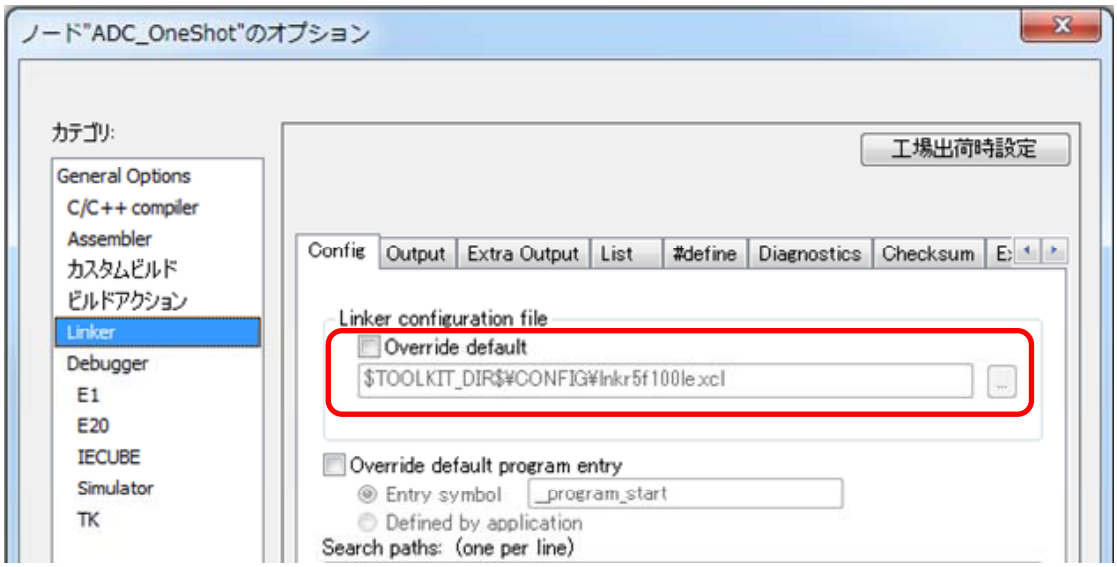
3.1　使用デバイス毎に準備されているリンカ設定ファイル

<u>使用デバイスの選択</u>

Project メニュー ＞ Options



デバイスを選択するとリンカ設定ファイルが自動で選択されます。

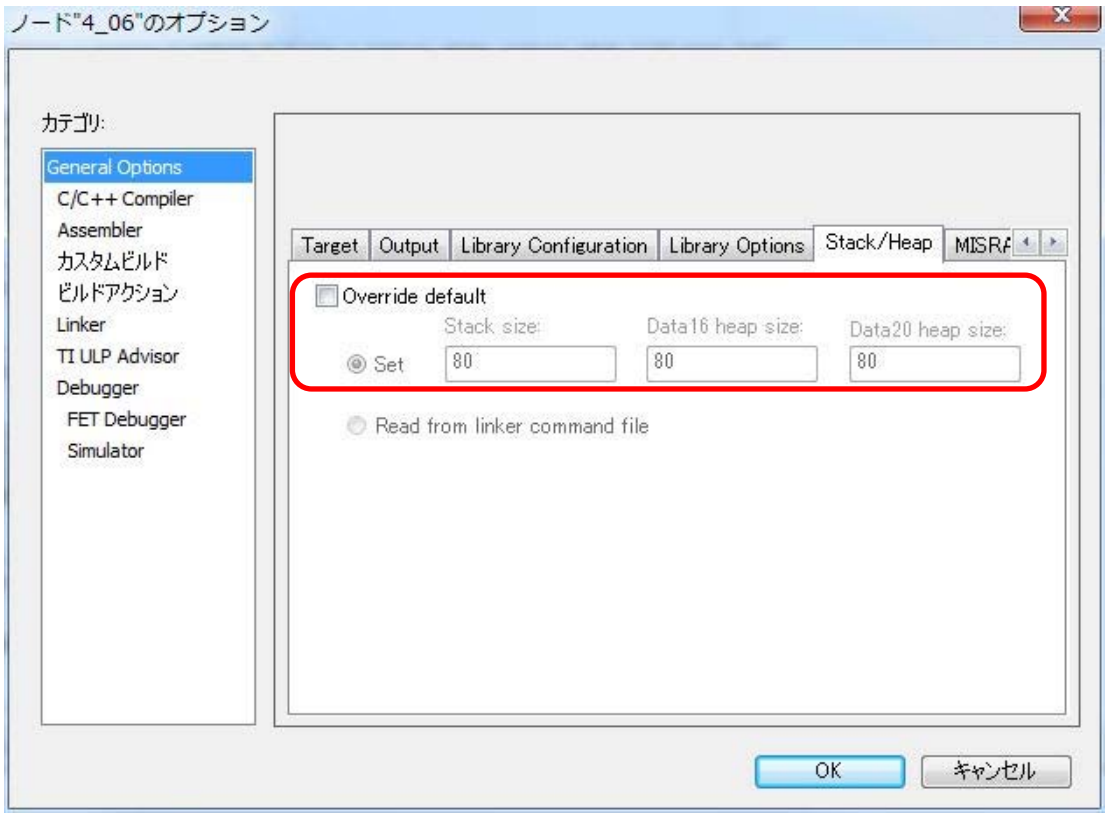


上記のチェックボックスにチェックを入れることで、予め用意されたリンカ設定ファイルの代わりに、独自にカスタマイズしたファイルを指定することができます。

3.2　プロジェクト毎に設定する情報

スタックサイズやヒープサイズ、メモリのどの番地を使うかといった具体的な値は、プロジェクトオプション画面を通じて設定します。設定できる値は、マイコン毎に異なります。

EW430 での設定例



3.3　リンカ設定ファイルの構成

自動で選択されるリンカ設定ファイルの例を示し、その構成について説明します。

```
//-------------------------------------------------------------------
//        XLINK command file template for RL78 microcontroller R5F100LE.
//
//        This file can be used to link object files from the RL78
//        Assembler, ARL78, and the C/C++ compiler ICCRL78.
//
//              This file is generated from the device file:
//              DR5F100LE.DVF
//              Copyright(C) 2012 Renesas
//              Format version V3.00, File version V1.12
//-------------------------------------------------------------------

//-------------------------------------------------------------------
//        The following segments are defined in this template link file:
//
//        INTVEC          -- Interrupt vectors.
//        CLTVEC          -- Calltable vectors.
//        RCODE           -- Code used by C/C++ startup and run-time library.
//        CODE            -- Code used by __near_func, __interrupt and __callt functions.
//        XCODE           -- Code used by __far_func functions.
//        NEAR_CONST_ -- Constants used by __near const.
//        FAR_CONST       -- Constants used by __far const.
//        SADDR_x         -- Variables used by __saddr.
//        NEAR_x          -- Variables used by __near.
//        FAR_x           -- Variables used by __far.
//        NEAR_HEAP       -- The heap used by near data model.
//        FAR_HEAP        -- The heap used by far data model.
//        SWITCH          -- Switch tables used by near code model.
//        FSWITCH         -- Switch tables used by far model.
//        DIFUNCT         -- Dynamic initialization vector used by C++
//
//        Where _x could be one of:
//
//        _Z          -- Initialized data (initvalue = 0 or without init value).
//        _I          -- Initialized data (initvalue != 0).
//        _ID         -- The initial values of _I.
//        _N          -- Uninitialized data, used by __no_init.
//-------------------------------------------------------------------

//-------------------------------------------------------------------
//        Define CPU
//-------------------------------------------------------------------

-cRL78

//-------------------------------------------------------------------
//        Size of the stack.
//        Remove comment and modify number if used from command line.
//-------------------------------------------------------------------

//-D_CSTACK_SIZE=80

//-------------------------------------------------------------------
//        Size of the heaps.
//        Remove comment and modify number if used from command line.
//-------------------------------------------------------------------

//-D_NEAR_HEAP_SIZE=400
//-D_FAR_HEAP_SIZE=4000

//-------------------------------------------------------------------
//        Near constant location.
//        0 -> 0xF0000-0xFFFFF mirrored in rom area 0x00000 - 0x0FFFF
//        1 -> 0xF0000-0xFFFFF mirrored in rom area 0x10000 - 0x1FFFF
//        2 -> Writeable constants
//        Remove comments and modify number if used from command line.
//-------------------------------------------------------------------

//-D_NEAR_CONST_LOCATION=0
//-D_NEAR_CONST_LOCATION_START=02000
//-D_NEAR_CONST_LOCATION_END=0EEFF
```

セグメントの一覧

CPU 指定
(変更しないで下さい)

スタックサイズの指定

ヒープサイズの指定

ミラーエリアの指定

注

XLINK は、RAM を、初期化無し、0 初期化、定数初期化の 3 種類に分けて配置します。

- “_Z”　　--- 初期化されるデータ(初期値 0 もしくは初期値の明示されていない変数)
- “_I”　　--- 0 以外の値で初期化されるデータ
- “_ID”　　--- “_I” のデータの初期値
- “_N”　　--- __no_init を付与した、初期化されないデータ

注

前述のとおり、スタックサイズやヒープサイズなどプロジェクト毎に設定する情報は EW が XLINK を起動する際に指定するものであるため、XLINK をコマンドラインから起動する場合には、EW の代わりに明示的に指定する必要が生じます。そのようなケースに備えて、リンカ設定ファイル中には、プロジェクト毎に設定する情報がコメントアウトされて埋め込まれています。よって、リンカ設定ファイル中の//を削除して直接指定することで、XLINK をコマンドラインからも使用することができます。

ライブラリの選択

```
//-------------------------------------------------------------------------
//          Define the format functions used by printf/scanf.
//          Default is auto formatting.
//          Remove appropriate comment(s) to get desired formatting
//          if used from command line.
//-------------------------------------------------------------------------

//-e_PrintfTiny=_Printf
//-e_PrintfSmall=_Printf
//-e_PrintfLarge=_Printf
//-e_PrintfFull=_Printf

//-e_ScanfSmall=_Scanf
//-e_ScanfLarge=_Scanf
//-e_ScanfFull=_Scanf

//-------------------------------------------------------------------------
//          Define replacement of the default library math functions.
//          Choose either of:
//          1) smaller and faster versions
```

ROM 配置

```
//-------------------------------------------------------------------------
//          Allocate the read only segments that are mapped to ROM.
//-------------------------------------------------------------------------
//-------------------------------------------------------------------------
//          Interrupt vector segment.
//-------------------------------------------------------------------------
-Z(CODE)INTVEC=00000-0007F

//-------------------------------------------------------------------------
//          CALLT vector segment.
//-------------------------------------------------------------------------
-Z(CODE)CLTVEC=00080-000BD

//-------------------------------------------------------------------------
//          OPTION BYTES segment.
//-------------------------------------------------------------------------
-Z(CODE)OPTBYTE=000C0-000C3

//-------------------------------------------------------------------------
//          SECURITY_ID segment.
//-------------------------------------------------------------------------
```

RAM 配置

```
//-------------------------------------------------------------------------
//          Allocate the read/write segments that are mapped to RAM.
//-------------------------------------------------------------------------
//-------------------------------------------------------------------------
//          EEPROM segment.
//          Note: This segment will not be automatically created
//                    and it will not be initialised by CSTARTUP!
//-------------------------------------------------------------------------
-Z(DATA)EEPROM=F1000-F1FFF

//-------------------------------------------------------------------------
//          Short address data and workseg segments.
//-------------------------------------------------------------------------
-Z(DATA)WRKSEG=FFE20-FFEDF
-Z(DATA)SADDR_I,SADDR_Z,SADDR_N=FFE20-FFEDF

//-------------------------------------------------------------------------
//          Near data segments.
//-------------------------------------------------------------------------
-Z(DATA)NEAR_I,NEAR_Z,NEAR_N=FEF00-FFE1F

//-------------------------------------------------------------------------
//          Stack segment.
//-------------------------------------------------------------------------
```

3.4　カスタマイズ用のリンカ設定ファイルの準備

デフォルトで選択されたファイルは、

EWRL78 のインストールフォルダ¥rl78¥config

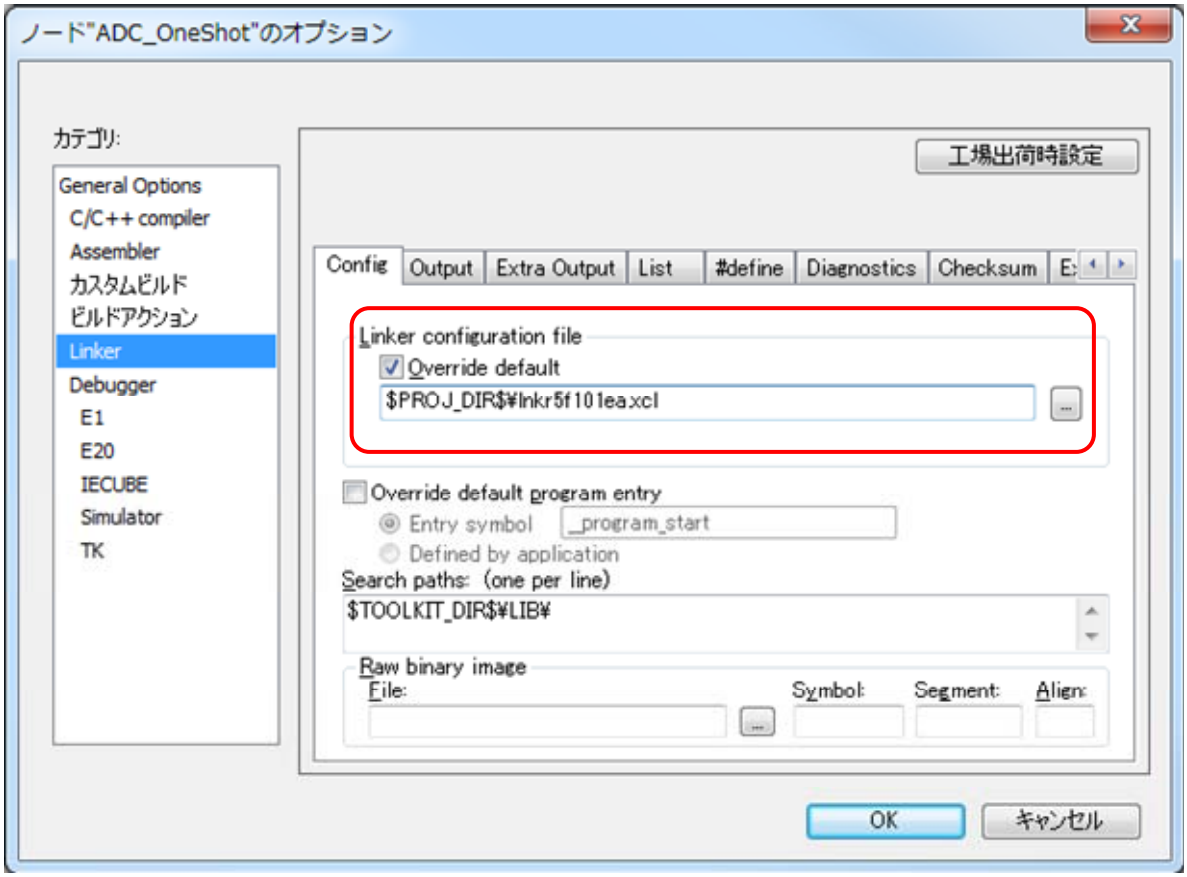
(EWRL78 の場合、他のマイコンの場合はその略称がフォルダ名になります)

フォルダにありますから、これを、プロジェクト名.ewp と同じフォルダにコピーし、下図のように、Override default にチェックを入れ、下のテキストボックスに、

$PROJ_DIR$¥lnkXXXXXXXX.xcl

と記入すると、コピーした xcl ファイルが使用できるようになります。「$PROJ_DIR$」は、プロジェクトファイルのパスを示す環境変数です。

(XXXXXXXXX はマイコンの品名で、下の図では、r5f10lea が選択されています。)



3.5　リンカ設定ファイルのカスタマイズ例

3.5.1　最も基本的な設定

次の例を用いてリンカ設定ファイルの基本的な記述例を示します。

```
int a[ ] = { 0x12345678, 0x23456789, 0x34567890, 0x45678901 };
int b[4];

void main( void )
{
    for(int i = 0; i < 3; i++) b[i] = a[i];
}
```

初期値付きで宣言された配列変数 a と、初期値なしで(デフォルトで 0 に初期化される)配列変数 b があります。
リンカ設定ファイルは次の通りです。

```
//target processor
-cRL78
//interrupt vector
-Z(CODE)INTVEC=00000-0007F
//runtime lib.
-Z(CODE)RCODE=000D8-0FFFF
//code segment
-Z(CODE)CODE=000D8-0FFFF
//near data initialize
-Z(CONST)NEAR_ID=[000D8-0FDFF]/10000
//data segment
-Z(DATA)NEAR_I=F9F00-FFE1F
-Z(DATA)NEAR_Z=F9F00-FFE1F
//far data
-Z(DATA)FAR_Z=[F9F00-FFE1F]/10000
//stack
-Z(DATA)CSTACK+_CSTACK_SIZE=F9F00-FFE1F
```

以下が、リンカのレポートしたシンボルリストとセグメントマップです。

```
                ****************************************
                *                                      *
                *              ENTRY LIST              *
                *                                      *
                ****************************************

Module  Entry                       Address
------  -----                       -------
＜途中省略＞

        a                           000F9F00
        b                           000F9F08
        main                        00000143

＜以下略＞
```

```
                ****************************************
                *                                      *
                *      SEGMENTS IN ADDRESS ORDER       *
                *                                      *
                ****************************************
SEGMENT              SPACE    START ADDRESS   END ADDRESS     SIZE  TYPE  ALIGN
=======              =====    =============   ===========     ====  ====  =====
__aseg                                00000000                     rel     0
INTVEC                           00000000 - 00000001           2   com     1
RCODE                            000000D8 - 00000142          6B   rel     0
CODE                             00000143 - 00000184          42   rel     0
NEAR_ID                          00000186 - 0000018D           8   rel     1
NEAR_I                           000F9F00 - 000F9F07           8   rel     1
NEAR_Z                           000F9F08 - 000F9F0F           8   rel     1
FAR_Z                                 000F9F10                     rel     0
CSTACK                           000F9F10 - 000F9F8F          80   rel     1
```

配列変数 a は、セグメントタイプ NEAR_I に配置されます。同時にその初期値は NEAR_ID に置かれます。配列変数 b は、セグメントタイプ NEAR_Z に配置されます。関数 main は、CODE セグメントに配置されます。

3.5.2 __no_init を付与した変数の配置

初期化しない変数が含まれる例を示します。

```
int a[ ] = { 0x12345678, 0x23456789, 0x34567890, 0x45678901 };
int b[4];
__no_init int c[4];
const int d = 0x56789012;

  : 以下略
```

配列変数 c は、_no_init が付与された、初期化しない変数です。

リンカ設定ファイルは次の通りです。

```
//target processor
-cRL78
//interrupt vector
-Z(CODE)INTVEC=00000-0007F
//near const. seg.
--segment_mirror @NEAR_CONST=NEAR_CONST_ID
-Z(CONST)NEAR_CONST_ID=_NEAR_CONST_LOCATION_START-_NEAR_CONST_LOCATION_END
-Z(DATA)NEAR_CONST=(_NEAR_CONST_LOCATION_START|F0000)-(_NEAR_CONST_LOCATION_END|F0000)
//runtime lib.
-Z(CODE)RCODE=000D8-0FFFF
//code segment
-Z(CODE)CODE=000D8-0FFFF
//near data initialize
-Z(CONST)NEAR_ID=[000D8-0FDFF]/10000
//data segment
-Z(DATA)NEAR_I,NEAR_Z=F9F00-FFE1F
-Z(DATA)NEAR_N=F9F00-FFE1F
//far data
-Z(DATA)FAR_Z=[F9F00-FFE1F]/10000
//stack
-Z(DATA)CSTACK+_CSTACK_SIZE=F9F00-FFE1F
```

以下が、リンカのレポートしたシンボルリストとセグメントマップです。

```
                ****************************************
                *                                      *
                *              ENTRY LIST              *
                *                                      *
                ****************************************

Module  Entry                       Address
------  -----                       -------
＜途中省略＞
        a                           000F9F00
        b                           000F9F08
        c                           000F9F10
        d                           000F3000
        main                        00000143

＜以下略＞
```

```
                ****************************************
                *                                      *
                *      SEGMENTS IN ADDRESS ORDER       *
                *                                      *
                ****************************************


SEGMENT             SPACE    START ADDRESS   END ADDRESS     SIZE  TYPE  ALIGN
=======             =====    =============   ===========     ====  ====  =====
__aseg                              00000000                        rel     0
INTVEC                          00000000 - 00000001            2    com     1
RCODE                           000000D8 - 00000142           6B    rel     0
CODE                            00000143 - 000001AE           6C    rel     0
NEAR_ID                         000001B0 - 000001B7            8    rel     1
NEAR_CONST_ID                   00003000 - 00003001            2    rel     1
NEAR_CONST                      000F3000 - 000F3001            2    rel     1
NEAR_I                          000F9F00 - 000F9F07            8    rel     1
NEAR_Z                          000F9F08 - 000F9F0F            8    rel     1
NEAR_N                          000F9F10 - 000F9F17            8    rel     1
FAR_Z                               000F9F18                        rel     0
CSTACK                          000F9F18 - 000F9F97           80    rel     1
```

3.5.3　RAM 上で実行する関数 (__ramfunc を使用)

関数を RAM に配置する例を示します。なお、以降で述べる方法は RL78 では制限されているため、本項では MSP430 を用いています。

```
int a[ ] = { 0x1234, 0x2345, 0x3456, 0x4567 };
int b[4];
__no_init int c[4];
#define d 0x5678

__ramfunc void sub(int *i, int *j)
{
  *i++ = *j++ + d;
  *i++ = *j++ + d;
  *i++ = *j++ + d;
  *i   = *j   + d;
}

void main(void)
{
  sub(&b[0], &a[0]);
  sub(&c[0], &b[0]);
}
```

関数 sub に__ramfunc を付与することで、RAM に配置されます。

リンカ設定ファイルはデフォルトのまま変更せずに使用できます。

以下が、リンカのレポートしたシンボルリストとセグメントマップです。関数 sub は、変数と同じメモリ領域(0x200～0x9ff の領域)に配置されています。

```
                ****************************************
                *                                      *
                *              ENTRY LIST              *
                *                                      *
                ****************************************

Module  Entry                       Address
------  -----                       -------
＜途中省略＞
        a                           0200
        b                           0208
        c                           0210
        sub                         0218
        main                        1176
＜以下略＞
```

```
              ****************************************
              *                                      *
              *      SEGMENTS IN ADDRESS ORDER       *
              *                                      *
              ****************************************

SEGMENT             SPACE    START ADDRESS   END ADDRESS     SIZE  TYPE  ALIGN
=======             =====    =============   ===========     ====  ====  =====
DATA16_I                           0200 - 0207                   8  rel     1
DATA16_Z                           0208 - 020F                   8  rel     1
DATA16_N                           0210 - 0217                   8  rel     1
CODE_I                             0218 - 024D                  36  rel     1
CSTACK                             09B0 - 09FF                  50  rel     1
DATA16_C                              1100                      0  dse     0
DATA16_ID                          1100 - 1107                   8  rel     1
CSTART                             1108 - 113F                  38  rel     1
ISR_CODE                              1140                      0  dse     0
CODE_ID                            1140 - 1175                  36  rel     1
<CODE> 1                           1176 - 11D5                  60  rel     1
RESET                              FFFE - FFFF                   2  rel     1
```

3.5.4　関数/データのセクション指定とメモリへの配置

関数や変数にセグメントを指定する例を示します。この例では、変数 a11、a12、a13、a14 にセグメント SRAM1 を、変数 a23、a24 にセグメント SRAM2 を、関数 f11、f12 にセグメント SROM1 を、関数 f2 にセグメント SROM2 を、それぞれ指定しています。

```
// セグメントへの配置
#define _SRAM1 _Pragma("location=¥"SRAM1¥"")
#define _SROM1 _Pragma("location=¥"SROM1¥"")

// メモリ
#pragma location="SRAM1"
       int a11,
           a12[16];                    // 記法 1-1

_SRAM1 int a13;                        // 記法 1-2
_SRAM1 int a14[] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16 };

       int a23     @"SRAM2";
       int a24[16] @"SRAM2"
                = {0,1,2,3,4,5,6,7,8, 9,10,11,12,13,14,15 };
                                       // 記法 2
// 関数
#pragma location="SROM1"
       int f11(void)          {return a11;} // 記法 1-1
_SROM1 int f12(void)          {return a13;} // 記法 1-2
       int f2 (void)@"SROM2" {return a23;} // 記法 2

void copy_RAMCODE(void);

void main()
{
  copy_RAMCODE();

  f11();
  f12();
  f2();
  a12[0] = a14[0];
  a14[0] = a24[0];
  return;
}
```

変数にセグメントを指定した場合、変数の初期値を保持する ROM 上のセグメントをリンカ設定ファイルに指定して確保する必要があります。また、プログラムの開始時に、初期値を格納している ROM 領域から、変数として使用する RAM 領域へ、初期値をコピーする操作が必要となります。上記プログラムでは、main 関数の最初で実行される copy_RAMCODE という関数で、RAM 領域へのコピーを行っています。copy_RAOMCODE 関数の記述は次の通りです。

```
#include <stdio.h>
#include <string.h>

/* copy all bytes between s (inclusive) and e (exclusive) to d */

void copy_mem (void * s, void * e, void * d)
{
  size_t size = (size_t) e - (size_t) s;
  memcpy(d, s, size);
}

/* copy the bytes from RAMCODE_ID to RAMCODE */

void copy_RAMCODE(void)
{
#pragma segment = "SRAM1"
#pragma segment = "SRAM1_ID"
#pragma segment = "SRAM2"
#pragma segment = "SRAM2_ID"
  copy_mem(__sfb("SRAM1_ID"), __sfe("SRAM1_ID"), __sfb("SRAM1"));
  copy_mem(__sfb("SRAM2_ID"), __sfe("SRAM2_ID"), __sfb("SRAM2"));
}
```

リンカ設定ファイルに追加する内容は次の通りです。

変数の初期値を保持するセグメントは、慣習的に末尾に_ID を付けています。

```
＜前半省略＞

// ------------------------------------------
// Support for placing functions in read/write memory
//
-QCODE_I=CODE_ID
-QSRAM1=SRAM1_ID
-QSRAM2=SRAM2_ID

＜途中省略＞

// ------------------------------------------
// Read/write memory
//

-Z(DATA)DATA16_I,DATA16_Z,DATA16_N,TLS16_I,DATA16_HEAP+_DATA16_HEAP_SIZE=0200-09FF
-Z(DATA)CODE_I
-Z(DATA)CSTACK+_STACK_SIZE#
-Z(DATA)SRAM1,SRAM2

＜途中省略＞

// ---------------------------------
// Code
//

-Z(CODE)CSTART,ISR_CODE,CODE_ID=1100-FFDF
-P(CODE)CODE=1100-FFDF
-Z(CODE)SROM1,SROM2
-Z(CODE)SRAM1_ID,SRAM2_ID

＜以下省略＞
```

以下が、リンカのレポートしたシンボルリストとセグメントマップです。

変数は 0x200～0x9ff の RAM 領域に配置されており、関数は 0x1100～0xffdf の ROM 領域に配置されています。

```
                ****************************************
                *                                      *
                *              ENTRY LIST              *
                *                                      *
                ****************************************

Module  Entry                       Address
------  -----                       -------
＜途中省略＞
        a11                 0200
        a12                 0222
        a13                 0222
        a14                 0224
        a23                 0244
        a24                 0246
        f11                 11BC
        f12                 11C2
        f2                  11C8
        main                1148
＜以下略＞
```

```
                ****************************************
                *                                      *
                *      SEGMENTS IN ADDRESS ORDER       *
                *                                      *
                ****************************************


SEGMENT             SPACE    START ADDRESS   END ADDRESS     SIZE  TYPE  ALIGN
=======             =====    =============   ===========     ====  ====  =====
DATA16_Z                              0200                          rel     1
DATA16_I                              0200                          rel     1
SRAM1                              0200 - 0243                44   rel     1
SRAM2                              0244 - 0265                22   rel     1
CSTACK                             09B0 - 09FF                50   rel     1
DATA16_ID                             1100                          rel     1
DATA16_C                              1100                          dse     0
CSTART                             1100 - 1127                28   rel     1
<CODE> 1                           1128 - 11BB                94   rel     1
SROM1                              11BC - 11C7                 C   rel     1
SROM2                              11C8 - 11CD                 6   rel     1
SRAM1_ID                           11CE - 1211                44   rel     1
SRAM2_ID                           1212 - 1233                22   rel     1
RESET                              FFFE - FFFF                 2   rel     1
```

3.5.5　RAM 上で実行する関数 (__ramfunc を使用しないバージョン)

__ramfun を使用せずに関数を RAM に配置する例を示します。

この例では、関数 sub にセグメント SRAM2 を指定し、このセグメントを RAM に配置します。

```
#pragma location="SRAM1"
int add(int i, int j)
{
  return i + j;
}

int sub(int i, int j) @"SRAM2"
{
  return i - j;
}

void copy_RAMCODE(void);

int main(void)
{
  copy_RAMCODE();
  return add(1, 2) + sub(3, 4);
}
```

関数が含まれるセグメントを、リンカ設定で RAM に配置する場合、そのコードを保持する ROM 上のセグメントをリンカ設定ファイルに指定して確保する必要があります。また、プログラムの開始時に、ROM 領域から RAM 領域へ、コードをコピーする操作が必要となります。上記プログラムでは、main 関数の最初で実行される copy_RAMCODE という関数で、コードのコピーを行っています。copy_RAOMCODE 関数の記述は次の通りです。

```
#include <stdio.h>
#include <string.h>

/* copy all bytes between s (inclusive) and e (exclusive) to d */

void copy_mem (void * s, void * e, void * d)
{
  size_t size = (size_t) e - (size_t) s;
  memcpy(d, s, size);
}

/* copy the bytes from RAMCODE_ID to RAMCODE */

void copy_RAMCODE(void)
{
#pragma segment = "SRAM2"
#pragma segment = "SRAM2_ID"
  copy_mem(__sfb("SRAM2_ID"), __sfe("SRAM2_ID"), __sfb("SRAM2"));
}
```

リンカ設定ファイルに追加する内容は次の通りです。

```
＜前半省略＞

// ------------------------------------------
// Support for placing functions in read/write memory
//
-QCODE_I=CODE_ID
-QSRAM2=SRAM2_ID

＜途中省略＞

// ------------------------------------------
// Read/write memory
//

-Z(DATA)DATA16_I,DATA16_Z,DATA16_N,TLS16_I,DATA16_HEAP+_DATA16_HEAP_SIZE=0200-09FF
-Z(DATA)CODE_I
-Z(DATA)CSTACK+_STACK_SIZE#
-Z(DATA)SRAM1,SRAM2

＜途中省略＞

// ------------------------------------
// Code
//

-Z(CODE)CSTART,ISR_CODE,CODE_ID=1100-FFDF
-P(CODE)CODE=1100-FFDF
-Z(CODE)SRAM2_ID

＜以下省略＞
```

以下が、リンカのレポートしたシンボルリストとセグメントマップです。関数 add および sub は、変数と同じメモリ領域(0x200～0x9ff の領域)に配置されています。

```
                ****************************************
                *                                      *
                *              ENTRY LIST              *
                *                                      *
                ****************************************

Module  Entry                       Address
------  -----                       -------
＜途中省略＞
        add                 0200
        sub                 0204
        main                110C

＜以下略＞
```

```
                ****************************************
                *                                      *
                *      SEGMENTS IN ADDRESS ORDER       *
                *                                      *
                ****************************************

SEGMENT              SPACE    START ADDRESS   END ADDRESS     SIZE  TYPE  ALIGN
=======              =====    =============   ===========     ====  ====  =====
SRAM1                                0200 - 0203               4   rel     1
SRAM2                                0204 - 0207               4   rel     1
CSTACK                               09B0 - 09FF              50   rel     1
CSTART                               1100 - 110B               C   rel     1
<CODE> 1                             110C - 117D              72   rel     1
SRAM2_ID                             117E - 1181               4   rel     1
RESET                                FFFE - FFFF               2   rel     1
```

3.5.6　変数を RAM の特定の領域に配置する

「myArray[]」という配列を、「MYARRAY」セクションに配置するためには、ソースファイルで以下のように記述します。

```
#pragma location = "MYARRAY"
int myArray[100];


または

int myArray[100] @ "MYARRAY";
```

リンカ設定ファイルでは以下のステートメントを追加します。

```
-Z(DATA)NEAR_I,NEAR_Z,NEAR_N=FEF00-FFE1F
-Z(DATA)MYARRAY=FFD00-FFE1F      <- 追加
```

3.5.7　変数を固定アドレスに配置する(1)

ソースコードを以下のように記述すると、リンカ側の設定は必要ありません。

```
unsiged char test[16] @0x20001520;

または

#pragma location=0x20001520
unsiged char test[16];
```

3.5.8　変数を固定アドレスに配置する(2)

セクションに配置し、そのセクションを固定アドレスに配置します。ソースコードは以下のように記述します。

```
int test @"MYSECTION" = 0x1234;

または

#pragma location="MYSECTION"
int test = 0x1234;
```

リンカ設定ファイルでは、以下のように追記します。

```
-Z(DATA)NEAR_I,NEAR_Z,NEAR_N=FEF00-FFE1F
-Z(DATA)MYSECTION=FFD00-FFE1F      <- 追加
```

3.5.9　関数を固定アドレスに配置する

ソースコードで以下のように記述します。

```
void MyFunc (void) @"MYSECTION" { }

または

#pragma location="MYSECTION"
void MyFunc (void) { }
```

リンカ設定ファイルでは、以下のように追記します。

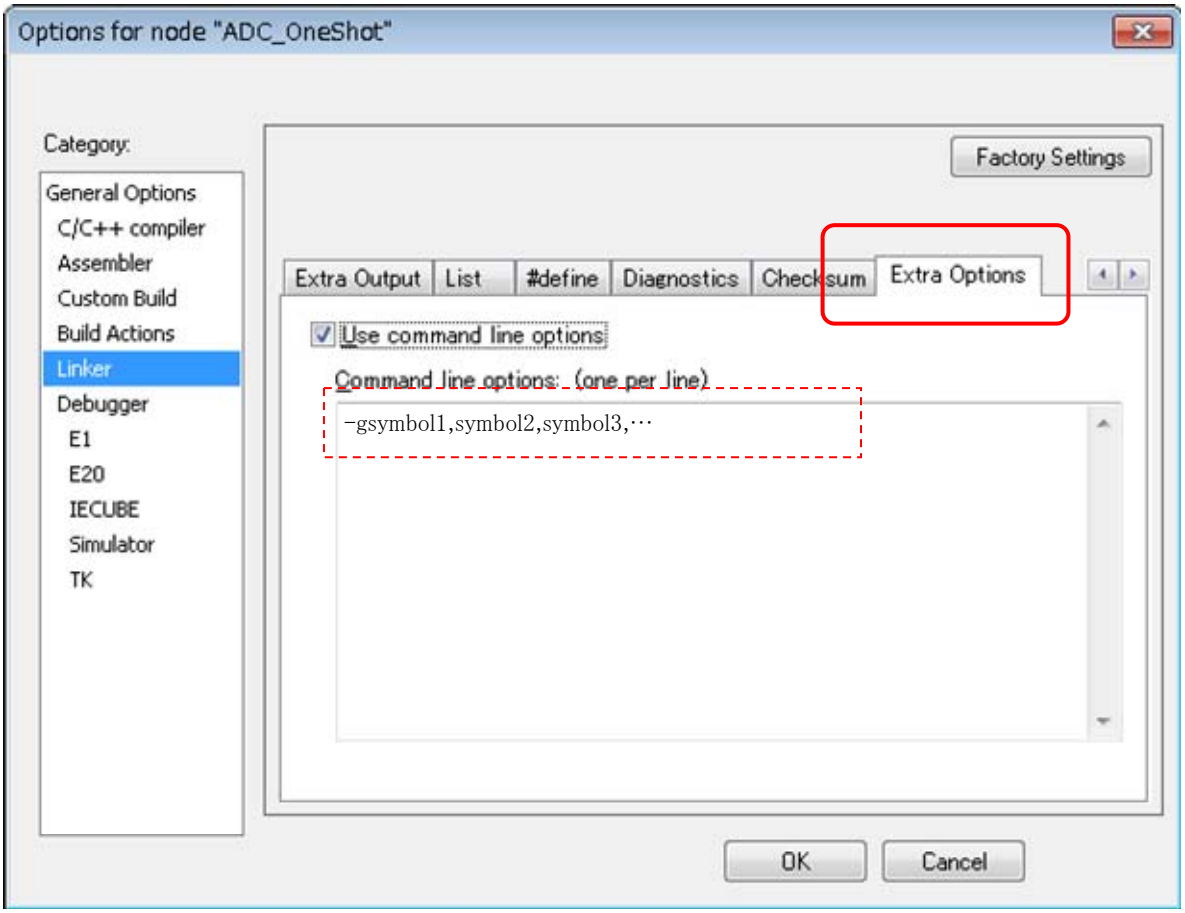
```
-Z(CODE)RCODE,CODE=000D8-0FBFF
-Z(CONST)SWITCH=000D8-0FBFF
-Z(CODE)MYSECTION=0F000      <- 追加
```

3.6　コードやデータの強制配置

XLINK は、デフォルトでは、参照されないコードやデータはリンクせず削除してしまいます。これを抑制する方法について記述します。

3.6.1　リンカオプションで設定する方法

Project メニュー ＞ Options... ＞ Linker ＞ Extra Options タブを開き、Use command line options に、
-gsymbol1[,symbol2,symbol3,…]
を記入します。ここで、symbol1,symbol2,symbol3,…は強制配置したいシンボルです。



3.6.2　ソースコードで指定する方法

「_root」キーワードまたは「#pragma required」が使用できます。

```
__root const char copyright[] = "Copyright by IAR Systems";

または

const char copyright[] = "Copyright by IAR Systems";
#pragma required=copyright
```