IAR EWARM 版本迁移指南

本指南介绍了 IAR Embedded Workbench for ARM 4.xx 版本和 5.xx 版本之间的主要区别，并列举了将在 4.xx 版本下创建的工程迁移到 5.xx 版本时所需进行的主要工作。用户一般只需按照说明进行改动，如遇到未在本指南中说明的特殊部份，可以查阅 IAR Embedded Workbench for ARM 5.xx 所带的 Migration Guide（EWARM_MigrationGuide.pdf）。

修改记录：

Version 1.00：	2008.04.25	by Vincent Hu	Initially created
Version 1.10：	2008.05.21	by Ryan Sheng	Revised and supplemented

目录

1 迁移概述

1.1 EWARM 版本 4.xx 与 5.xx 的区别

IAR EWARM 在版本 4.xx 与 5.xx 之间的主要区别是建立目标代码所用的文件格式不同。在 4.xx 版本下，IAR 使用的是私有的 UBROF 格式，而 5.xx 版本下使用的是业界标准格式 ELF/DWARF。遵循 ARM 公司提出的 ABI（Application Binary Interface）标准，EWARM 5.xx 提供了目标文件级别的兼容性，即其它 ABI 兼容工具生成的目标库可以与 EWARM 生成的目标文件一起链接并调试；同时 EWARM 生成的目标库也能在其它 ABI 兼容工具里参与链接和调试，使得应用程序的开发更具灵活性。当然，这也意味着 EWARM 5.xx 里使用了全新版本的链接器 ILINK 来取代原先所用的 XLINK，从而导致链接器配置文件也使用了新的格式：ICF，而不再是原先的 XCL。关于这两种配置文件的格式和内容，下面还会详细介绍。

1.2 迁移工作

因为 EWARM 4.xx 和 5.xx 之间存在的上述差异，使得在 4.xx 版本下面创建的 Project 不能直接在 5.xx 版本中使用。对于 EWARM 的新用户，或者有经验的用户开发新的项目来说，首选的建议是基于 5.xx 中的相关例程来建立新的 Project。如果在某些情况下，不得不基于以前在 4.xx 下所创建的 Project 进行工作，则一般来说可能会有以下几个方面需要修改：

- C/C++语言源代码
- 汇编语言源代码
- 链接器配置文件
- 运行时环境和目标文件
- 工程配置文件

对于具体的应用程序来说，通常并不是上面提到的每个部份都需要考虑。下面主要针对链接器配置文件的修改来介绍迁移过程中需要注意的内容。

2 链接器和链接器的配置

2.1 EWARM 4.xx 的链接器 XLINK 及其配置文件.xcl

XLINK 链接器可以把 IAR 汇编器或编译器所产生的可重定位的 UBROF 目标文件转换成针对目标处理器的机器码。XLINK 一般通过外部链接器命令文件（*.xcl）来配置，当然也可以在命令行中直接在 xlink 命令之后输入链接选项，或者也可以在 XLINK_ENVPAR 环境变量中设置链接选项。下面介绍 XCL 文件中常用的链接选项，以便在版本迁移之前，确切地了解 XCL 文件的含义。

2.2 XLINK 选项

下面介绍几个XCL文件中常见的链接器配置选项。更详细的内容请查阅XLINK的参考手册：IAR Linker and Library Tools Reference Guide。

-D　-Dsymbol=value

作用：

使用-D 选项可以定义一些纯粹的符号，一般用于声明常数。

参数：

symbol 是未在其它地方定义过的外部符号，value 是 symbol 所代表的值。例如：

```
// Code memory in FLASH
-DROMSTART=0x8000000
-DROMEND=0x801FFFF
```

就定义了 2 个标识了 ROM 起始和结束地址的符号，这样以后关于 ROM 地址的配置都可以直接使用这 2 个符号，使得配置文件的可读性增强。

-Z　-Z [@] [(SPLIT-)type] segments [=|#] range [, range] …

作用：

使用-Z 命令的目的是规定 segments 在存储空间中占据的位置和区间。如果链接器发现某个 segment 没有使用-Z，-b 或者-P 中的任何一个命令进行定义，则会报错。

参数：

@　　使用@参数，表示为 segments 分配空间时不考虑任何已经被使用的地址空间。这适用于当某些 segments 的地址空间需要发生重叠的情形。

type　　参数 type 规定了 segments 的存储类型，默认为 UNTYPED。表 1 列举了 IAR 的 ARM C/C++编译器所支持的 segments 类型。

Segment 存储类型	描述
CODE	可执行代码
CONST	存放在 ROM 中的数据
DATA	存放在 RAM 中的数据

表 1　Segment 存储类型

segments　　参数 segments 列举了参与链接的一个或多个 segment，中间可用逗号分隔。这些 segments 在存储空间中的顺序和被列举的先后顺序一致。在 segment 名后面添加‘+nnnn’，可以给 XLINK 为该 segment 所分配的空间增加 nnnn 字节。表 2 列举了预定义的 segment 名。

Segment	描述
CODE	程序代码
CODE_I	由__ramfunc 声明的，运行在 RAM 中的代码
CODE_ID	用于初始化 CODE_I 的代码
DATA_C	常量
DATA_I	初始化的静态和全局变量
DATA_ID	用于初始化 DATA_I 的数据
DATA_N	由__no_init 声明的静态和全局变量
DATA_Z	未初始化的静态和全局变量
DIFUNCT	C++所要求的动态初始化代码
SWITAB	软件中断向量表
INITTAB	需要被初始化的 segments 地址和大小列表
INTVEC	异常向量表
ICODE	初始化代码
CSTACK	User 和 System 模式所用到的栈
IRQ_STACK	IRQ 模式所用到的栈
HEAP	堆

表 2　Segment 名称以及描述

=|#　　规定了 segments 在存储空间中如何分配，其中‘=’从指定范围的起始处开始为 segments 分配空间，而‘#’从指定范围的结尾处开始为 segments 分配空间。如果这两个参数都没有出现，则 segments 会被分配在当前最后一个有确定链接地址的 segment 后面；如果当前还没有任何 segment 被链接，就分配在 0 地址。

range　　参数 range 规定了分配 segments 时的地址范围。

SPLIT　　参数 SPLIT 用于分隔存储 segments。

-Q　-Q *segment = initializer_segment*

作用：

自动设置 segment 的拷贝初始化。链接器会产生一个新的 *initializer_segment*（如 CODE_ID），其内容与 *segment*（如 CODE_I）完全一致。相关的符号表和调试信息都会和 *segment* 相关联（如 CODE_I）。*initializer_segment* 的内容（通常在 ROM 中）必须在初始化阶段被复制到 *segment*（通常在 RAM 中）。

-c　-cprocessor

作用：

规定目标处理器的类型。如-carm。

2.3 XCL 文件举例

```
// Code memory in FLASH
-DROMSTART=0x8000000
-DROMEND=0x801FFFF

// Data in RAM
-DRAMSTART=0x20000000
-DRAMEND=0x20004FFF
```

图 1　XCL（1）

图 1 使用-D 命令，定义了标识 ROM、RAM 起止地址空间的符号，便于以后使用。

```
-carm
```

图 2　XCL（2）

图 2 使用-c 命令，规定了目标处理器是 arm 处理器。

```
//*************************************************
// Address range for reset and exception
// vectors (INTVEC).
//*************************************************

-Z(CODE)INTVEC=ROMSTART-ROMEND

//*************************************************
// Startup code and exception routines (ICODE).
//*************************************************

-Z(CODE)ICODE,DIFUNCT=ROMSTART-ROMEND
-Z(CODE)SWITAB=ROMSTART-ROMEND

//*************************************************
// Code segments may be placed anywhere.
//*************************************************

-Z(CODE)CODE=ROMSTART-ROMEND
```

图 3　XCL（3）

图 3 使用-Z 命令，为常用的各代码段分配了空间，如 INTVEC，ICODE，CODE 等。

```
//*************************************************
// Original ROM location for __ramfunc code copied
// to and executed from RAM.
//*************************************************

-Z(CONST)CODE_ID=ROMSTART-ROMEND

//*************************************************
// Various constants and initializers.
//*************************************************

-Z(CONST)INITTAB,DATA_ID,DATA_C=ROMSTART-ROMEND
-Z(CONST)CHECKSUM=ROMSTART-ROMEND
```

```
//*************************************************
// Data segments.
//*************************************************

-Z(DATA)DATA_I,DATA_Z,DATA_N=RAMSTART-RAMEND

//*************************************************
// __ramfunc code copied to and executed from RAM.
//*************************************************

-Z(DATA)CODE_I=RAMSTART-RAMEND
```

图 4　XCL（4）

图 4 使用-Z 命令，为预定义的各常量和变量数据段分配了空间，如 DATA_C，DATA_I，DATA_Z 等。其中 DATA_ID 用于在初始化之前存放 DATA_I 的数据；CODE_ID 用于在初始化之前存放 CODE_I 的数据。

```
-QCODE_I=CODE_ID
```

图 5　XCL（5）

图 5 使用-Q 命令，自动设置 CODE_ID 和 CODE_I 之间的初始化关系。由__ramfunc 声明的代码在运行期间从 CODE_ID 被复制到 CODE_I 中。链接器会忽略 CODE_ID 中的符号地址和调试信息，而使用 CODE_I 中的信息来进行链接定位。CODE_ID 和 CODE_I 之前在图 4 中已经被分配了各自的存储空间。

```
-D_CSTACK_SIZE=800
-D_HEAP_SIZE=400

-Z(DATA)CSTACK+_CSTACK_SIZE=RAMSTART-RAMEND
-Z(DATA)HEAP+_HEAP_SIZE=RAMSTART-RAMEND
```

图 6　XCL（6）

图 6 使用-Z 和-D 命令，在 RAM 中为堆、栈分配了指定大小的空间。

2.4 EWARM 5.xx 的链接器 ILINK 及其配置文件.icf

EWARM 5.xx 中的链接器称为 ILINK。ILINK 可以从 ELF/DWARF 格式的目标文件中提取代码和数据，并生成可执行映像。在 EWARM 4.xx 中，基本的代码和数据链接单元是 segment，而对于 ELF/DWARF 格式而言，基本链接单元是 section。ILINK 根据 ILINK Configuration File（*.icf）来分配这些 sections。由于 XLINK 与 ILINK 是两个完全不同的链接器，所以 XCL 和 ICF 也是两种完全不同的配置文件。下面简要介绍 ICF 文件的格式和内容，以协助用户完成版本迁移。

2.5 ICF 格式浅析

sections 在地址空间中的存放是由 ILINK 链接器来实现的，而 ILINK 链接器是按照用户在 ICF 文件中的规定来放置 sections 的，所以理解 ICF 文件的内容尤其重要。

一个标准的 ICF 文件可包括下面这些内容：

- 可编址的存储空间（memory）
- 不同的存储器地址区域（region）
- 不同的地址块（block）
- Section 的初始化与否
- Section 在存储空间中的放置

下面介绍了几条 ICF 文件中常见的指令，详细内容请参考 ILINK 相关说明文档（EWARM_DevelopmentGuide.pdf）：

define [exported] symbol *name* = *expr*;
作用：
指定某个符号的值。
参数：
exported　　导出该 symbol，使其对可执行镜像可用
name　　符号名
expr　　符号值
举例：
define symbol RAM_START_ADDRESS　= 0x40000000;
define symbol RAM_END_ADDRESS　　= 0x4000FFFF;

define memory *name* with size = expr [, *unit-size*];
作用：
定义一个可编址的存储地址空间（memory）。
参数：
name　　memory 的名称
expr　　地址空间的大小
unit-size　　expr 的单位，可以是位（unitbitsize），缺省是字节（unitbytesize）
举例：
define memory MEM with size = 4G;

define region *name* = *region-expr*;
作用：
定义一个存储地址区域（region）。一个区域可由一个或多个范围组成，每个范围内地址必须连续，但几个范围之间不必是连续的。
参数：
name　　region 的名称
region-expr　　*memory*:[from *expr* **{** to *expr* **|** size *expr*}]，可以定义起止范围，也可以定义起始地址和 region 的大小
举例：
define region ROM = MEM:[from 0x0 size 0x10000];
define region ROM = MEM:[from 0x0 to 0xFFFF];

define block *name*[with *param, param*...]
{
extended-selectors
};
作用：
定义一个地址块（block）；它可以是个空块，比如栈、堆；也可以包含一系列 sections。
参数：
name　block 的名称
param　可以是：　size = *expr*　（块的大小）
maximum size = *expr*　（块大小的上限）
alignment = *expr*　（最小对齐字节数）
fixed order　（按照固定顺序放置 sections）
extended-selector　**[** first | last **] {** *section-selector* | block *name* | overlay *name* **}**
first　最先存放
last　最后存放
section-selector　**[** *section-attribute* **][** section *sectionname* **][**object *filename* **]**
section-attribute　**[** readonly **[** code | data **]** | readwrite **[** code | data **]** | zeroinit **]**
sectionname　section的名称
filename　目标文件的名称
即可以按照section的属性，名称及其所在目标文件的名称这三个过滤条件中，任意选取一个条件，或选取多个条件进行组合，来圈定所要求的sections。
name　block或overlay的名称
举例：
define block HEAP with size = 0x1000, alignment = 4 { };
define block MYBLOCK1 = { section mysection1, section mysection2, readwrite };
define block MYBLOCK2 = { readonly object myfile2.o };

initialize { by copy | manually } [with *param, param*...]
{
section-selectors
};
作用：
初始化 sections。
参数：
by copy　在程序启动时自动执行初始化。
manually　在程序启动时不自动执行初始化。
param　可以是：　packing = **{** none | compress1 | compress2 | auto **}**
copy routine = *functionname*
packing表示是否压缩数据，缺省是auto。
functionname表示是否使用自己的拷贝函数来取代缺省函数。
section-selector　同上
举例：
initialize by copy { rw };

do not initialize
{
section-selectors
};
作用：
规定在程序启动时不需要初始化的 sections。一般用于__no_init 声明的变量段（.noinit）。
参数：
section-selector　　同上
举例：
do not initialize { .noinit };

place at { address *memory*[: *expr*] | start of *region_expr* | end of *region_expr* }
{
extended-selectors
};
作用：
把一系列 sections 和 blocks 放置在某个具体的地址，或者一个 region 的开始或者结束处。
参数：
memory　　memory 的名称
expr　　地址值，该地址必须在 memory 所定义的范围内
region_expr　　region 的名称
extended-selector　同上
举例：
place at start of ROM { section .cstart };
place at end of ROM { section .checksum };
place at address MEM:0x0 { section .intvec };

place in *region-expr*
{
extended-selectors
};
作用：
把一系列 sections 和 blocks 放置在某个 region 中。sections 和 blocks 将按任意顺序放置。
参数：
region-expr　　region 的名称
extended-selector　同上
举例：
place in ROM { readonly };　　/* all readonly sections */
place in RAM { readwrite };　　/* all readwrite sections */
place in RAM { block HEAP, block CSTACK, block IRQ_STACK };
place in ROM { section .text object myfile.o };　/* the .text section of myfile.o */
place in ROM { readonly object myfile.o };　　/* all read-only sections of myfile.o */
place in ROM { readonly data object myfile.o };　/* all read-only data sections myfile.o */

下面列举一些基本的 section 和 block 的功能及其通常所在的存储空间。

Section 名称	描述	存储空间
CSTACK	User 和 System 模式所用到的栈	RAM
IRQ_STACK	IRQ 模式所用到的栈	RAM
HEAP	堆	RAM
.intvec	异常向量表	ROM
.cstart	初始化代码	ROM
.text	程序代码	ROM
.data	初始化的静态和全局变量	RAM
.bss	未初始化的静态和全局变量	RAM
.noinit	由__no_init 声明的静态和全局变量	RAM
.rodata	常量	ROM

表 3　Section 名称以及描述

2.6 ICF 文件举例

```
/*-Specials-*/
define symbol __ICFEDIT_intvec_start__ = 0x08000000;
/*-Memory Regions-*/
define symbol __ICFEDIT_region_ROM_start__ = 0x08000000;
define symbol __ICFEDIT_region_ROM_end__   = 0x0801FFFF;
define symbol __ICFEDIT_region_RAM_start__ = 0x20000000;
define symbol __ICFEDIT_region_RAM_end__   = 0x20004FFF;
/*-Sizes-*/
define symbol __ICFEDIT_size_cstack__ = 0x800;
define symbol __ICFEDIT_size_heap__   = 0x400;
```

图 7　ICF（1）

图 7 中首先定义了一些为了增加可读性的符号，包括异常向量表的起始地址，ROM、RAM 的起止地址和堆、栈的大小等。以前缀__ICFEDIT_开头的符号是由 ICF Editor 自动定义的（见 2.7 节），如果不用 ICF Editor 编辑 ICF 文件，这些符号名可以任意定义。

```
define memory mem with size = 4G;
define region ROM_region   = mem:[from __ICFEDIT_region_ROM_start__   to __ICFEDIT_region_ROM_end__];
define region RAM_region   = mem:[from __ICFEDIT_region_RAM_start__   to __ICFEDIT_region_RAM_end__];
```

图 8　ICF（2）

图 8 定义了可编址的存储空间最大为 4G 字节，以及 ROM 和 RAM 所对应的地址区域。

```
define block CSTACK    with alignment = 8, size = __ICFEDIT_size_cstack__   { };
define block HEAP      with alignment = 8, size = __ICFEDIT_size_heap__     { };
```

图 9　ICF（3）

图 9 中创建了 2 个块：CSTACK、HEAP，用于栈和堆的放置，均为 8 字节对齐。

```
initialize by copy { readwrite };
do not initialize  { section .noinit };
```

图 10　ICF（4）

图 10 表示对所有 readwrite 属性的 sections（如.data，.bss 等）进行自动初始化，而对于.noinit 这个 section 则不做初始化处理。

```
place at address mem:__ICFEDIT_intvec_start__ { readonly section .intvec };

place in ROM_region   { readonly };
place in RAM_region   { readwrite,
                        block CSTACK, block HEAP };
```

图 11　ICF（5）

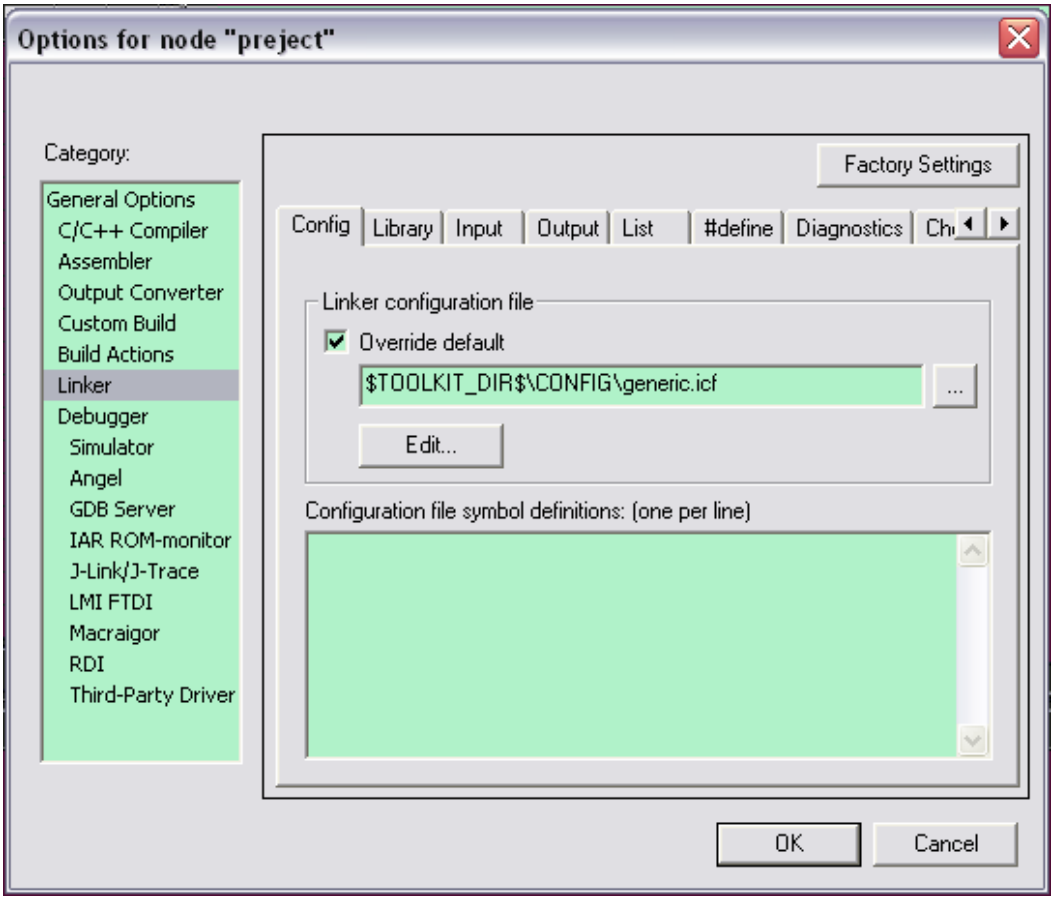
图 11 对所有的 sections 在地址空间中所处的位置进行了配置。首先将只读的异常向量表.intvec 放置在 0x08000000 地址处，然后将余下的只读 sections 以任意顺序存放在 ROM 中，将可读写的 sections 和栈、堆这些块以任意顺序存放在 RAM 中。

上面就是一个基本的 ICF 配置文件的全部内容。

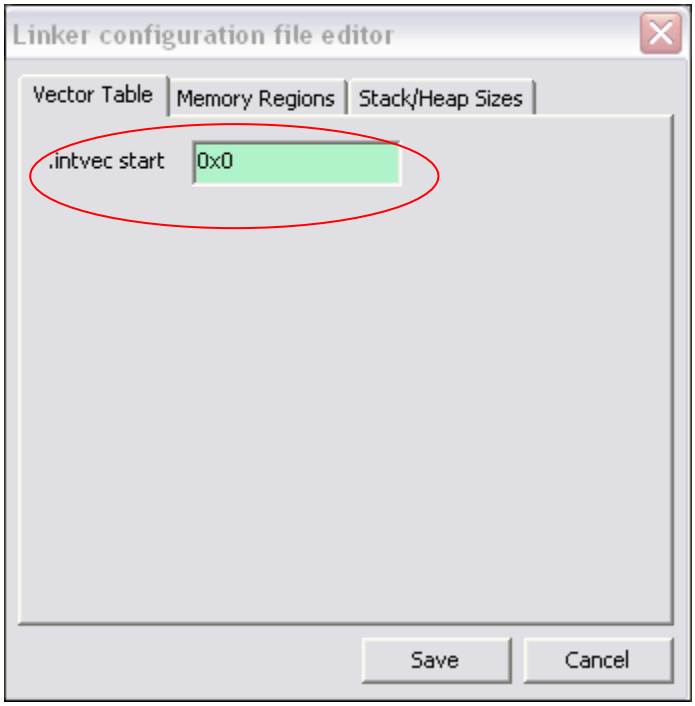
2.7 图形化工具 ICF Editor 的使用

除了完全手工撰写 ICF 文件来完成链接器配置之外，EWARM 5.xx 还为用户提供了界面友好的图形化工具 ICF Editor，用于编辑 ICF 文件的一些基本选项。

1） 选择菜单项 Project -> Options，在 Linker 目录下点击 Config 选项卡，选中 Override default 单选框，输入 ICF 文件名和所在路径，再点击 Edit 按钮：



2） 在 Linker configuration file editor 对话框中可以定义：异常向量表的起始地址、RAM/ROM 存储区域的起止地址、栈和堆的大小等配置数据。



3） 结束后按 Save 键，设置的参数就写入了指定的 ICF 文件中。可以进一步手工编辑该文件以达到其它配置目的。

3 其它

如果在 EWARM 5.xx 中直接打开 4.xx 所创建的工程文件，会有对话框询问是否自动将其转换成 5.xx 的工程文件；若选择 OK，4.xx 的工程文件会被转换成 5.xx 的工程文件，当然原来的 4.xx 工程文件也会自动生成一个备份。某些配置信息无法被自动带入 5.xx 的工程，如链接器配置文件的路径等，因此请仔细检查相关的编译、汇编或链接选项，确保它们具有正确的设置。各配置选项的含义可参阅 EWARM_UserGuide.pdf 的 Part 7.

在 EWARM 5.xx 版本中，默认的程序入口符号（Program Entry）由原先的__program_start 更改为__iar_program_start，因此对于旧的 4.xx 汇编代码而言，需要更改这个入口符号名；当然也可以在 EWARM 5.xx 的 Linker 配置选项中修改默认的 Program Entry。

在 EWARM 4.xx 版本中，在 Linker 配置选项的 Output 和 Extra Output 选项卡中都可以选择生成除 UBROF 格式之外的其他格式输出文件。在 EWARM 5.xx 中，Linker 只能生成 ELF/DWARF 格式的输出文件，若需要 Motorola S-Record，Intel HEX 或简单 Binary 等其它格式的文件，可在 Output Converter 配置选项中设置。

在从 EWARM 4.xx 向 5.xx 版本的迁移过程中，除了需要更改链接器配置文件之外，可能还需要根据应用程序的具体情况，对某些其它方面做小的修改，如 C/C++源代码和汇编语言源代码等，必要时可以参考 IAR Embedded Workbench for ARM 5.xx 所带的 Migration Guide（EWARM_MigrationGuide.pdf）。